Building a System Management Service

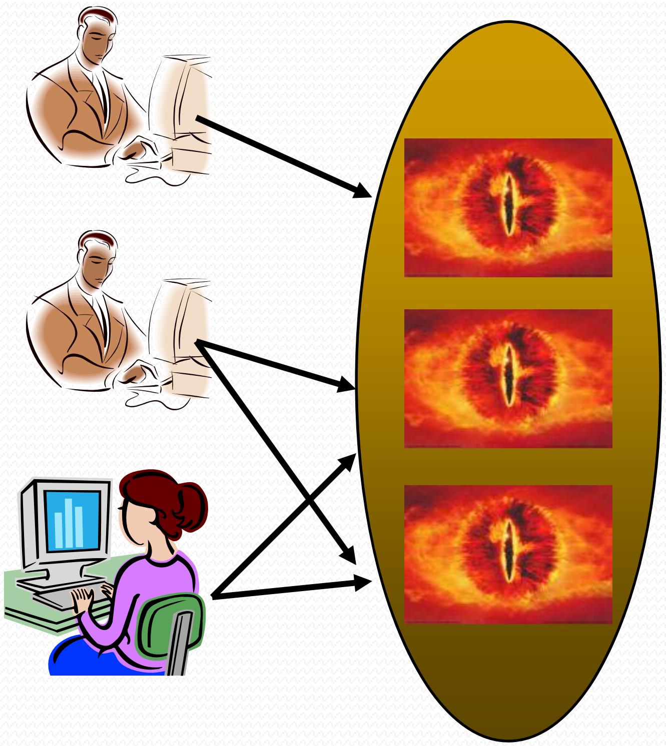
Ken Birman

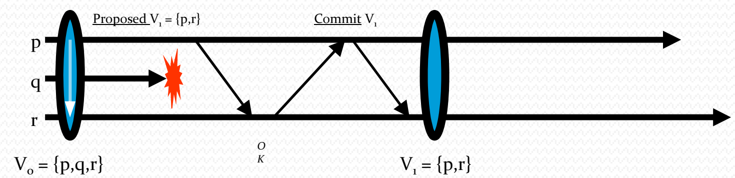
Cornell University. CS5410 Fall 2008.

Monday: Designed an Oracle

- We used a state machine protocol to maintain consensus on events
- Structured the resulting system as a tree in which each node is a group of replicas
- Results in a very general management service
  - One role of which is to manage membership when an application needs replicated data
  - Today continue to flesh out this idea of a group communication abstraction in support of replication

Turning the GMS into the Oracle

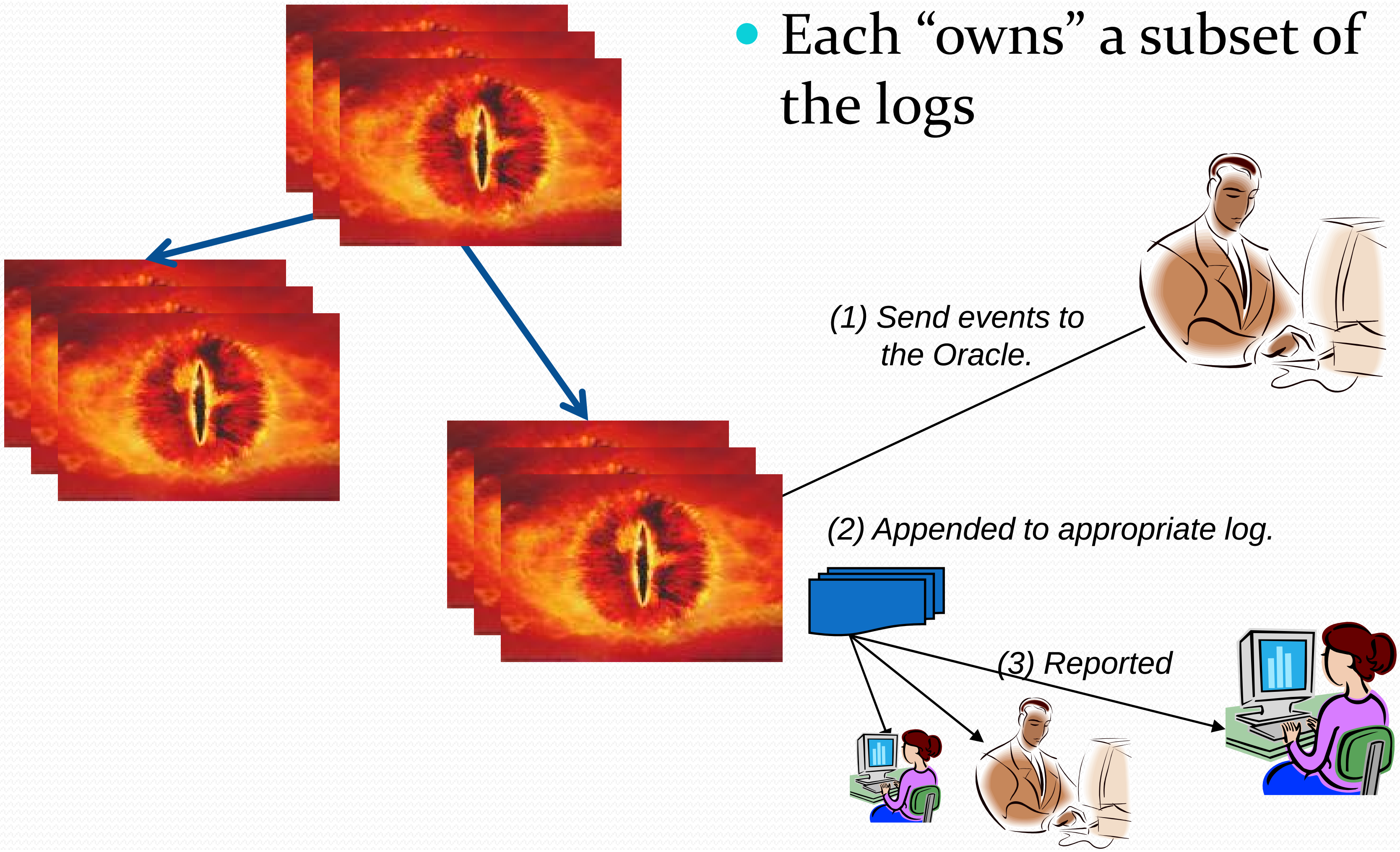




Here, three replicas cooperate to implement the GMS as a fault-tolerant state machine. Each client platform binds to some representative, then rebinds to a different replica if that one later crashes….

Tree of state machines

- Each "owns" a subset of the logs



Use scenario

- Application A wants to connect with B via "consistent TCP"
  - A and B register with the Oracle – each has an event channel of its own, like /status/biscuit.cs.cornell.edu/pid=12421
  - Each subscribes to the channel of the other (if connection breaks, just reconnect to some other Oracle member and ask it to resume where the old one left off)
  - They break the TCP connections if (and only if) the Oracle tells them to do so.

Use scenario

- For locking
  - Lock is “named” by a path
    - /x/y/z…
  - Send “lock request” and “unlock” messages
  - Everyone sees the same sequence of lock, unlock messages… so everyone knows who has the lock

- Garbage collection?
  - Truncate prefix after lock is granted

Use scenario

- For tracking group membership
  - Group is "named" by a path
    - /x/y/z…
  - Send "join request" and "leave" messages
  - Report failures as "forced leave"
  - Everyone sees the same sequence of join, leave messages… so everyone knows the group view

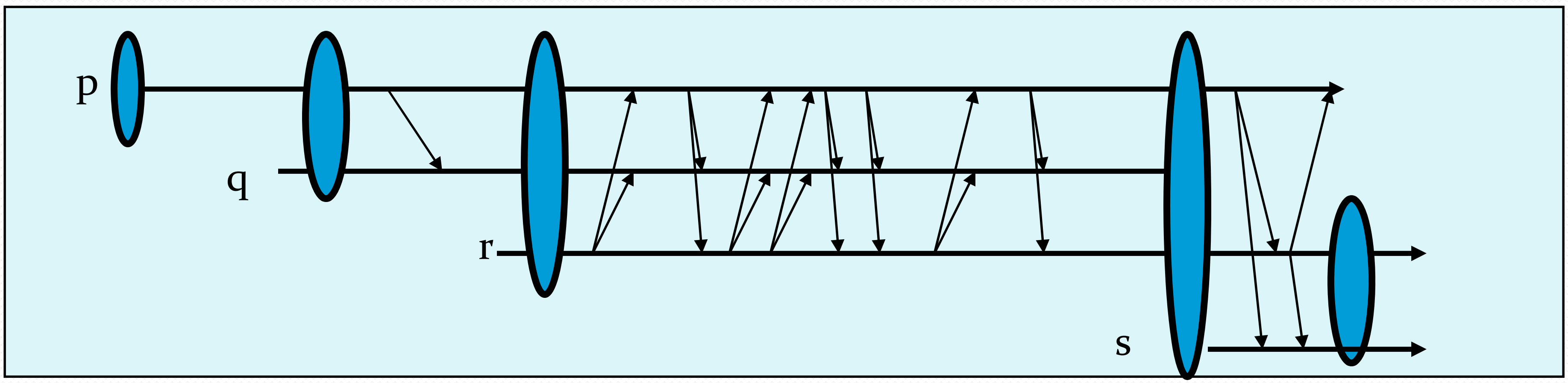
- Garbage collection?
  - Truncate old view-related events

A primitive "pub/sub" system

- The Oracle is very simple but quite powerful
  - Everyone sees what appears to be a single, highly available source of reliable "events"
  - XML strings can encode all sorts of event data
  - Library interfaces customize to offer various abstractions
- Too slow for high-rate events (although the Spread system works that way)
- But think of the Oracle as a bootstrapping tool that helps the groups implement their own direct, peer-to-peer protocols in a nicer world that if they didn't have it.

Building group multicast

- Any group can use the Oracle to track membership
- Enabling reliable multicast!

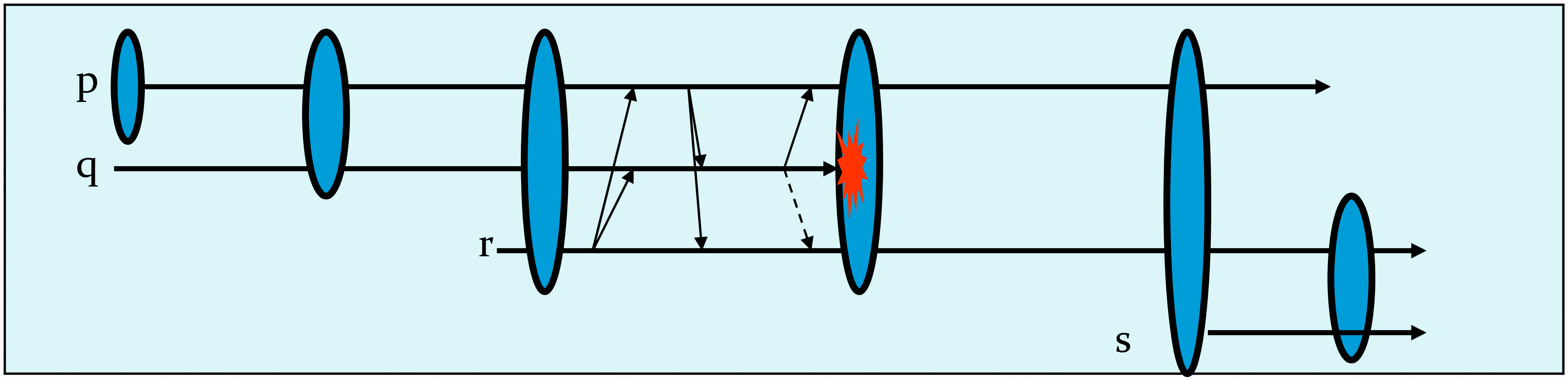


- Protocol: Unreliable multicast to current members. ACK/NAK to ensure that all of them receive it

Concerns if sender crashes

- Perhaps it sent some message and only one process has seen it
- We would prefer to ensure that
  - All receivers, in "current view"
  - Receive any messages that <u>any</u> receiver receives (unless the sender and all receivers crash, erasing evidence…)

An interrupted multicast



- A message from q to r was “dropped”
- Since q has crashed, it won’t be resent

Flush protocol

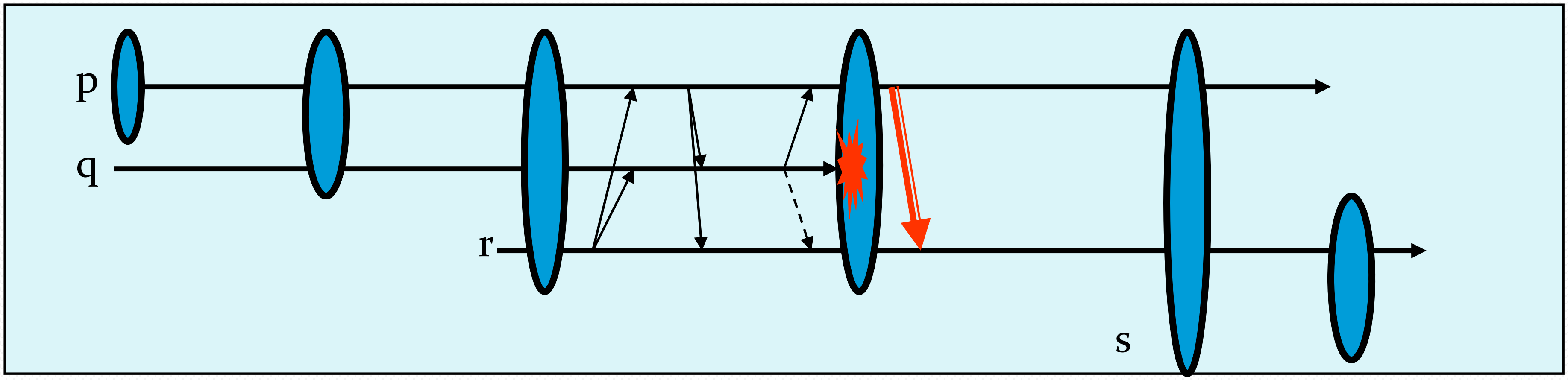
- We say that a message is *unstable* if some receiver has it but (perhaps) others don't
  - For example, q's message is unstable at process r
- If q fails we want to "flush" unstable messages out of the system

How to do this?

- Easy solution: all-to-all echo
  - When a new view is reported
  - All processes echo any unstable messages on all channels on which they haven't received a copy of those messages
- A flurry of $O(n^2)$ messages

- *Note: must do this for <u>all</u> messages, not just those from the failed process. This is because more failures could happen in future*

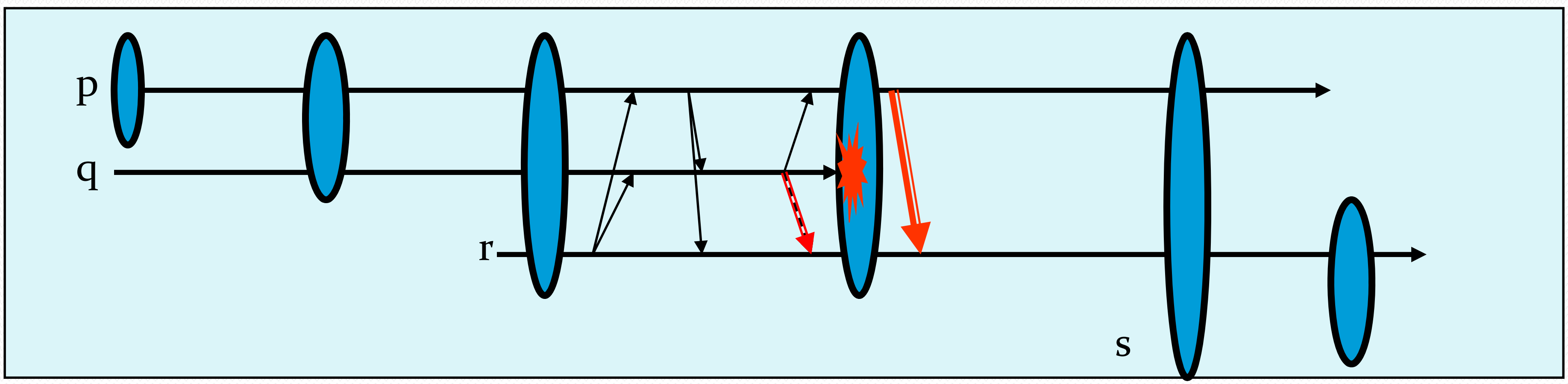
An interrupted multicast



- p had an unstable message, so it echoed it when it saw the new view

Event ordering

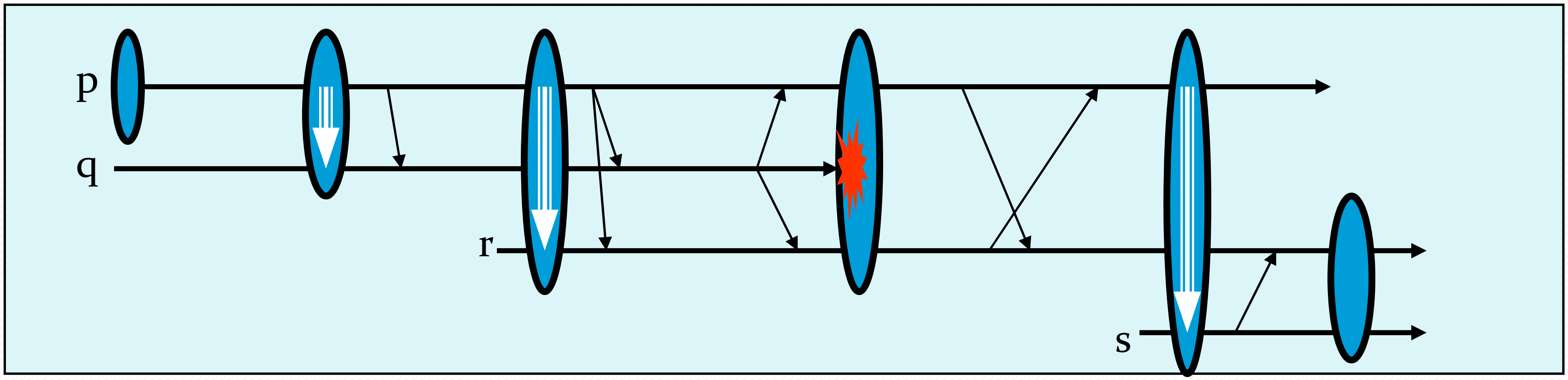
- We should *first* deliver the multicasts to the application layer and *then* report the new view
- This way all replicas see the same messages delivered "in" the same view
  - Some call this "view synchrony"



State transfer

- At the instant the new view is reported, a process already in the group makes a checkpoint
- Sends point-to-point to new member(s)
- It (they) initialize from the checkpoint

State transfer and reliable multicast



- After re-ordering, it looks like each multicast is reliably delivered <u>in the same view</u> at each receiver
- Note: if sender *and all receivers* fails, unstable message can be "erased" even after delivery to an application
    - This is a price we pay to gain higher speed

State transfer

- New view initiated, it adds a process
- We run the flush protocol, but as it ends…
- … some existing process creates a checkpoint of group
  - Only state specific to the group, not ALL of its state
  - Keep in mind that one application might be in many groups at the same time, each with its own state
- Transfer this checkpoint to joining member
  - It loads it to initialize the state of its instance of the group – that object. One state transfer per group.

Ordering: The missing element

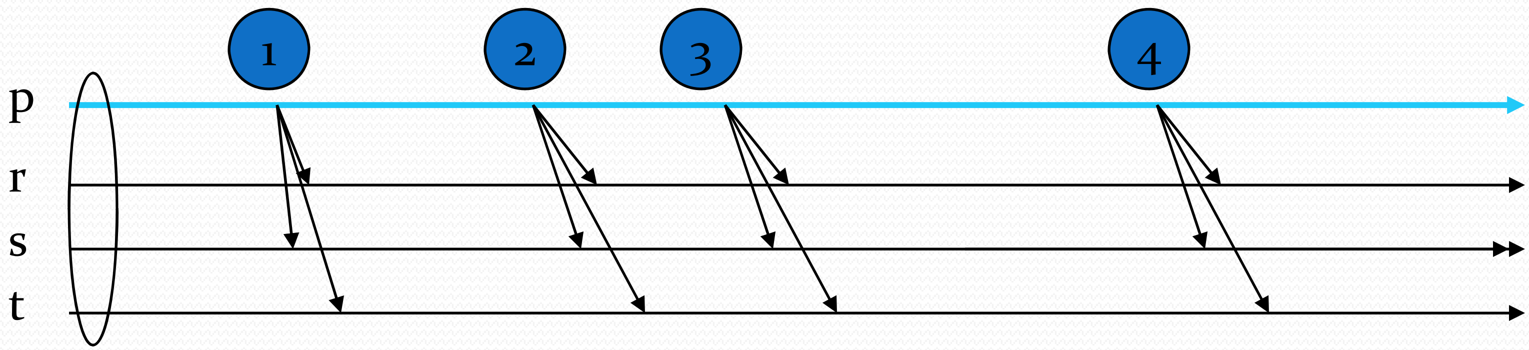
- Our fault-tolerant protocol was
  - FIFO ordered: messages *from a single sender* are delivered in the order they were sent, even if someone crashes
  - View synchronous: everyone receives a given message in the same group view
- This is the protocol we called **fbcast**

Other options

- **cbcast**: If cbcast(a)→cbcast(b), deliver a before b at common destinations
- **abcast:** Even if a and b are concurrent, deliver in some agreed order at common destinations
- **gbcast:** Deliver this message like a new group view: agreed order w.r.t. multicasts of all other flavors

Single updater

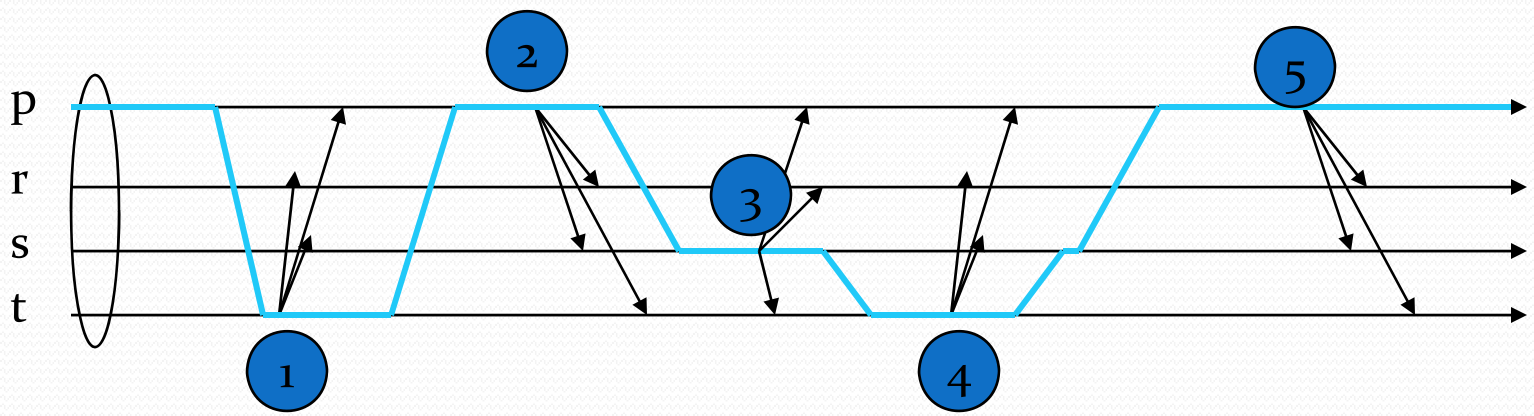
- If p is the only update source, the need is a bit like the TCP "fifo" ordering



- **fbcast** is a good choice for this case

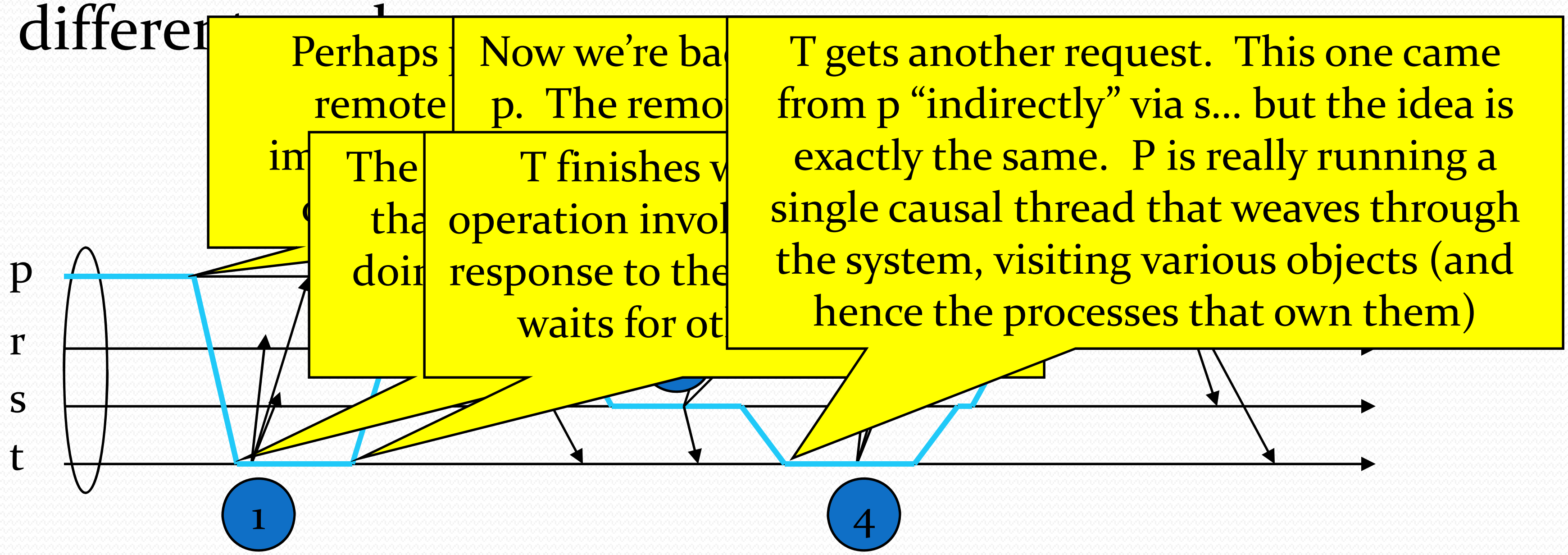
Causally ordered updates

- Events occur on a "causal thread" but multicasts have different senders



Causally ordered updates

- Events occur on a “causal thread” but multicasts have differen

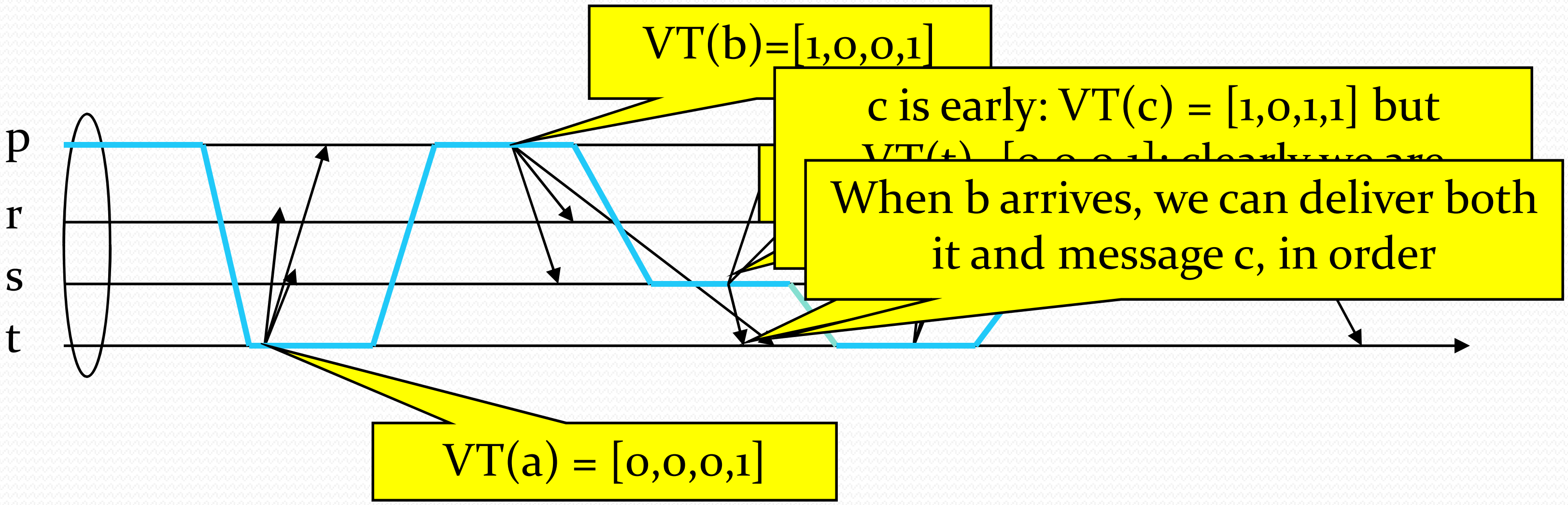


How to implement it?

- Within a single group, the easiest option is to include a vector timestamp in the header of the message
  - Array of counters, one per group member
  - Increment your personal counter when sending
  - iSend these “labeled” messages with fbcast
- Delay a received message if a causally prior message hasn’t been seen yet

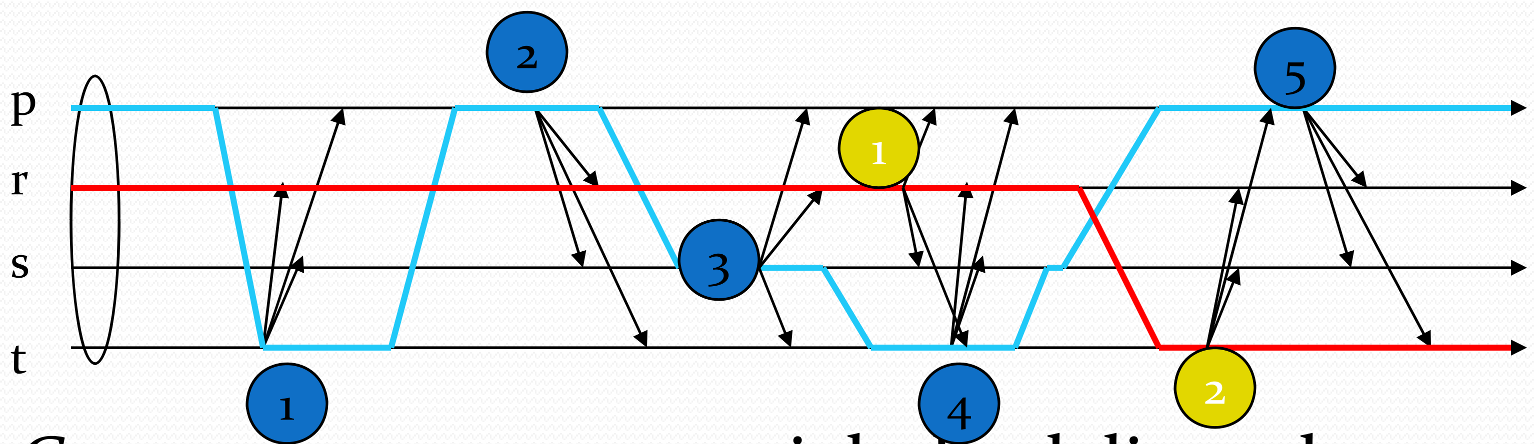
Causally ordered updates

- Example: messages from p and s arrive out of order at t



Causally ordered updates

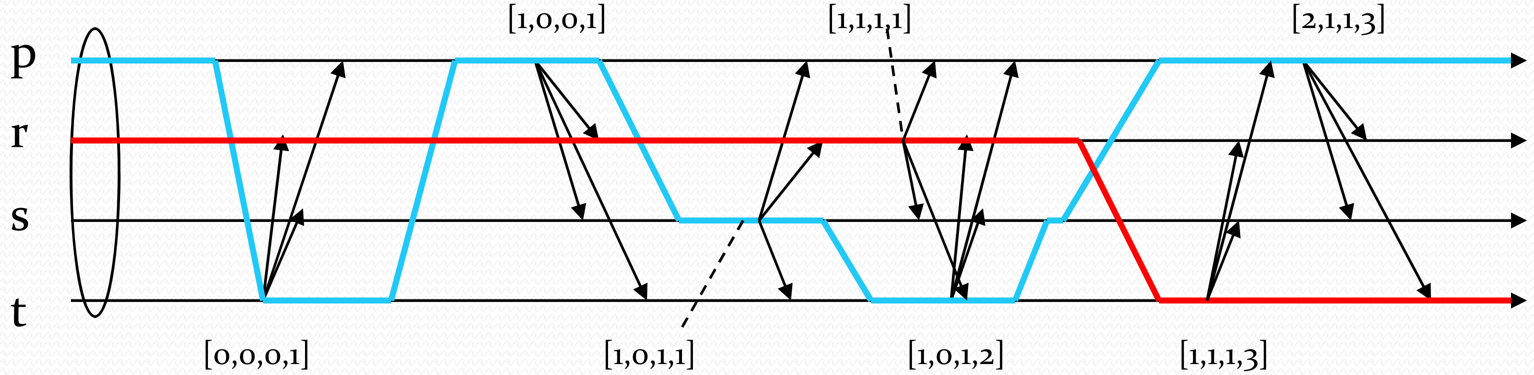
- This works even with multiple causal threads.



- Concurrent messages might be delivered to different receivers in different orders
  - Example: green 4 and red 1 are concurrent

Causally ordered updates

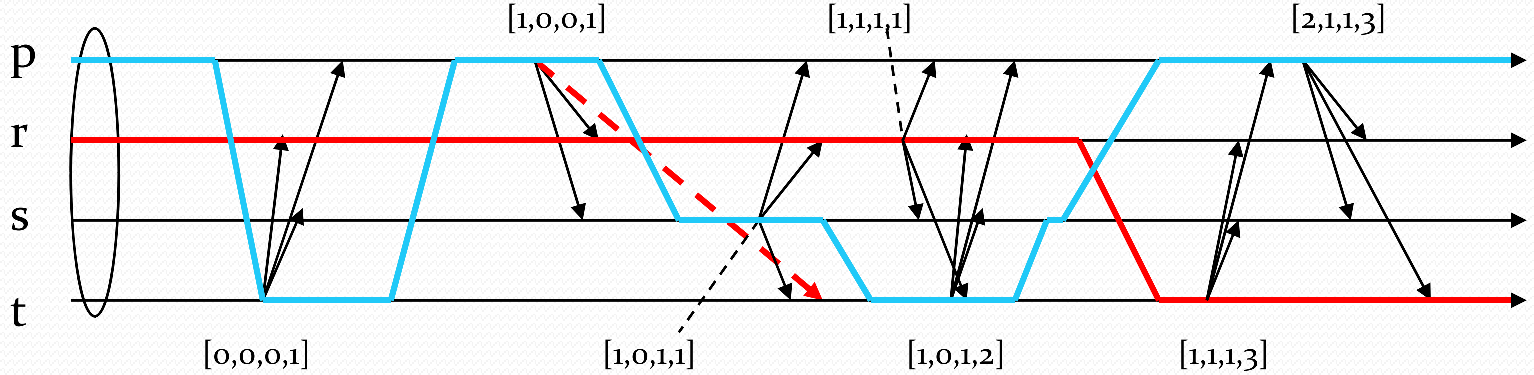
- Sorting based on vector timestamp



- In this run, everything can be delivered immediately on arrival

Causally ordered updates

- Suppose p's message [1,0,0,1] is "delayed"



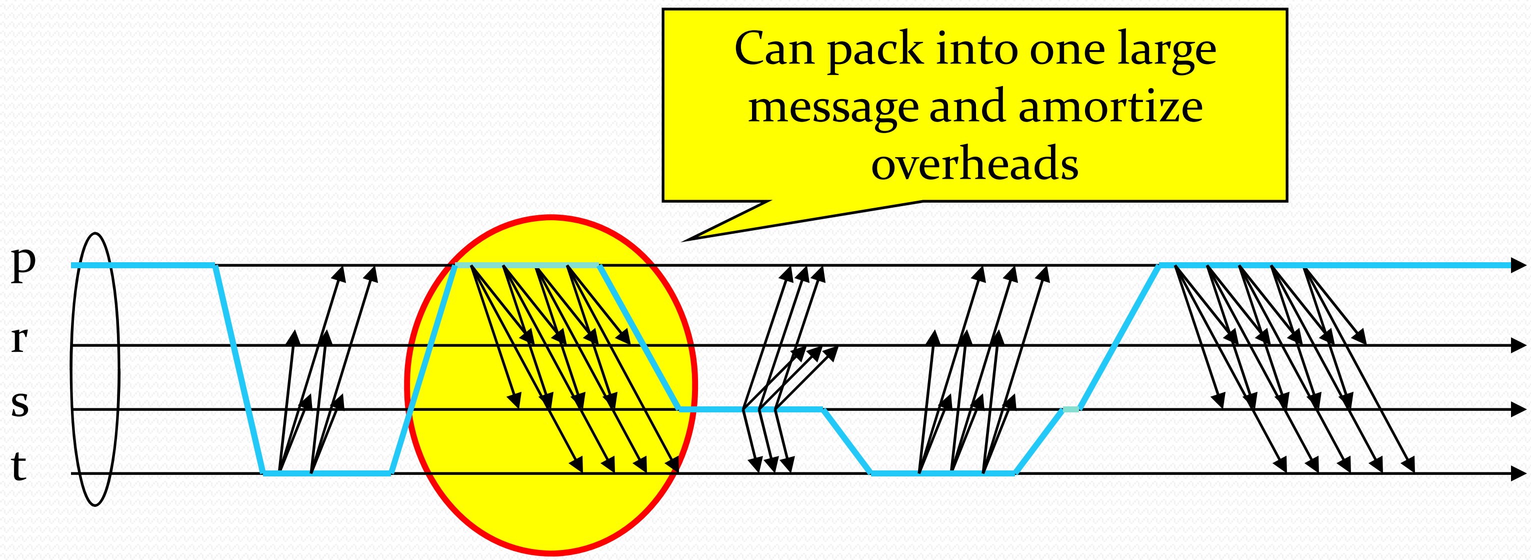
- When t receives message [1,0,1,1], t can "see" that one message from p is late and can delay deliver of s's message until p's prior message arrives!

Other uses for cbcast?

- The protocol is very helpful in systems that use locking for synchronization
  - Gaining a lock gives some process mutual exclusion
  - Then it can send updates to the locked variable or replicated data
- Cbcast will maintain the update order

Causally ordered updates

- A bursty application



Other forms of ordering

- Abcast (total or "atomic" ordering)
  - Basically, our locking protocol solved this problem
  - Can also do it with fbcast by having a token-holder send out ordering to use

- Gbcast
  - Provides applications with access to the same protocol used when extending the group view
  - Basically, identical to "Paxos" with a leader

Algorithms that use multicast

- Locked access to shared data
  - Multicast updates, read *any local copy*
  - This is very efficient… 100k updates/second not unusual
- Parallel search
- Fault-tolerance (primary/backup, coordinator-cohort)
- Publish/subscribe
- Shared “work to do” tuples
- Secure replicated keys
- Coordinated actions that require a leader

Modern high visibility examples

- Google's Chubby service (uses Paxos == gbcast)
- Yahoo! Zookeeper
- Microsoft cluster management technology for Windows Enterprise clusters
- IBM DCS platform and Websphere

- Basically: stuff like this is all around us, although often hidden inside some other kind of system

Summary

- Last week we looked at two notions of time
  - Logical time is more relevant here
  - Notice the similarity between delivery of an ordered multicast and computing something on a consistent cut
- We're starting to think of "consistency" and "replication" in terms of events that occur along time-ordered event histories
  - The GMS (Oracle) tracks "management" events
  - The group communication system supports much higher data rate replication