CS514: Intermediate Course in Operating Systems

Professor Ken Birman
Vivek Vishnumurthy: TA

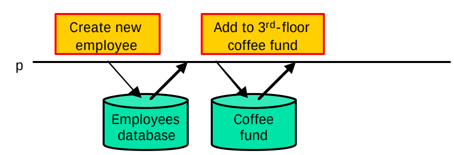
Applications of these ideas

- Over the past three weeks we've heard about
  - Gossip protocols
  - Distributed monitoring, search, event notification
  - Agreement protocols, such as 2PC and 3PC
- Underlying theme: some things need stronger forms of consistency, some can manage with weaker properties
- Today, let's look at an application that could run over several of these options, but where the consistency issue is especially "clear"

Let's start with 2PC and transactions

- The problem:
  - Some applications perform operations on multiple databases
  - We would like a guarantee that either *all* the databases get updated, or *none* does
- The relevant paradigm? 2PC

Problem: Pictorial version



- Goal? Either p succeeds, and both lists get updated, or something fails and neither does

Issues?

- P could crash part way through…
- … a database could throw an exception, e.g. "invalid SSN" or "duplicate record"
- … a database could crash, then restart, and may have "forgotten" uncommitted updates (presumed abort)

2PC is a good match!

- Adopt the view that each database votes on its willingness to commit
  - Until the commit actually occurs, update is considered temporary
  - In fact, database is permitted to discard a pending update (covers crash/restart case)
- 2PC covers all of these cases

Solution

- P runs the transactions, but warns databases that these are part of a transaction on multiple databases
  - They need to retain locks & logs
- When finished, run a 2PC protocol
  - Until they vote "ok" a database can abort
- 2PC decides outcome and informs them

Low availability?

- One concern: we know that 2PC blocks
  - It can happen if two processes fail
  - It would need to happen at a particular stage of execution and be the "right" two… but this scenario isn't all that rare
- Options?
  - Could use 3PC to reduce (not eliminate!) this risk, but will pay a cost on every transaction
  - Or just accept the risk
  - Can eliminate the risk with special hardware but may pay a fortune!

Drilling down

- Why would 3PC reduce but not eliminate the problem?
  - It adds extra baggage and complexity
  - And the result is that if we had a perfect failure detector, the bad scenario is gone
    - … but we only have timeouts
    - … so there is *still* a bad scenario! It just turns out to be less likely, if we estimate risks
- So: risk of getting stuck is "slashed"

Drilling down

- Why not just put up with this risk?
  - Even the 3PC solution can still sometimes get stuck
  - Maybe the "I'm stuck" scenario should be viewed as basic property of this kind of database replication!
- This approach leads towards "wizards" that sense the problem and then help DB administrator relaunch database if it does get stuck

Drilling down

- What about special hardware?
  - Usually, we would focus on dual ported disks that have a special kind of switching feature
  - Only one node "owns" a disk at a time. If a node fails, some other node will "take over" its disk
- Now we can directly access the state of a failed node, hence can make progress in that mystery scenario that worried us
- But this can add costs to the hardware

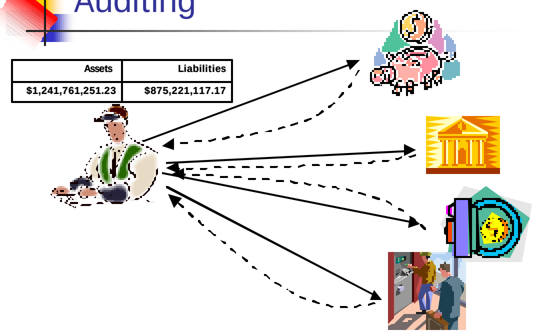
Connection to consistency

- We're trying to ensure a form of "all or nothing" consistency using 2PC
- Idea for our database is to either do the transaction on all servers, or on none
- But this concept can be generalized

Auditing

- Suppose we want to “audit” a system
  - Involves producing a summary of the state
  - Should look as if system was idle
- Some options (so far)…
  - Gossip to aggregate the system state
  - Use RPC to ask everyone to report their state.
  - With 2PC, first freeze the whole system (phase 1), then snapshot the state.

Auditing



Uses for auditing

- In a bank, may be the only practical way to understand “institutional risk”
  - Need to capture state at some instant in time. If branches report status at closing time, a bank that operates world-wide gets inconsistent answers!
- In a security-conscious system, might audit to try and identify source of a leak
- In a hospital, want ability to find out which people examined which records
- In an airline, might want to know about maintenance needs, spare parts inventory

Other kinds of auditing

- In a complex system that uses locking might want to audit to see if a deadlock has arisen
- In a system that maintains distributed objects we could “audit” to see if objects are referenced by anyone, and garbage collect those that aren’t

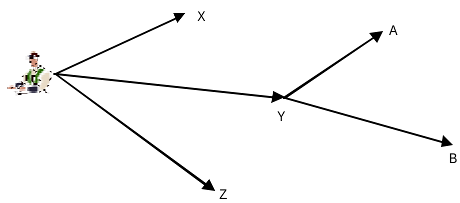
Challenges

- In a complex system, such as a big distributed web services system, we won’t “know” all the components
  - The guy starting the algorithm knows it uses servers X and Y
  - But server X talks to subsystem A, and Y talks to B and C…
- Algorithms need to “chase links”

Implications?

- Our gossip algorithms might be ok for this scenario: they have a natural ability to chase links
- A simple RPC scheme (“tell me your state”) becomes a nested RPC

Nested RPC

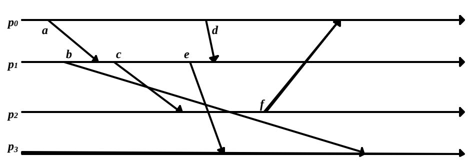


Temporal distortions

- 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

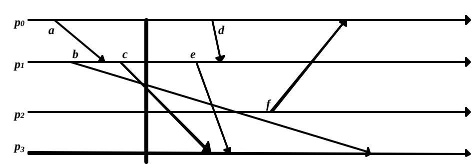
Temporal distortions

- What does "now" mean?



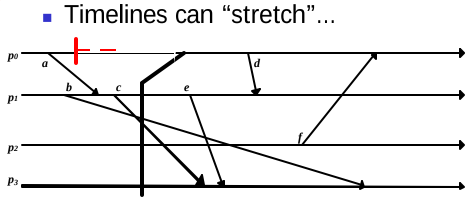
Temporal distortions

- What does "now" mean?



Temporal distortions

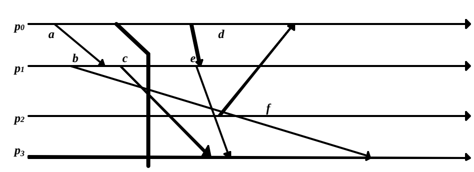
- Timelines can "stretch"...



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

Temporal distortions

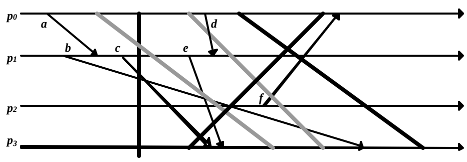
- Timelines can "shrink"



- E.g. something lets a machine speed up

Temporal distortions

- Cuts represent instants of time.



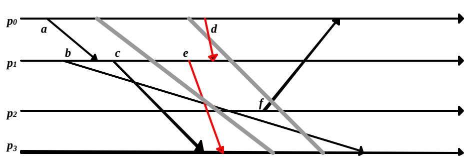
- But not every “cut” makes sense
  - Black cuts could occur but not gray ones.

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

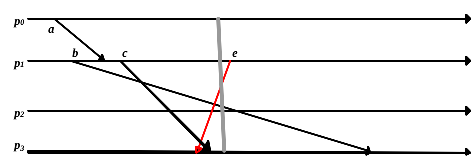
Temporal distortions

- 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”

Who cares?

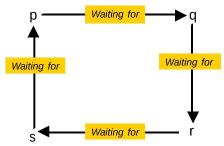
- In our auditing example, we might think some of the bank’s money is missing
- Or suppose that we want to do distributed deadlock detection
  - System lets processes “wait” for actions by other processes
  - A process can only do one thing at a time
  - A deadlock occurs if there is a circular wait

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…

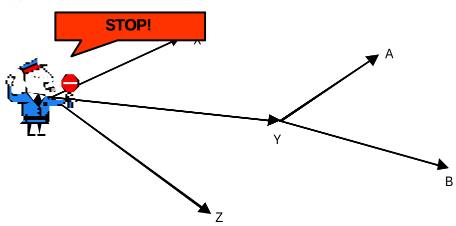


- … but is it a deadlock?

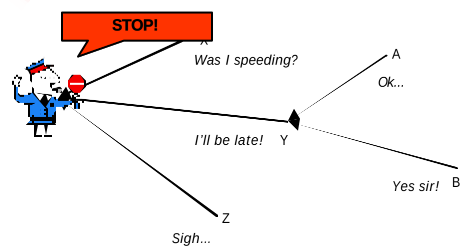
Phantom deadlocks!

- 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.

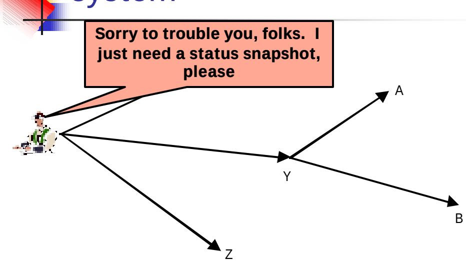
One solution is to "freeze" the system



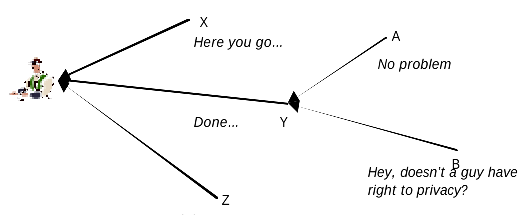
One solution is to "freeze" the system



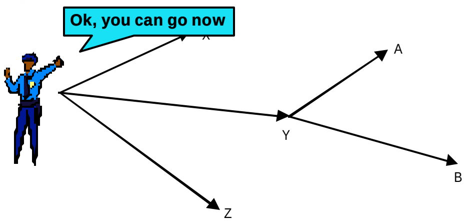
One solution is to "freeze" the system



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

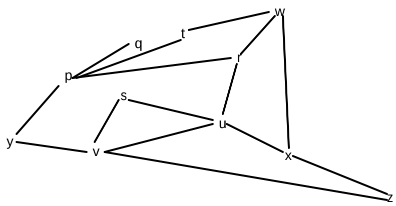
Turn idea into an algorithm

- 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

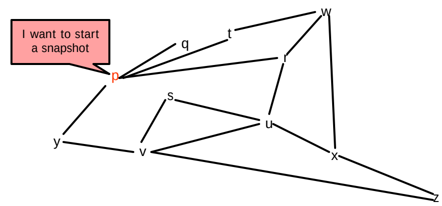
- Outgoing wave of requests... incoming wave of snapshots and channel state
- Snapshot ends up accumulating at the initiator, p_i
- Algorithm doesn't tolerate process failures or message failures.

Chandy/Lamport



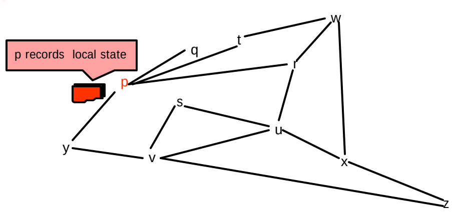
A network

Chandy/Lamport



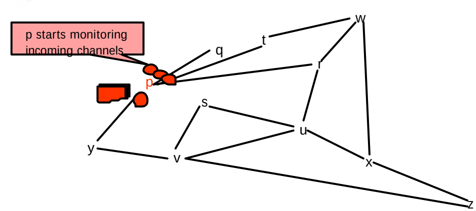
A network

Chandy/Lamport



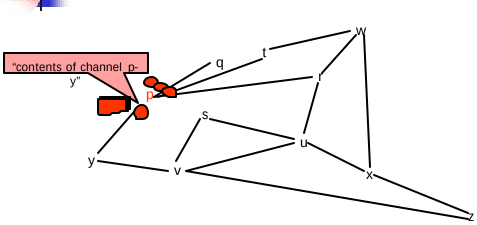
A network

Chandy/Lamport



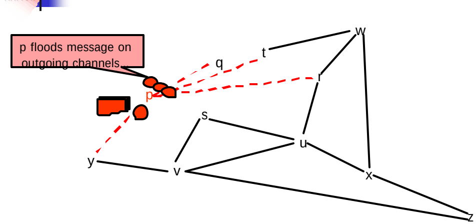
A network

Chandy/Lamport



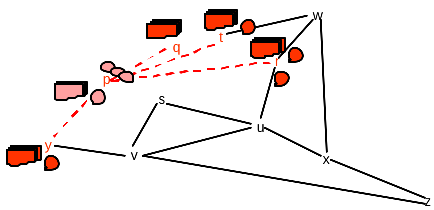
A network

Chandy/Lamport



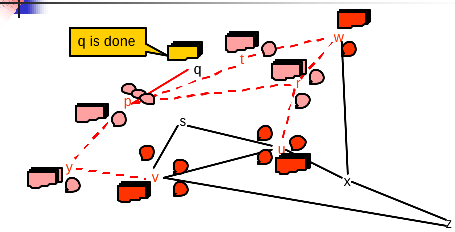
A network

Chandy/Lamport



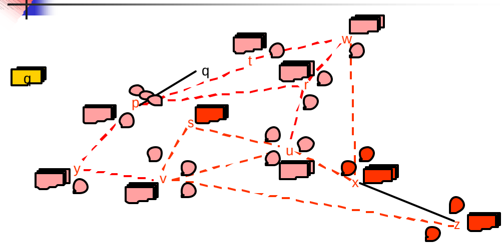
A network

Chandy/Lamport



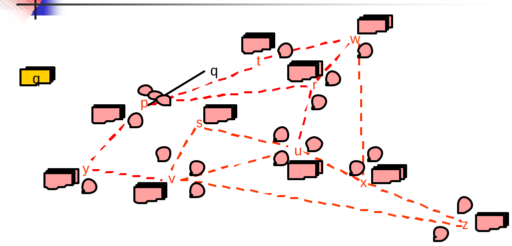
A network

Chandy/Lamport



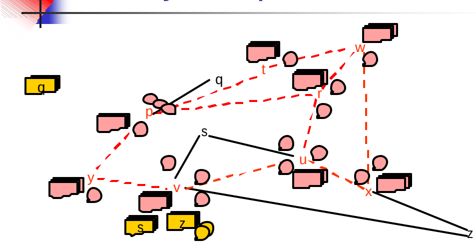
A network

Chandy/Lamport



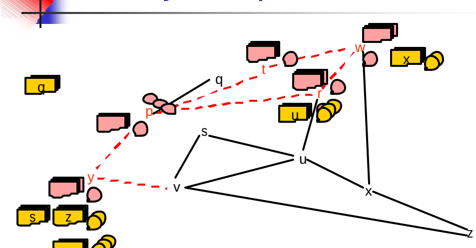
A network

Chandy/Lamport



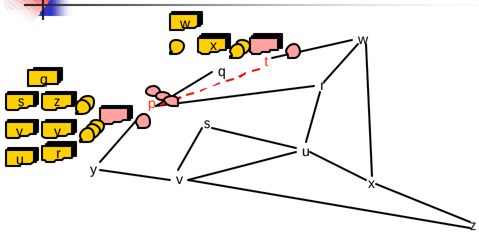
A network

Chandy/Lamport



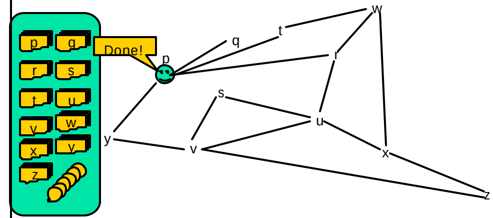
A network

Chandy/Lamport



A network

Chandy/Lamport



A snapshot of a network

Practical implication

- Snapshot won't occur at a point in *real* time
  - Could be noticeable to certain kinds of auditors
  - In some situations only a truly instantaneous audit can be accepted, but this isn't common
- What belongs in the snapshot?
  - Local states… namely "status of X when you asked"
  - Messages in transit… e.g. of we're transferring $1M from X to Y (otherwise that money would be missing)

Recap and summary

- We've begun to develop powerful, general tools
  - They aren't always of a form that the platform can (or should) standardize
  - But we can understand them as templates that can be specialized to our needs
  - Thinking this way lets us see that many practical questions are just instances of the templates we've touched on in the course

What next?

- We'll resume the development of primitives for replicating data
  - First, notion of group membership
    - Turns out to have a very strong connection to our snapshot algorithm!
  - Then fault-tolerant multicast
  - Then ordered multicast delivery
  - Finally leads to virtual synchrony "model"
- Then tackle more practical problems