Ordering and Consistent Cuts

Dan Deng

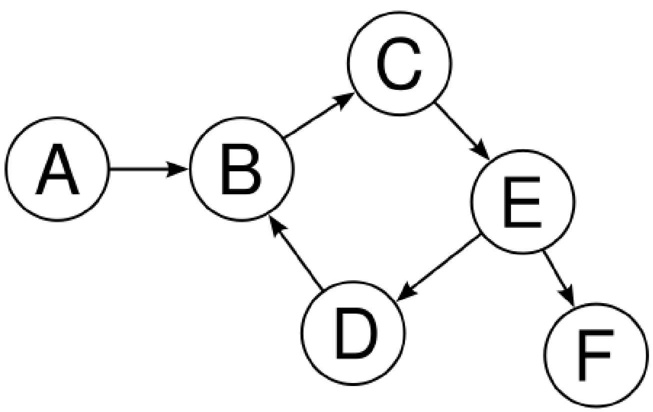
10/30/11

Introduction

- Distributed systems
  - Loosely coupled processes cooperating to solve a bigger problem
- Novelty of distributed systems
  - Lamport published his paper in 1978
  - ARPANET was just “operational” in 1975
  - Temporal characteristics poorly understood
- Need a mechanism for processes to agree on time

Distributed System Model

- Distributed system of sets of processes and channels
- Processes communicate by sending and receiving messages
- A process can observe:
  - Its own state
  - Messages it sends
  - Messages it receives
- Must enlist other processes to determine global state

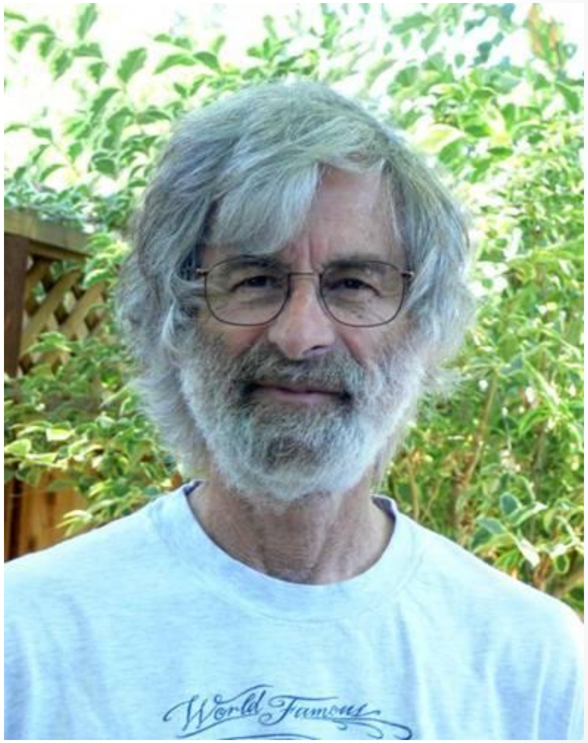


Time, Clocks, and the Ordering of events in a Distributed System – PODC influential paper award (2000)

Leslie Lamport

(Massachusetts Computer Associates)

- B.S. in math from MIT (1960)
- Ph.D. in math from Brandeis (1972)
- Microsoft Research (2001-Current)
- Distributed systems, LaTeX
- *“a distributed system is one in which the failure of a computer you didn’t even know existed can render your own computer unusable”*



Takeaways

- Happened-before using logical clocks to totally order events
- Logical clocks used to implement mutual exclusion
- Physical clocks for anomalous behavior

Discussion points:

- Useful model for reasoning about temporal events
- Logical clock overflow not considered
- Does not answer precisely questions of concurrency or dependency

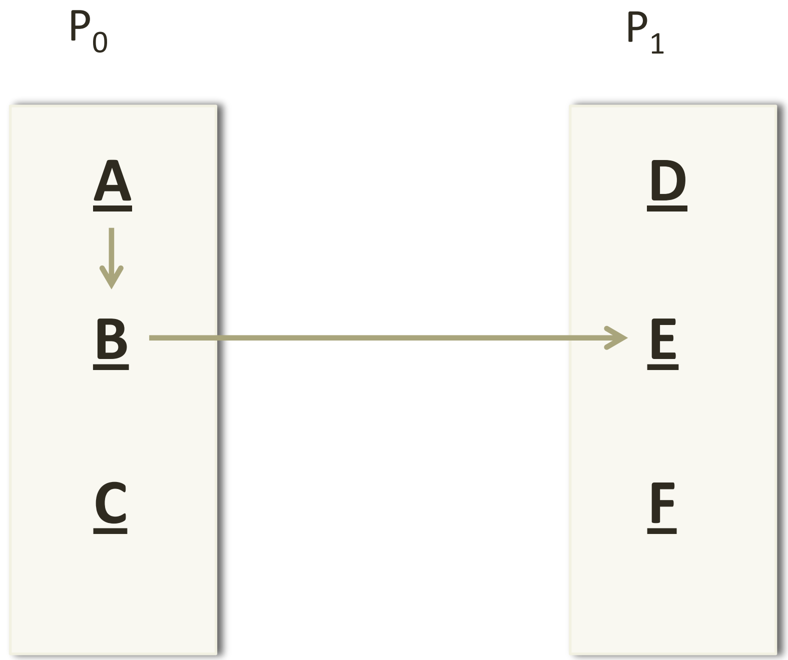
Outline

- Motivation
- Partial ordering
- Logical clocks
- Total ordering
- Mutual exclusion
- Anomalous behavior
- Physical clocks

Motivation

- Our notion of event ordering is derived using time
- Time is implemented on machines using clocks
- Local clocks on machines may not be accurate
- Need another mechanism to agree on time

Partial Ordering



$A \rightarrow B$, $B \rightarrow E$, $A \rightarrow E$

$A \nrightarrow D$, $D \nrightarrow A$, A and D are concurrent

Logical Clocks

- Used to implement the happened-before relation

$$A \Rightarrow B \qquad C_i(A) < C_i(B)$$

- Between successive events in a process:
  - Each process increments its logical clock
- On event A of sending of a message from process P_i
  - P_i sends $T_m = C_i(A)$ with message
- On event B of receiving of a message by process P_j
  - B advances $C_j(B)$ to $MAX(T_m, C_j(B))+1$

Total Ordering

- Happens-before gives only a partial ordering of events
- Can totally order events by
  - Ordering events by the logical times they occur
  - Break ties using an arbitrary total ordering of processes
- Specifically A happens before B if
  - $C_i(A) < C_j(B)$
  - $C_i(A) == C_j(B)$ and $P_i < P_j$

Total Ordering

- Can be used to solve the mutual exclusion problem in a fully distributed fashion

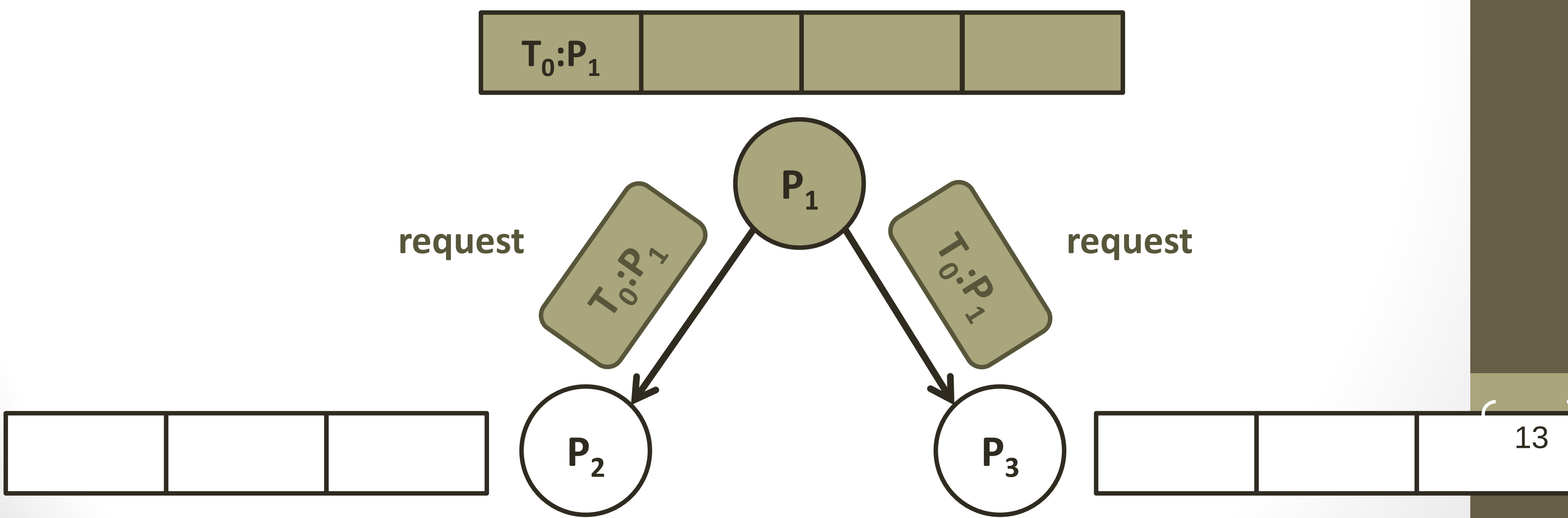
- Problem description:
  - Fixed number of processes
  - A single resource
  - Processes must synchronize to avoid conflict
  - Requests must be granted in order

Mutual Exclusion

- Each process maintains its own request queue
- Process P_i – To request the resource
  - Add Request T_m:P_i to its queue
  - Send Requests T_m:P_i to all P_j
- Process P_j – On receiving Request T_m:P_i
  - Add Request T_m:P_i to its queue
  - Send Acknowledge message to P_i
- Process P_i is granted resource when
  - Request T_m:P_i is earliest in request queue
  - Acknowledge is received from all P_j

Mutual Exclusion

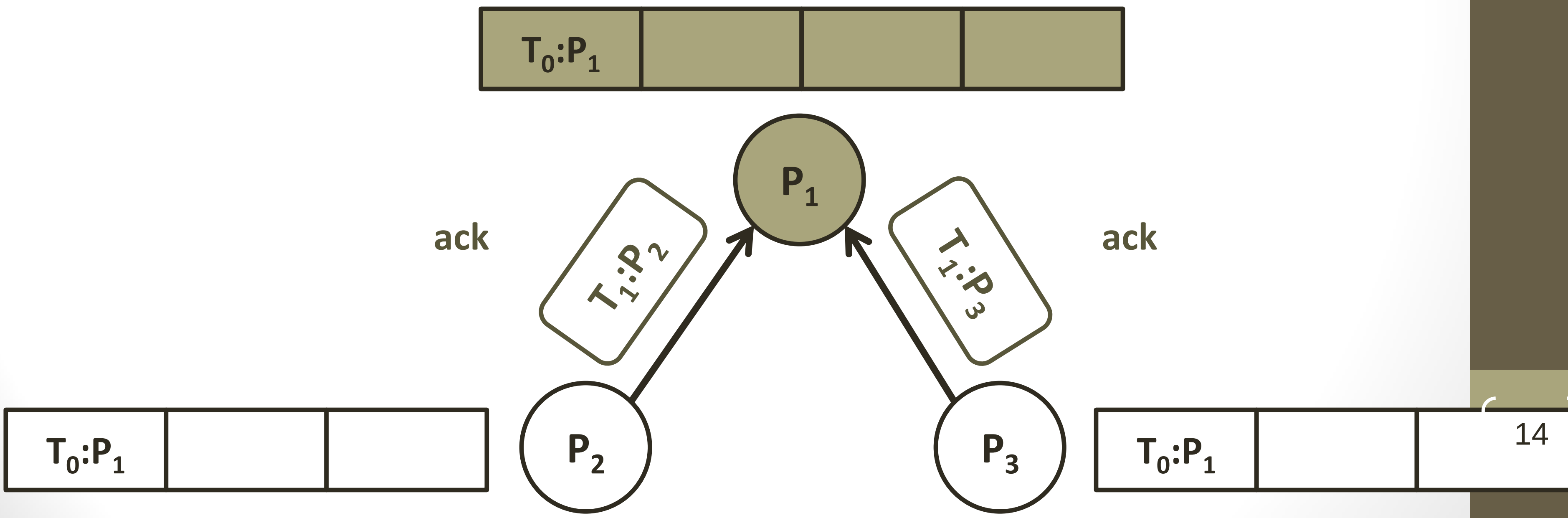
- **Step 1: P_i Sends Request Resource**
  - **P_i** puts **Request T_m:P_i** on its request queue
  - **P_i** sends **Request T_m:P_i** to **P_j**



Source: Nicole Caruso's F09 CS6410 Slides

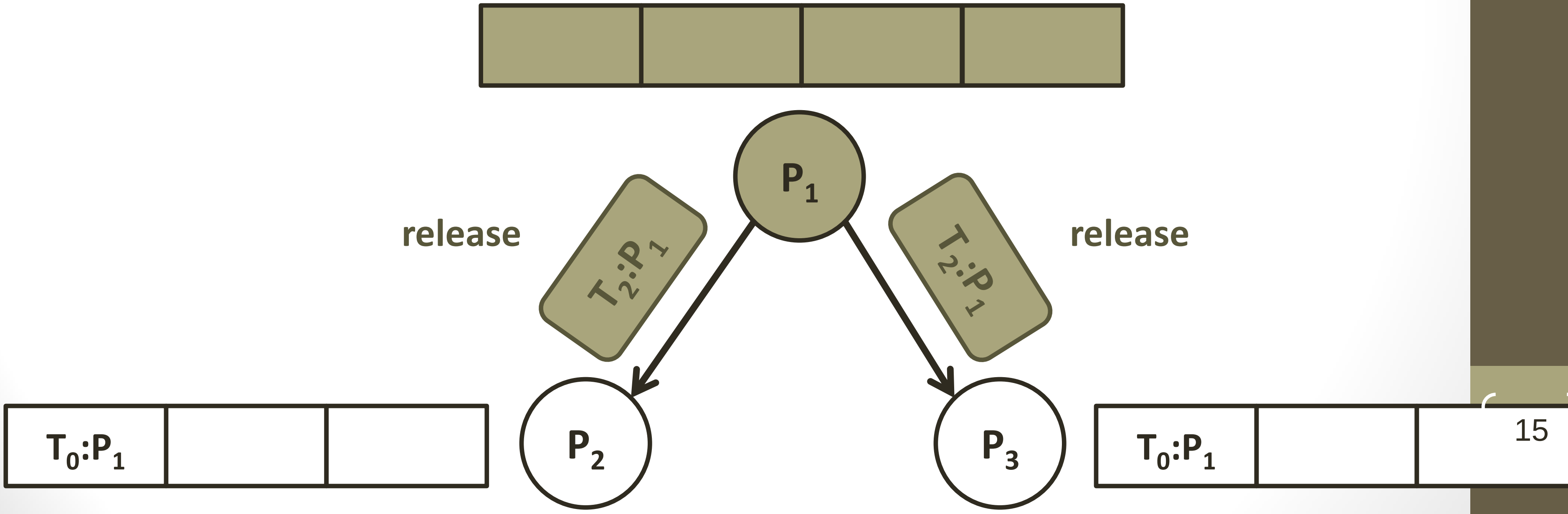
Mutual Exclusion

- **Step 2: P_j Adds Message**
  - **P_j** puts **Request T_m:P_i** on its request queue
  - **P_j** sends **Acknowledgement T_m:P_j** to **P_i**



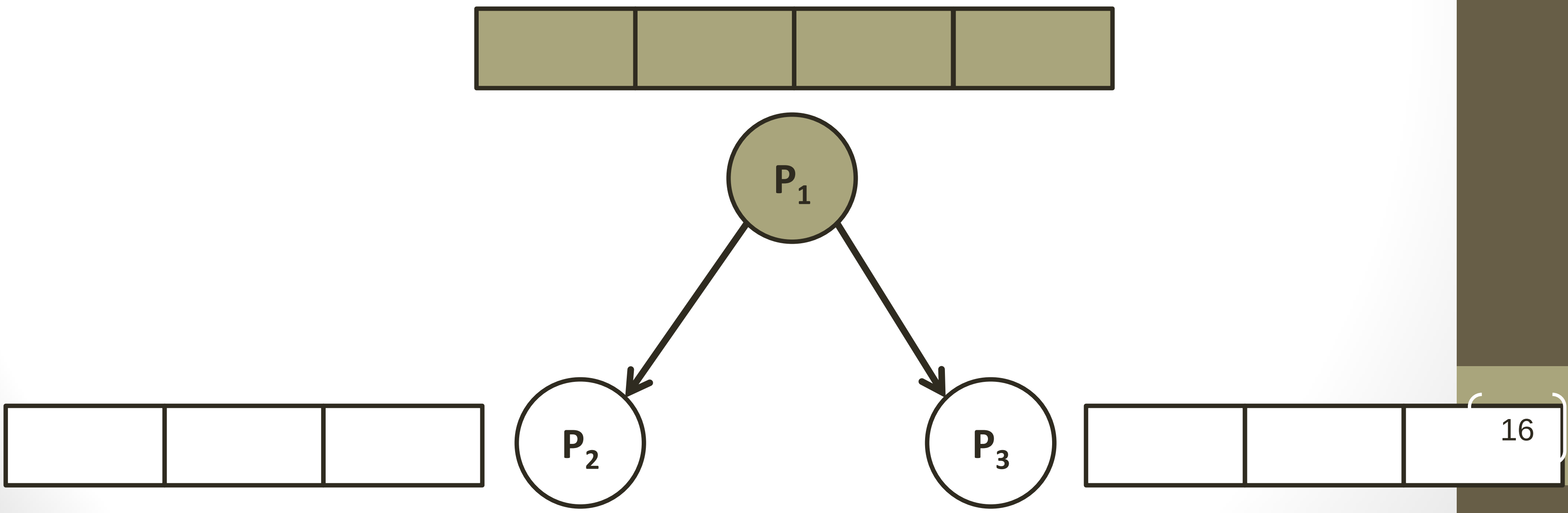
Mutual Exclusion

- **Step 3: P_i Sends Release Resource**
  - **P_i** removes **Request T_m:P_i** from request queue
  - **P_i** sends **Release T_m:P_i** to each **P_j**

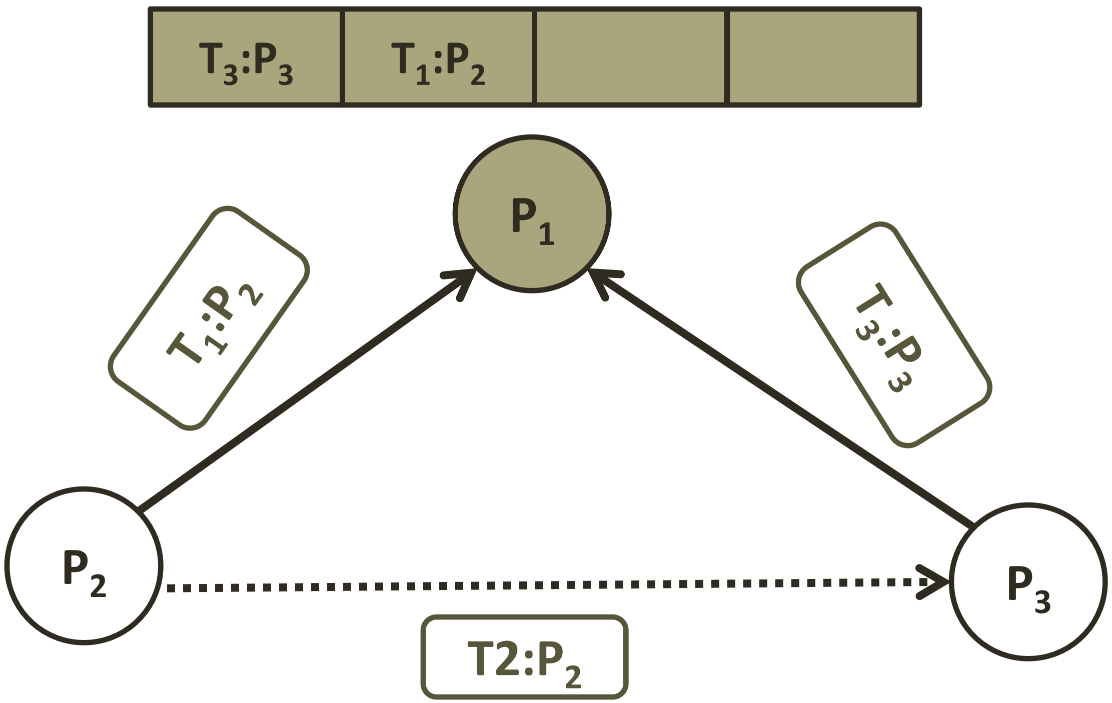


Mutual Exclusion

- **Step 4: P_j Removes Message**
  - **P_j** receives **Release T_m:P_i** from **P_i**
  - **P_j** removes **Request T_m:P_i** from request queue



Anomalous Behavior



- Can occur if some messages are not observed

those events in any way with the other events in $\underline{\mathscr{L}}$, can guarantee that request A is ordered before request B.

There are two possible ways to avoid such anomalous

Physical Clocks

- A physical clock (C) must run at about the right rate
  - $|dC_i(t) / dt - 1| < k$ where $k << 1$
- Physical clocks must be somewhat synchronized
  - $| Ci(t) - Cj(t) | < \varepsilon$
- Let μ < shortest transmission time for interprocess messages

- To prevent anomalous behavior
  - $C_i(t + \mu) - C_j(t) > 0$
  - $\varepsilon < \mu * (1 - k)$

Distributed snapshots: determining global states of distributed systems

K. Mani Chandy (UT-Austin)

- Indian Institute of Technology (B.E. 1965)
- Polytechnic Institute of Brooklyn (M.S. 1966)
- MIT (Ph.D. 1969)
- CS Department at UT-Austin (1970-1989) (department chair 1978-79 and 1983-85)
- CS Professor at CalTech (1989-Current)

Leslie Lamport (SRI, 1977-1985)

Takeaways

- Distributed algorithm to determine global state
- Detect stable conditions such as deadlock and termination
- Defines relationships among local process state, global system state, and points in a distributed computation

Discussion points:

- Scheme accurately captures state
- Algorithm introduces communication overheads
- Related to Vector clocks


PGC Photo/Hal Korber

Outline

- Motivation
- Distributed system model
- Consistent cuts
- Global state detection
- Stable state detection

Motivation

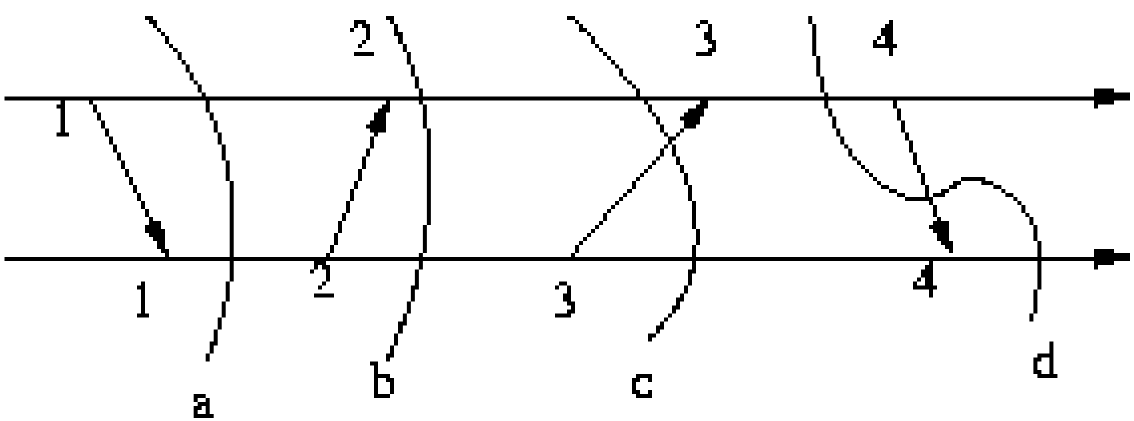
- Algorithms for determining global states are incorrect
  - Relationships among local process states, global system states, and points in a distributed computation are not well understood
- Attempt to define those relationships
- Correctly identify stable states in a distributed system

Distributed system model

- Processes
  - Defined in terms of states; states change on events
- Channels
  - State changes when messages are sent along the channel
- Events **e <p, s, s', M, c>** defined by
  - Process **P** in which event occurs
  - State **S** of P before event
  - State **S'** of P after event
  - Channel **C** altered by event
  - Message **M** sent/received along c

Consistent Cuts

- Snapshot of global state in a distributed system
- Defined as snapshots where no event after the cut happened before an event before the cut
- Forbids situations where effect is seen without its cause
- Useful for debugging, deadlock detection, termination detection, and global checkpoints



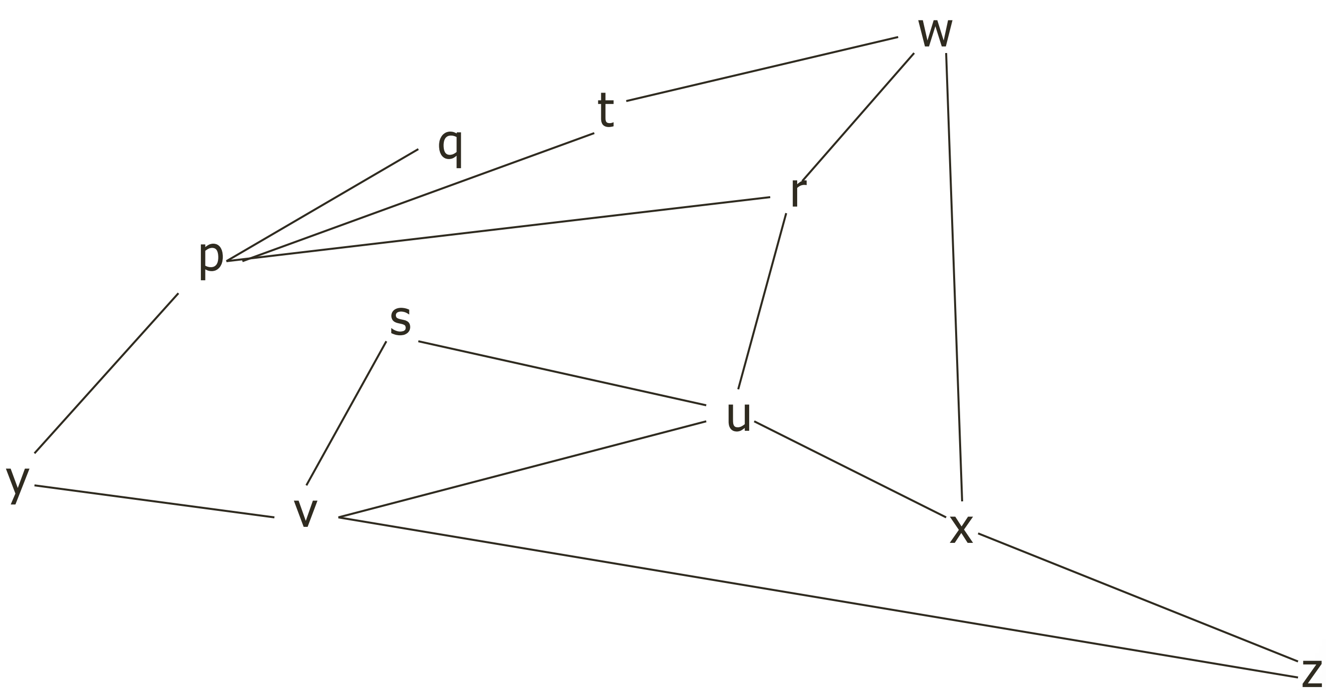
Global State Detection

- Superimposed on the computation
- Each process records its own state
- Processes of a channel cooperate on recording channel state
- Use a marker to synchronize global state recording

Global State Detection

- Process decides to take a snapshot
  - Save its state and sends marker through its outgoing channels
  - Save messages it receives on its in channels
- Process receives a marker for the first time
  - Save state and send marker on out channels
  - Save messages it receives on its in channels
- Algorithm terminates when:
  - Each node received markers through all its incoming channels

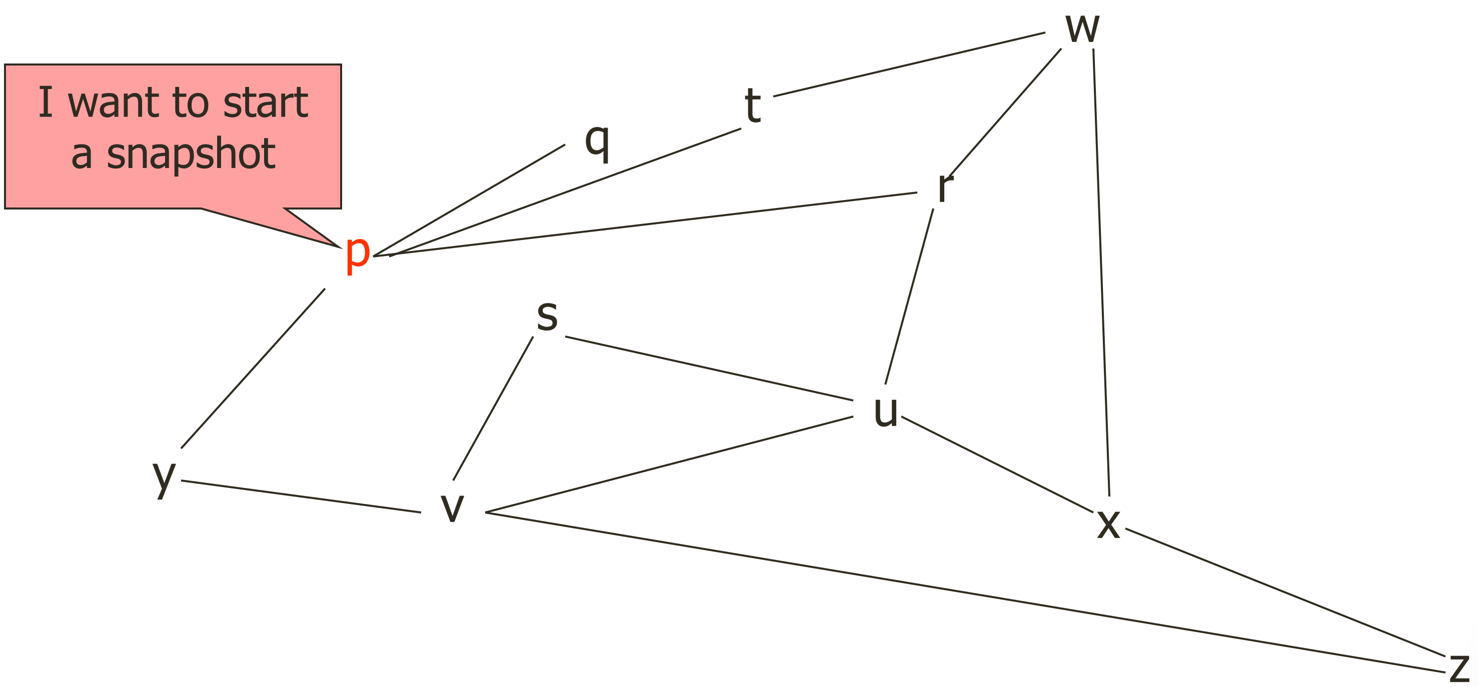
Global State Detection



A network

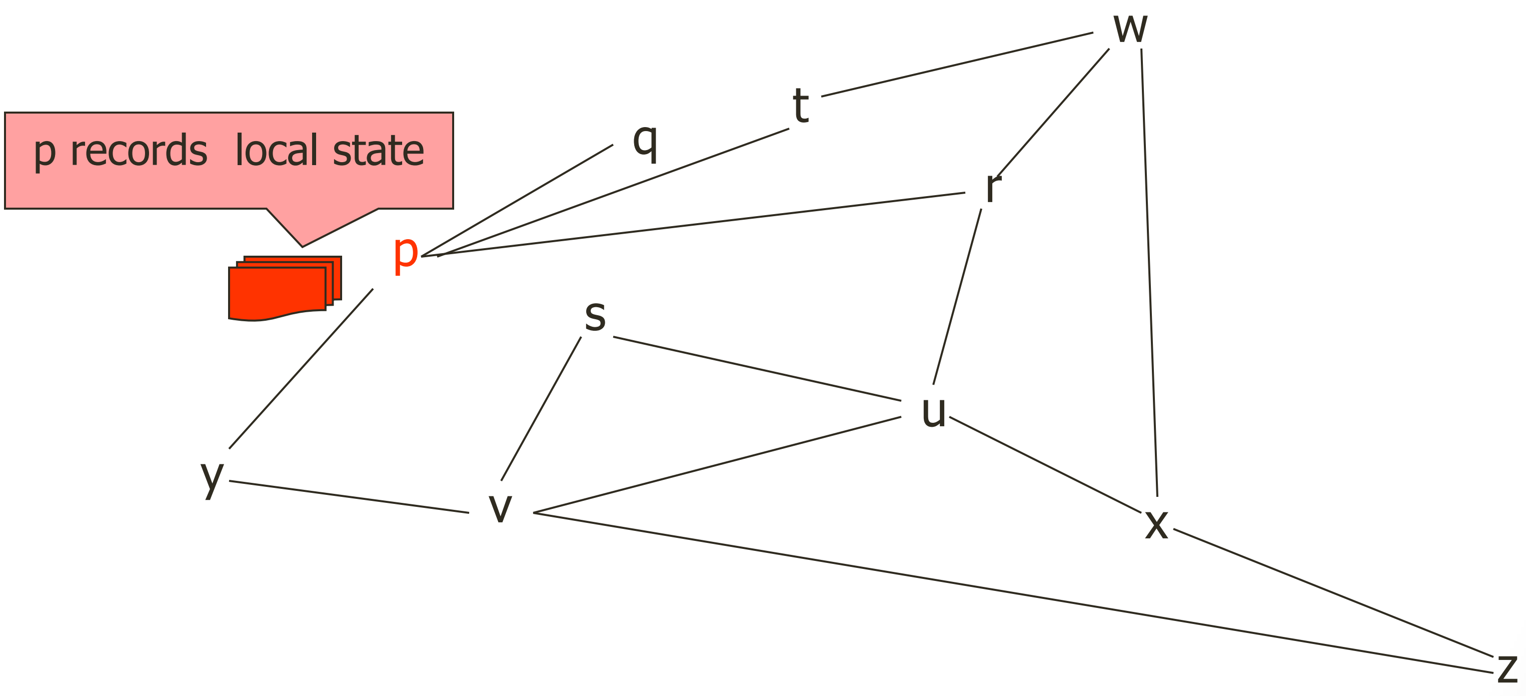
Source: Professor Hakim Weatherspoon's CS4410 F08 Lectures

Global State Detection



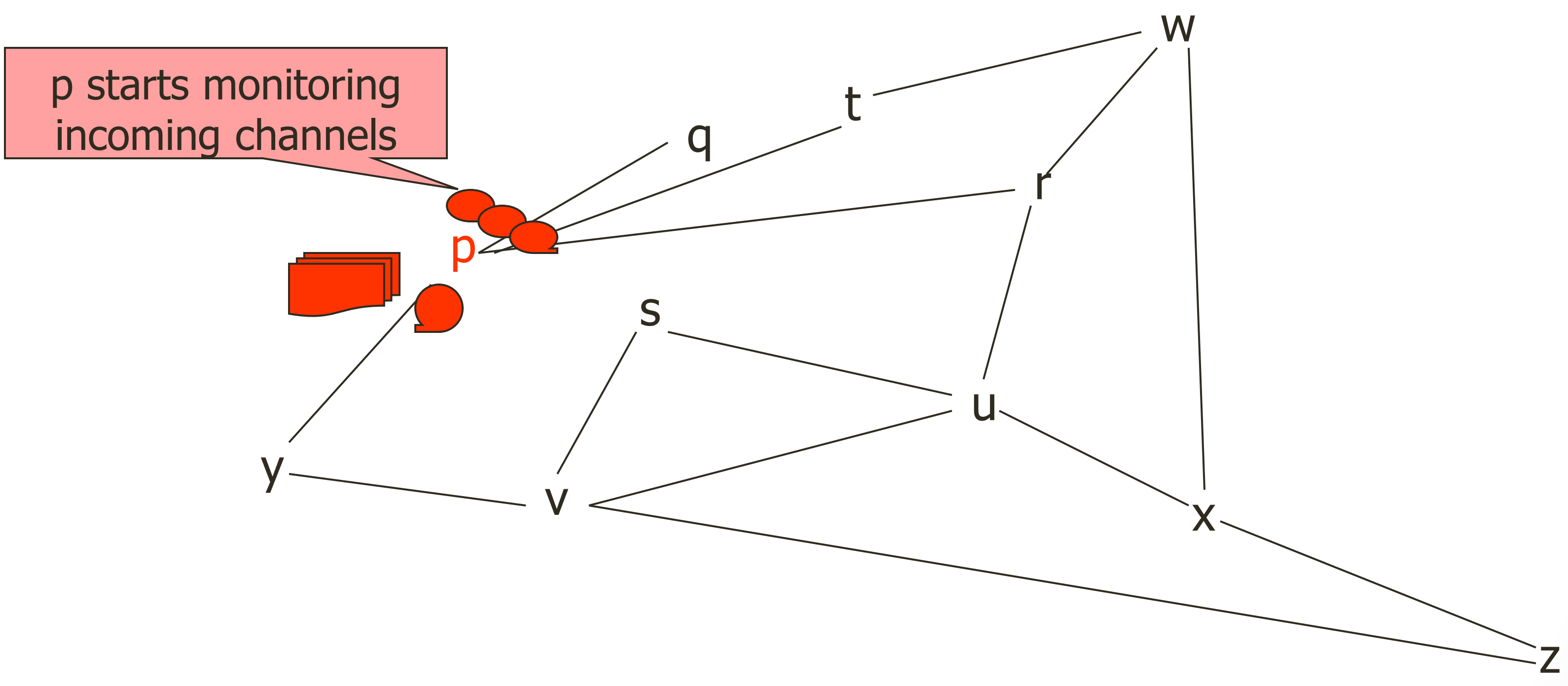
A network

Global State Detection



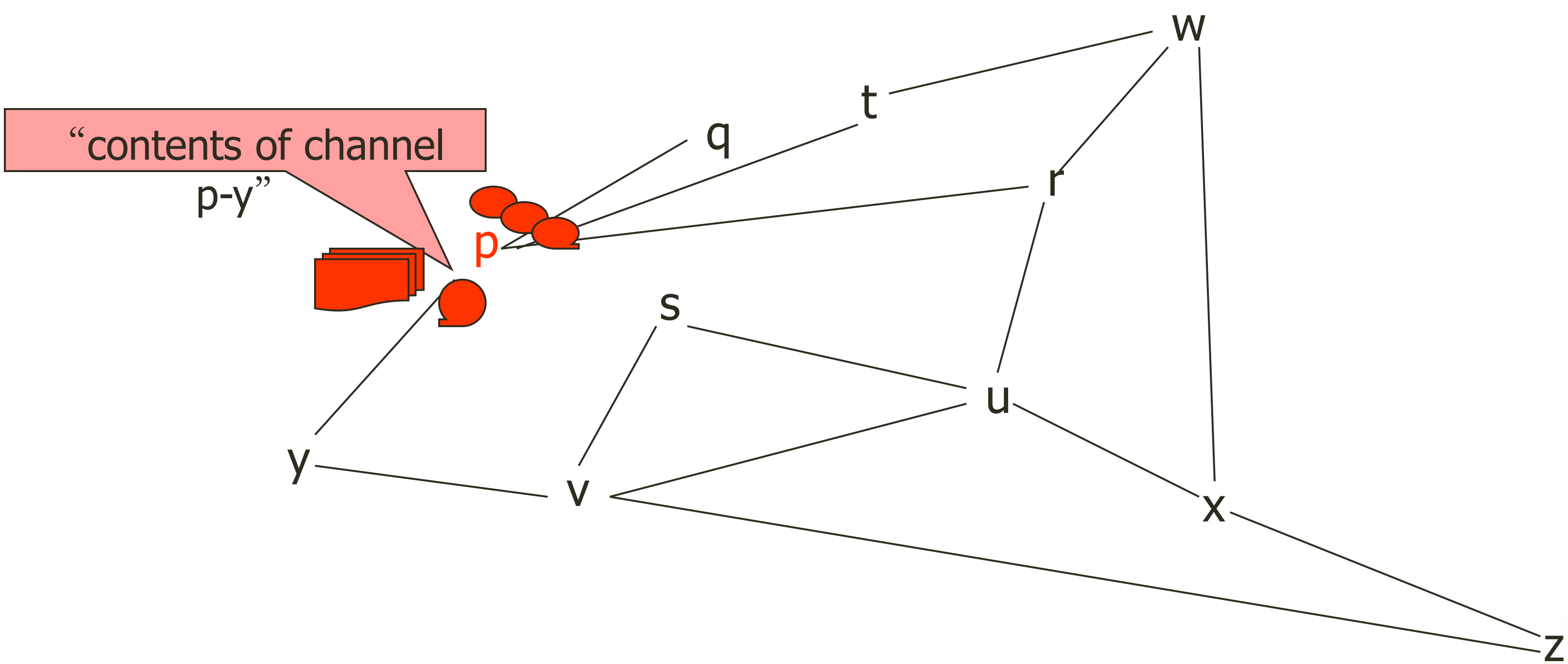
A network

Global State Detection



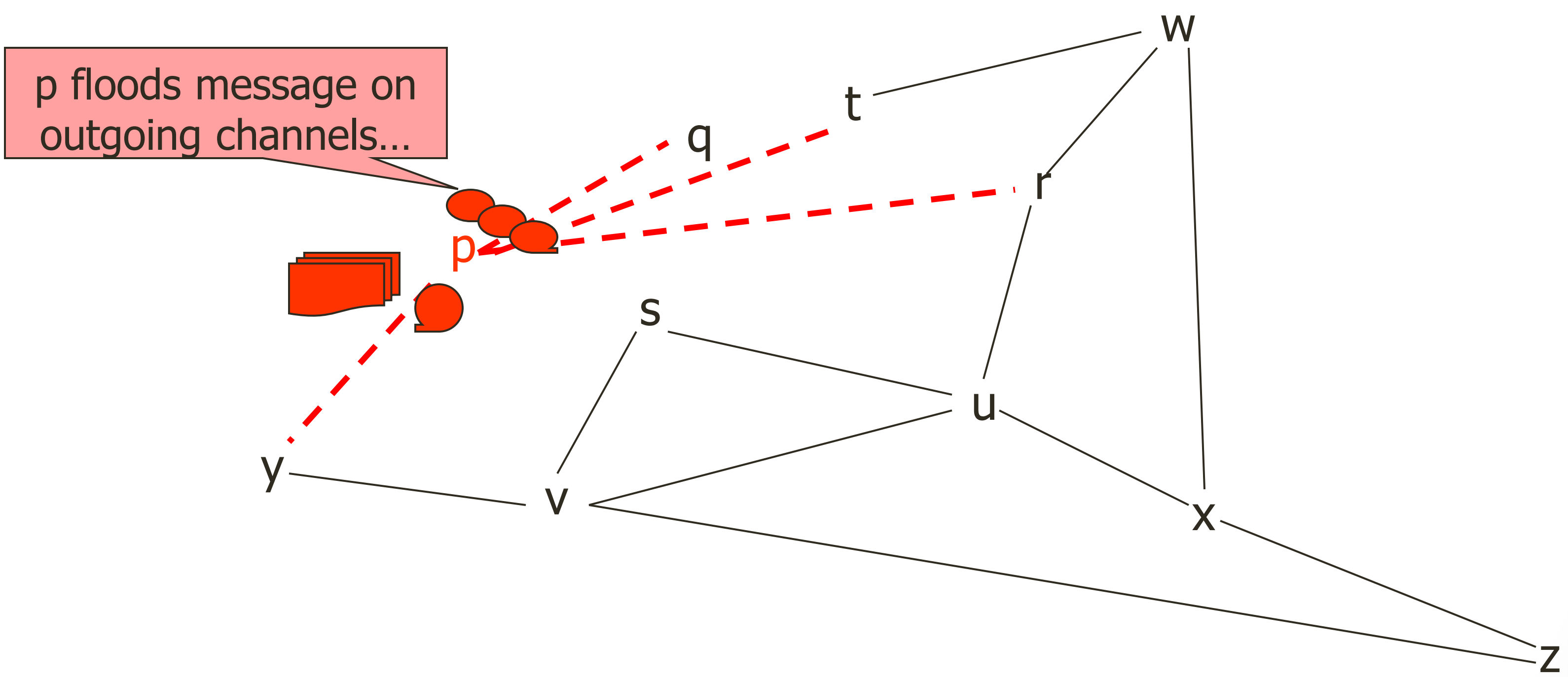
A network

Global State Detection



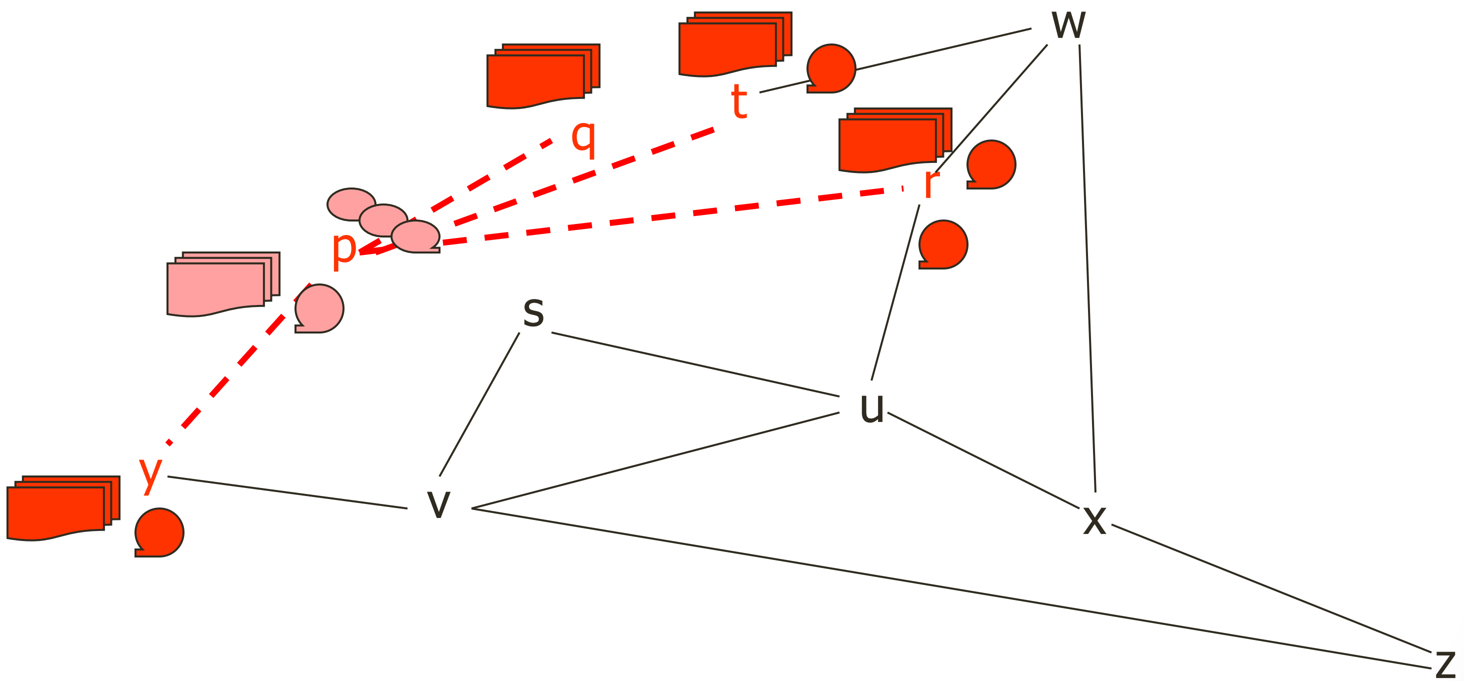
A network

Global State Detection



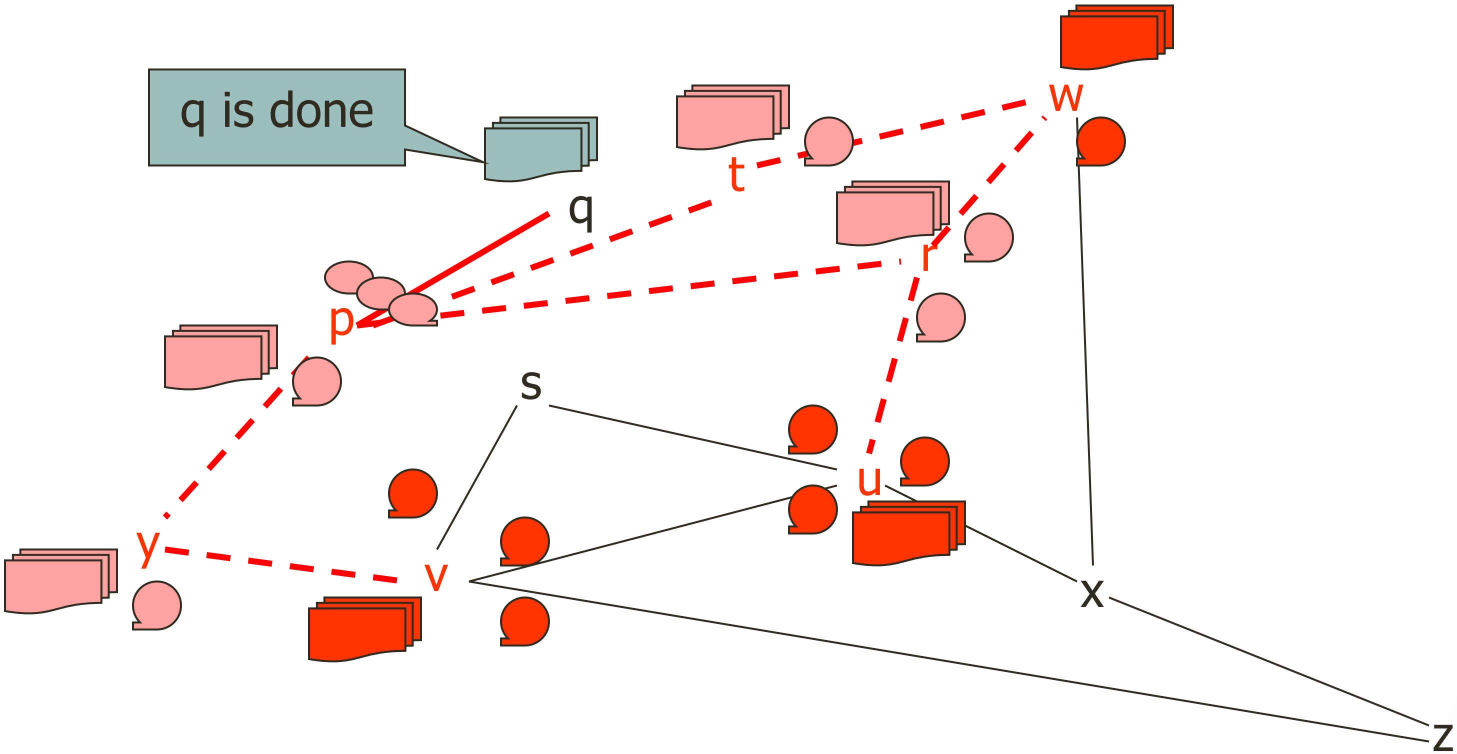
A network

Global State Detection



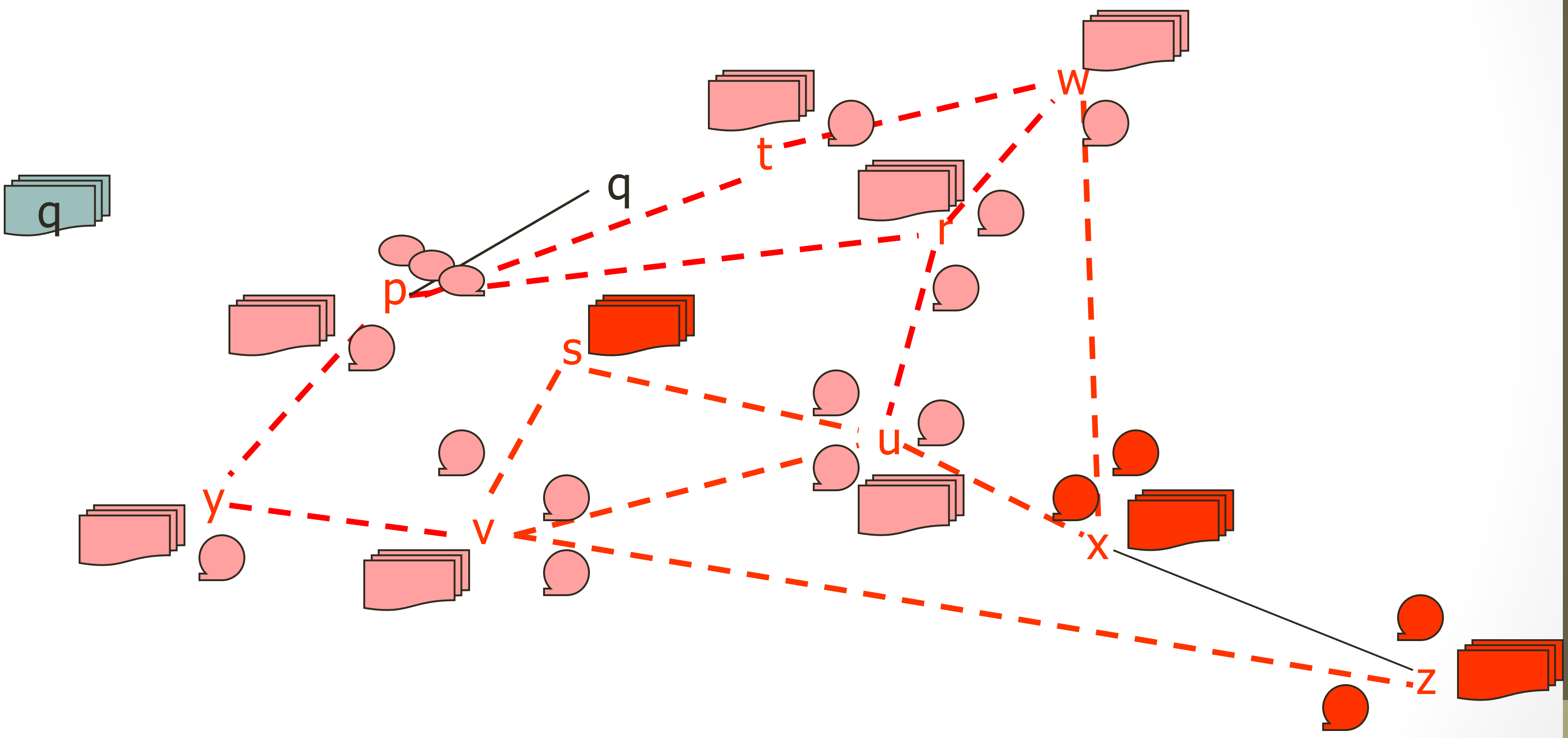
A network

Global State Detection



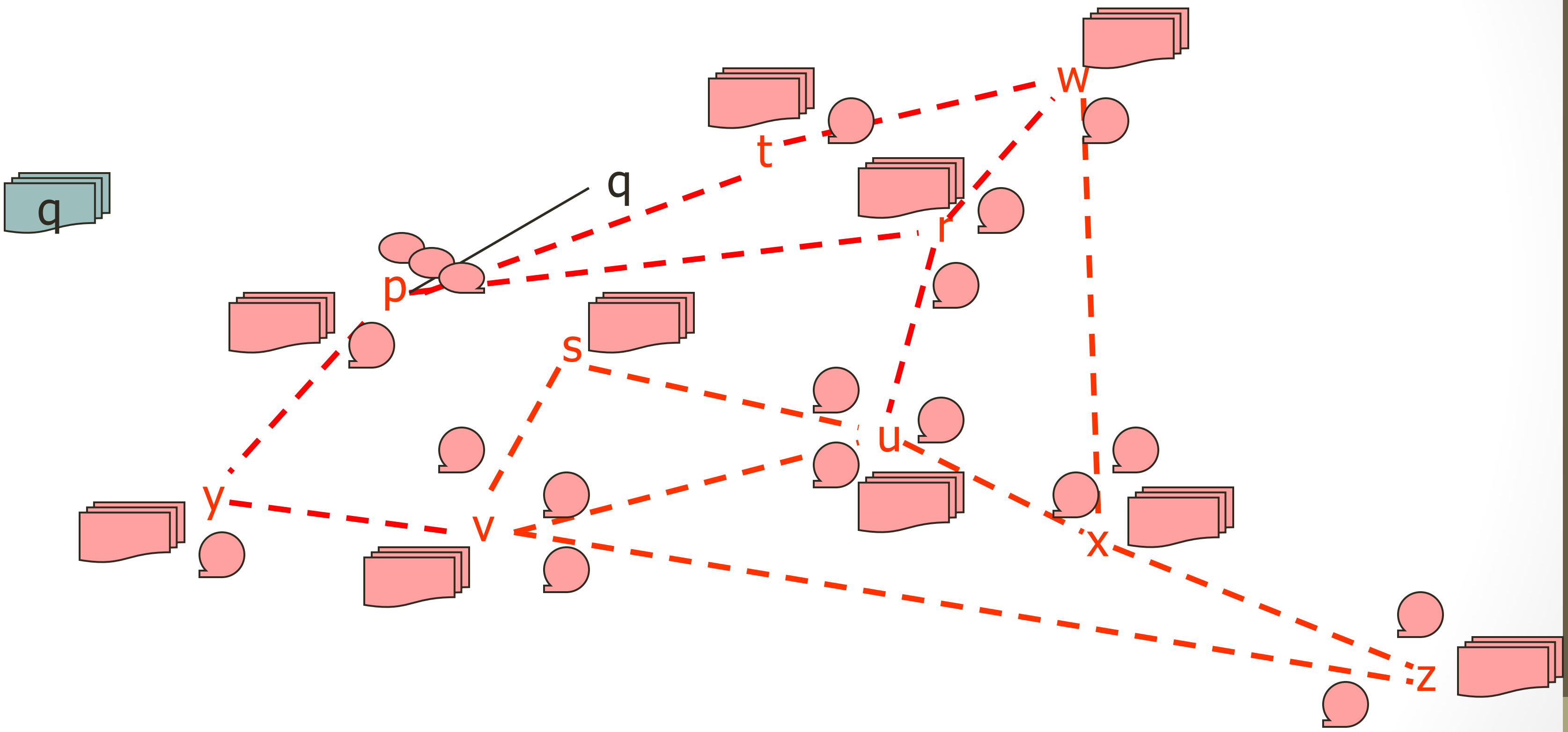
A network

Global State Detection



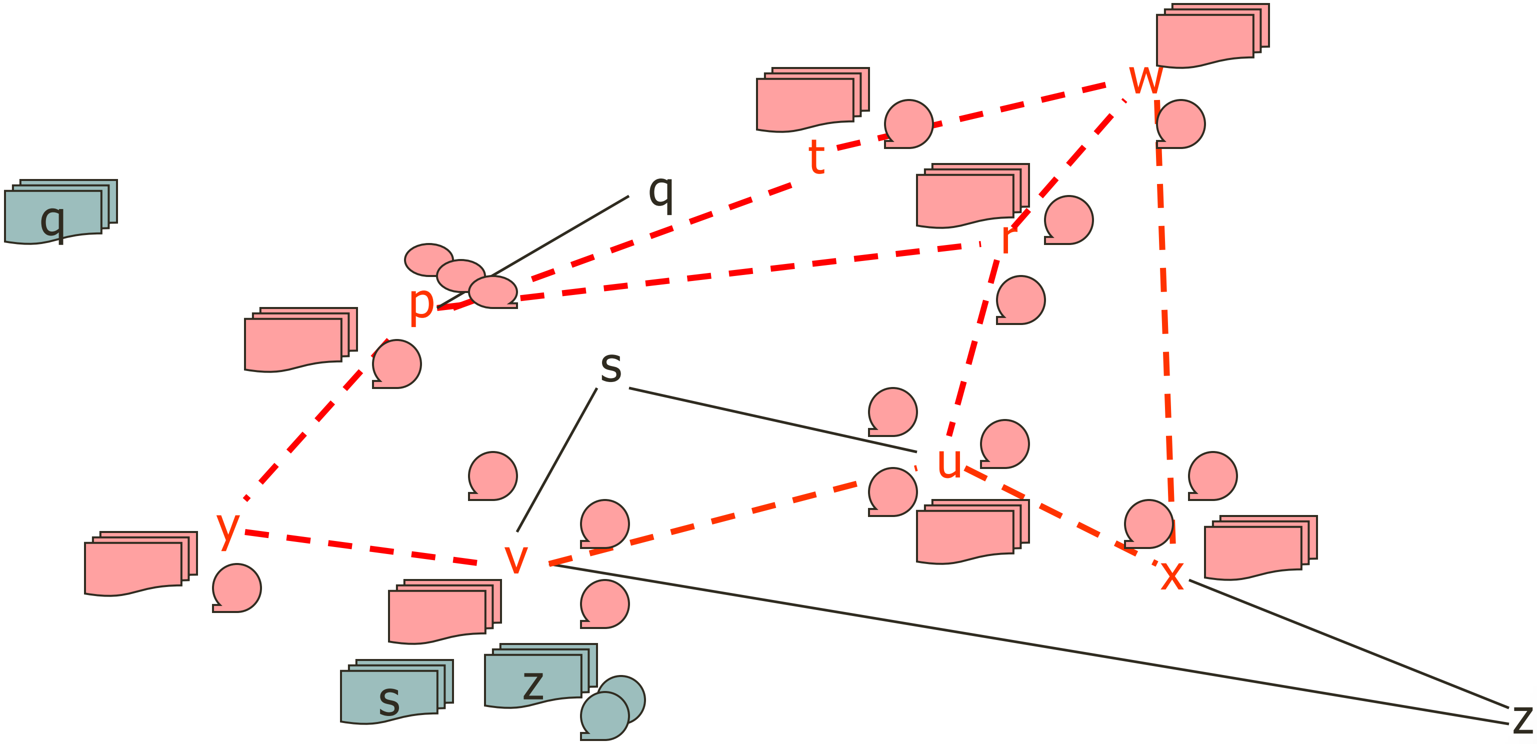
A network

Global State Detection



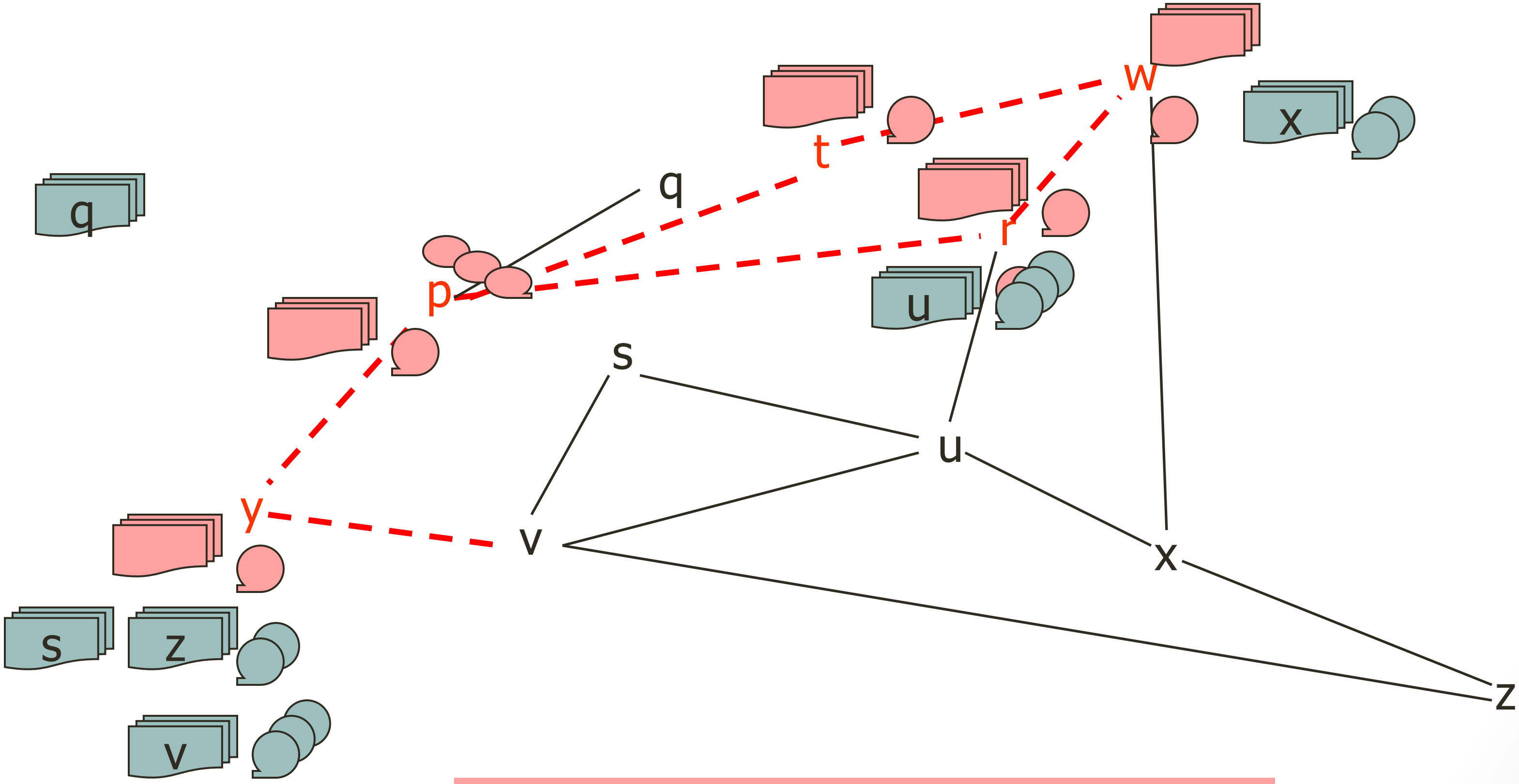
A network

Global State Detection



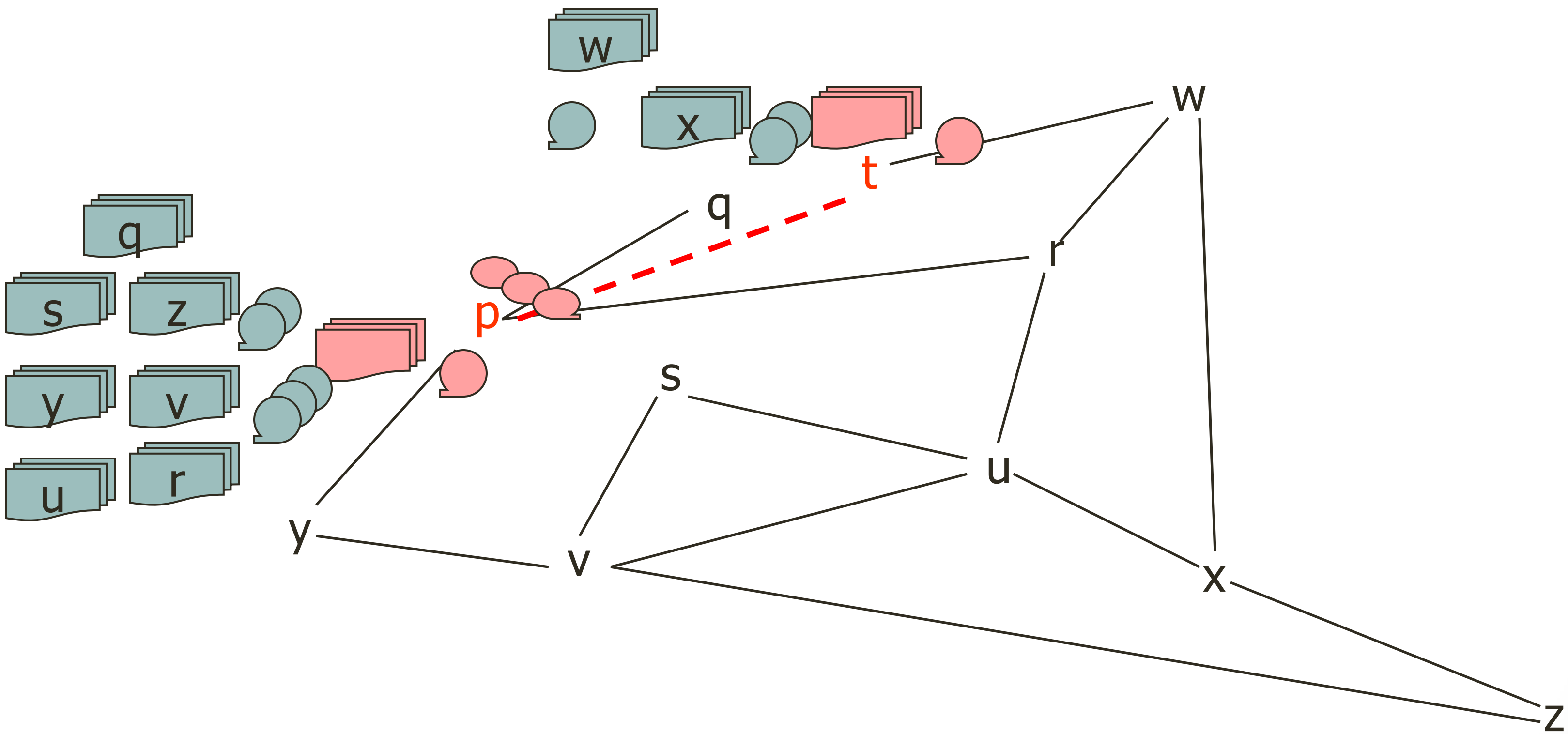
A network

Global State Detection



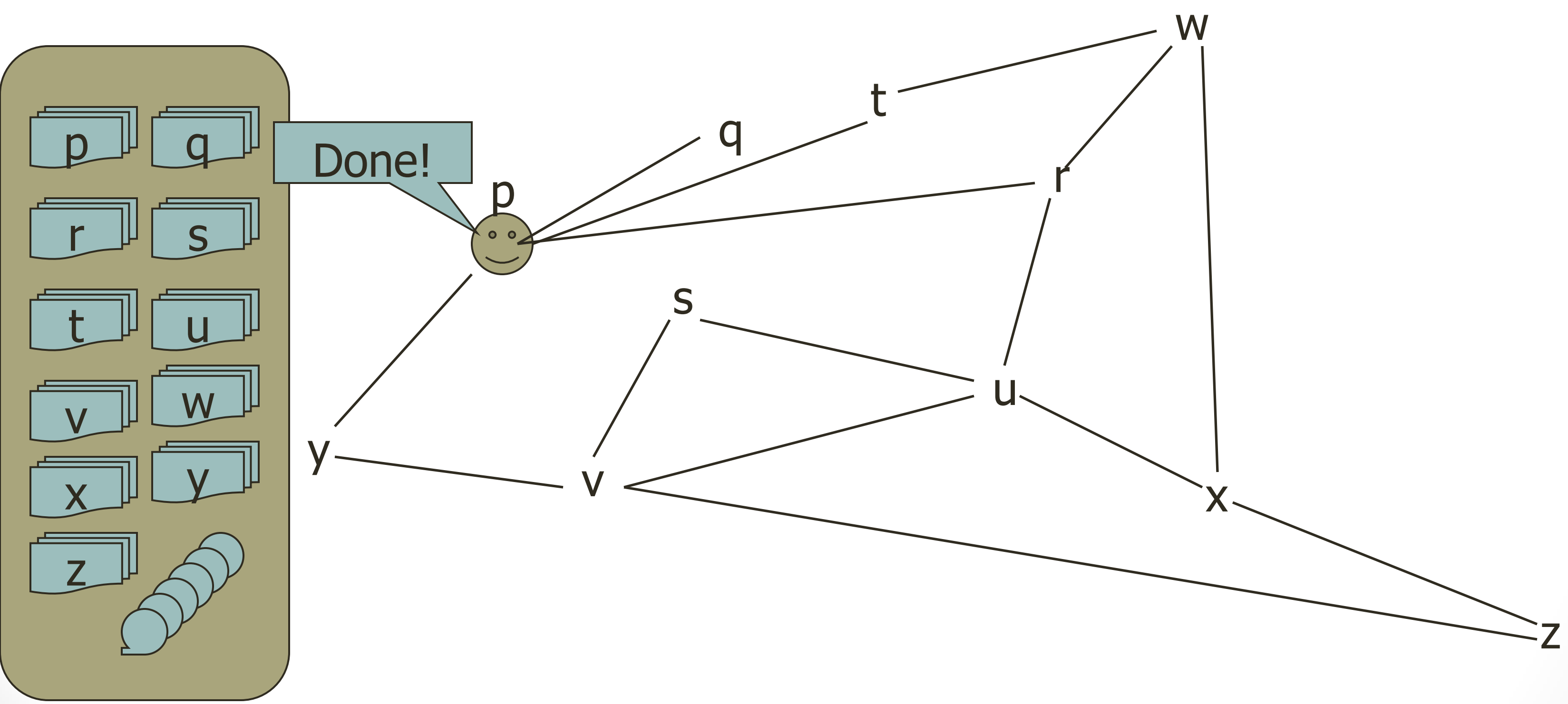
A network

Global State Detection



A network

Global State Detection



A snapshot of a network

Stable State Detection

- Input: A stable property function Y
- Output: A Boolean value definite:
  - ($Y(S_i)$ -> definite) and ($Y(S_\Phi)$ -> definite)
  - Implications of "definite"
    - definite == false: cannot say YES/NO stability
    - definite == true:stable property at termination
- Correctness
  - Initial state -> recorded state -> terminating state
  - for all j: $y(S_j) = y(S_j+1)$ – state is stable

Takeaways

- Temporal characteristics of distributed systems was poorly understood
- Lamport proposed logical clocks for ordering
- Chandy/Lamport proposed a distributed snapshot algorithm
  - Snapshot algorithm can be used to accurately detect stable events