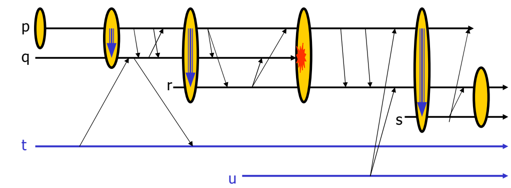
CS514: Intermediate Course in Operating Systems

Professor Ken Birman
Vivek Vishnumurthy: TA

Reminder: Group Communication



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

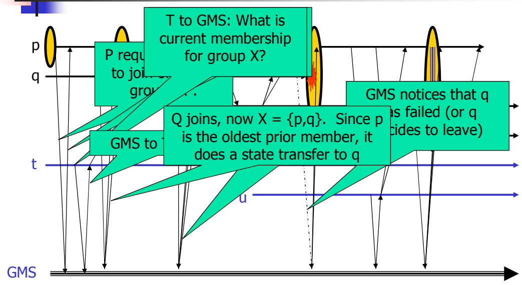
Recipe for a group communication system

- Back one pie shell
  - *Build a service that can track group membership and report "view changes"*
- 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



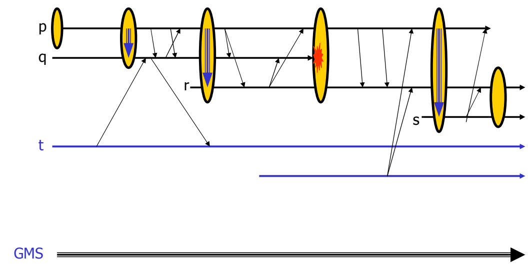
Group membership service

- 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

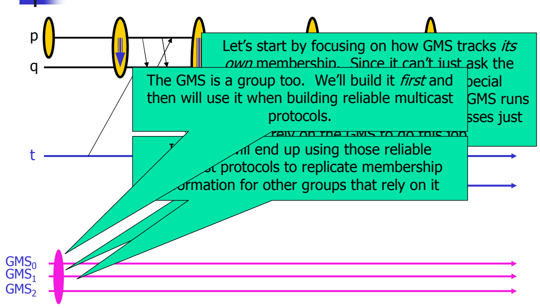
Issues?

- 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



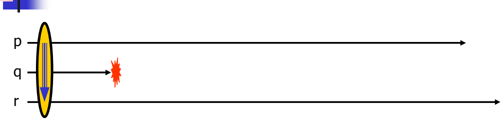
Group picture… with GMS



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

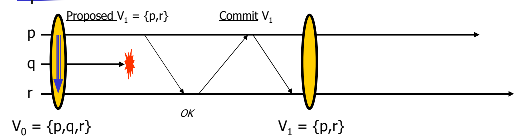
Failure detection: may make mistakes

- 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 majority 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

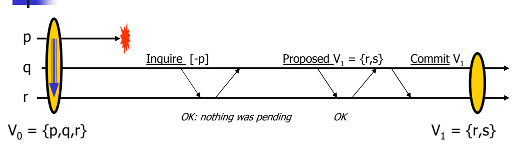
Special concerns?

- 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

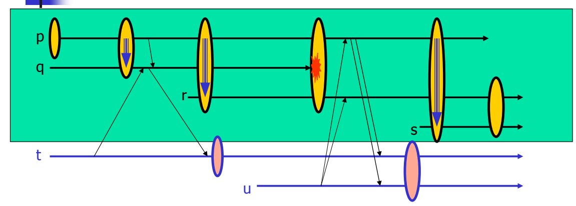
Properties of GMP

- We end up with a single service shared by the entire system
  - In fact every process can participate
  - But more often we just designate a few processes and they run the GMP
- Typically the GMS runs the GMP and also uses replicated data to track membership of *other* groups

Use of GMS

- A process t, not in the GMS, wants to join group "Upson309_status"
  - It sends a request to the GMS
  - GMS updates the "membership of group Upson309_status" to add t
  - Reports the new view to the current members of the group, and to t
  - Begins to monitor t's health

Processes t and u "using" a GMS



- The GMS contains p, q, r (and later, s)
- Processes t and u want to form some other group, but use the GMS to manage membership on their behalf

We have our pie shell

- Now we've got a group membership service that reports identical views to all members, tracks health
- Can we build a reliable multicast?

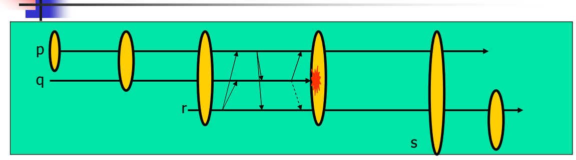
Unreliable multicast

- Suppose that to send a multicast, a process just uses an unreliable protocol
  - Perhaps IP multicast
  - Perhaps UDP point-to-point
  - Perhaps TCP
- … some messages might get dropped. If so it eventually finds out and resends them (various options for how to do 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

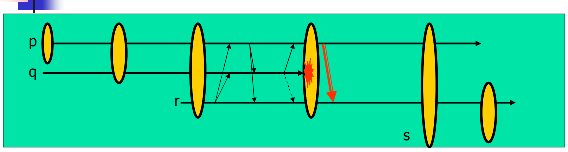
Flush protocol

- 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

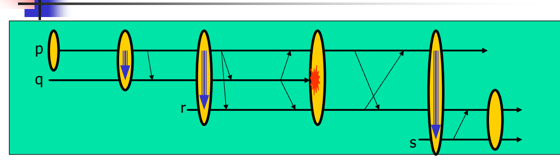
Event ordering

- 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

What about ordering?

- It is trivial to make our protocol FIFO wrt other messages from same sender
  - If we just number messages from each sender, they will "stay" in order
- Concurrent messages are unordered
  - If sent by different senders, messages can be delivered in different orders at different receivers
- This is the protocol called "fbcast"

Preview of coming attractions

- Next time we'll add richer ordering properties
- Group communication platforms often offer a range
  - Idea is that developer will pick the cheapest solution that meets needs of a given use