Ordering and Consistent Cuts

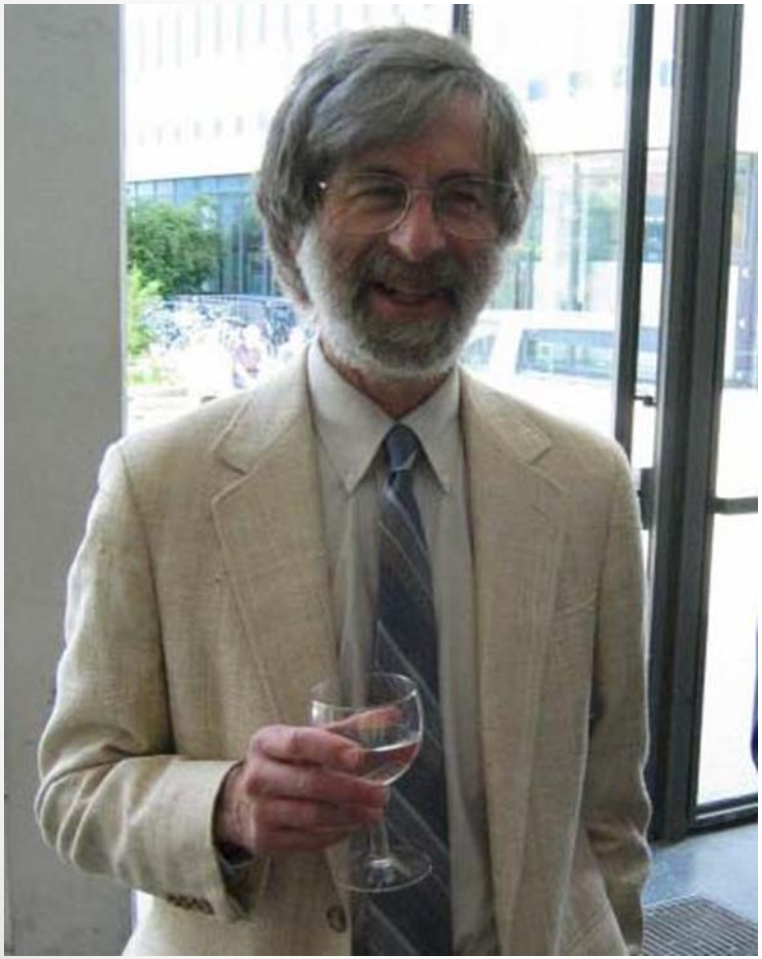
Edward Tremel

11/7/2013

Synchronizing Distributed Systems

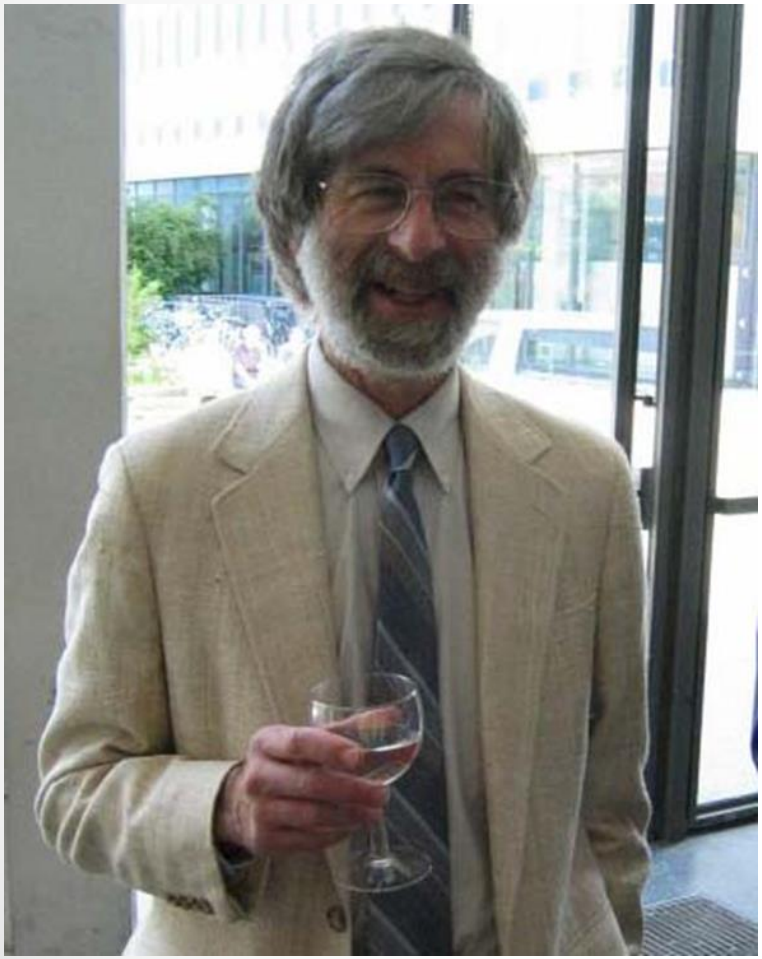
- *Time, Clocks, and the Ordering of Events in Distributed Systems*
  - How to agree on an order of events across asynchronous processes
  - Synchronized concurrent execution of a state machine
  - Synchronizing clocks across a network
- *Distributed Snapshots: Determining Global States of Distributed Systems*
  - How to record state of a distributed system without losing information
  - Determining when a stable property is satisfied
  - Synchronizing phases of distributed computation

Or: Leslie Lamport Invents Things



- Two of the most influential papers in distributed systems
- Almost entirely original work by Leslie Lamport
- Distributed Systems was still a brand-new field
  - PODC not until 1982

Also Invented by Lamport



- Sequential consistency
- Bakery algorithm
  - Mutual exclusion without hardware support
- Atomic registers
- Byzantine Generals' Problem
- Paxos Algorithm
- Temporal Logic of Actions
- LaTeX

Leslie Lamport chilling on a boat with Andy van Dam

(left: Hector Garcia-Molina, distributed systems researcher)

Biographical Highlights

- BS in Math from MIT, 1960
- MA, PhD in Math from Brandeis, 1972
- Taught math at Marlboro College 1965-69
- Research in industry
  - Massachusetts Computer Associates (1970-77)
  - SRI International (1977-85)
  - DEC/Compaq (1985-2001)
  - Microsoft Research (2001-)
- Many awards, including 2000 PODC Influential Paper award for *Time, Clocks, and the Ordering of Events*

Time, Clocks, and the Ordering of Events in a Distributed System

- Written by Lamport in 1978
- Inspired by *The Maintenance of Duplicate Databases* (Paul Johnson and Bob Thomas)
  - Database update messages must be timestamped
  - Updates are ordered by timestamp, not message receive order
  - Did not account for clock inconsistency
- Intended goal: Show how to implement arbitrary distributed state machine

Setup

- Assumption: Distributed systems don't have a common (physical) clock
- Still need to agree on when events happened
- Nodes in distributed system have shared state, must apply updates in same order to stay consistent
- Examples:
  - Bank servers at different branches, need to know order of transactions
  - Distributed database, need to know when value was added or changed
  - Distributed filesystem, need to know order of writes
  - Distributed lock manager, need to agree on who got the lock

Setup

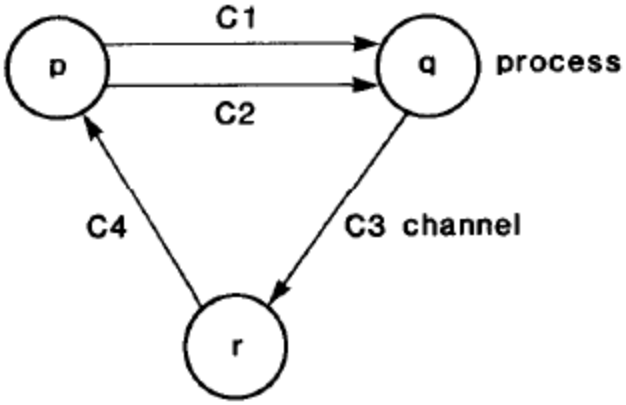


Fig. 1. A distributed system with processes p, q, and r and channels c1, c2, c3, and c4.

- Assumption: Distributed systems communicate by sending messages over directed channels
  - No other way to share state between nodes
  - No Ethernet (shared line)
  - Basically the same model as microkernel processes
- Assumption: Channels are FIFO ordered and reliable

"Happened Before"

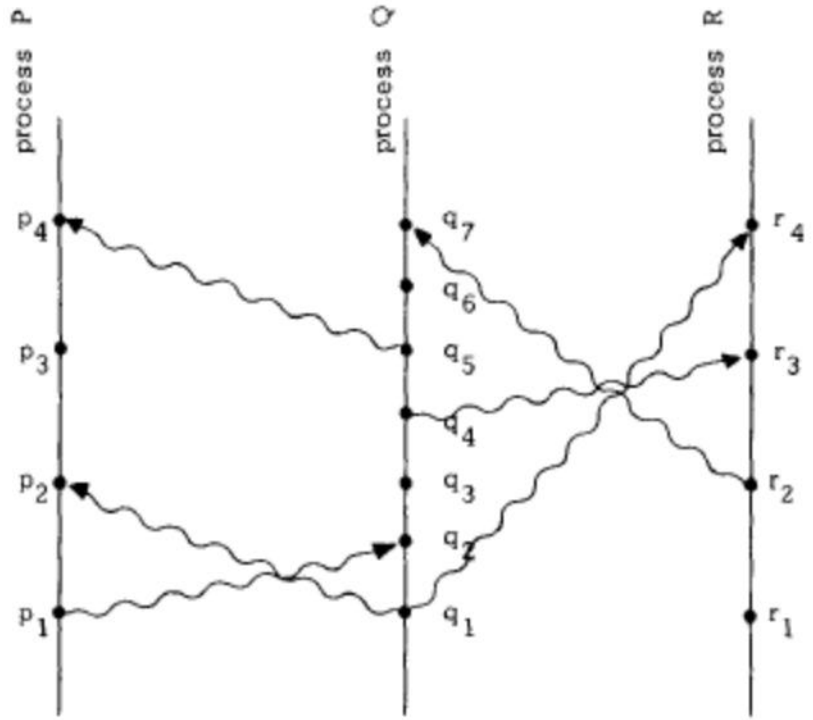


Fig. 1 from *Time, Clocks, and the Ordering of Events*

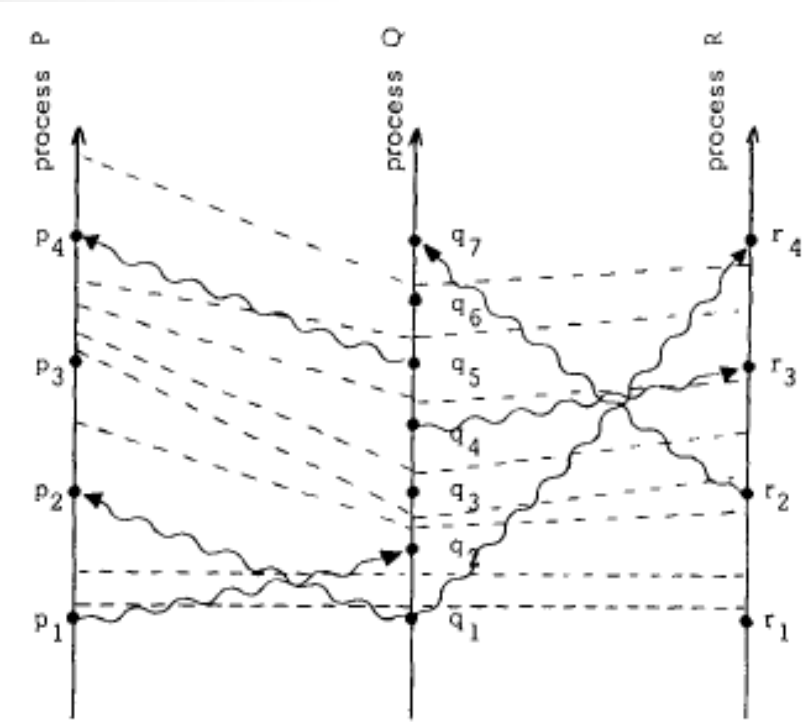
"Happened Before"

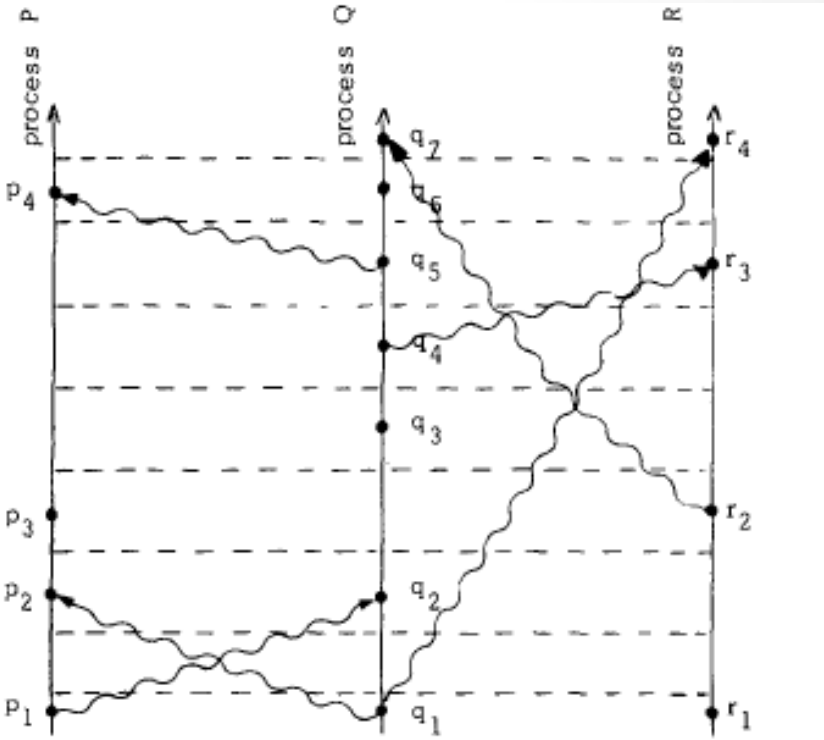
- Natural, straightforward partial order on events
- $a \rightarrow b$ if *a* and *b* are in the same process and *a* precedes *b* in execution
- $a \rightarrow b$ if *a* is sending of message by one process and *b* is receipt of message by another process
- Events in different processes that are not message sends/receives cannot be ordered
- Relation is transitive: $a \rightarrow b$ and $b \rightarrow c$ means $a \rightarrow c$
- Not reflexive: $a \rightarrow a$ is impossible

Logical Clocks

- Concrete representation of "happened before"
- Each process has a clock
  - Assigns a number to an event
  - Event = send message, receive message, computation (internal)
  - Monotonically increasing
- If *a* and *b* are events in process *i* and *a* comes before *b*, then $C_i(a) < C_i(b)$
- If *a* is the sending of a message by process *i*, and *b* is the receipt of the message by process *j*, then $C_i(a) < C_j(b)$

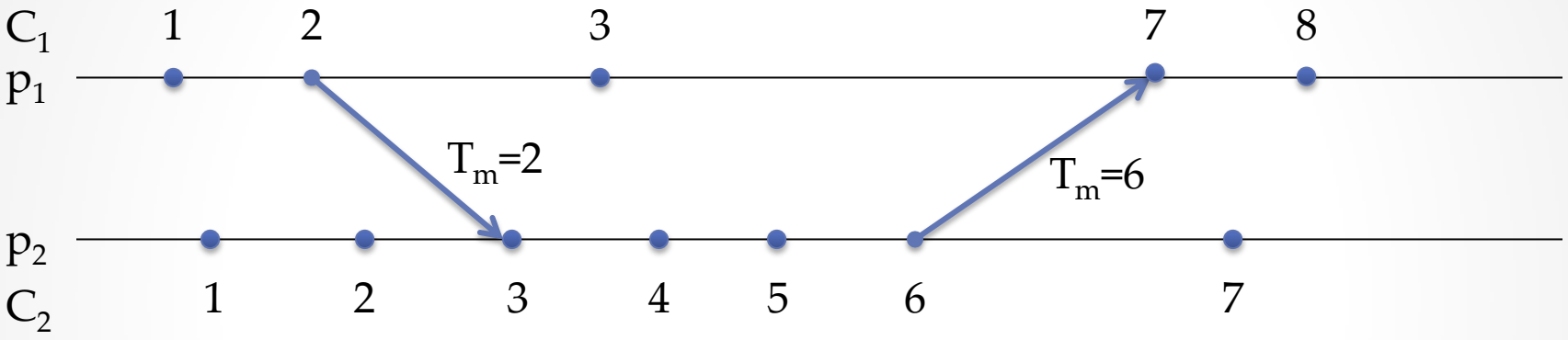
Visualizing Clock Ticks





Figs. 2 and 3 from *Time, Clocks, and the Ordering of Events*

Synchronizing Clocks



- Clock increments between events
- Every message sent with timestamp of sending process
- When process receives message, it must advance its clock to greater than message's timestamp

Ordering Events

- Clocks by themselves are still a partial order on events
- Total Order: Clocks plus arbitrary tiebreaking
- Given a total order on processes, can construct a total order on events
- $a \Rightarrow b$ if $C_i(a) < C_j(b)$
- $a \Rightarrow b$ if $C_i(a) = C_j(b)$ and process *i* is ordered before process *j*
- Total order on processes: process IDs, machine IPs

State Machine Replication

- Each process keeps its own copy of the state
- Processes send messages with commands
- Command messages are cached and acknowledged
- A process can execute a command when it has learned of all commands issued before that command's timestamp
- Progress guaranteed because communication channels are reliable and FIFO
- State machine replication without reliable channels: much harder problem, also solved by Lamport

Physical Clocks

- Can use physical clocks instead of logical clocks, as long as they can only be set forward
- Assume μ_m = minimum duration of message transit
- Each process's physical clock ticks continuously
- When a process receives a message, it advances its clock to message timestamp + μ_m
- Difference between any two clocks can be bounded if error in clock rates and unpredictable message delay can be bounded
  - Requires sending a message at least once every τ seconds

Significance

- Lamport's opinion:

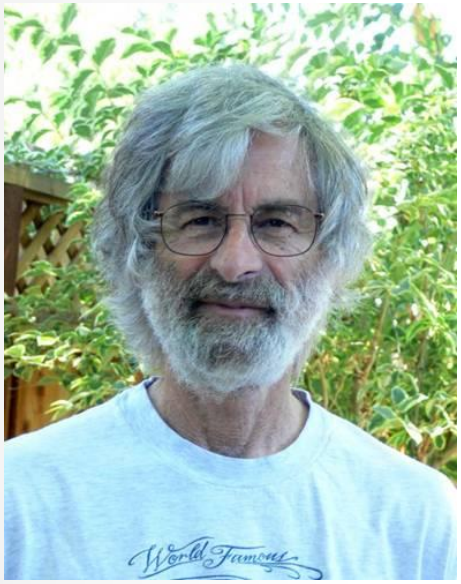
  "Jim Gray once told me that he had heard two different opinions of this paper: that it's trivial and that it's brilliant. I can't argue with the former, and I am disinclined to argue with the latter."

- References: 4
- Citations: 8196
- Basis of vector clocks (Fidge), which are often used in distributed systems
- Also network time, Paxos protocol
- But most people remember it for causality relation or distributed mutual exclusion, not state machines

Questions

- Is this brilliant? Trivial? Both?
- What's more important: intended goal or remembered result?
- Is application to physical clocks necessary or helpful?
  - What about inescapable forward drift? Clocks can't be set back…

Distributed Snapshots



Leslie Lamport

- At this point, working at Stanford Research Institute (SRI International)

K. Mani Chandy

- PhD from MIT in EE, 1969
- Professor at UT Austin 1970-89
- Professor at Caltech since 1989

Origins of the Paper

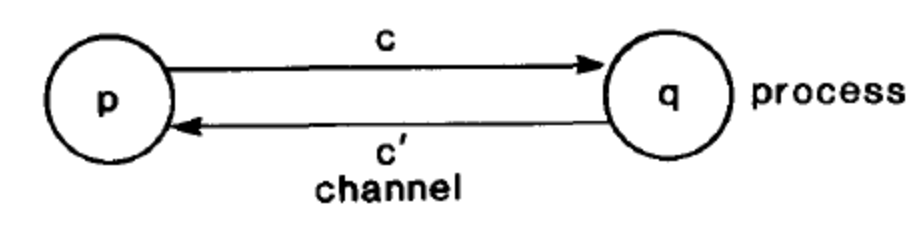
"The distributed snapshot algorithm described here came about when I visited Chandy, who was then at the University of Texas in Austin. He posed the problem to me over dinner, but we had both had too much wine to think about it right then. The next morning, in the shower, I came up with the solution. When I arrived at Chandy's office, he was waiting for me with the same solution. I consider the algorithm to be a straightforward application of the basic ideas from [Time, Clocks, and the Ordering of Events in Distributed Systems]."

—Leslie Lamport

- Acknowledgements: Dijkstra, Hoare, Fred Schneider

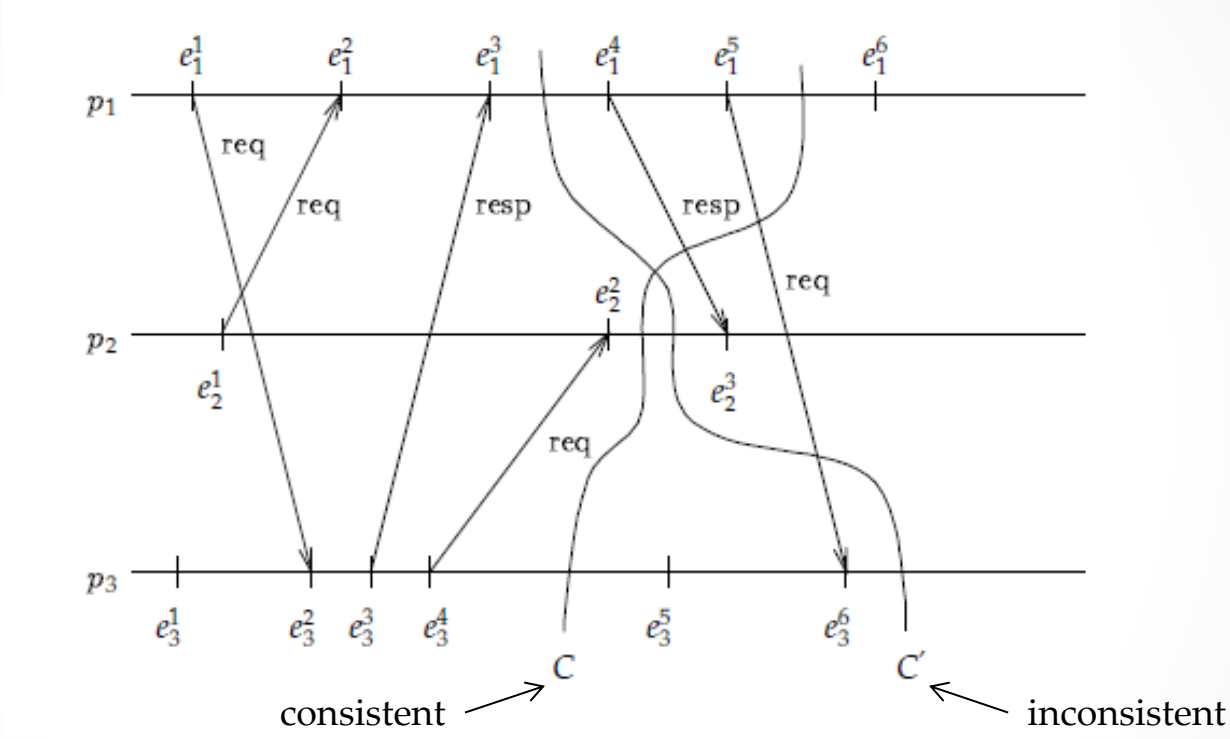
The Problem

- Recording state of a distributed system is important
  - Determining stable properties, such as "phase completed"
- No way to ensure all nodes record state at "exactly" the same time
- Naïve solution can record an impossible state



  - Record state of *p* and *c'* while *p* has token
  - Then *p* sends token along *c*
  - Record state of *q* and *c*, showing token is in *c*
  - Snapshot shows token in two places, but only one token exists!

Consistent Cuts



- Need a *consistent cut*: If an event is in the snapshot, all events that happen before it must be in snapshot

(image copied from Dinesh Bhat's 2010 presentation)

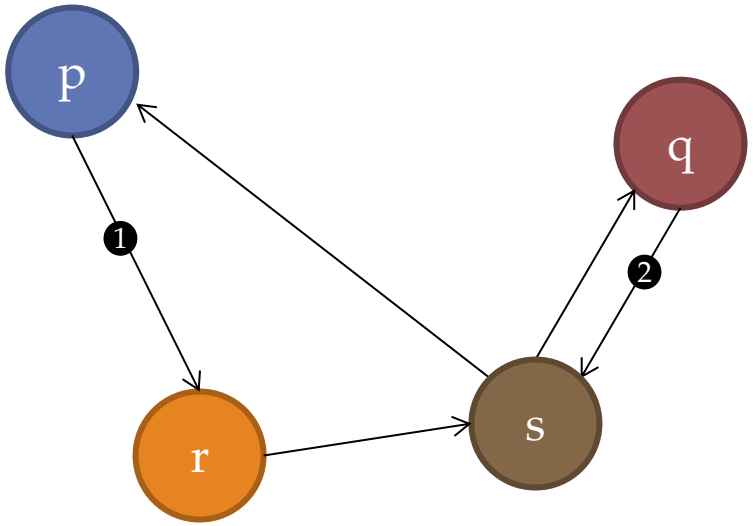
The Solution

- Send a *marker* along all channels immediately after recording state
- Upon receipt of a marker along channel c:
  - Record process state if not already recorded
  - Record state of c as all messages received between recording process state and receiving marker
- Eventually markers will reach all processes, so all state will be recorded

Assumptions

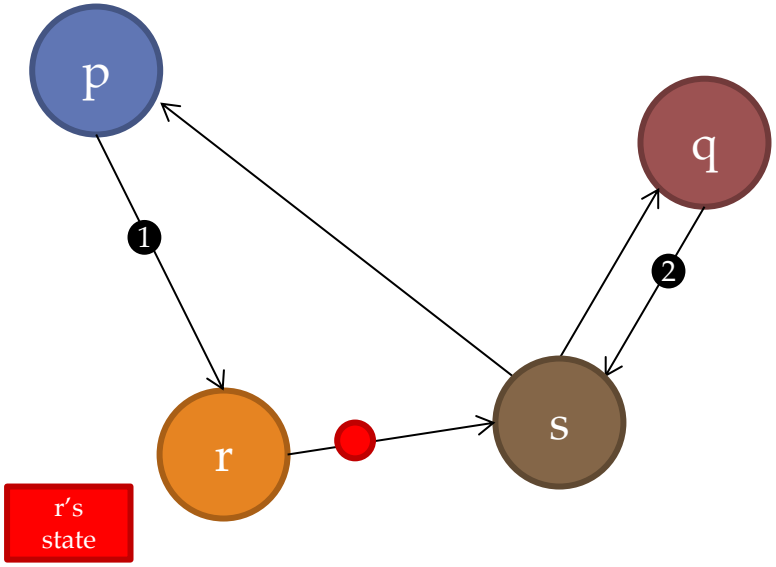
- Graph of processes is strongly connected
  - If your network is really Ethernet, it is
- Processes can atomically record their own state
- Processes keep log of messages received
- Processes do not fail
- Channels are still reliable and FIFO
- There is some way to collect the snapshot from all nodes once done recording

Example

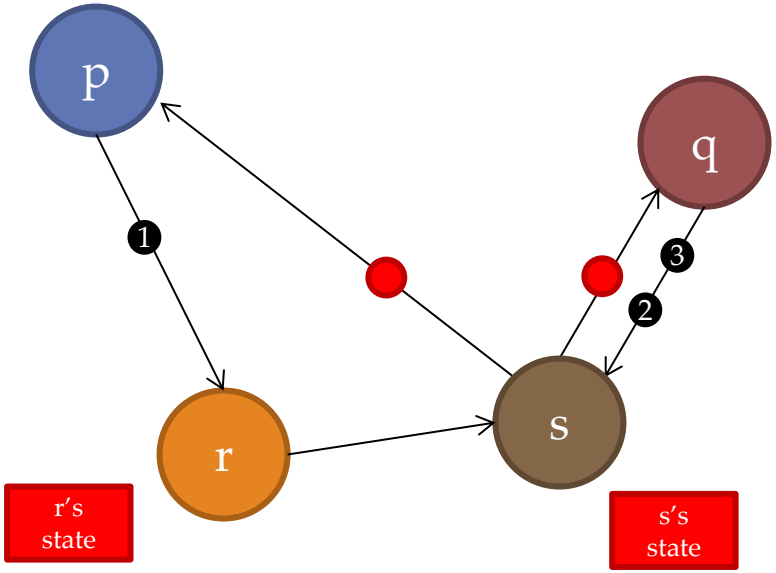


(shamelessly stolen from Isaac's presentation last year)

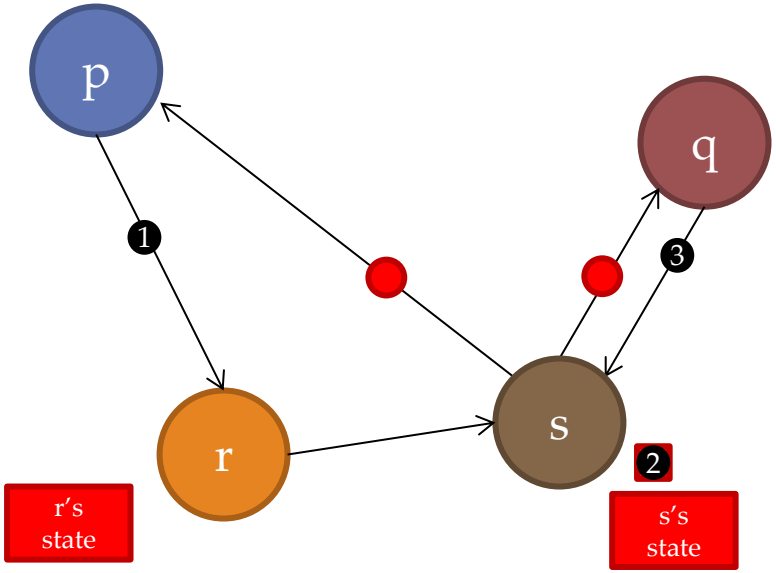
Example



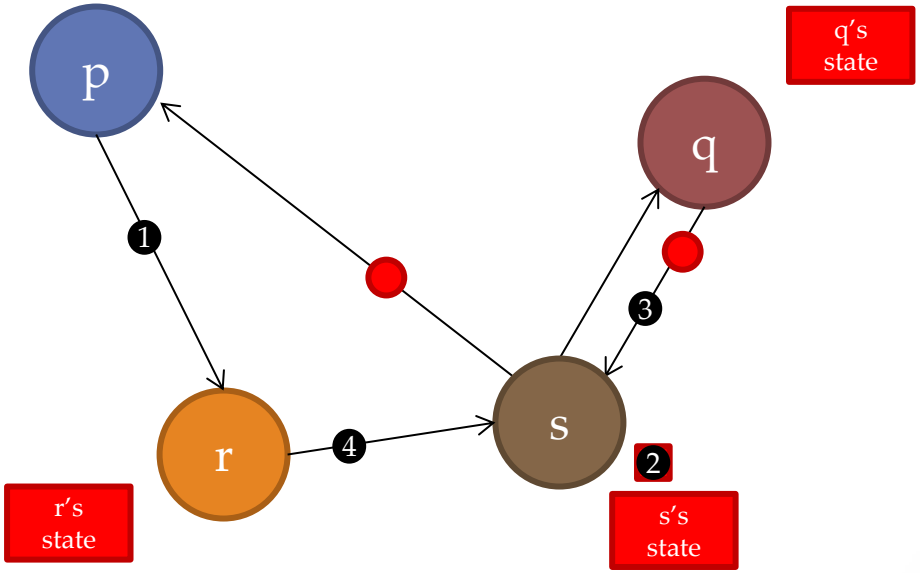
Example



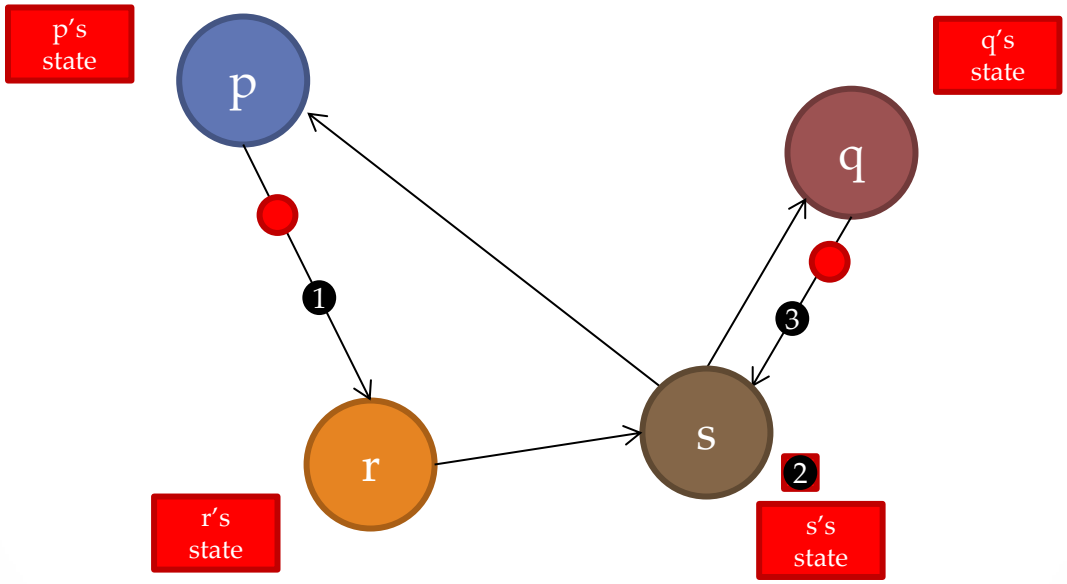
Example



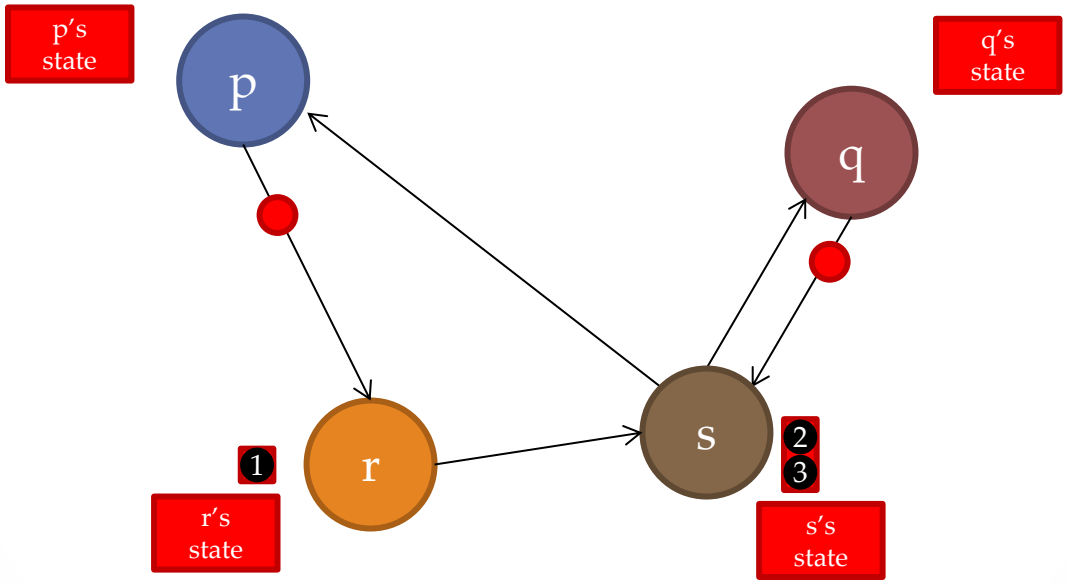
Example



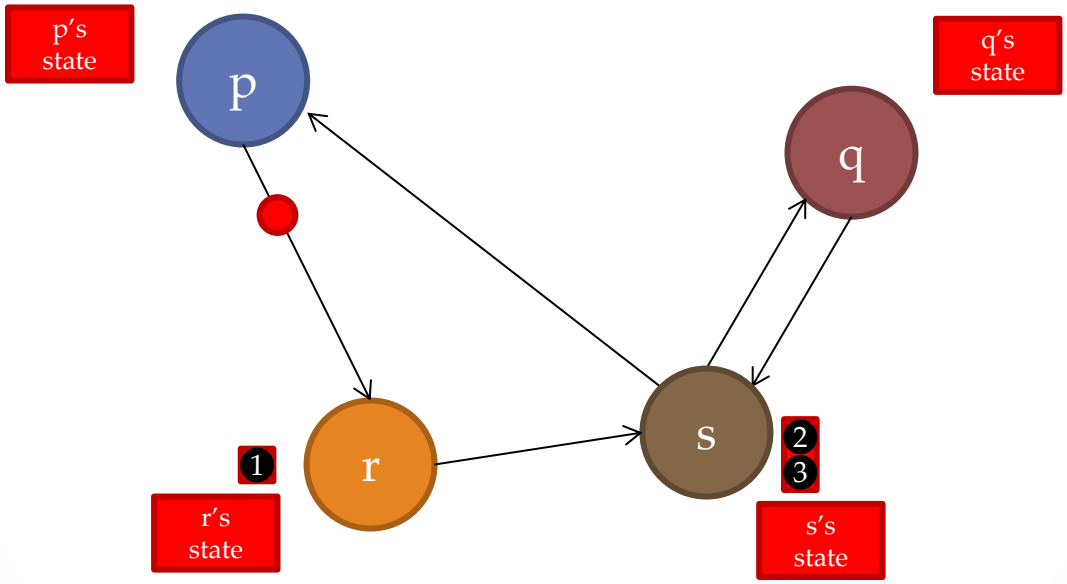
Example



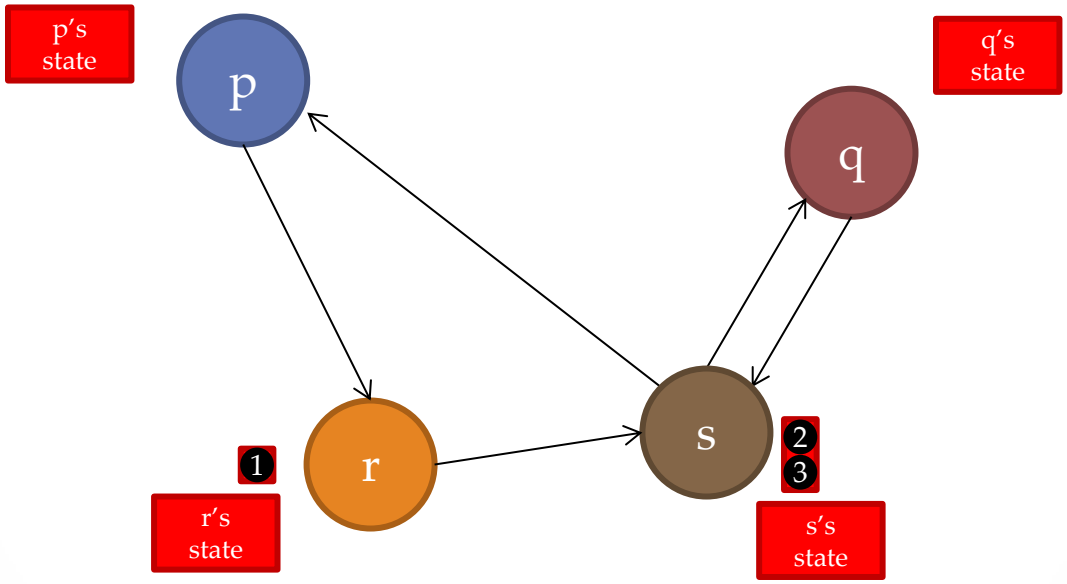
Example



Example



Example



Properties of Snapshot

- Produces a consistent cut: If a message's receipt is recorded, then its sending is also recorded (FIFO)
- Global system state recorded in snapshot may not have ever occurred
- But it will be possible and reachable by a re-ordering of events within snapshot window
- Sequentially consistent with state actually reached by system during snapshot
  - Only relative order of unconnected events on different processes is changed
- Only events within snapshot window affected

Significance

- References: 11, one of which doesn't exist
- Citations: 2564
- First "consistent cut" algorithm, basis of more complex ones such as Mattern's
- Useful for logging, fault-tolerance

Questions

- This still assumes reliable, FIFO channels between processes. Is that reasonable?
- Is the "sequentially consistent" snapshot good enough?
- Are clocks actually necessary for this algorithm? (Lamport claims it's an extension of clocks).
- Is it efficient to use this algorithm for stability detection?