Reliable Distributed Systems

Models for distributed computing.
Keeping time in a distributed system.
The Fischer, Lynch and Paterson Result.
The Byzantine Generals Problem.

What time is it?

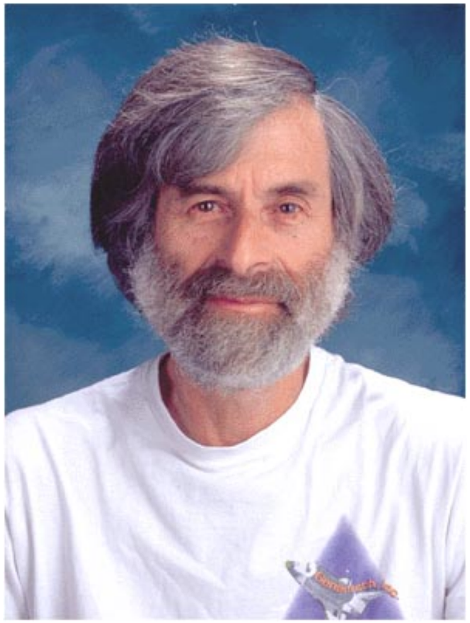
- In distributed system we need practical ways to deal with time
  - E.g. we may need to agree that update A occurred before update B
  - Or offer a “lease” on a resource that expires at time 10:10.0150
  - Or *guarantee* that a time critical event will reach all interested parties within 100ms

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

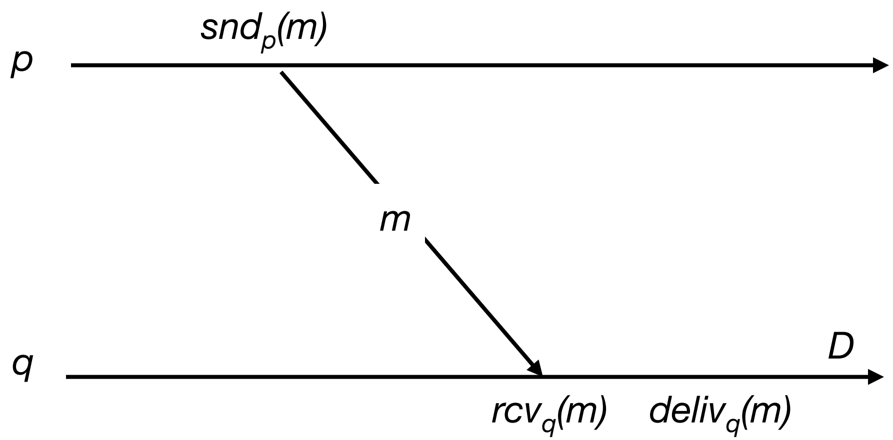
Lamport's approach

- 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.

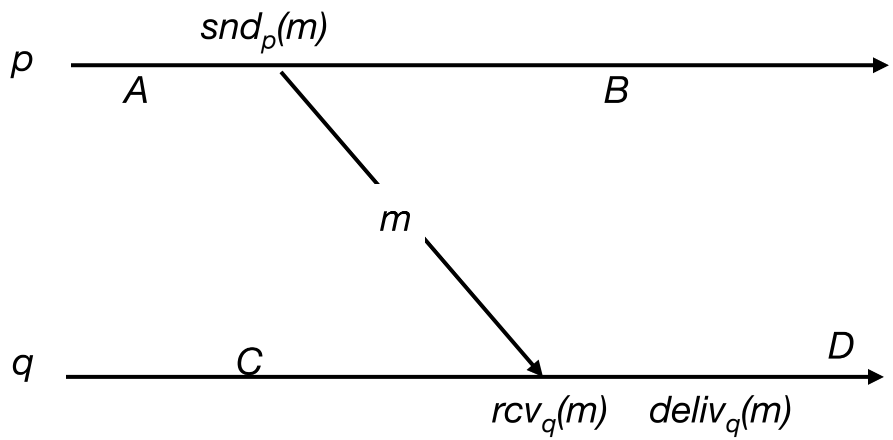


Time, Clocks and the Ordering of Events in a Distributed System. **Communications of the ACM** 21, 7 (July 1978), 558-565.

Drawing time-line pictures:

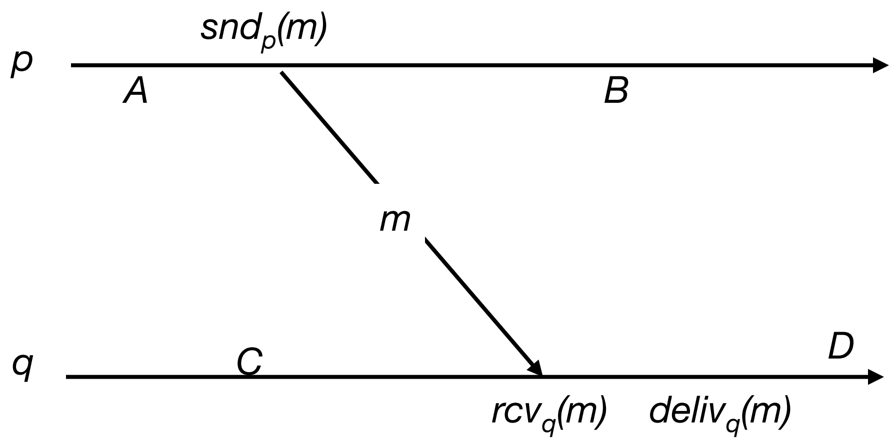


Drawing time-line pictures:



- 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?

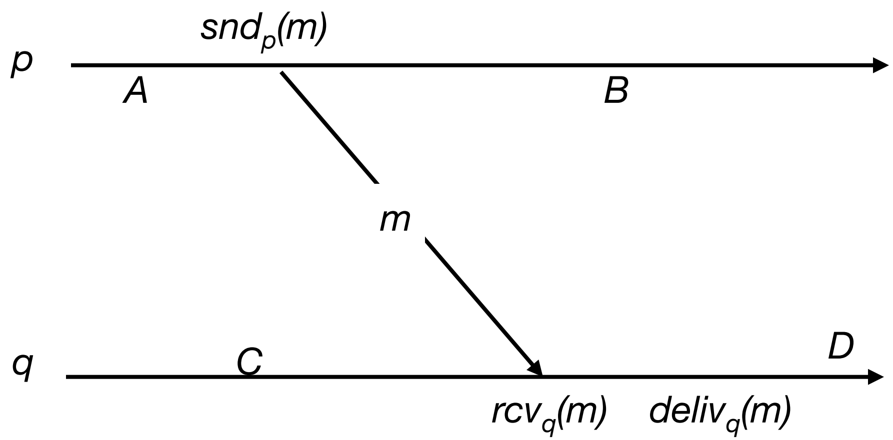
Drawing time-line pictures:



$A \xrightarrow{p} B \qquad C \xrightarrow{q} D$

- A happens before B, and C before D.
  - “Local ordering” at a single process.

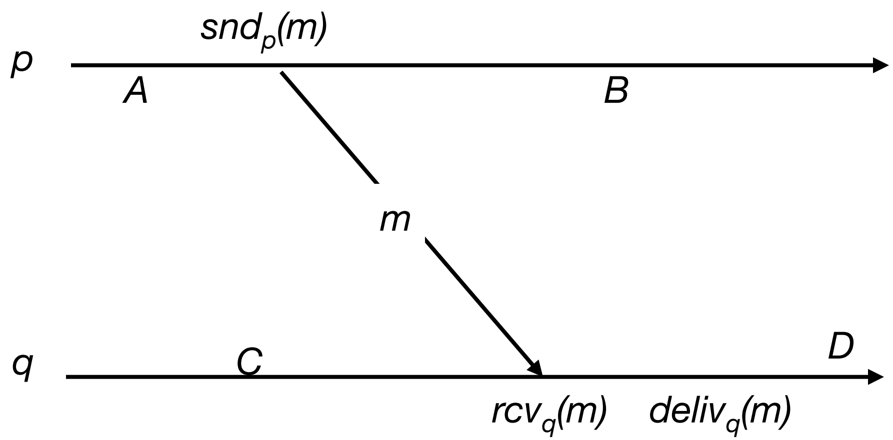
Drawing time-line pictures:



$$snd_p(m) \xrightarrow{M} rcv_q(m)$$

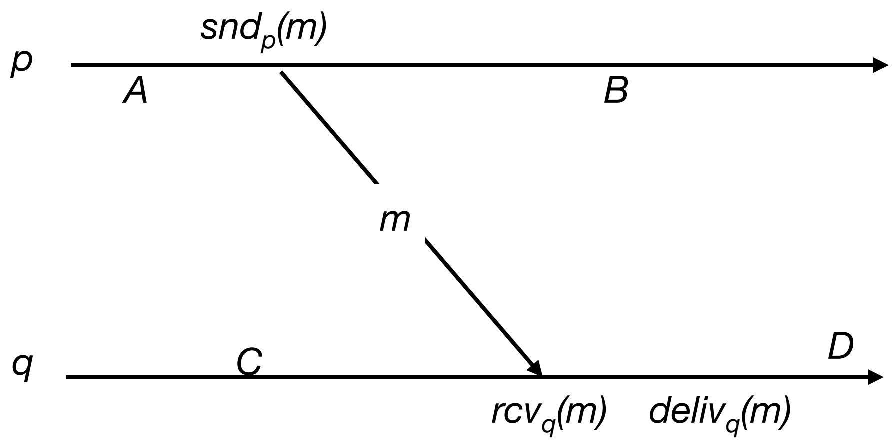
- $snd_p(m)$ also happens before $rcv_q(m)$.
  - “Distributed ordering” introduced by a message.

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 will say that "A happens before B", written A→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."

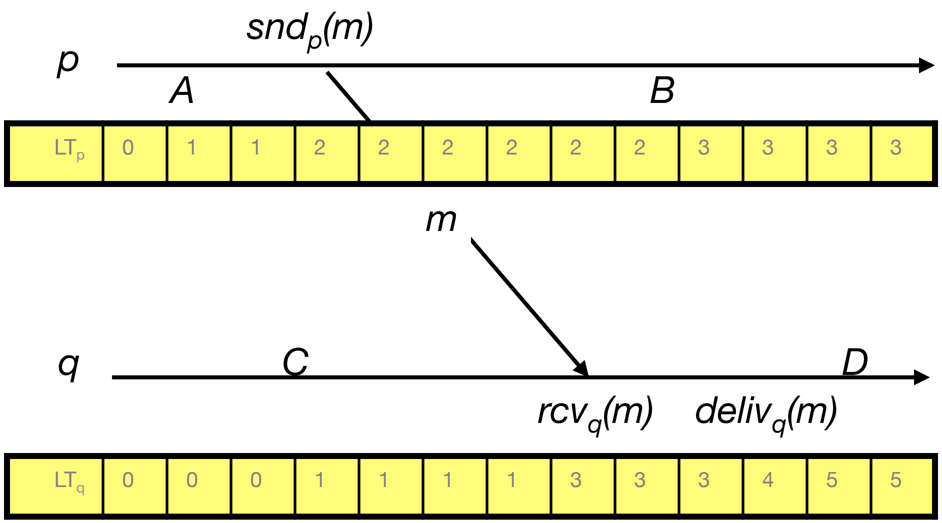
Logical clocks

- 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→B, then LT(A)<LT(B).
- But converse might not be true:
  - If LT(A)<LT(B) can not be sure that A→B.
  - This is because processes that do not communicate still assign timestamps and hence events will “seem” to have an order.

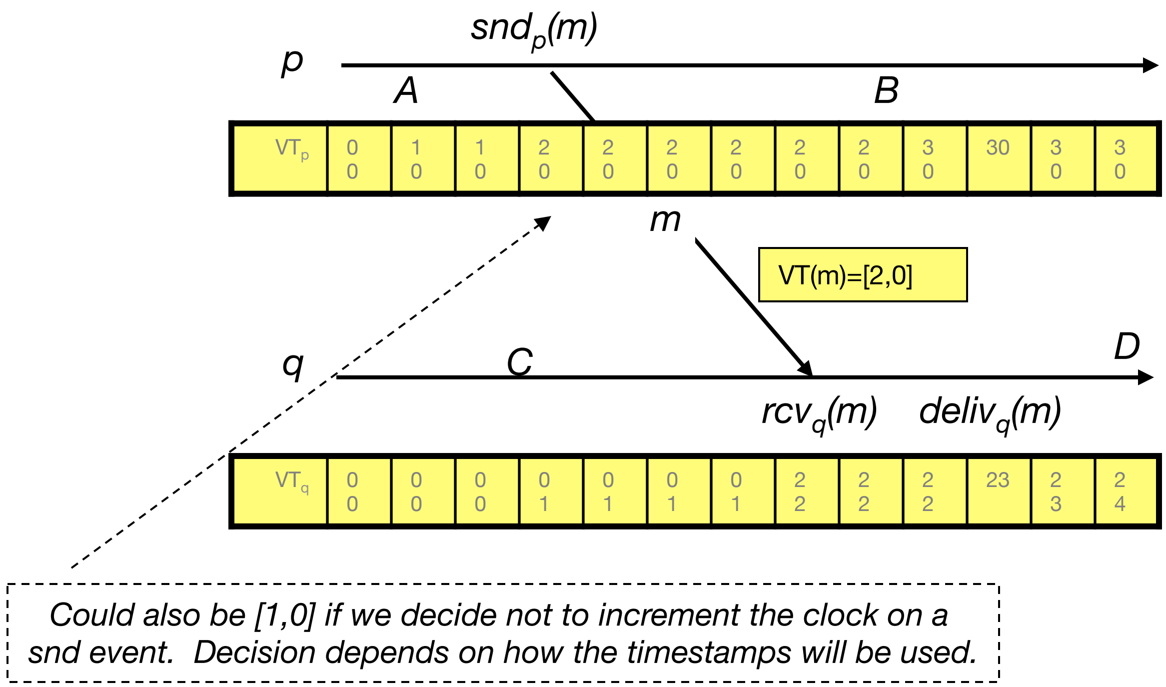
Can we do better?

- 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 will just assign p index 0 and q index 1.
  - Vector clocks require either agreement on the numbering, or that the actual process ids 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 $VT_q=max(VT_q, VT(m))$.

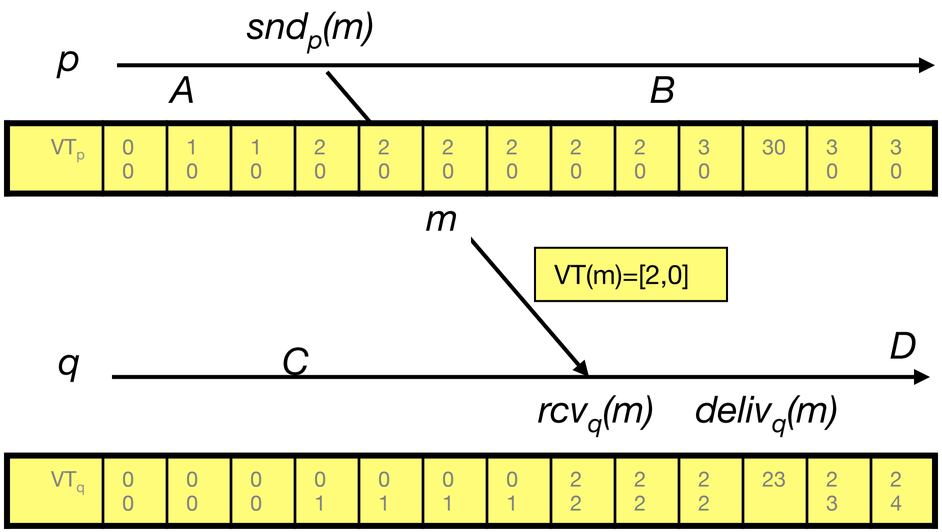
Time-line with VT annotations



Rules for comparison of VTs

- We will say that $VT_A \leq VT_B$ if
  - $\forall_I$, $VT_A[i] \leq VT_B[i]$.
- And we will 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.

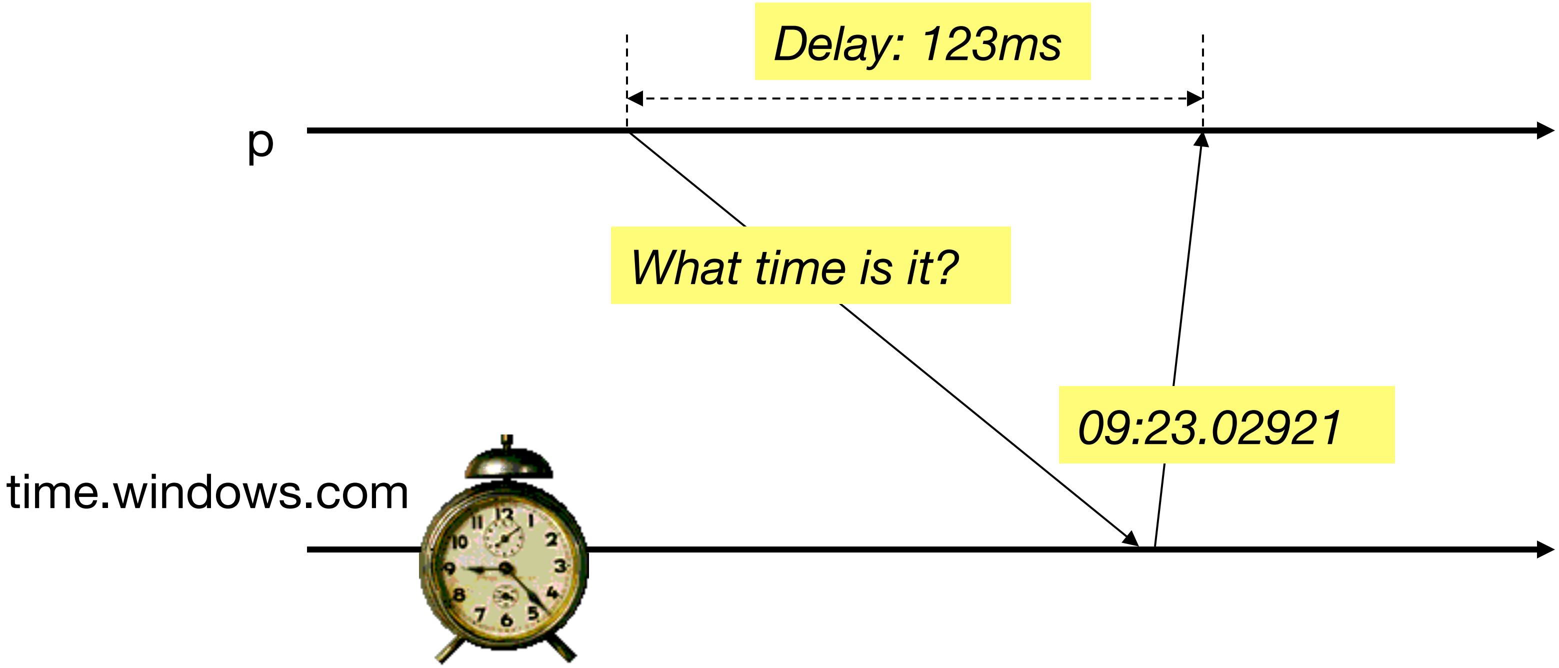
Introducing “wall clock time”

- There are several options:
  - “Extend” a logical clock or vector clock with the clock time and use it to break ties.
    - Makes meaningful statements like “B and D were concurrent, although B occurred first.”
    - But unless clocks are closely synchronized such statements could be erroneous!
  - We use a clock synchronization algorithm to reconcile differences between clocks on various computers in the network.

Synchronizing clocks

- Without help, clocks will often differ by many milliseconds.
  - Problem is that when a machine downloads time from a network clock it can not be sure what the delay was.
  - This is because the “uplink” and “downlink” delays are often very different in a network.
- Outright failures of clocks are rare…

Synchronizing clocks



- *Suppose p synchronizes with time.windows.com and notes that 123 ms elapsed while the protocol was running… what time is it now?*

Synchronizing clocks

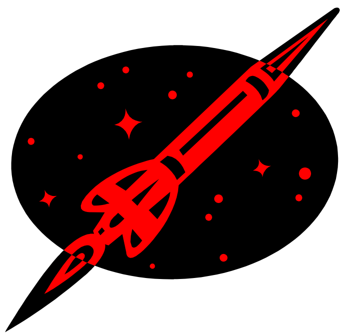
- Options?
  - P could guess that the delay was evenly split, but this is rarely the case in WAN settings (downlink speeds are higher).
  - P could ignore the delay.
  - P could factor in only “certain” delay, e.g. if we *know* that the link takes at least 5ms in each direction. Works best with GPS time sources!
- In general, can not do better than uncertainty in the link delay from the time source down to P.

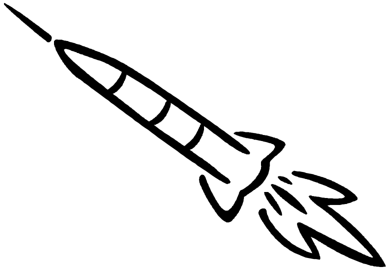
Consequences?

- In a network of processes, we must assume that clocks are
  - Not perfectly synchronized. Even GPS has uncertainty, although small.
    - We say that clocks are "inaccurate."
  - And clocks can drift during periods between synchronizations.
    - Relative drift between clocks is their "precision."

Thought question

- We are building an anti-missile system.
- Radar tells the interceptor where it should be and what time to get there.
- Do we want the radar and interceptor to be as accurate as possible, or as precise as possible?





Thought question

- We want them to agree on the time but it is not important whether they are accurate with respect to “true” time.
  - “Precision” matters more than “accuracy.”
  - Although for this, a GPS time source would be the way to go.
    - Might achieve higher precision than we can with an “internal” synchronization protocol!

Real systems?

- Typically, some “master clock” owner periodically broadcasts the time.
- Processes then update their clocks.
  - But they can drift between updates.
  - Hence we generally treat time as having fairly low accuracy.
  - Often precision will be poor compared to message round-trip times.

Clock synchronization

- To optimize for precision we can:
  - Set all clocks from a GPS source or some other time “broadcast” source.
    - Limited by uncertainty in downlink times.
  - Or run a protocol between the machines.
    - Many have been reported in the literature.
    - Precision limited by uncertainty in message delays.
    - Some can even overcome arbitrary failures in a subset of the machines!

“Simultaneous” actions

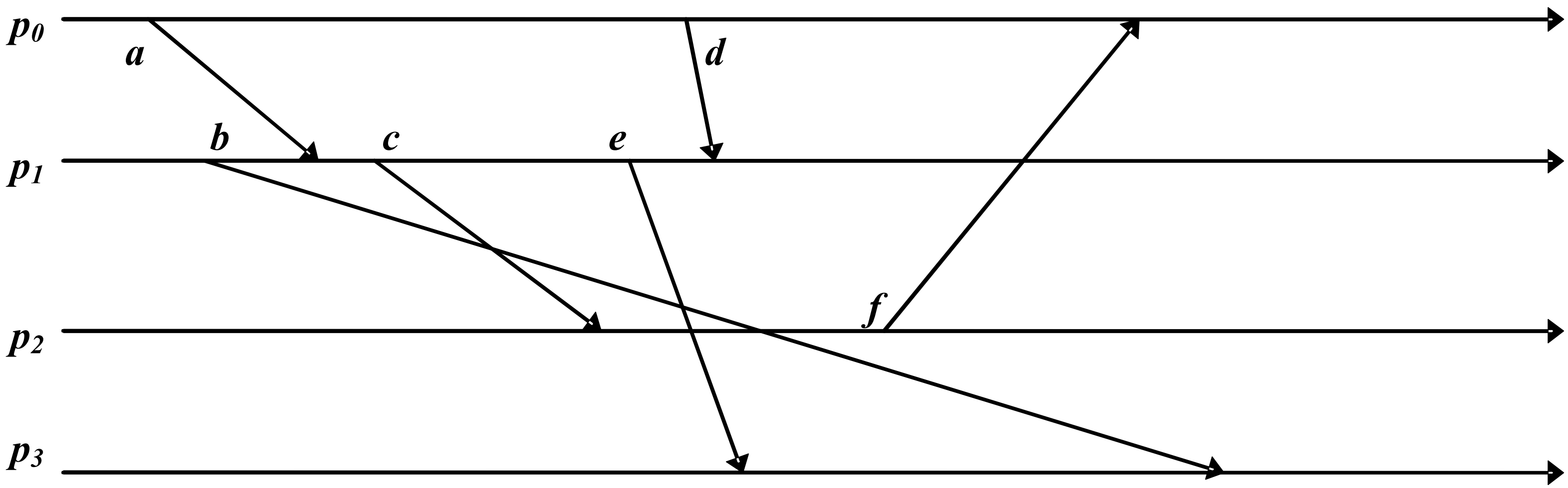
- There are many situations in which we want to talk about some form of simultaneous event
  - Our missile interceptor is one case
  - But think about updating replicated data
    - Perhaps we have multiple conflicting updates
    - The need is to ensure that they will happen in the same order at all copies
    - This “looks” like a kind of simultaneous action

Temporal distortions

- Things can be complicated because we can not 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.

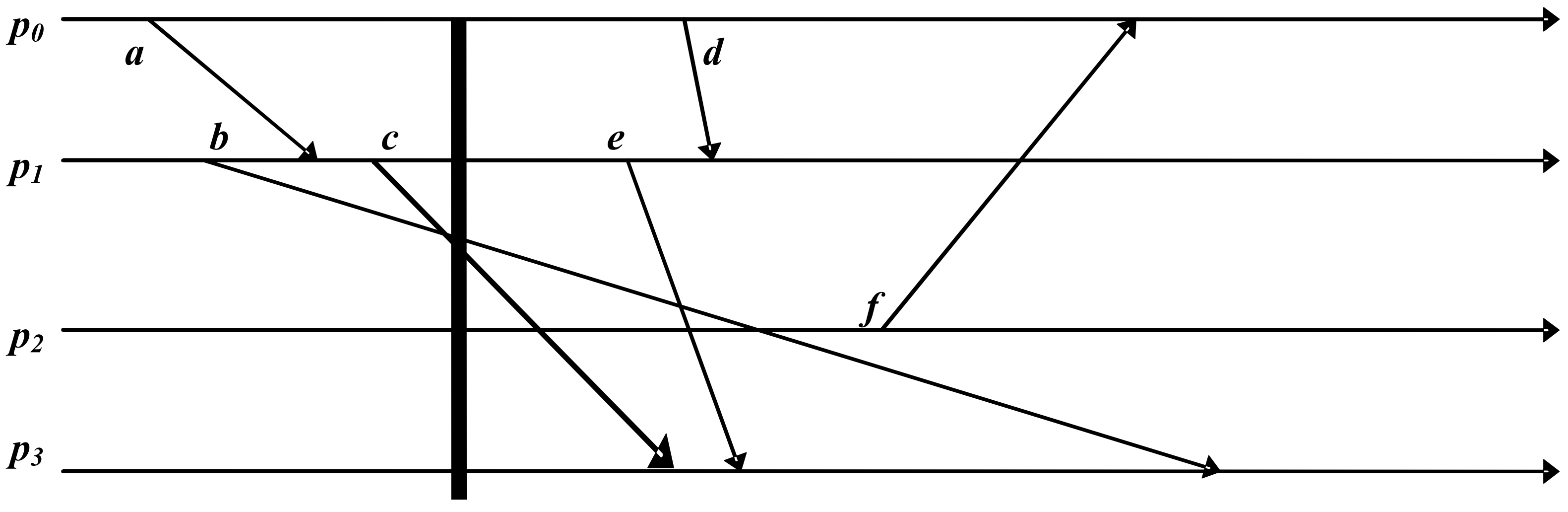
Temporal distortions

- What does "now" mean?



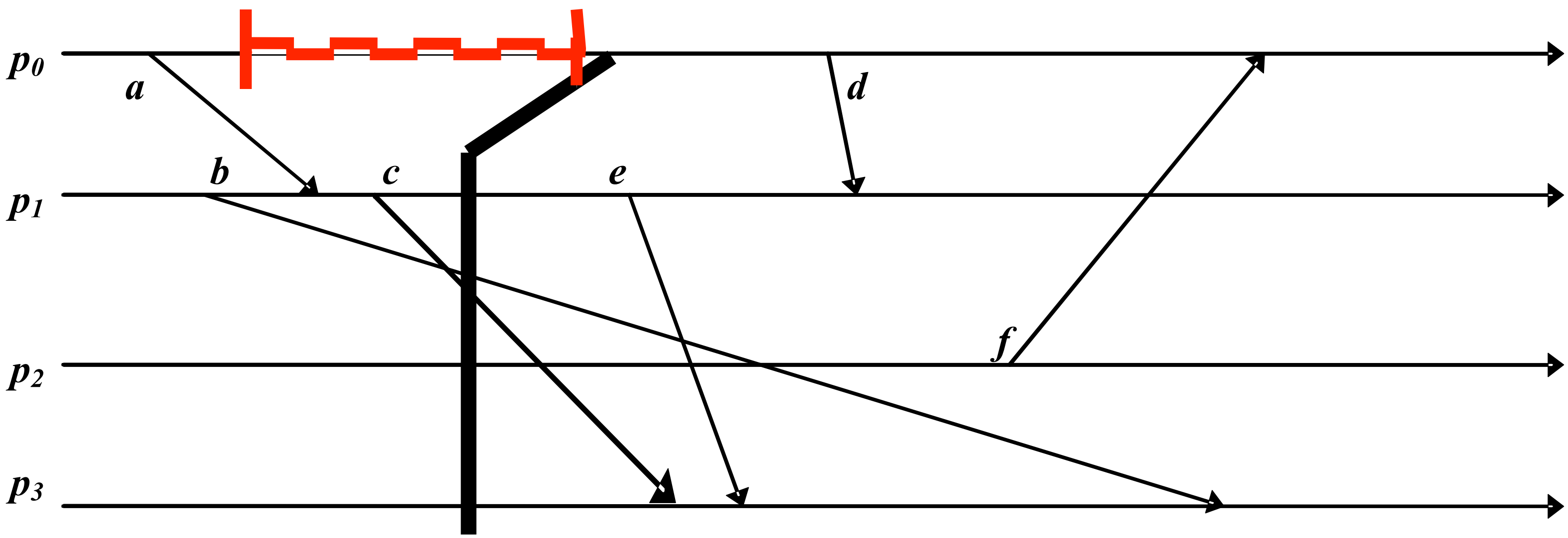
Temporal distortions

- What does "now" mean?



Temporal distortions

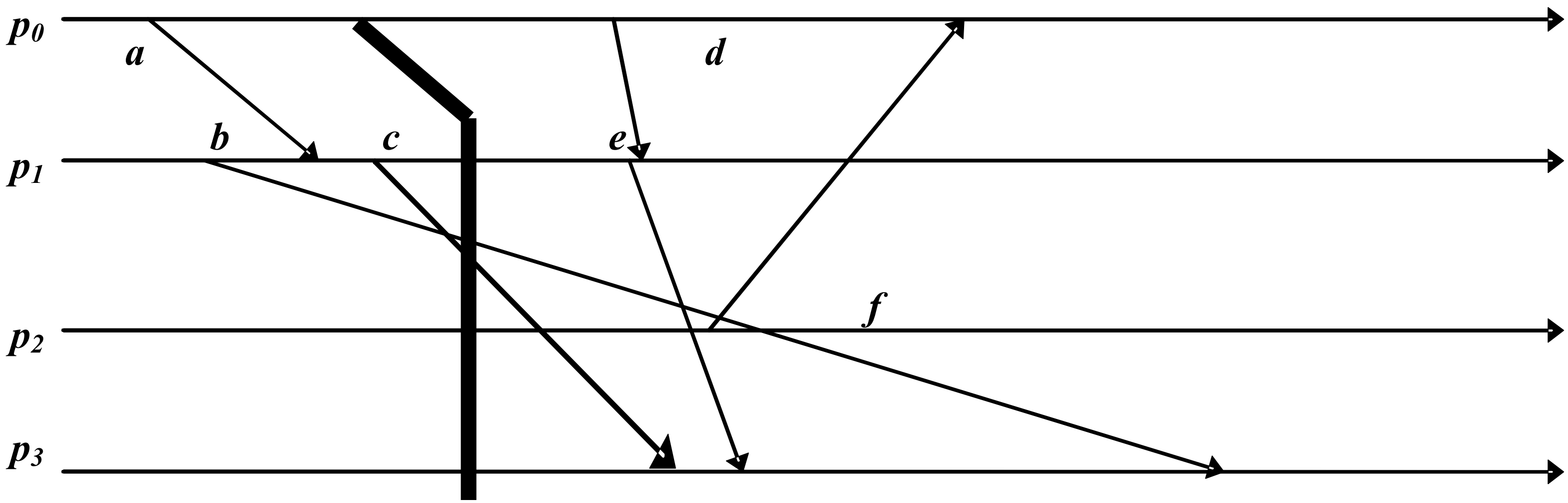
- Timelines can “stretch”…



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

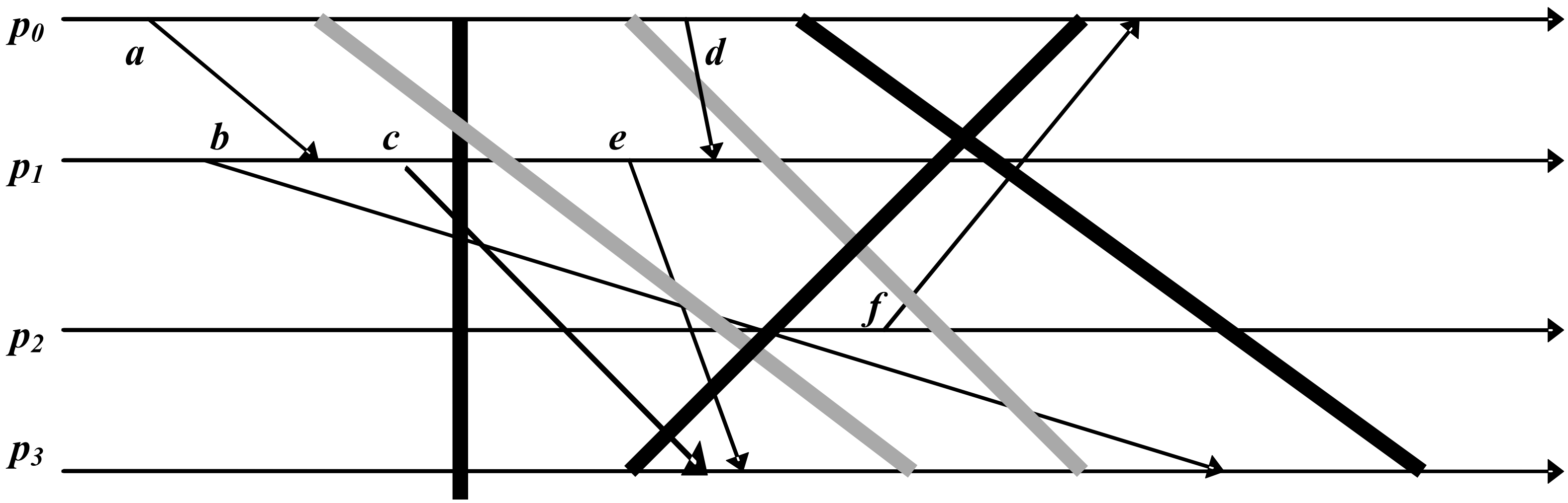
Temporal distortions

- Timelines can "shrink."



- e.g. something lets a machine speed up

Temporal distortions

- Cuts represent instants of time.
- 

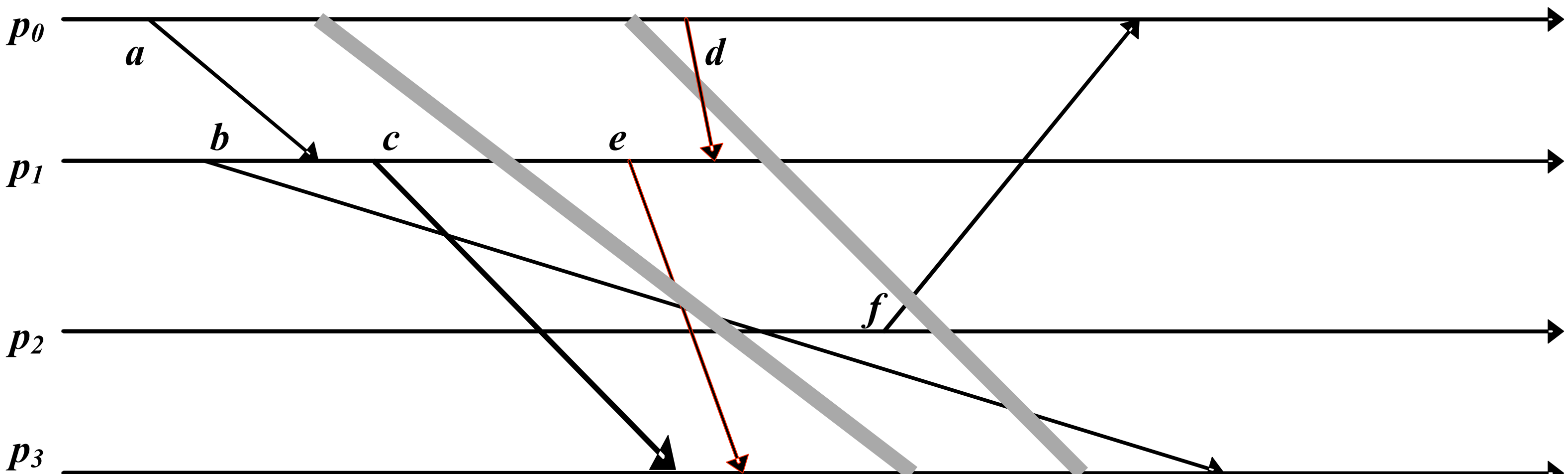
- But not every “cut” makes sense.
  - Black cuts could occur but not gray ones.

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.

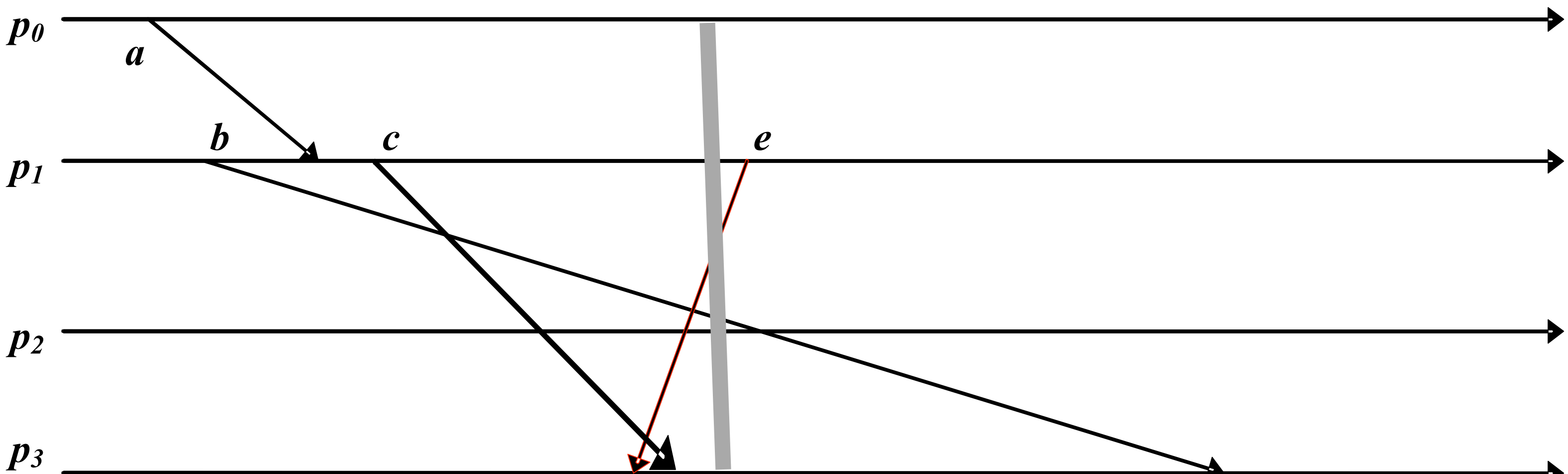
Temporal distortions

- 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.”

Who cares?

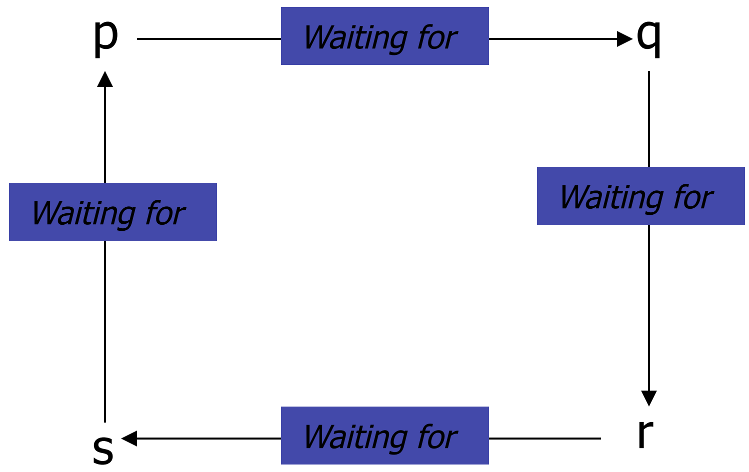
- Suppose, for example, that we want to do distributed deadlock detection.
  - System lets processes “wait” for actions by other processes.
  - A process can only do one thing at a time.
  - A deadlock occurs if there is a circular wait.

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.

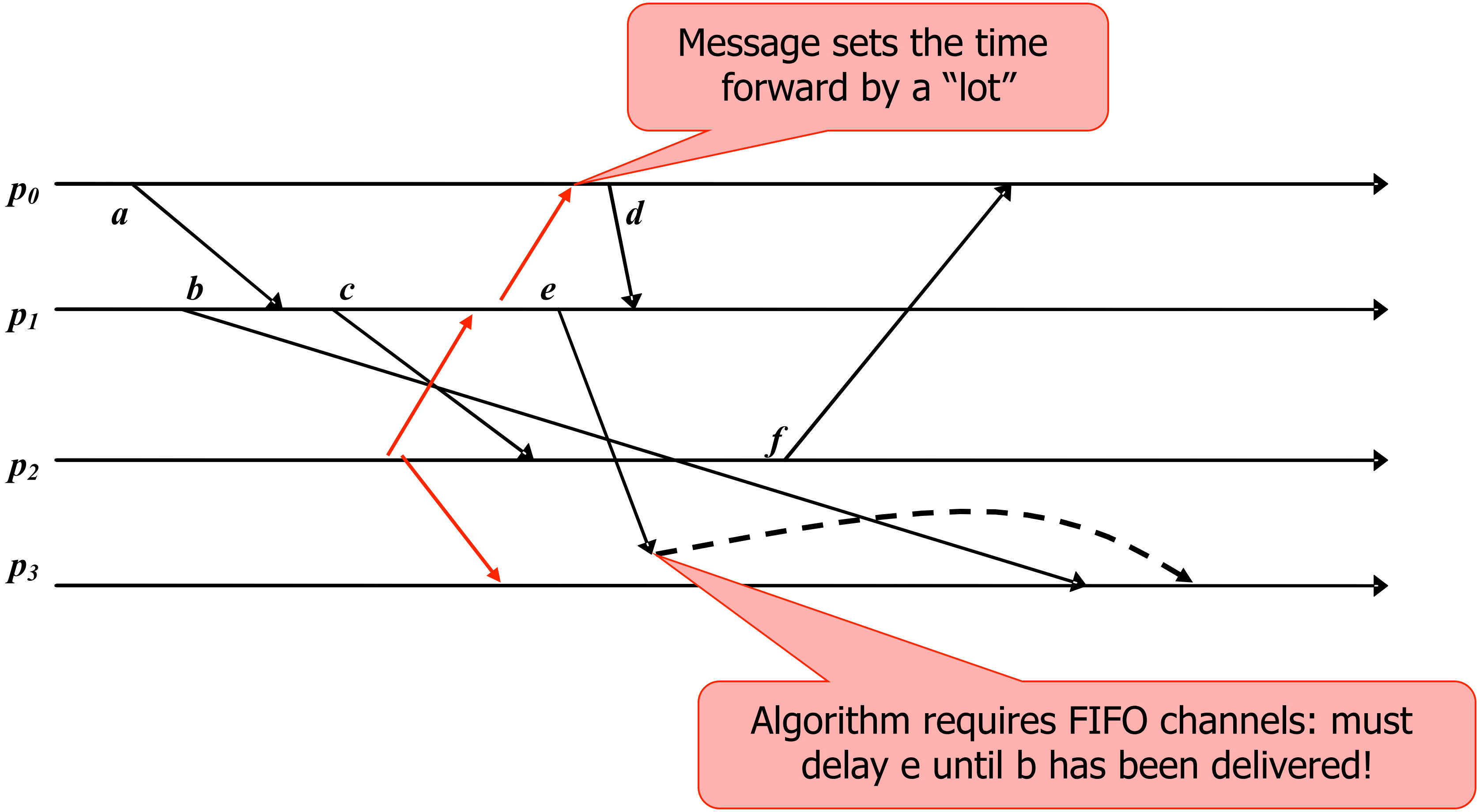
Consistent cuts and snapshots

- 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.
- A "cut" is the frontier of a "snapshot."

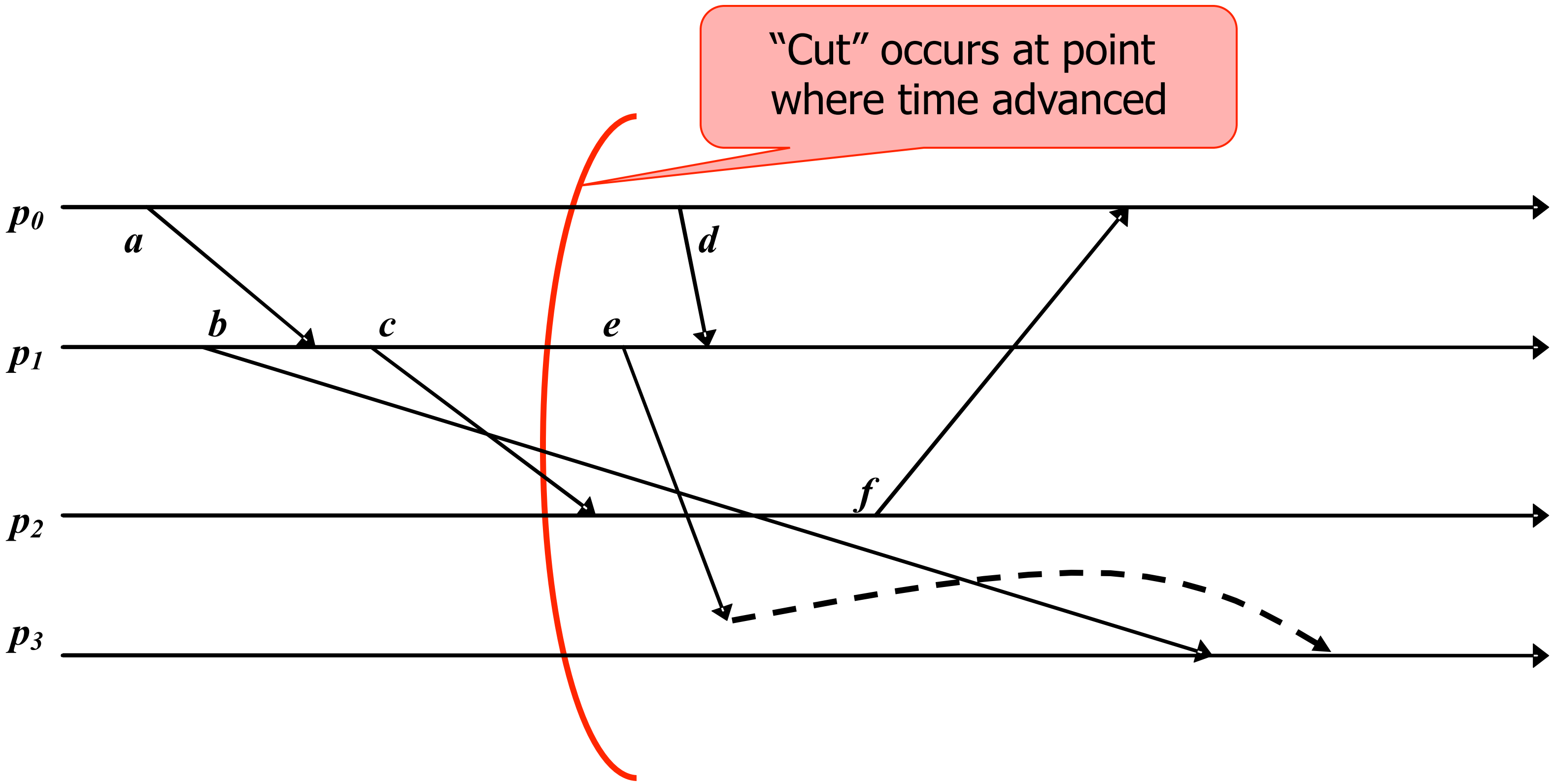
Chandy/Lamport Algorithm

- Assume that if p_i can talk to p_j they do so using a lossless, FIFO connection.
- Now think about logical clocks:
  - Suppose someone sets his clock way ahead and triggers a “flood” of messages.
  - As these reach each process, it advances its own time… eventually all do so.
- The point where time jumps forward is a consistent cut across the system.

Using logical clocks to make cuts



Using logical clocks to make cuts



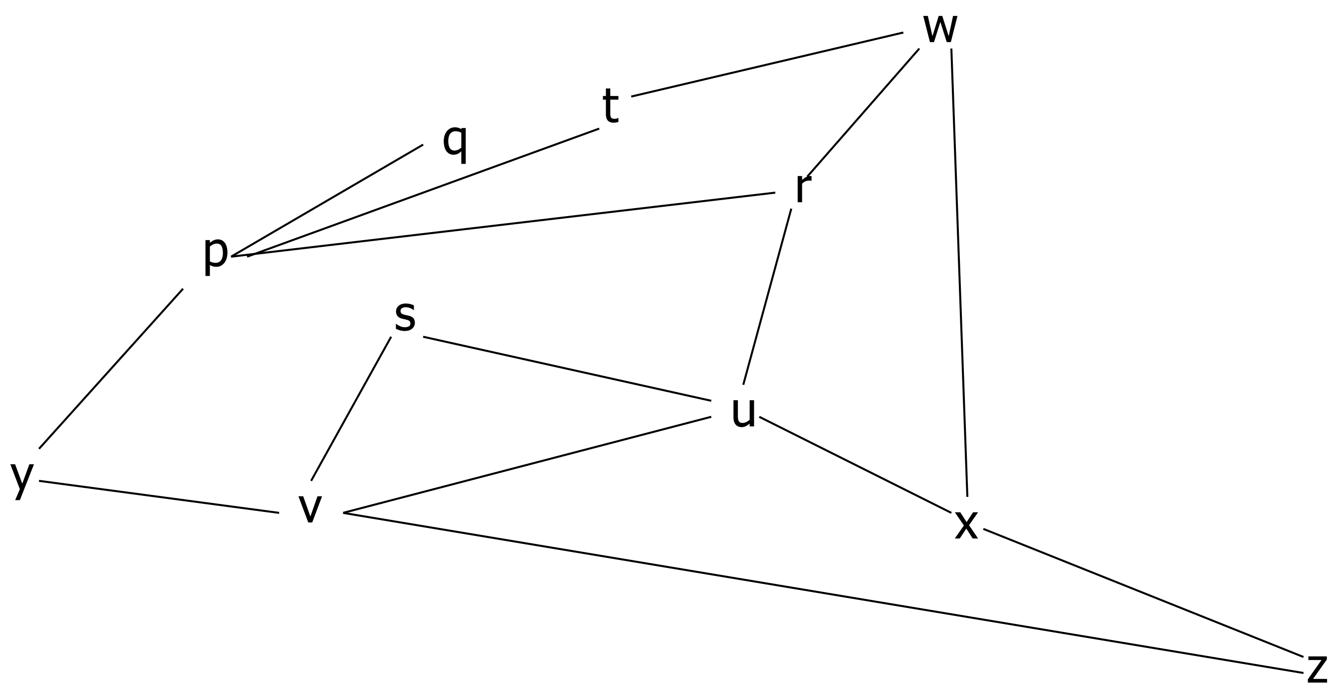
Turn idea into an algorithm

- 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.
- 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

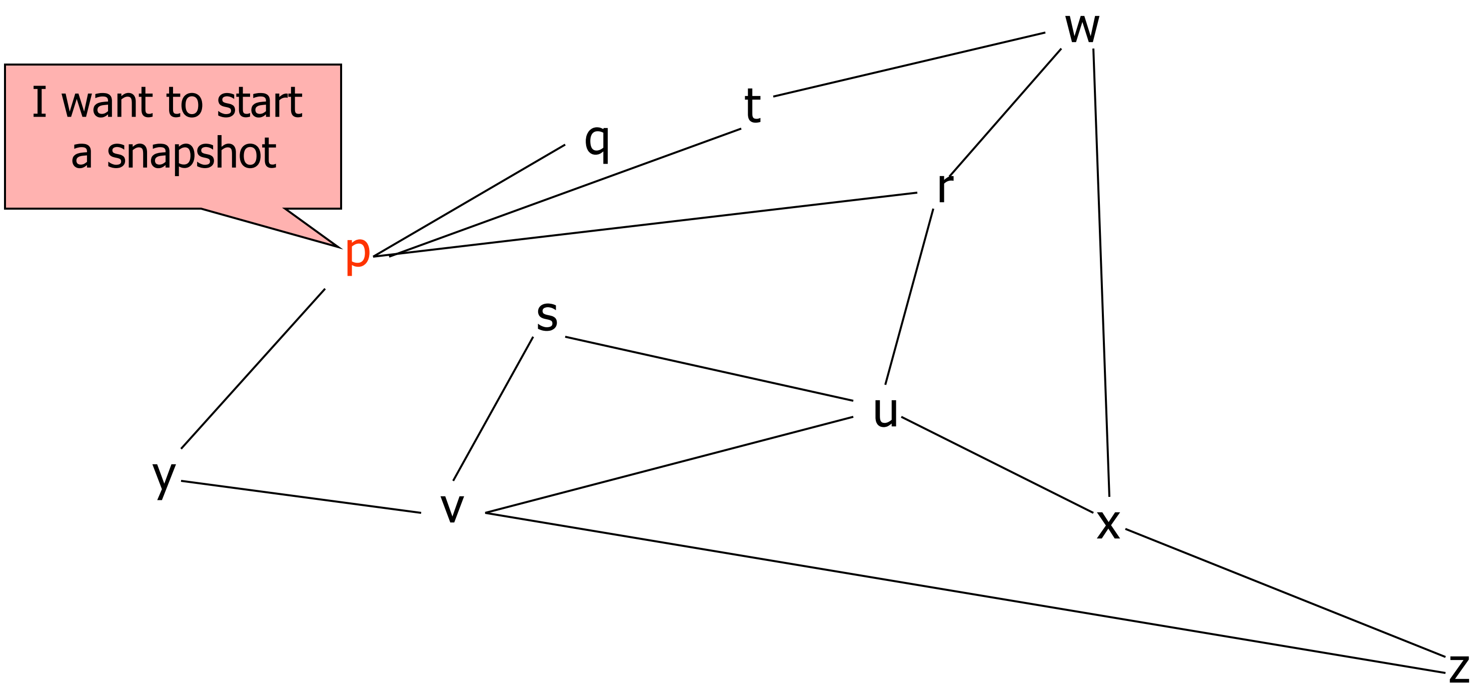
- This algorithm, but implemented with an outgoing flood, followed by an incoming wave of snapshot contributions.
- Snapshot ends up accumulating at the initiator, $p_{i.}$
- Algorithm does not tolerate process failures or message failures.

Chandy/Lamport



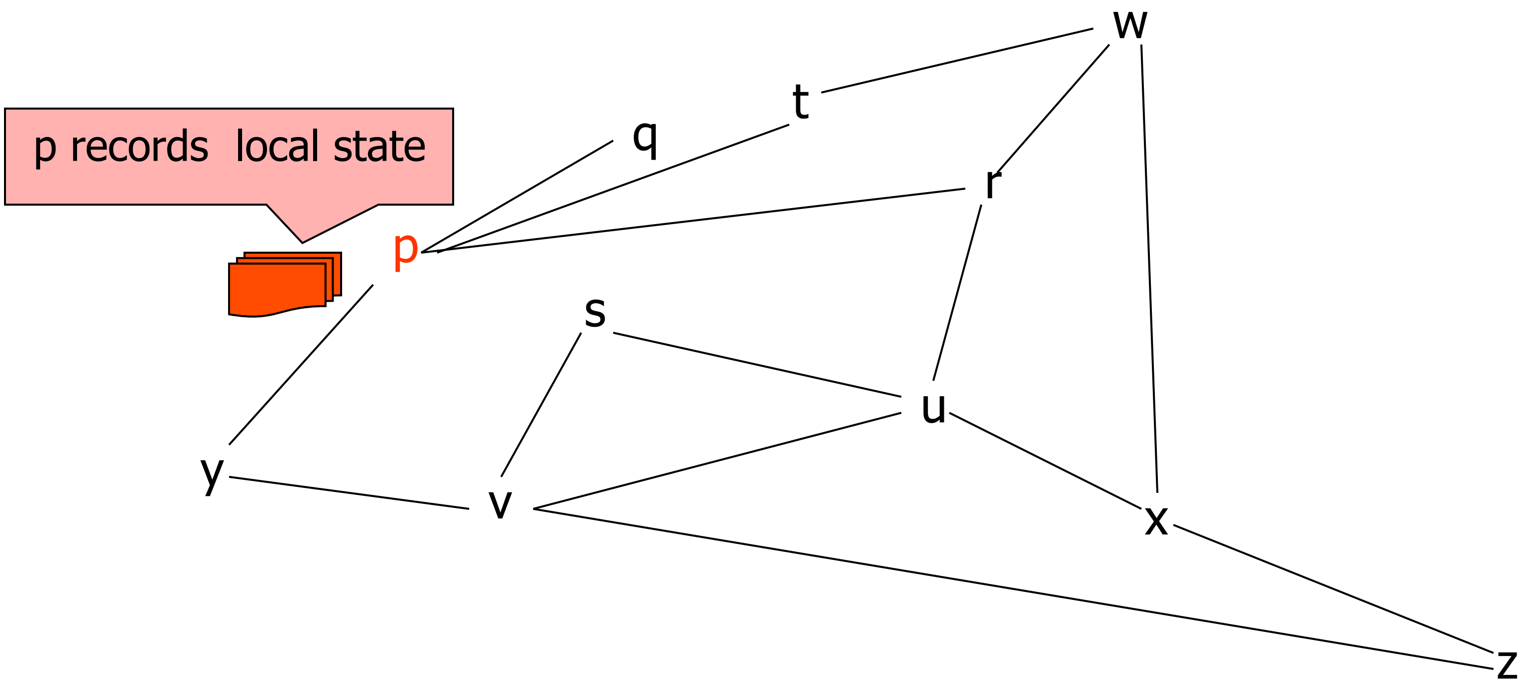
A network

Chandy/Lamport



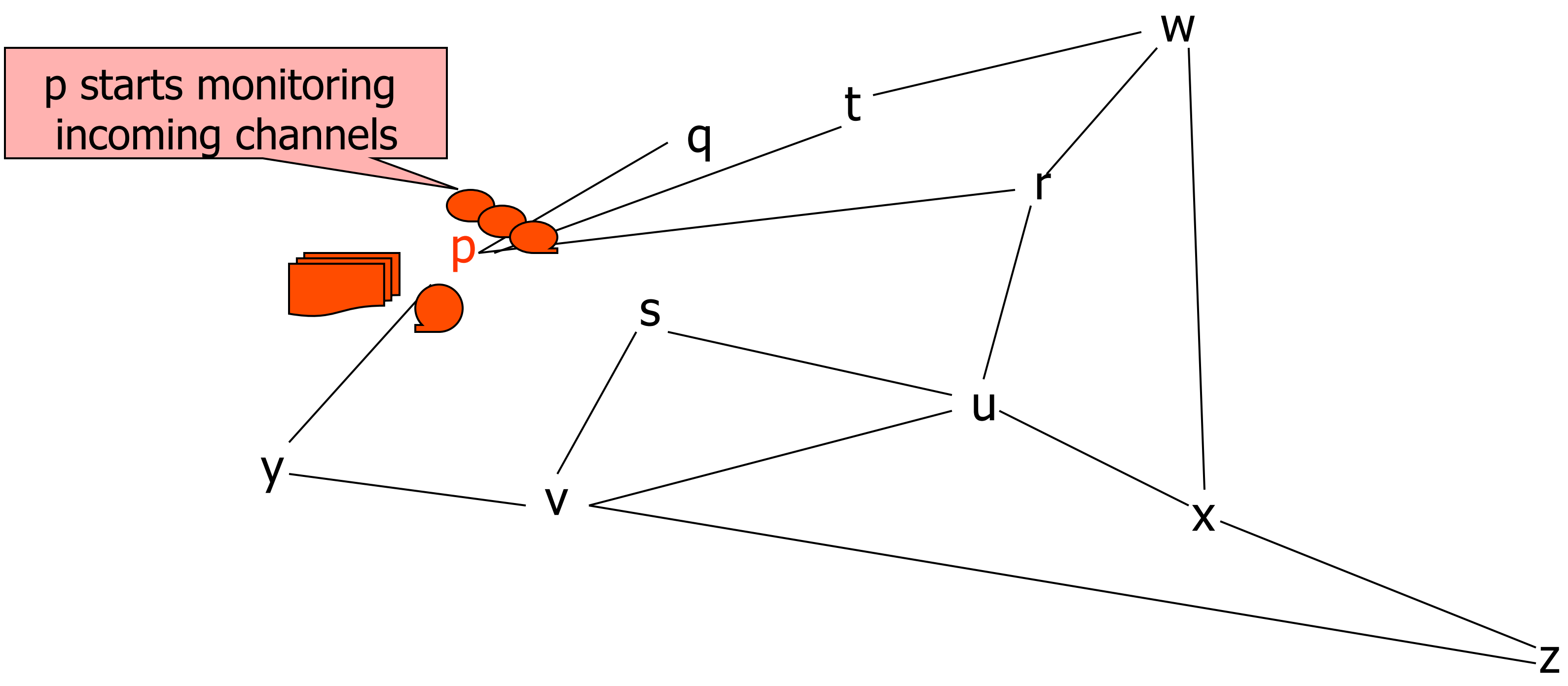
A network

Chandy/Lamport



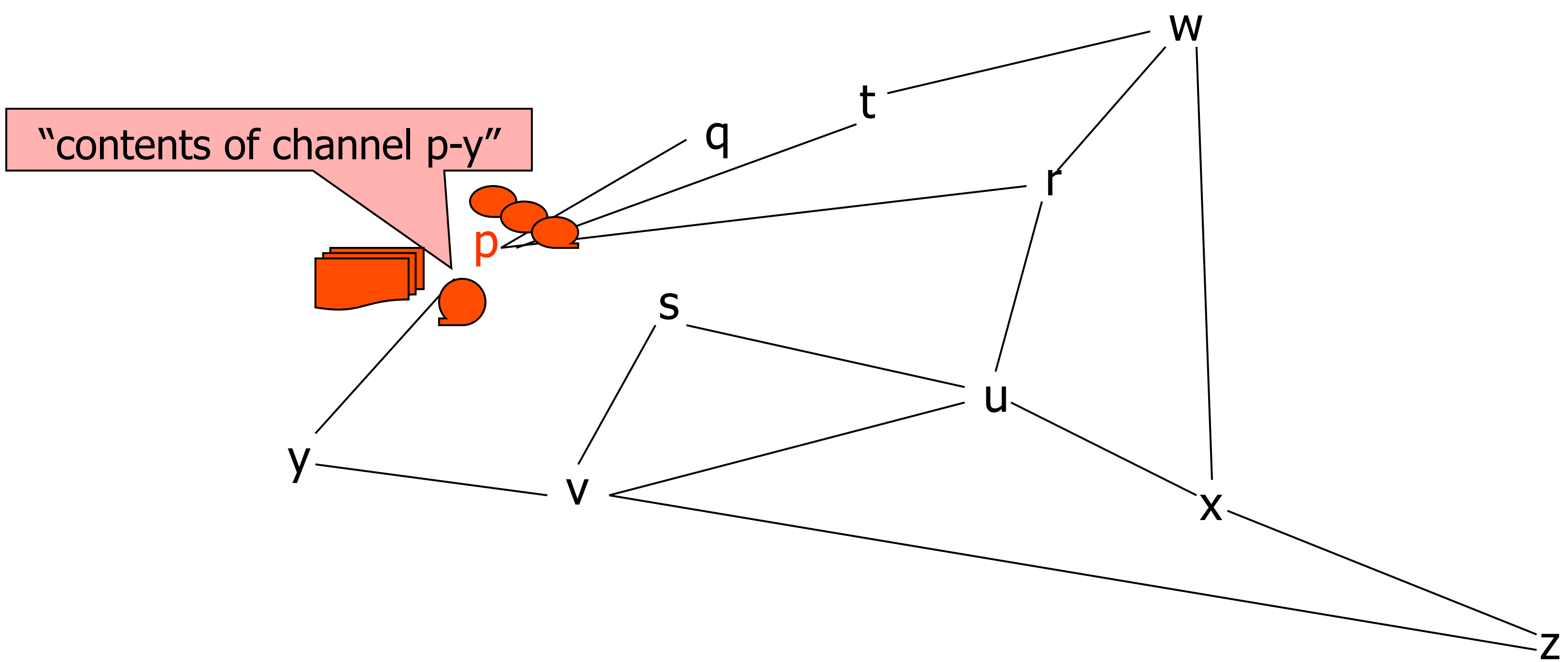
A network

Chandy/Lamport



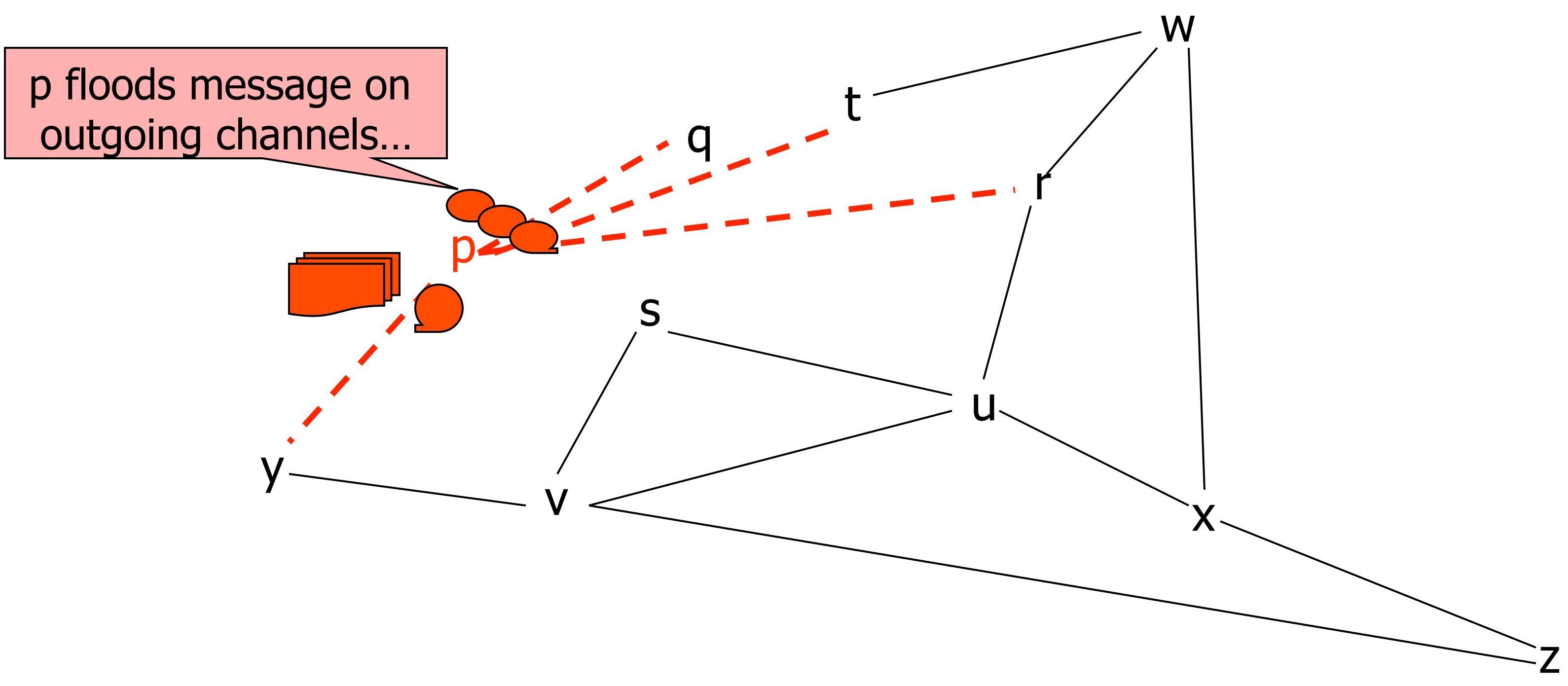
A network

Chandy/Lamport



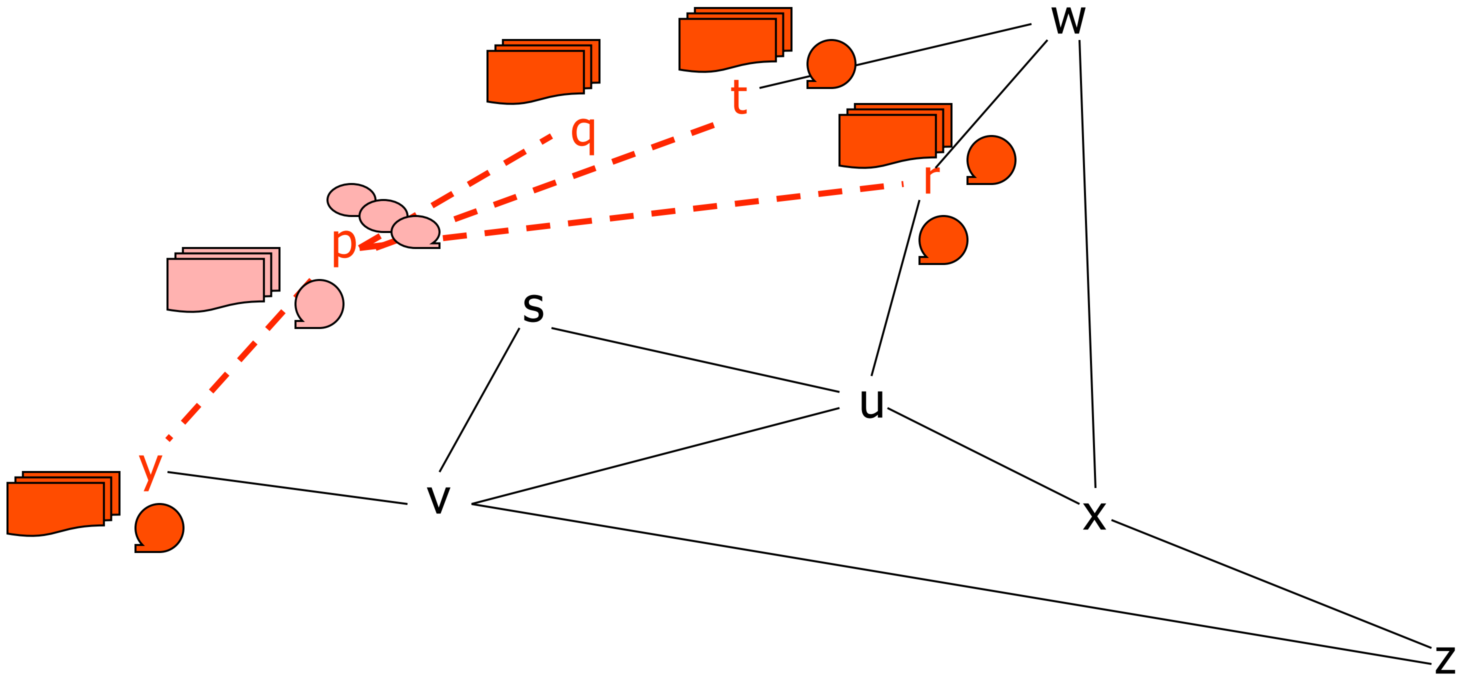
A network

Chandy/Lamport



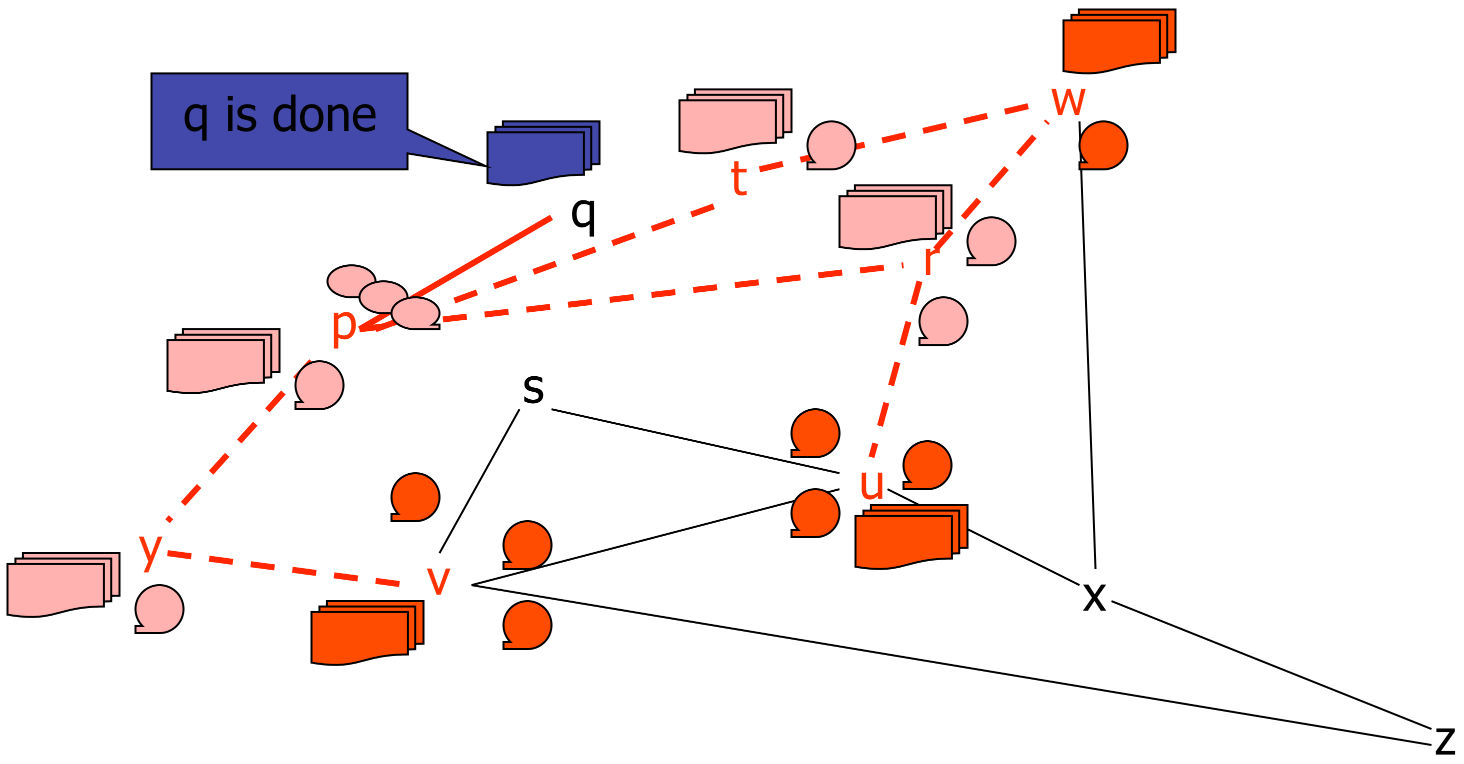
A network

Chandy/Lamport



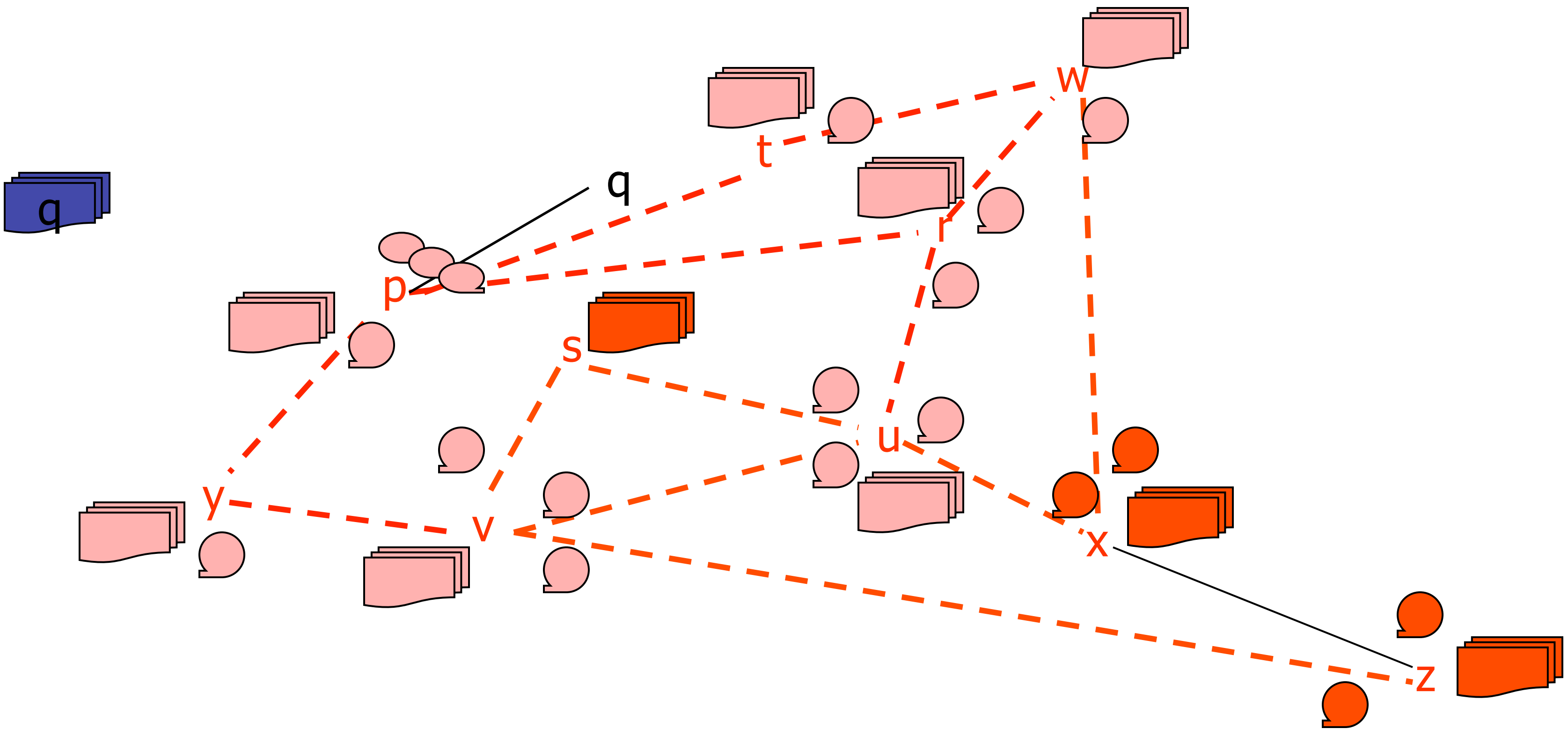
A network

Chandy/Lamport



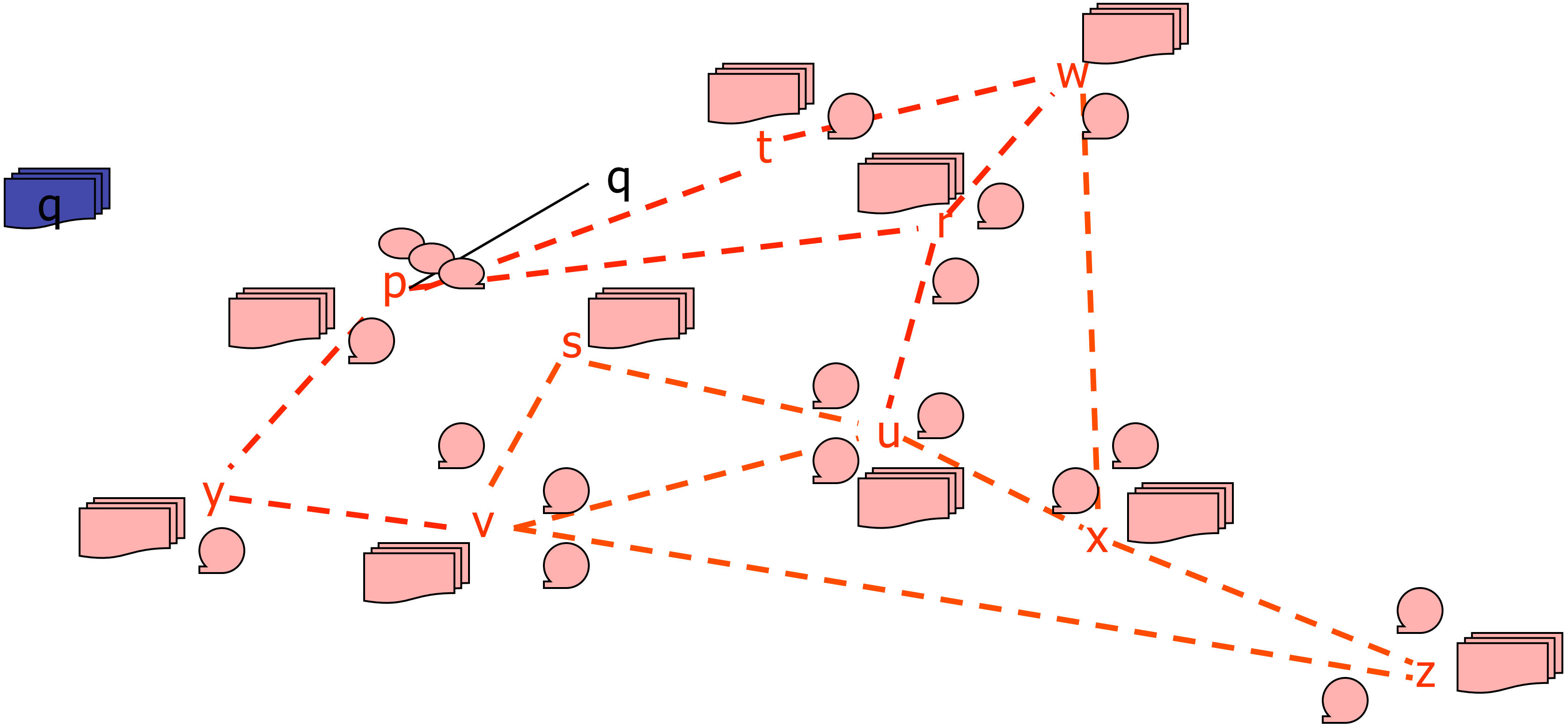
A network

Chandy/Lamport



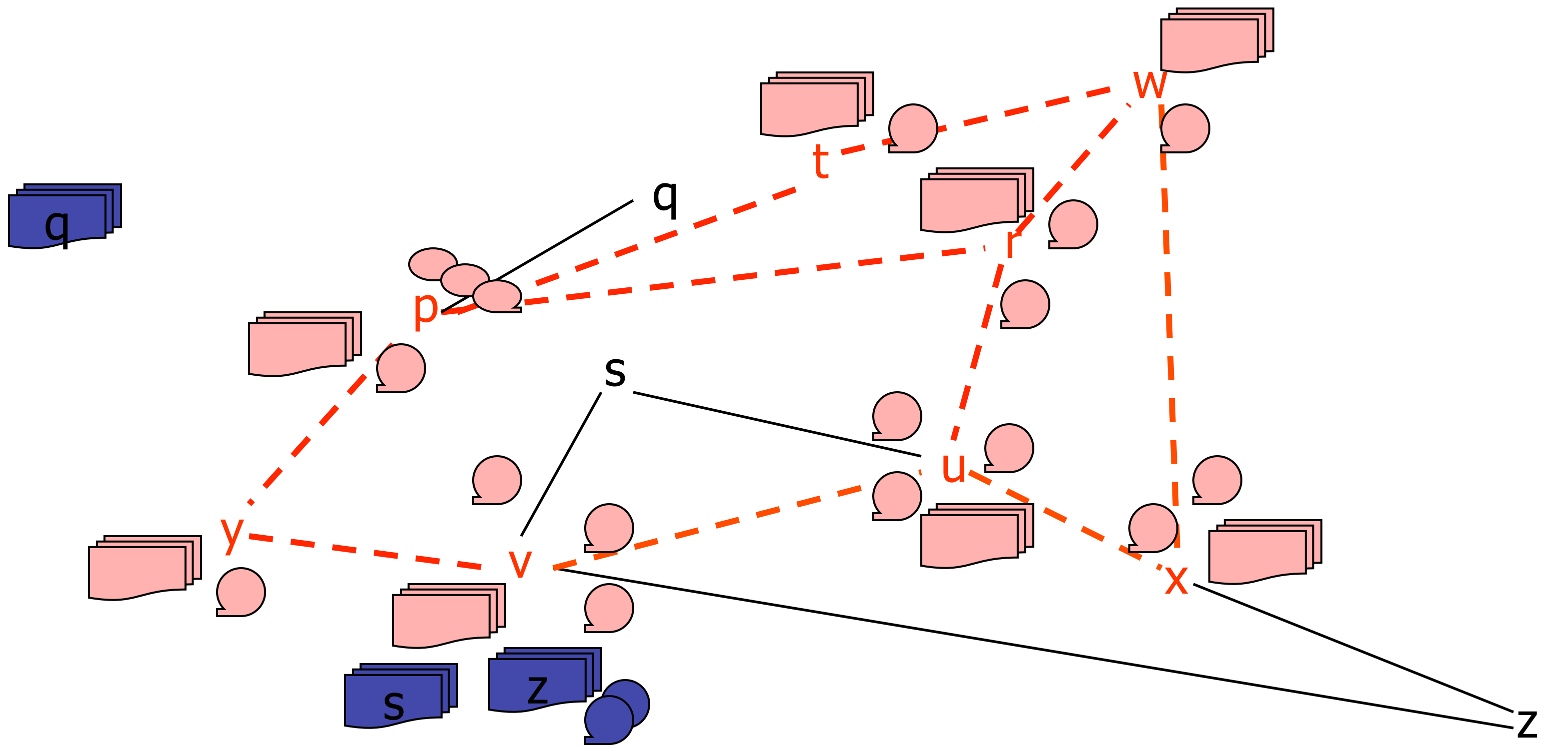
A network

Chandy/Lamport



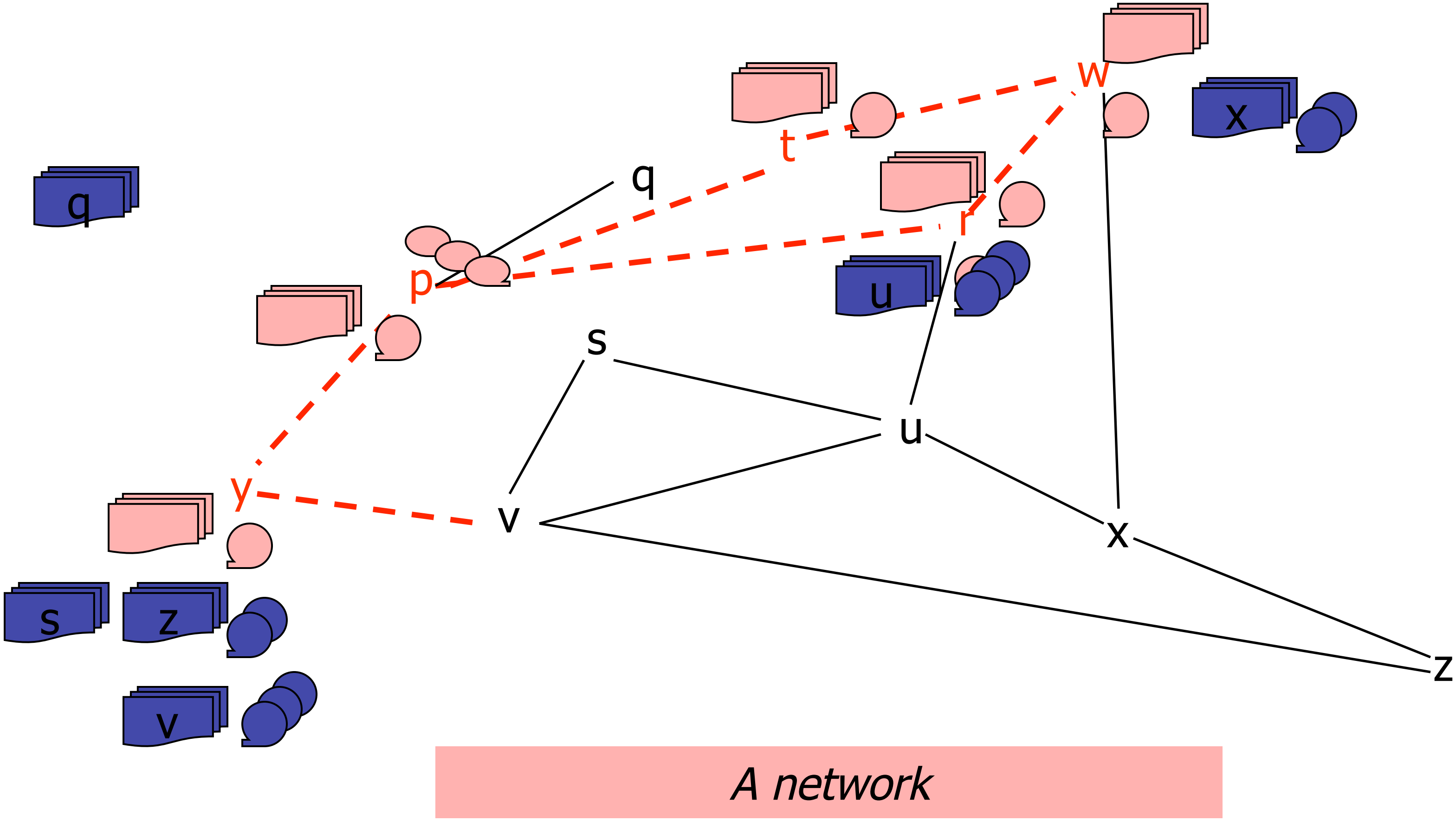
A network

Chandy/Lamport

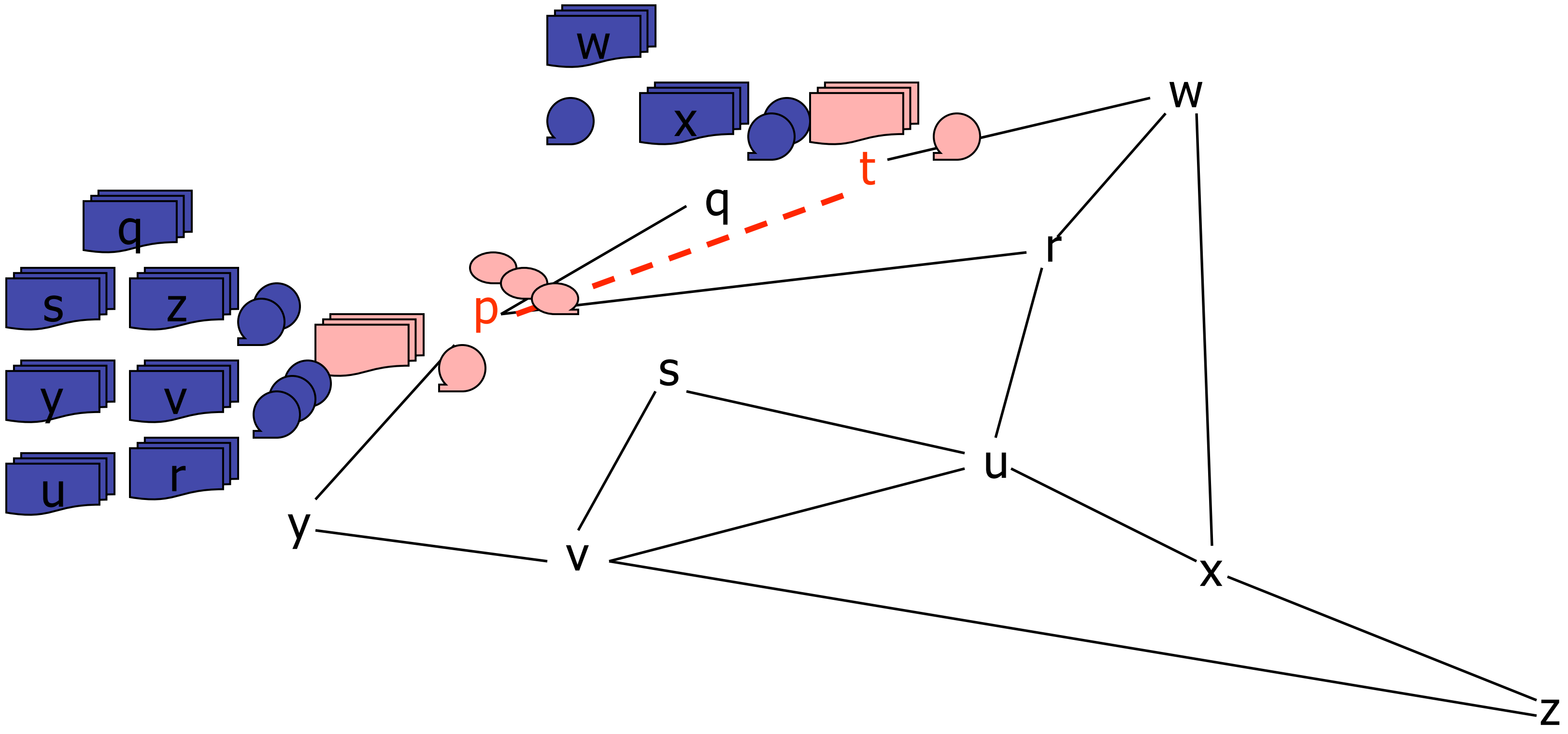


A network

Chandy/Lamport

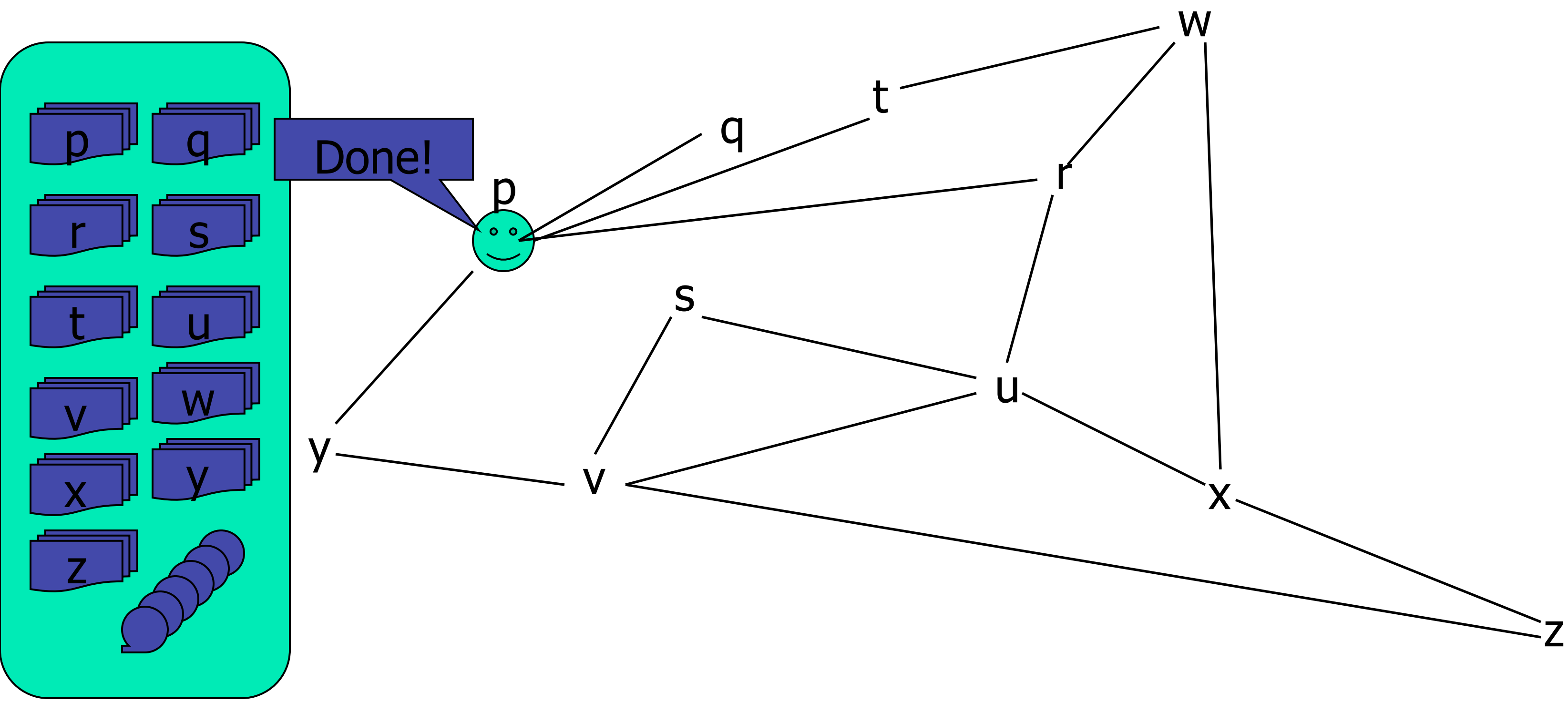


Chandy/Lamport



A network

Chandy/Lamport



A snapshot of a network

What’s in the “state”?

- In practice we only record things important to the application running the algorithm, not the “whole” state,
  - e.g. “locks currently held”, “lock release messages.”
- Idea is that the snapshot will be
  - Easy to analyze, letting us build a picture of the system state,
  - And will have everything that matters for our real purpose, like deadlock detection.

Other algorithms?

- Many algorithms have a consistent cut mechanism hidden within.
  - More broadly, notions of time are *sometimes* explicit in algorithms,
  - But are often used as the insight that motivated the developer.
  - By thinking about time, he or she was able to reason about a protocol.

Who needs failure “models”?

- Role of a failure model
  - Lets us reduce fault-tolerance to a mathematical question.
    - In model *M*, can problem *P* be solved?
    - How costly is it to do so?
    - What are the best solutions?
    - What tradeoffs arise?
  - And clarifies what we are saying.
    - Lacking a model, confusion is common.

Categories of failures

- Crash faults, message loss
  - These are common in real systems.
  - Crash failures: process simply stops, and does nothing wrong that would be externally visible before it stops.
- These faults can not be directly detected.

Categories of failures

- Fail-stop failures.
  - These require system support.
  - Idea is that the process fails by crashing, and the system notifies anyone who was talking to it.
  - With fail-stop failures we can overcome message loss by just resending packets, which must be uniquely numbered.
  - Easy to work with… but rarely supported.

Categories of failures

- Non-malicious Byzantine failures.
  - This is the best way to understand many kinds of corruption and buggy behaviors.
  - A program can do pretty much anything, including sending corrupted messages.
  - But it does not do so with the intention of messing up our protocols.
- Unfortunately, a pretty common mode of failure.

Categories of failure

- Malicious, true Byzantine, failures.
  - Model is of an attacker who has studied the system and wants to break it.
  - She can corrupt or replay messages, intercept them at will, compromise programs and substitute hacked versions.
- This is a worst-case scenario mindset.
  - In practice, doesn't actually happen.
  - Very costly to defend against; typically used in very limited ways (e.g., key mgt. server).

Models of failure

- The question here concerns how failures appear in formal models used when proving things about protocols.
- Lamport's happens-before relationship [$\rightarrow$]
  - Model already has processes, messages, temporal ordering.
  - Assumes messages are reliably delivered.

Recall: Two kinds of models

- We tend to work within two models:
  - Asynchronous model makes no assumptions about time
    - Lamport's model is a good fit.
    - Processes have no clocks, will wait indefinitely for messages, could run arbitrarily fast/slow.
    - Distributed computing at an "eons" timescale.
  - Synchronous model assumes a lock-step execution in which processes share a clock.

Adding failures in Lamport's model

- Also called the asynchronous model.
- Normally we just assume that a failed process "crashes": it stops doing anything.
  - Notice that in this model, a failed process is indistinguishable from a delayed process.
  - In fact, the decision that something has failed takes on an arbitrary flavour.
    - Suppose that at point e in its execution, process p decides to treat q as faulty…

What about the synchronous model?

- Here, we also have processes and messages.
  - But communication is usually assumed to be reliable: any message sent at time t is delivered by time $t+\delta$.
  - Algorithms are often structured into rounds, each lasting some fixed amount of time Δ, giving time for each process to communicate with every other process.
  - In this model, a crash failure is easily detected.
- When people have considered malicious failures, they often used this model.

Neither model is realistic

- Value of the asynchronous model is that it is so stripped down and simple.
  - If we can do something "well" in this model we can do at least as well in the real world.
  - So we will want "best" solutions.
- Value of the synchronous model is that it adds a lot of "unrealistic" mechanism.
  - If we can not solve a problem with all this help, we probably can not solve it in a more realistic setting!
  - So seek impossibility results.

Examples of results

- We saw an algorithm for taking a global snapshot in an asynchronous system.
- And it is common to look at problems like agreeing on an ordering.
  - Often reduced to “agreeing on a bit” (0/1).
  - To make this non-trivial, we assume that processes have an input and must pick some legitimate input value.

Connection to consistency

- We started by talking about consistency.
  - We found that many (not all) notions of consistency reduce to forms of agreement on the events that occurred and their order.
  - Could imagine that our "bit" represents
    - Whether or not a particular event took place;
    - Whether event A is the "next" event.
  - Thus fault-tolerant consensus is deeply related to fault-tolerant consistency.

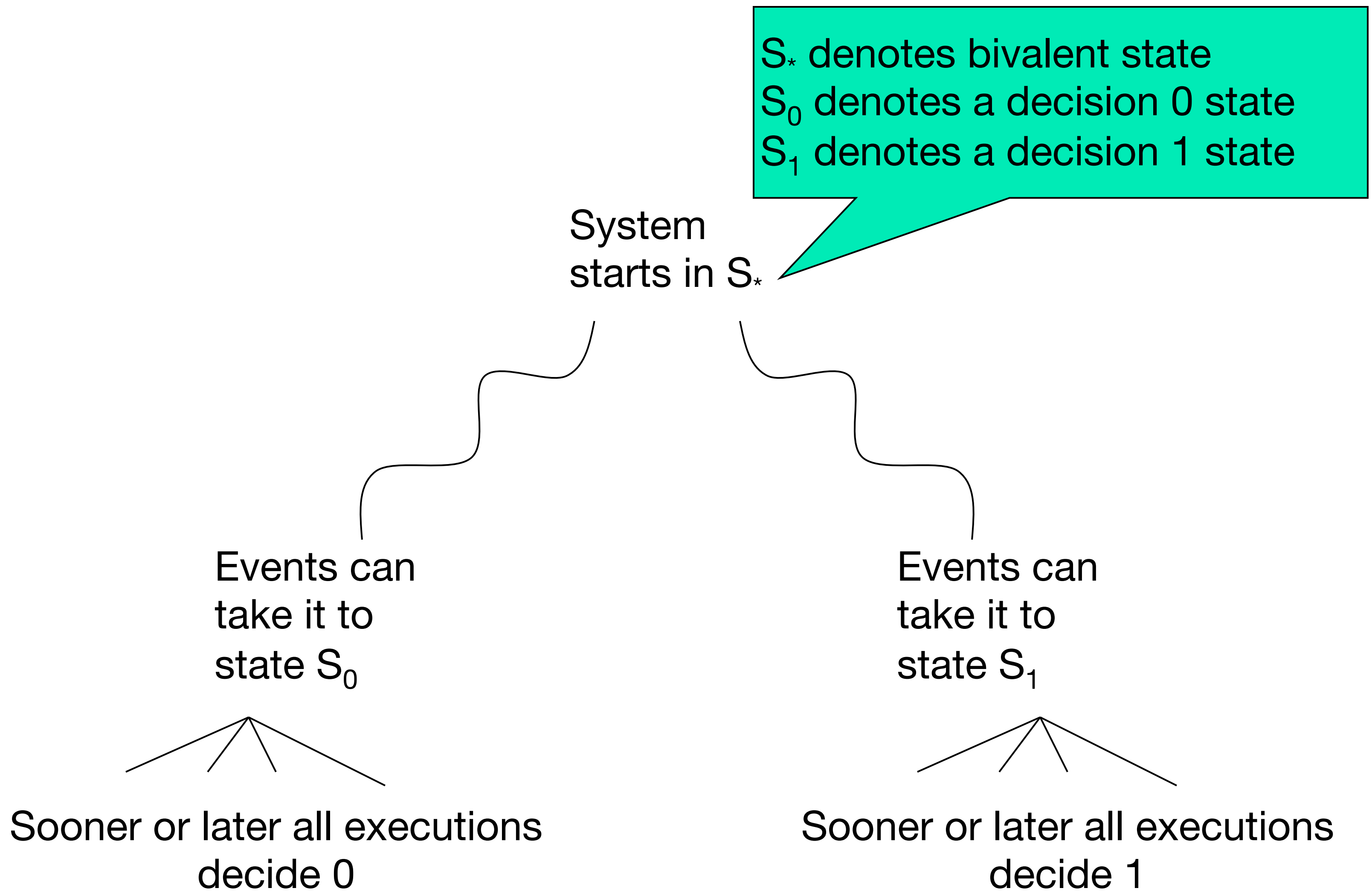
Fischer, Lynch and Patterson

- A surprising result:
  - **Impossibility of asynchronous distributed consensus with a single faulty process.**
- They prove that no asynchronous algorithm for agreeing on a one-bit value can guarantee that it will terminate in the presence of crash faults.
  - And this is true even if no crash actually occurs!
  - Proof constructs infinite non-terminating runs.

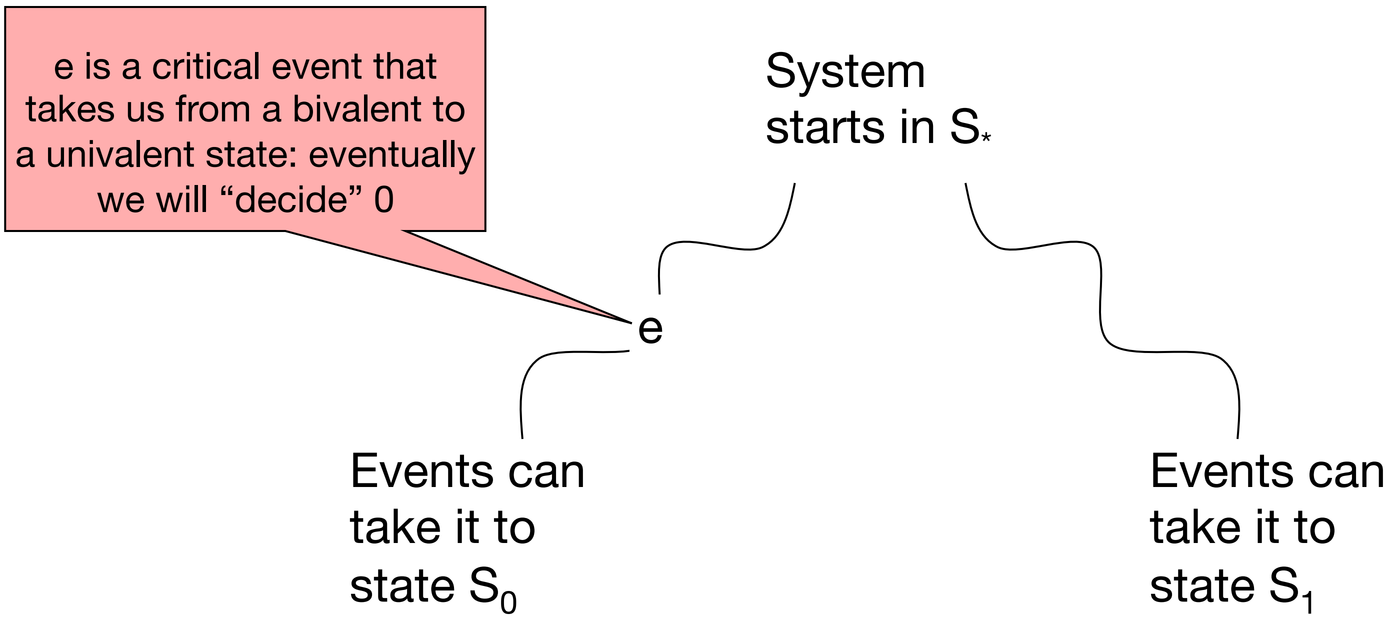
Core of the FLP result

- They start by looking at a system with inputs that are all the same.
  - All 0s must decide 0, all 1s decide 1.
- Now they explore mixtures of inputs and find some initial set of inputs with an uncertain (“bivalent”) outcome.
- They focus on this bivalent state.

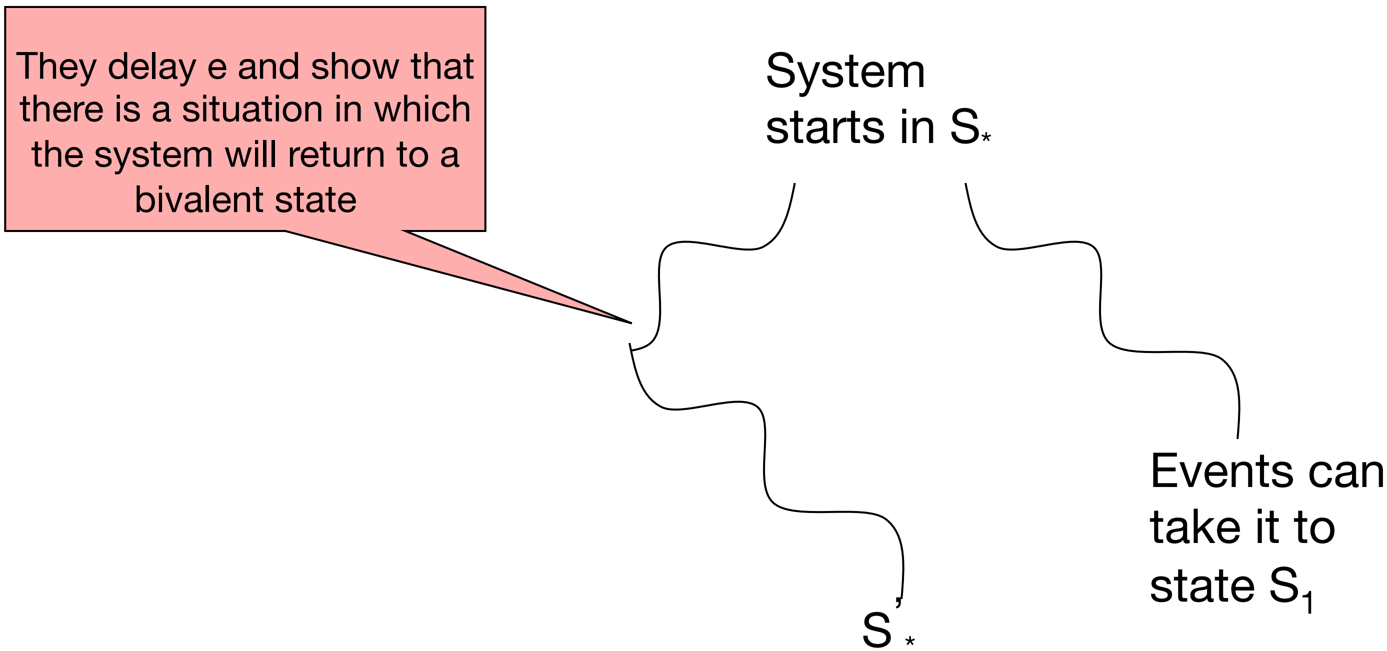
Bivalent state



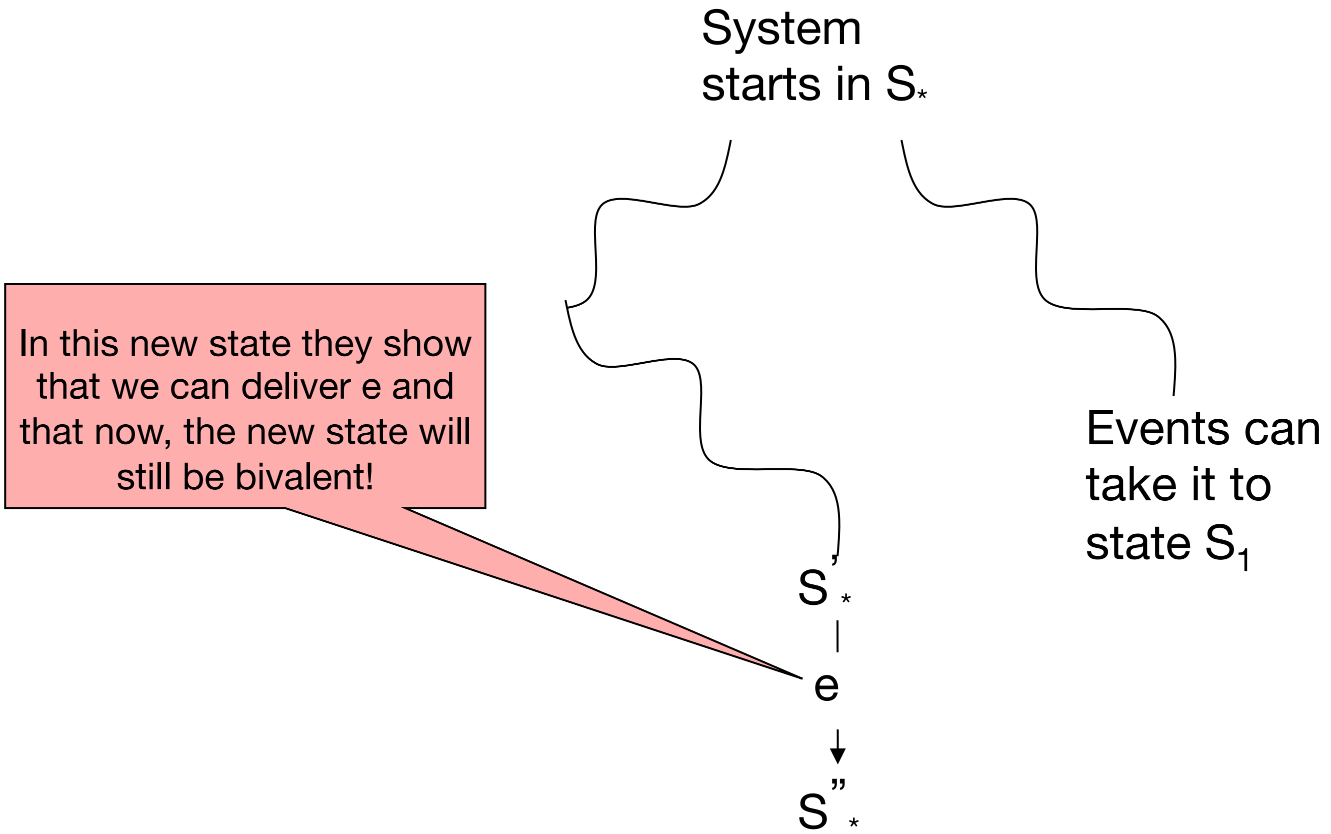
Bivalent state



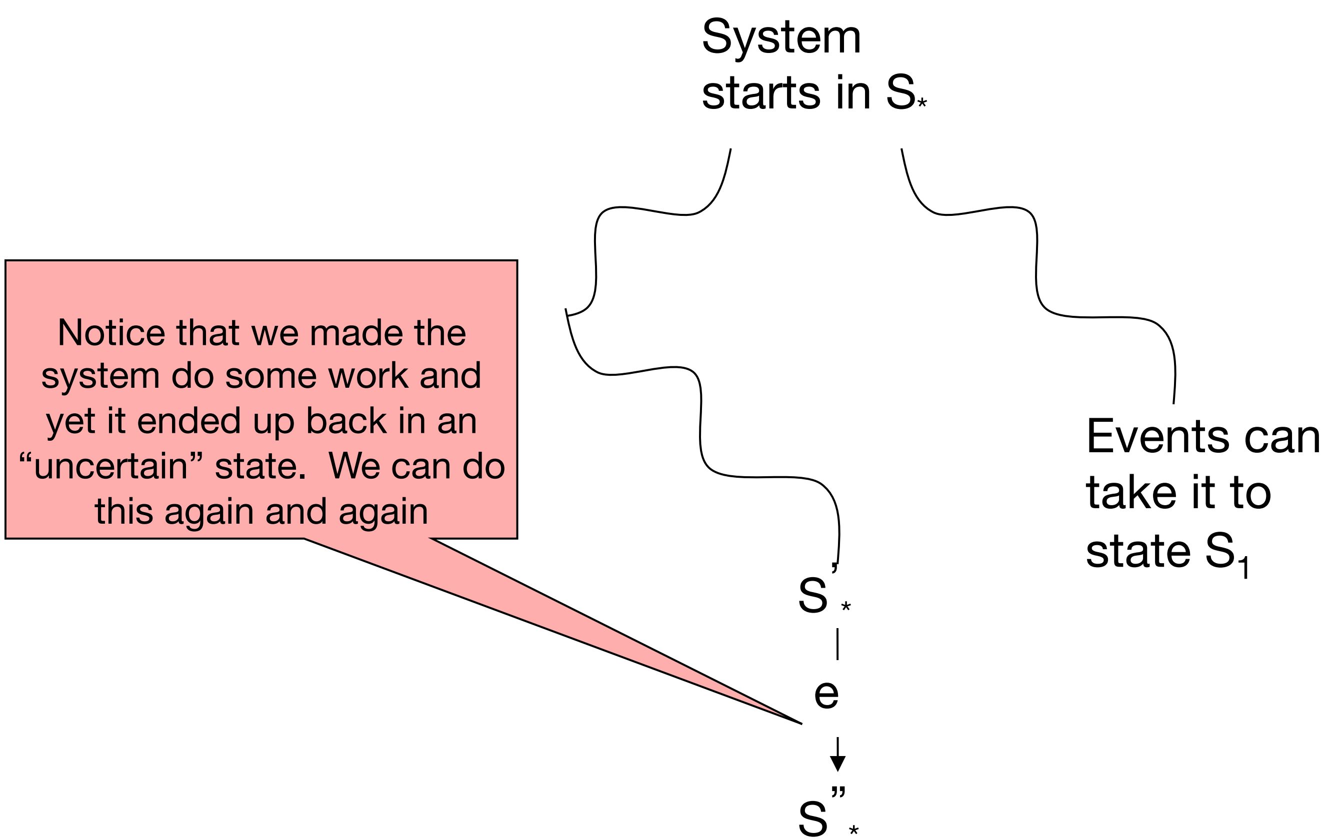
Bivalent state



Bivalent state



Bivalent state



Core of the FLP result in words

- In an initially bivalent state, they look at some execution that would lead to a decision state, say “0”.
  - At some step this run switches from bivalent to univalent, when some process receives some message *m*.
  - They now explore executions in which *m* is delayed.

Core of FLP result

- So:
  - Initially in a bivalent state.
  - Delivery of m would make us univalent but we delay m.
  - They show that if the protocol is fault-tolerant there must be a run that leads to the <u>other</u> univalent state.
  - And they show that you can deliver m in this run without a decision being made.
- This proves the result: they show that a bivalent system can be forced to do some work and yet remain in a bivalent state.
  - If this is true once, it is true as often as we like.
  - In effect: we can delay decisions indefinitely.

But how did they “really” do it?

- Our picture just gives the basic idea.
- Their proof actually proves that there is a way to force the execution to follow this tortured path.
- But the result is very theoretical.
  - For details: Michael J. Fischer, Nancy A. Lynch and Michael S. Paterson, *Impossibility of Distributed Consensus with One Faulty Process*, **Journal of the ACM**, April 1985, 32(2):374-382.
- So we will skip the real details.

Intuition behind this result?

- Think of a real system trying to agree on something in which process p plays a key role.
- But the system is fault-tolerant: if p crashes it adapts and moves on.
- Their proof “tricks” the system into treating p as if it had failed, but then lets p resume execution and “rejoin”.
- This takes time… and no real progress occurs.

But what did “impossibility” mean?

- In formal proofs, an algorithm is totally correct if
  - It computes the right thing,
  - And it always terminates.
- When we say something is possible, we mean “there is a totally correct algorithm” solving the problem.
- FLP proves that any fault-tolerant algorithm solving consensus has runs that never terminate.
  - These runs are extremely unlikely (“probability zero”).
  - Yet they imply that we can not find a totally correct solution.
  - And so “consensus is impossible” (“not always possible”).

Recap

- We have an asynchronous model with crash failures.
  - A bit like the real world!
- In this model we know how to do some things.
  - Tracking “happens before” & making a consistent snapshot.
- But now we also know that there will always be scenarios in which our solutions can not make progress.
  - Often can engineer system to make them extremely unlikely.
  - Impossibility does not mean these solutions are wrong – only that they live within this limit.

Tougher failure models

- We have focused on crash failures.
  - In the synchronous model these look like a “farewell cruel world” message.
  - Some call it the “failstop model”. A faulty process is viewed as first saying goodbye, then crashing
- What about tougher kinds of failures?
  - Corrupted messages;
  - Processes that do not follow the algorithm, and
  - Malicious processes out to cause havoc?

Here the situation is much harder

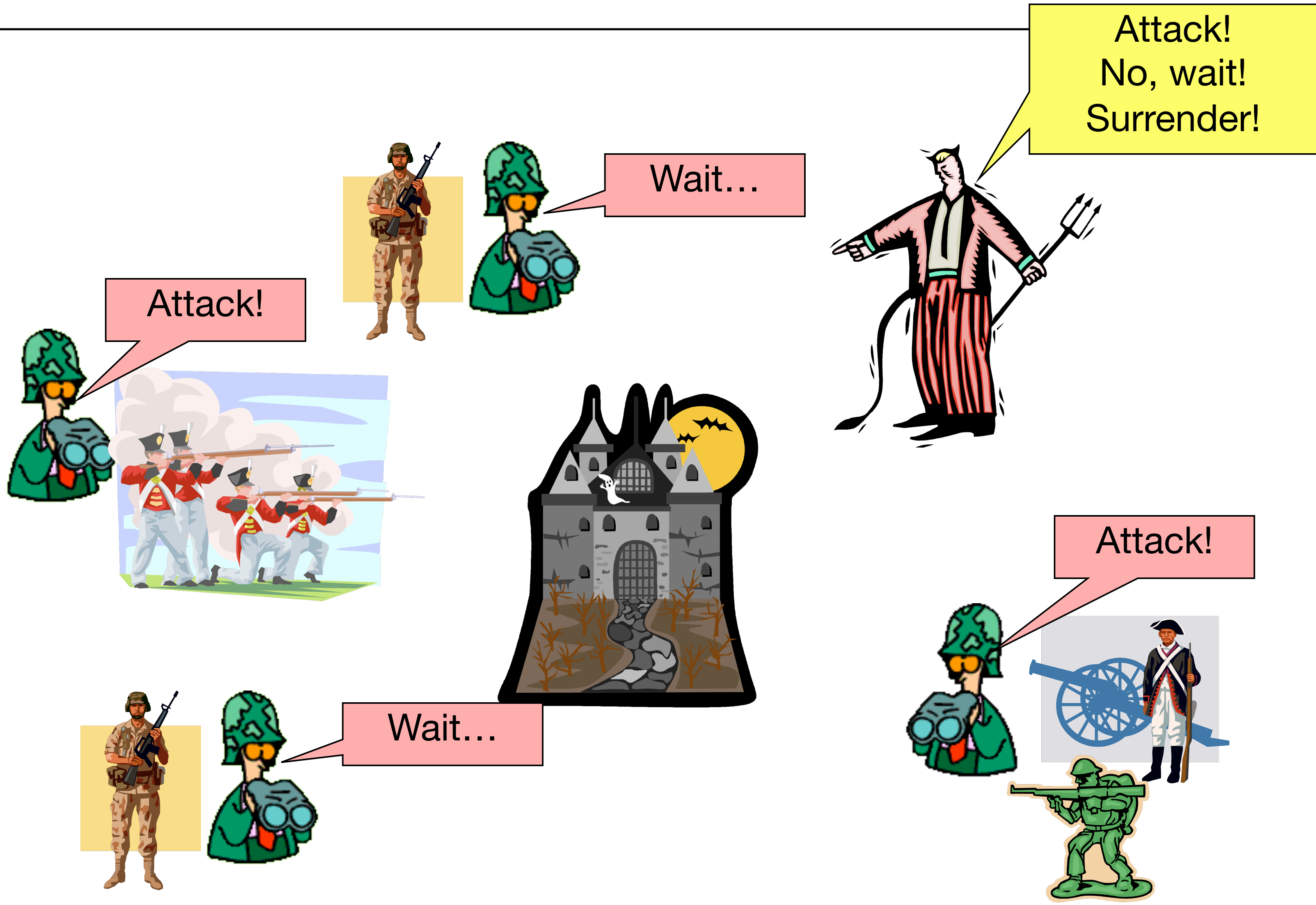
- Generally we need at least $3f+1$ processes in a system to tolerate f Byzantine failures.
  - For example, to tolerate 1 failure we need 4 or more processes.
- We also need $f+1$ “rounds”.
- Let’s see why this happens.

Reaching Agreement in the Presence of Faults.
Leslie Lamport, Marshall Pease and Robert Shostak.
Journal of the Association for Computing Machinery 27, 2 (April 1980).

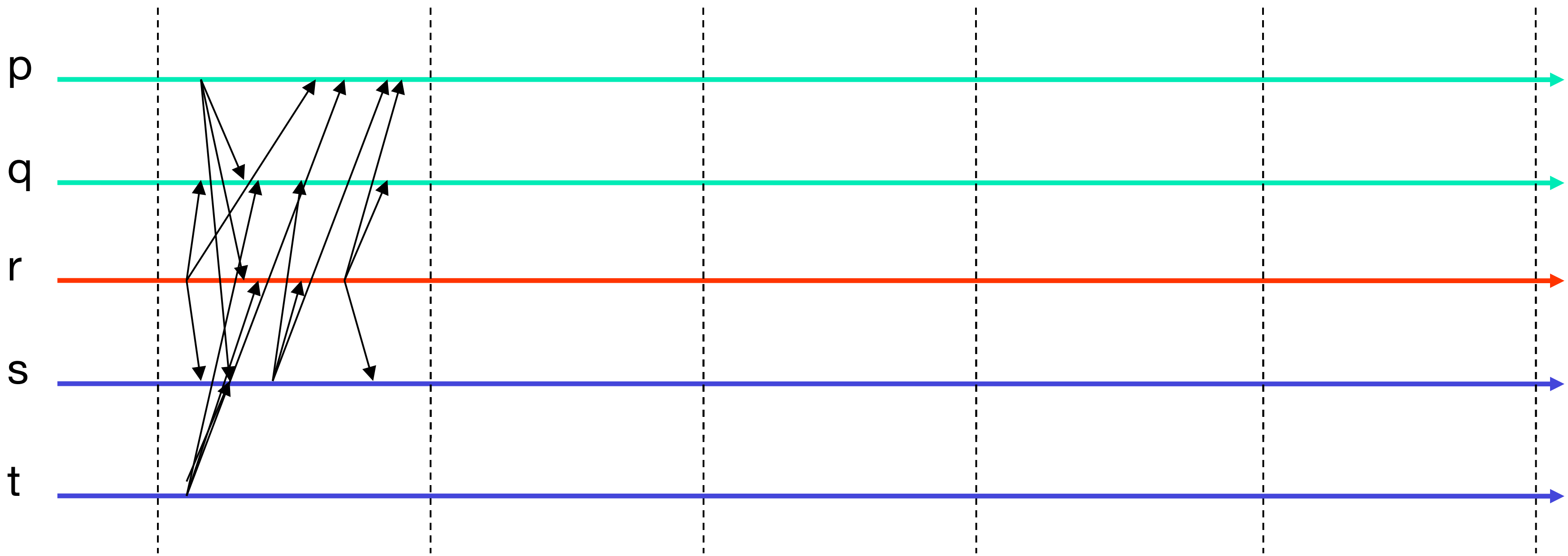
Byzantine scenario

- Generals (*n* of them) surround a city
  - They communicate by courier
- Each has an opinion: “attack” or “wait”
  - In fact, an attack would succeed: the city will fall.
  - Waiting will succeed too: the city will surrender.
  - But if some attack and some wait, disaster ensues
- Some Generals (*f* of them) are traitors… it doesn’t matter if they attack or wait, but we must prevent them from disrupting the battle
  - Traitor can not forge messages from other Generals

Byzantine scenario

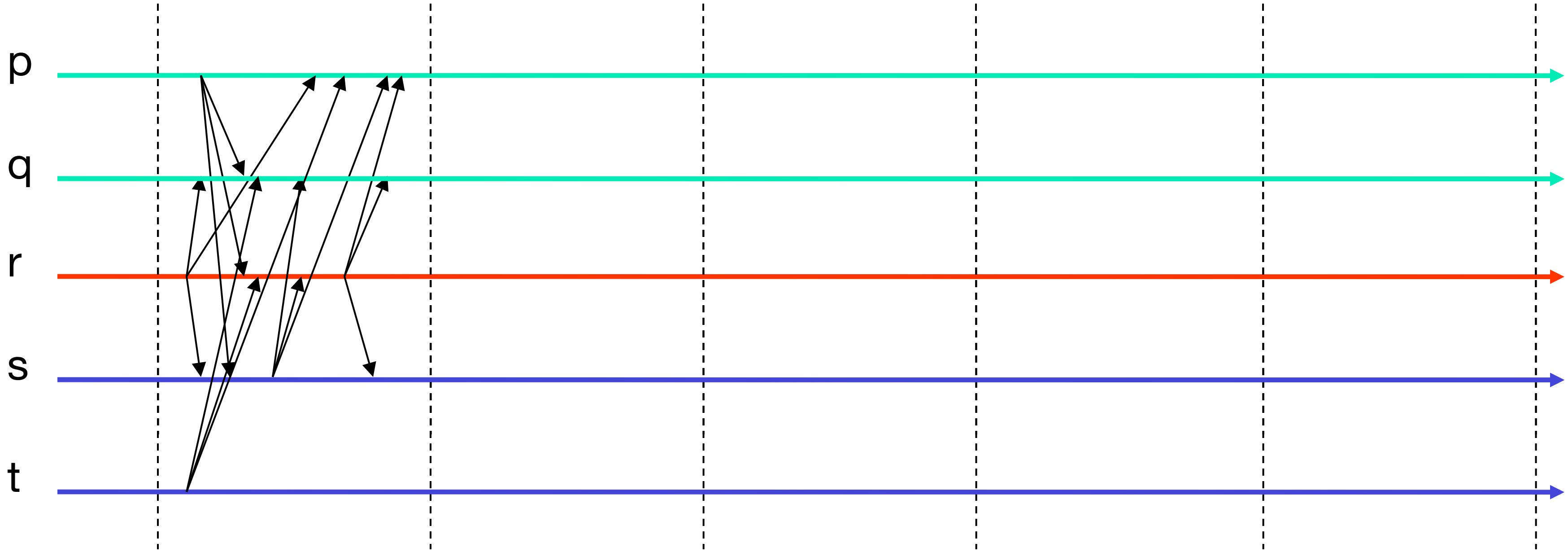


A timeline perspective



- Suppose that p and q favor attack, r is a traitor and s and t favor waiting… assume that in a tie vote, we attack.

A timeline perspective



- After first round collected votes are:
    - {attack, attack, wait, wait, traitor’s-vote}

What can the traitor do?

- Add a legitimate vote of “attack.”
  - Anyone with 3 votes to attack knows the outcome.
- Add a legitimate vote of “wait.”
  - Vote now favors “wait.”
- Or send different votes to different folks.
- Or do not send a vote, at all, to some.

Outcomes?

- Traitor simply votes:
  - Either all see {a,a,a,w,w}.
  - Or all see {a,a,w,w,w}.
- Traitor double-votes.
  - Some see {a,a,a,w,w} and some {a,a,w,w,w}.
- Traitor withholds some vote(s).
  - Some see {a,a,w,w}, perhaps others see {a,a,a,w,w,} and still others see {a,a,w,w,w}.
- Notice that traitor can not manipulate votes of loyal Generals!

What can we do?

- Clearly we can not decide yet; some loyal Generals might have contradictory data.
  - In fact if anyone has 3 votes to attack, they can already "decide".
  - Similarly, anyone with just 4 votes can decide.
  - But with 3 votes to "wait" a General is not sure (one could be a traitor…)
- So: in round 2, each sends out "witness" messages: here is what I saw in round 1:
  - General Smith sent me: "$\text{attack}_{\text{(signed) Smith}}$"

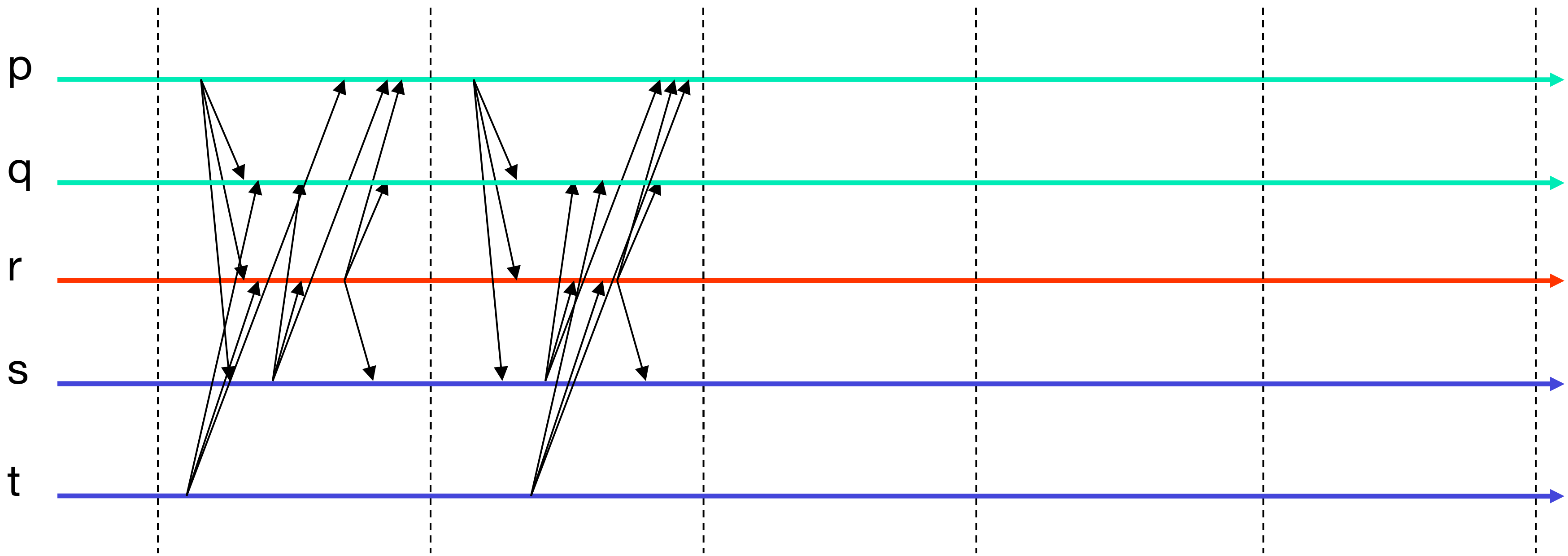
Digital signatures

- These require a cryptographic system.
  - For example, RSA.
  - Each player has a secret (private) key K^{-1} and a public key K.
    - She can publish her public key.
  - RSA gives us a single “encrypt” function:
    - $Encrypt(Encrypt(M,K),K^{-1}) = Encrypt(Encrypt(M,K^{-1}),K) = M$.
    - Encrypt a hash of the message to “sign” it.

With such a system

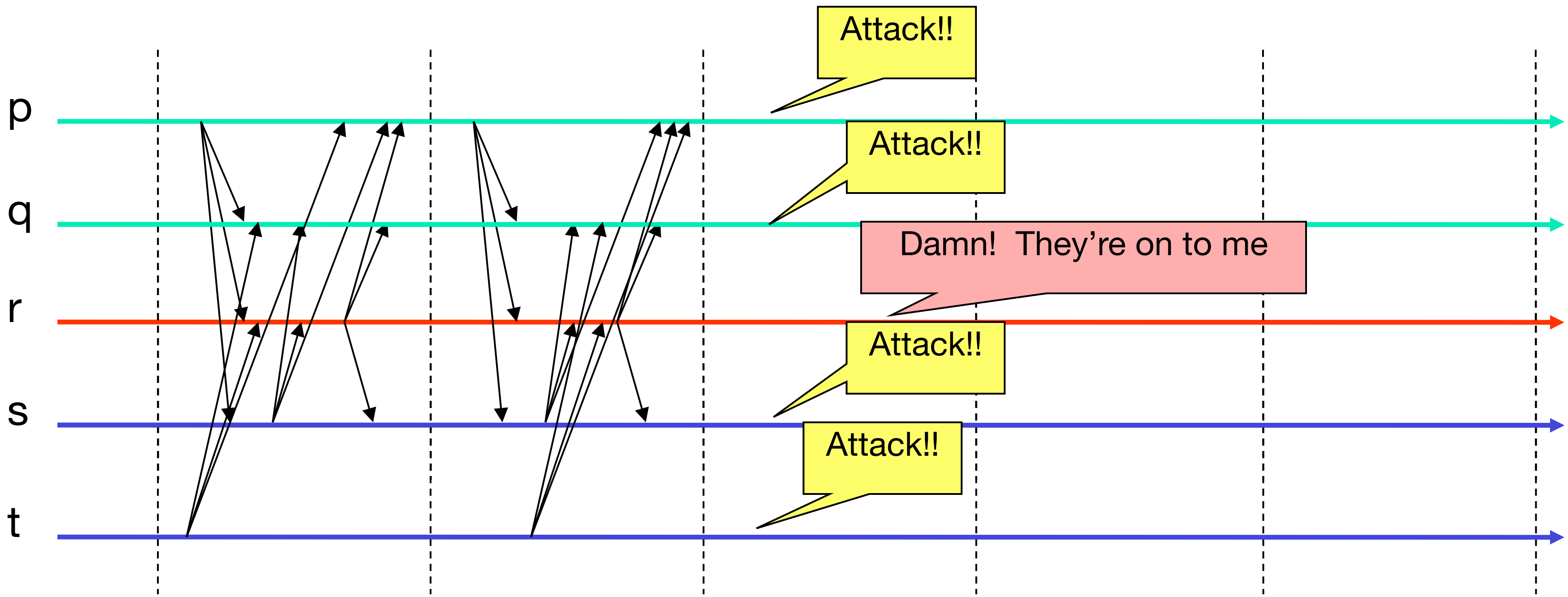
- A can send a message to B that only A could have sent,
  - A just encrypts the body with her private key.
- … or one that only B can read,
  - A encrypts it with B’s public key.
- Or can sign it as proof she sent it.
  - B can recompute the signature and decrypt A’s hashed signature to see if they match.
- These capabilities limit what our traitor can do: he can not forge or modify a message.

A timeline perspective



- In second round if the traitor did not behave identically with all Generals, we can weed out his faulty votes.

A timeline perspective



- We attack!

Traitor is stymied

- Our loyal generals can deduce that the decision was to attack.
- Traitor can not disrupt this…
  - Either forced to vote legitimately, or is caught.
  - But costs were steep!
    - $(f+1)*n^2$,messages!
    - Rounds can also be slow….
  - "Early stopping" protocols: min(t+2, f+1) rounds; t is true number of faults.

Recent work with the Byzantine model

- Focus is typically on using it to secure particularly sensitive, ultra-critical services.
  - For example the “certification authority” that hands out keys in a domain,
  - Or a database maintaining top-secret data.
- Researchers have suggested that for such purposes, a “Byzantine Quorum” approach can work well.
- They are implementing this in real systems by simulating rounds using various tricks.

Byzantine Quorums

- Arrange servers into a $\sqrt{n}$ x $\sqrt{n}$ array.
  - Idea is that any row or column is a quorum.
  - Then use Byzantine Agreement to access that quorum, doing a read or a write.
- Separately, Castro and Liskov have tackled a related problem, using BA to secure a file server.
  - By keeping BA out of the critical path, can avoid most of the delay BA normally imposes.

Split secrets

- In fact BA algorithms are just the tip of a broader "coding theory" iceberg.
- One exciting idea is called a "split secret".
  - Idea is to spread a secret among n servers so that any k can reconstruct the secret, but no individual actually has all the bits.
  - Protocol lets the client obtain the "shares" without the servers seeing one-another's messages.
  - The servers keep but can not read the secret!
- Question: In what ways is this better than just encrypting a secret?

How split secrets work

- They build on a famous result.
    - With k+1 distinct points you can uniquely identify an order-k polynomial.
        - i.e. 2 points determine a line.
        - 3 points determine a unique quadratic.
    - The polynomial is the “secret”.
    - And the servers themselves have the points – the “shares”.
    - With coding theory the shares are made just redundant enough to overcome n-k faults.

Byzantine Broadcast (BB)

- Many classical research results use Byzantine Agreement to implement a form of fault-tolerant multicast.
  - To send a message I initiate “agreement” on that message.
  - We end up agreeing on content and ordering w.r.t. other messages.
- Used as a primitive in many published papers.

Pros and cons to BB

- On the positive side, the primitive is very powerful.
  - For example this is the core of the Castro and Liskov technique.
- But on the negative side, BB is slow.
  - we will see ways of doing fault-tolerant multicast that run at 150,000 small messages per second.
  - BB: more like 5 or 10 per second.
- The right choice for infrequent, very sensitive actions… but wrong if performance matters.

Take-aways?

- Fault-tolerance matters in many systems
  - But we need to agree on what a “fault” is.
  - Extreme models lead to high costs!
- Common to reduce fault-tolerance to some form of data or “state” replication.
  - In this case fault-tolerance is often provided by some form of broadcast.
  - Mechanism for detecting faults is also important in many systems.
    - Timeout is common… but can behave inconsistently.
    - “View change” notification is used in some systems. They typically implement a fault agreement protocol.