Building a System Management Service

Ken Birman

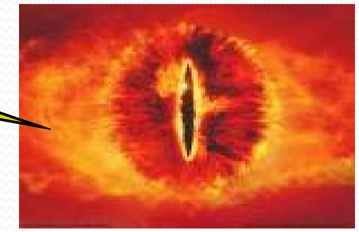
Cornell University. CS5410 Fall 2008.

Last week looked at time

- In effect, we asked "can we build a time service for a data center"?
  - Reached two conclusions
  - One focused on event ordering
  - The other was a true synchronized clock
- This week, we'll use some of the ideas from the time service to build a powerful system management service

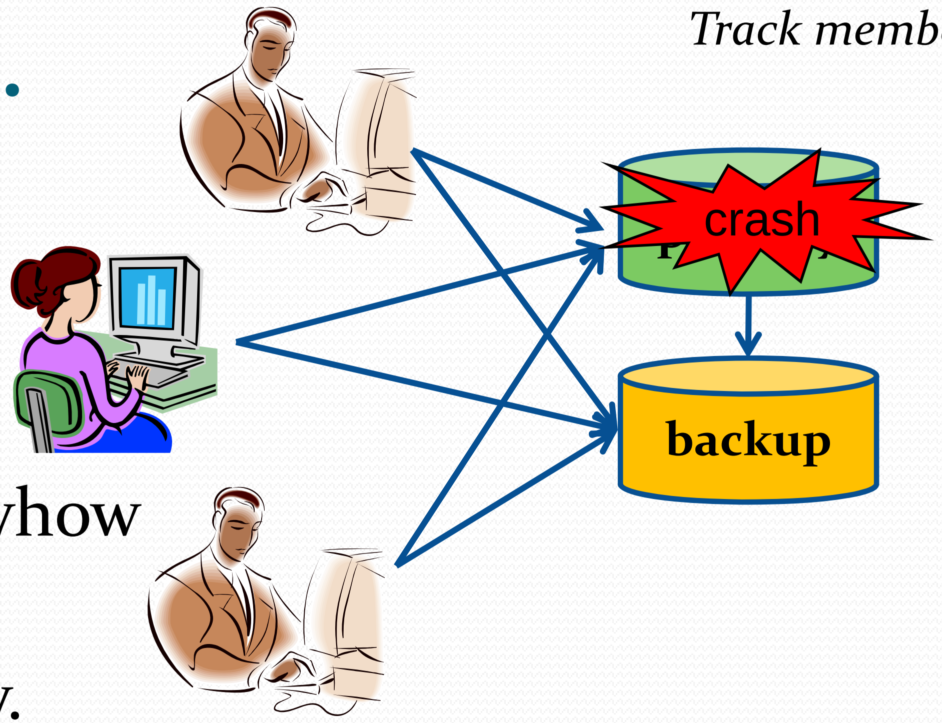
Oracle

Hear and obey. The primary is down. I have spoken!!!



Track membership

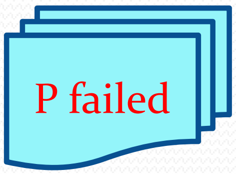
An all-seeing eye.



- Clients obey it
- If the oracle errs we "do as it says" anyhow
- This eliminated our fear of inconsistency.

Using the Oracle to manage a system

- For many purposes, Oracle can “publish decrees”
  - “Failure” and “Recovery” don’t need to be the only cases
- For example
  - “Engines at warp-factor two!”
  - “Reject non-priority requests”
  - “Map biscuit.cs.cornell.edu to 128.57.43.1241”
- Imagine this as an append-only log



Using the Oracle to manage a system

- If we give the records "names" (like file paths) we can treat the log as a set of logs
  - /process-status/biscuit.cs.cornell.edu/pid12345
  - /parameters/peoplesoft/run-slow=true
  - /locks/printqueue

- Thus one log can "look" like many logs
  - Clients append to logs
  - And they also "subscribe" to see reports as changes occur

Many roles for Oracles

- Track membership of a complex system
  - Which applications are up? Which are down?
  - Where are service instances running? (“GMS” function)
  - Use it as “input” for group applications, TCP failure sensing, load-balancing, etc.
- Lock management
- Parameter and status tracking
- Assignment of roles, keys
- DNS functionality

Scalability

- Clearly, not everything can run through one server
  - It won't be fast enough
- Solutions?
  - Only use the Oracle "when necessary" (will see more on this later)
  - Spread the role over multiple servers
    - One Oracle "node" could be handled by, say, three servers
    - And we could also structure the nodes as a hierarchy, with different parts of our log owned by different nodes
  - Requires "consensus" on log append operations

Consensus problem

- A classic (and well understood) distributed computing problem, arises in a few variant forms (agreement, atomic broadcast, leader election, locking)

- Core question:
  - A set of processes have inputs $v_i \in \{0,1\}$
  - Protocol is started (by some sort of trigger)
  - Objective: all *decide v*, for some v in the input set
  - Example solution: “vote” and take the majority value

Consensus with failures

- The so-called FLP (Fischer, Lynch and Patterson) result proves that any consensus protocol capable of tolerating even a single failure must have non-terminating runs (in which no decision is reached)
- Proof is for an asynchronous execution; flavor similar to that of the pumping lemma in language theory
- Caveat: the run in question is of *probability zero*

Aside: FLP Proof

- The actual proof isn't particularly intuitive
  - They show that any fault-tolerant consensus protocol has infinite runs that consist of purely bivalent states

- The intuition is that delayed messages can force a consensus protocol to "reconfigure"
  - The implicit issue is that consensus requires a unique leader to reaches the decision on behalf of the system.
  - FLP forces repeated transient message delays
  - These isolate the leader, forcing selection of a new leader, and thus delaying the decision indefinitely

Aside: "Impossibility"

- A perhaps-surprising insight is that for theory community, "impossible" doesn't mean "can't be done"
  - In normal language, an impossible thing can *never* be done. It is impossible for a person to fly (except on TV)
  - In the formal definitions used for FLP, impossible means *can't always be done.* If there is even one run in which decisions aren't reached, it is "impossible" to decide.
  - In fact, as a practical matter, consensus can *always be reached* as long as a majority of our system is operational

Consensus is impossible. But why do we care?

- The core issue is that so many problems are equivalent to consensus
  - Basically, any consistent behavior
- FLP makes it hard to be rigorous about correctness
  - We can prove partial but not total correctness
  - For the theory community, this is frustrating – it is "impossible" to solve consensus or equivalent problems
  - At best we talk about progress in models with Oracles

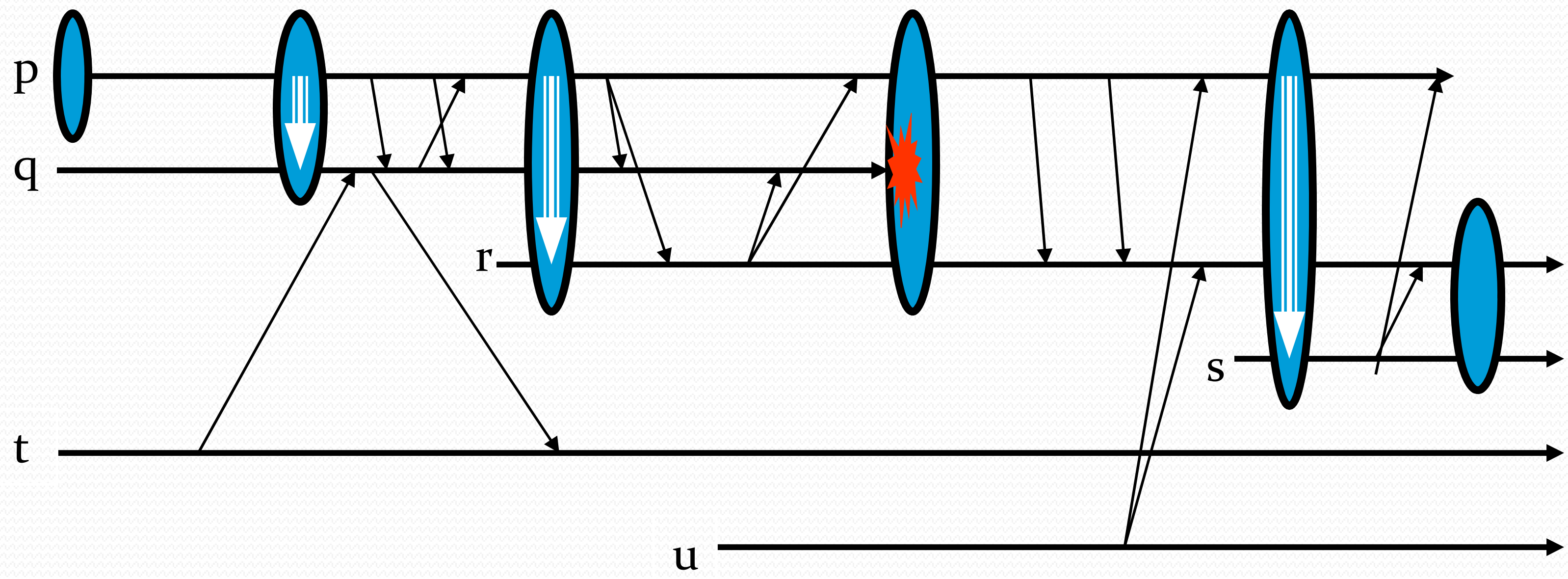
Consensus-like behavior

- We'll require that our log behave in a manner indistinguishable from a non-replicated, non-faulty single instance running on some accessible server
- But we'll *implement* the log using a group of components that run a simple state-machine append protocol
  - This abstraction matches the "Paxos" protocol
  - But the protocol we'll look at is older and was developed in the Isis system for "group view management"

Group communication

- We want the Oracle itself to be a tree, nodes of which are groups of servers
- In fact we can *generalize* this concept
  - The general version is a group of processes
  - … supported by some form of management service
- Turtles all the way down, again?
  - At the core we'll have a "root" group

Group Communication illustration



- Terminology: group create, view, join with state transfer, multicast, client-to-group communication
- "Dynamic" membership model: processes come & go

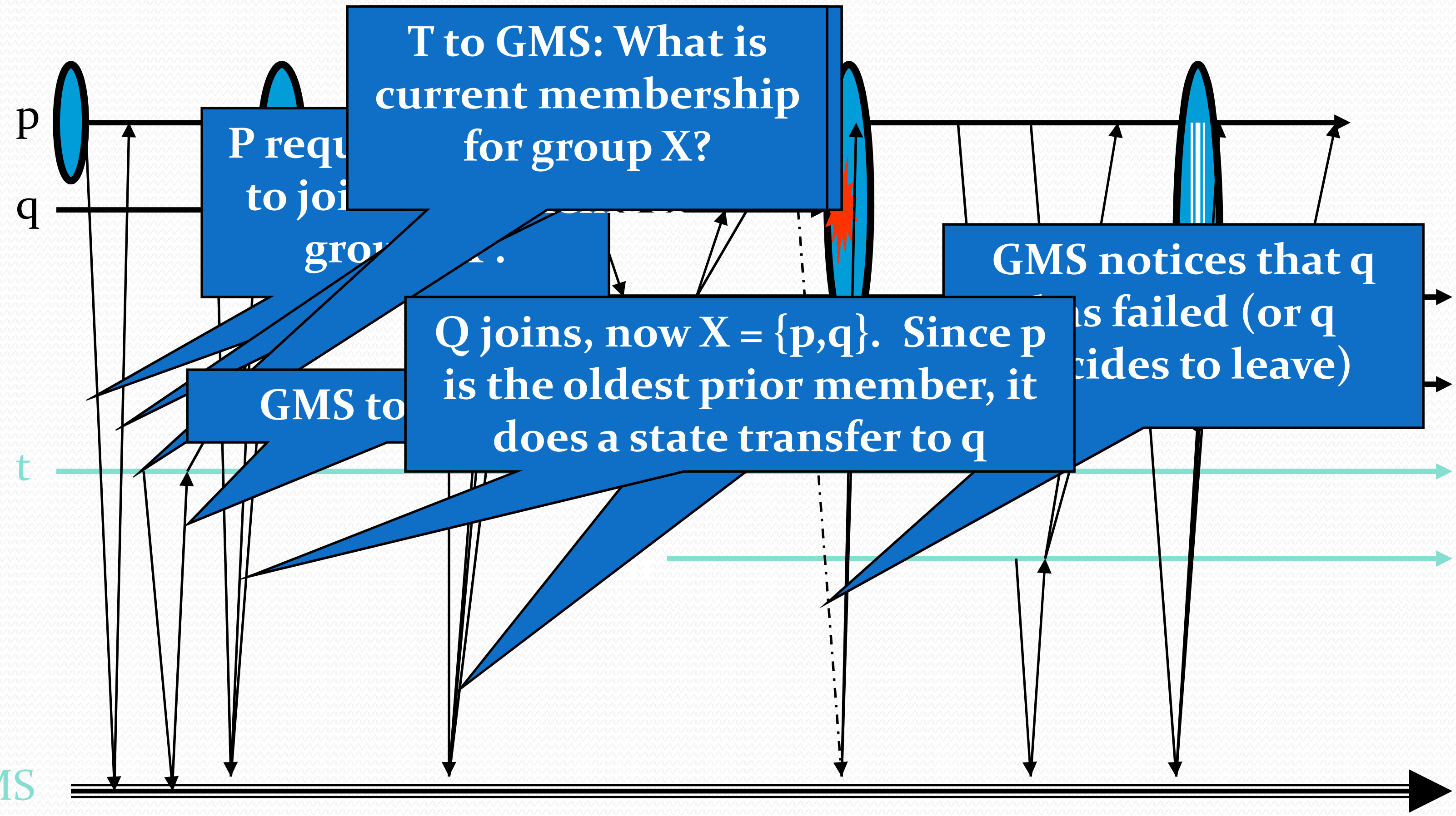
Recipe for a group communication system

- Back one pie shell
  - *Build a service that can track group membership and report "view changes" (our Oracle)*
- Prepare 2 cups of basic pie filling
  - *Develop a simple fault-tolerant multicast protocol*
- Add flavoring of your choice
  - *Extend the multicast protocol to provide desired delivery ordering guarantees*
- Fill pie shell, chill, and serve
  - *Design an end-user "API" or "toolkit". Clients will "serve themselves", with various goals…*

Role of GMS

- We'll add a new system service to our distributed system, like the Internet DNS but with a new role
  - Its job is to track membership of groups
  - To join a group a process will ask the GMS
  - The GMS will also monitor members and can use this to drop them from a group
  - And it will report membership changes

Group picture… with GMS



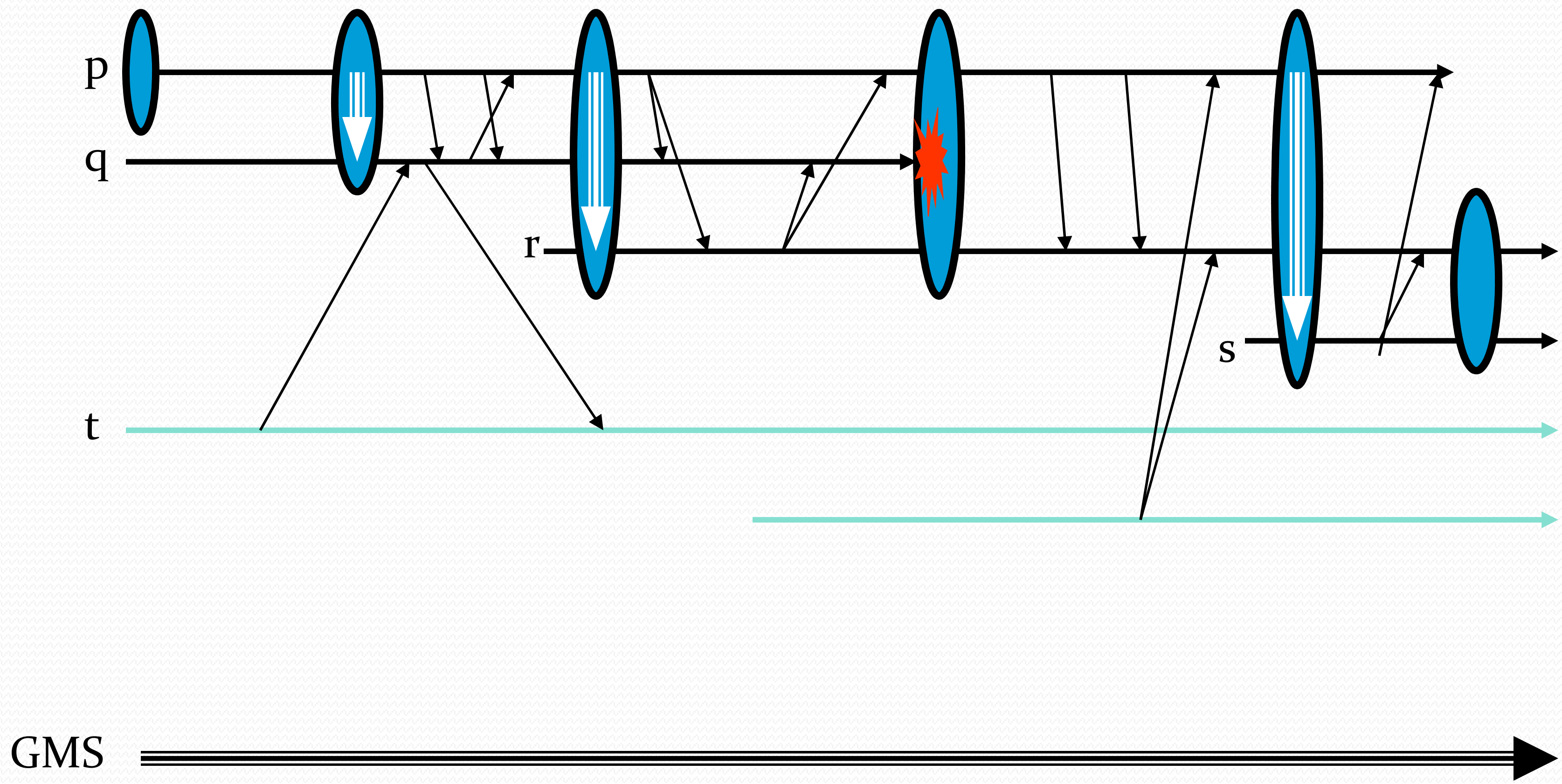
Group membership service

- Runs on some sensible place, like the server hosting your DNS
- Takes as input:
  - Process "join" events
  - Process "leave" events
  - Apparent failures
- Output:
  - Membership views for group(s) to which those processes belong
  - Seen by the protocol "library" that the group members are using for communication support

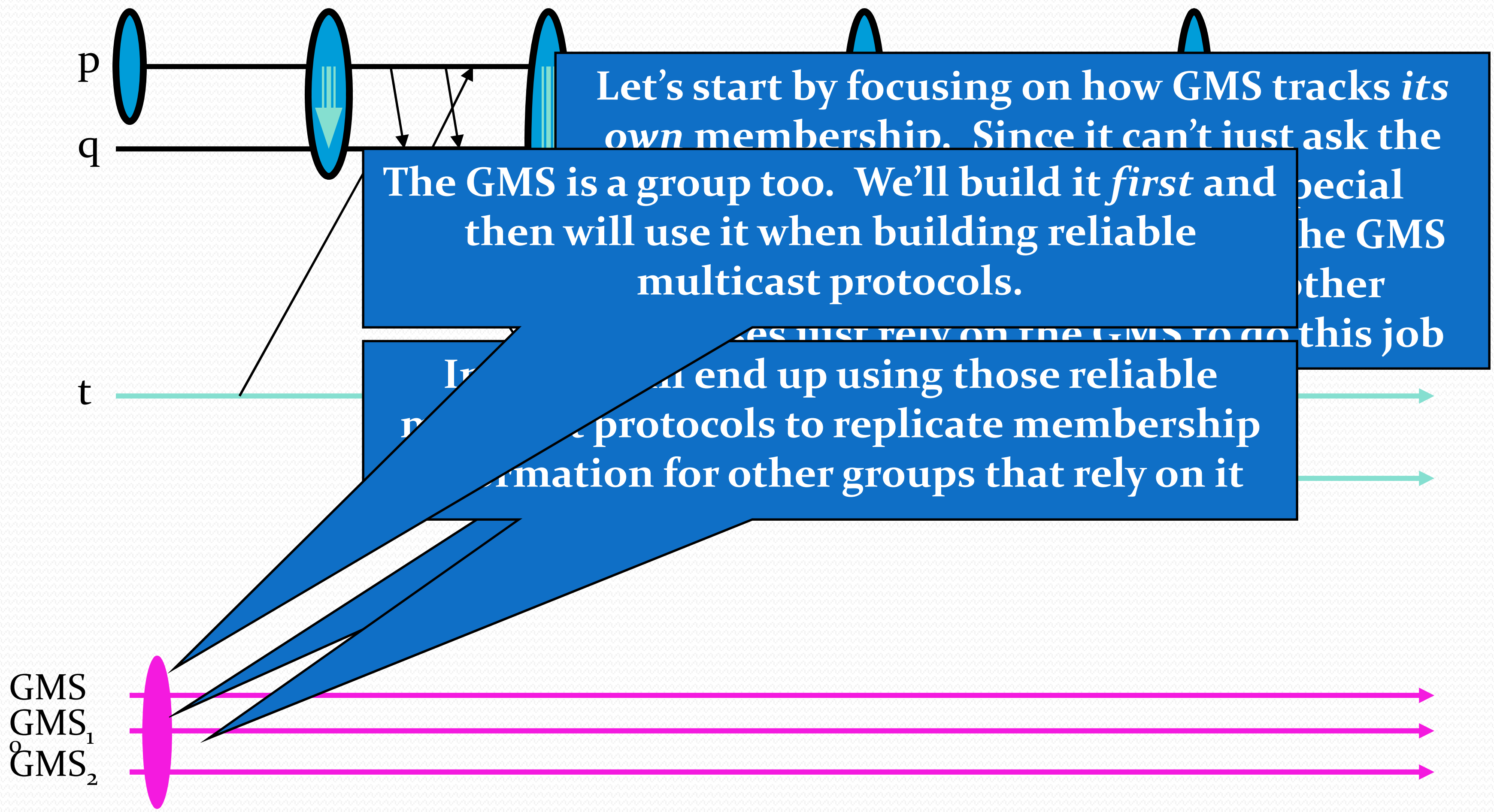
Issues?

- The service *itself* needs to be fault-tolerant
  - Otherwise our entire system could be crippled by a single failure!
- So we'll run two or three copies of it
  - Hence Group Membership Service (GMS) must run some form of protocol (GMP)

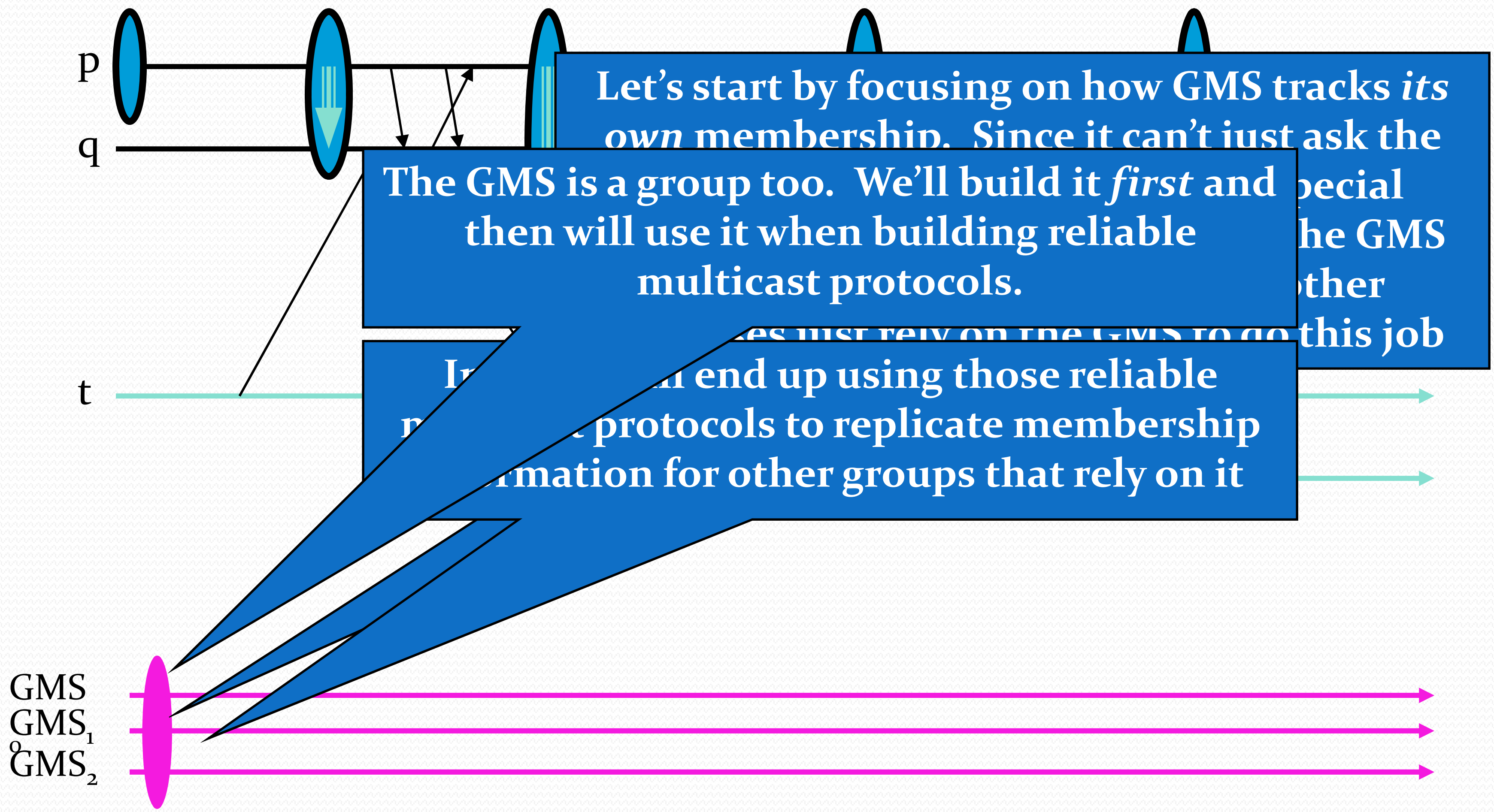
Group picture… with GMS



Group picture… with GMS



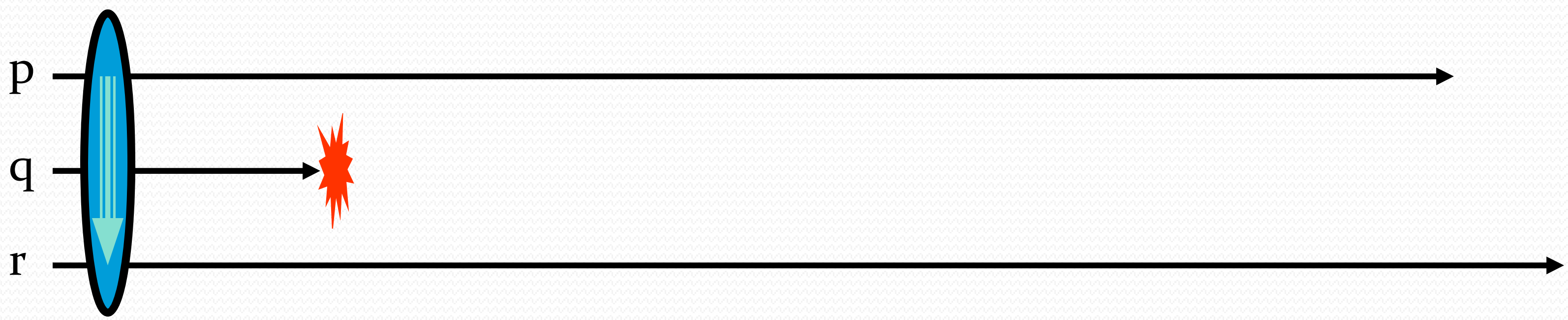
Group picture… with GMS



Approach

- We'll assume that GMS has members {p,q,r} at time t
- Designate the "oldest" of these as the protocol "leader"
  - To initiate a change in GMS membership, leader will run the GMP
  - Others can't run the GMP; they report events to the leader

GMP example



- Example:
  - Initially, GMS consists of {p,q,r}
  - Then q is believed to have crashed

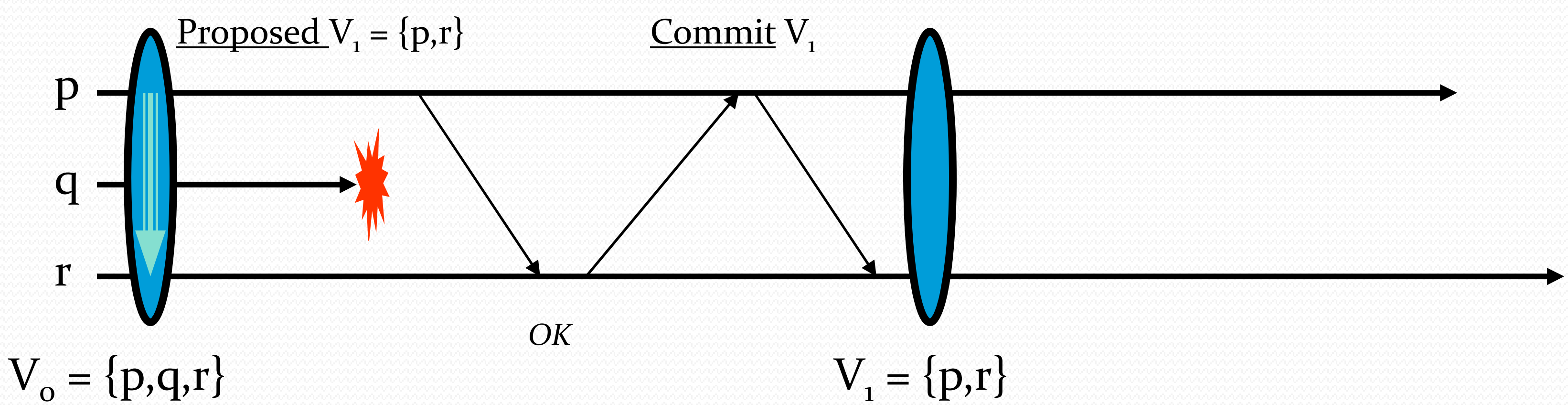
Failure detection: may make mistakes

- Recall that failures are hard to distinguish from network delay
  - So we accept risk of mistake
  - If p is running a protocol to exclude q because "q has failed", all processes that hear from p will cut channels to q
    - Avoids "messages from the dead"
  - q must rejoin to participate in GMS again

Basic GMP

- Someone reports that "q has failed"
- Leader (process p) runs a 2-phase commit protocol
  - Announces a "proposed new GMS view"
    - Excludes q, or might add some members who are joining, or could do both at once
  - Waits until a <u>majority</u> of members of current view have voted "ok"
  - Then commits the change

GMP example



- Proposes new view: {p,r} [-q]
- Needs majority consent: p itself, plus one more ("current" view had 3 members)
- Can add members at the same time

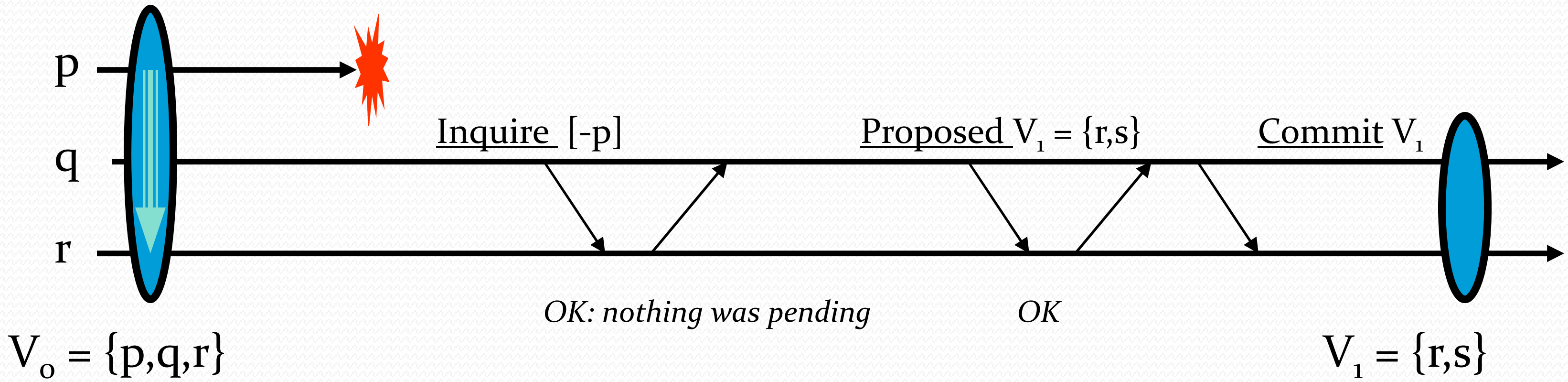
Special concerns?

- What if someone doesn't respond?
  - P can tolerate failures of a minority of members of the current view
    - New first-round "overlaps" its commit:
      - "Commit that q has left. Propose add s and drop r"
  - P must wait if it can't contact a majority
    - Avoids risk of partitioning

What if leader fails?

- Here we do a 3-phase protocol
  - New leader identifies itself based on age ranking (oldest surviving process)
  - It runs an inquiry phase
    - “The adored leader has died. Did he say anything to you before passing away?”
    - Note that this causes participants to cut connections to the adored previous leader
  - Then run normal 2-phase protocol but “terminate” any interrupted view changes leader had initiated

GMP example

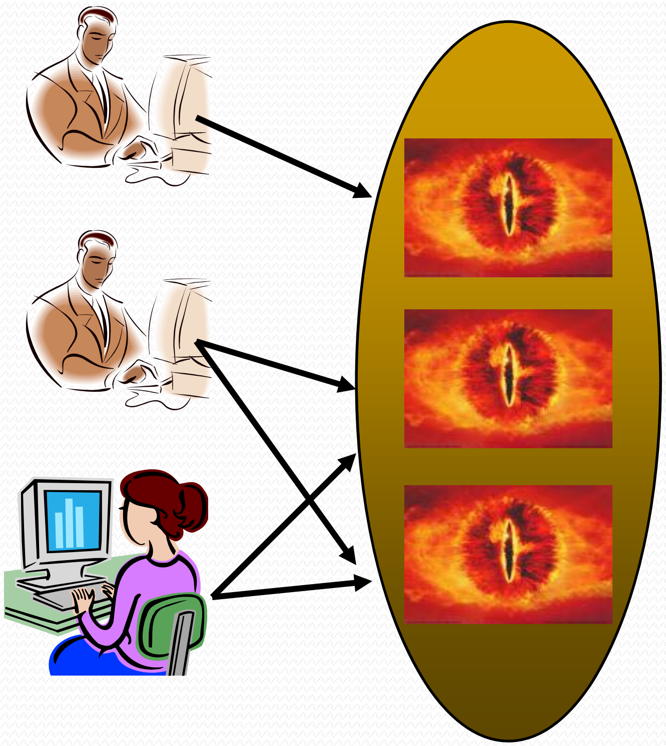


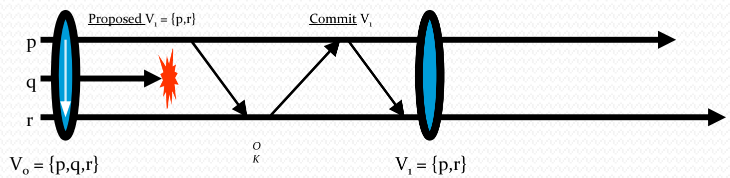
- New leader first sends an inquiry
- Then proposes new view: {r,s} [-p]
- Needs majority consent: q itself, plus one more ("current" view had 3 members)
- Again, can add members at the same time

Turning the GMS into the Oracle

- Build a tree of GMS servers
  - Each node will be a small replicated state machine
- In addition to the group view, members maintain a set of replicated logs
  - Log has a name (like a file pathname)
  - View change protocol used to extend the log with new events
- Various "libraries" allow us to present the service in the forms we have in mind: locking, load-balancing, etc

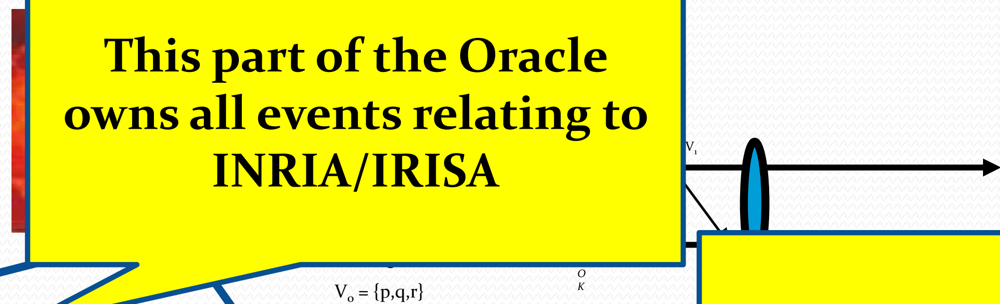
Turning the GMS into the Oracle



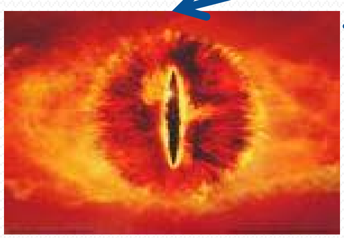


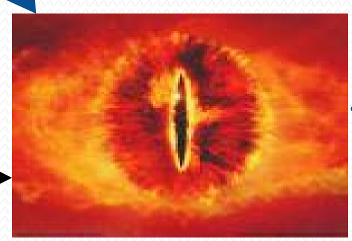
Here, three replicas cooperate to implement the GMS as a fault-tolerant state machine. Each client platform binds to some representative, then rebinds to a different replica if that one later crashes….

Turning the GMS into the Oracle

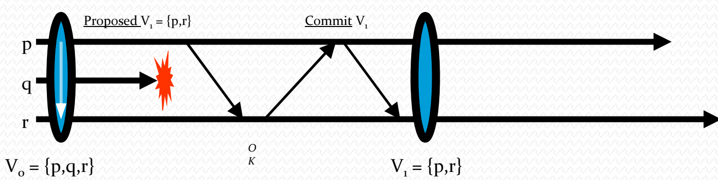


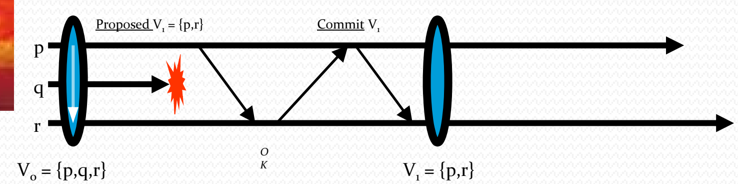
This part of the Oracle owns all events relating to INRIA/IRISA



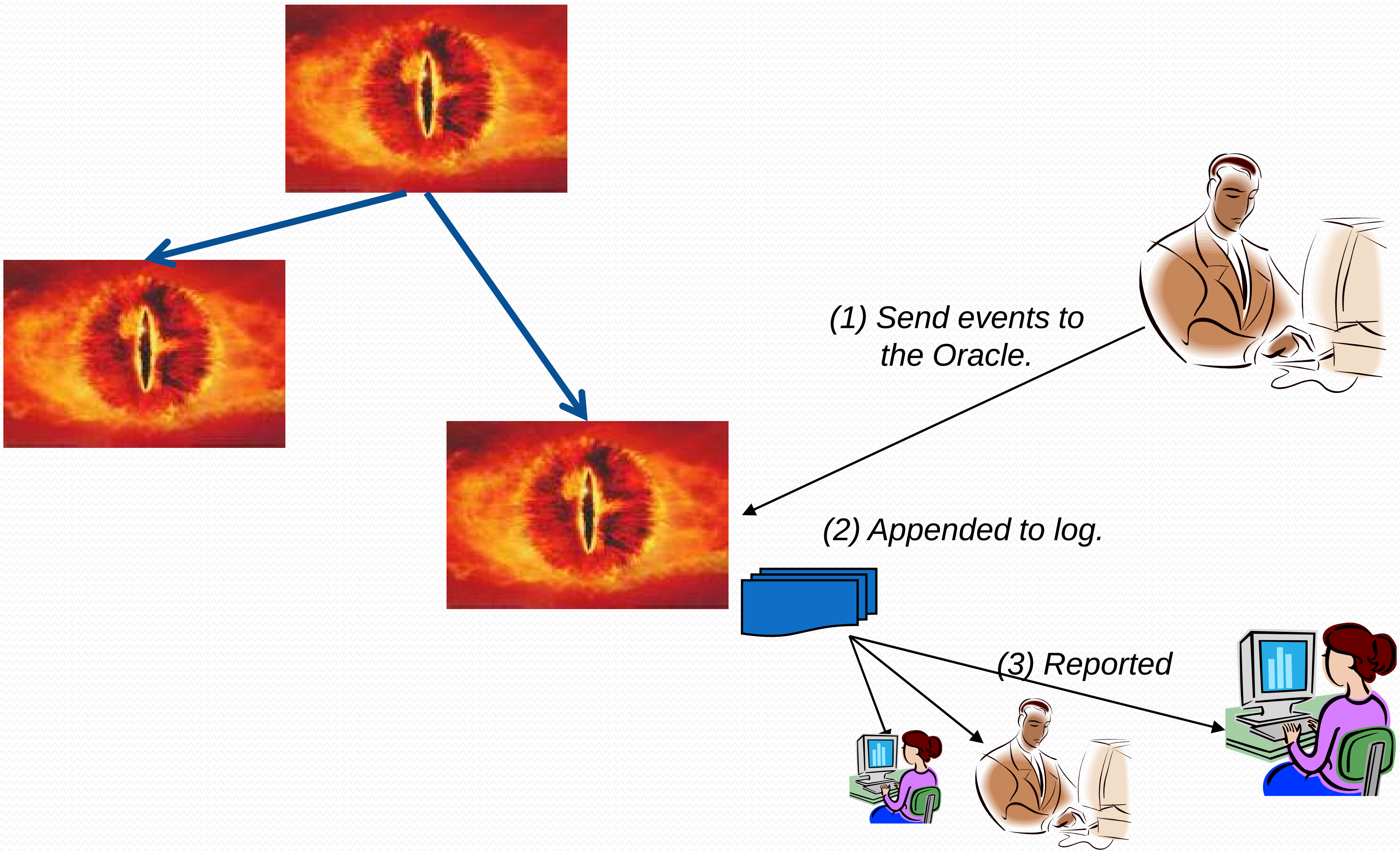


This part of the Oracle owns all events relating to Cornell University





Turning the GMS into the Oracle



Summary

- We're part way down the road to a universal management service
  - We know how to build the core Oracle and replicate it
  - We can organize the replica groups as a tree, and split the roles among nodes (each log has an "owner"
  - The general class of solutions gives us group communication supported by a management layer
- Next lecture: we'll finish the group communication subsystem and use it to support service replication