CS5412: VIRTUAL SYNCHRONY

Lecture XIV

Ken Birman

Group Communication idea

- System supports a new abstraction (like an object)
  - A “group” consisting of a set of processes (“members”) that join, leave and cooperate to replicate data or do parallel processing tasks
  - A group has a name (like a filename)
  - … and a state (the data that its members are maintaining)
    - The state will often be *replicated* so each member has a copy
    - Note that this is in contrast to Paxos where each member has a partial copy and we need to use a “learner algorithm” to extract the actual current state
    - Think of state much as you think of the value of a variable, except that a group could track many variables at once

Group communication Idea

- The members can send each other
  - Point-to-point messages
  - Multicasts that go from someone to *all* the members
- They can also do RPC style queries
  - Query a single member
  - Query the whole group, with all of them replying

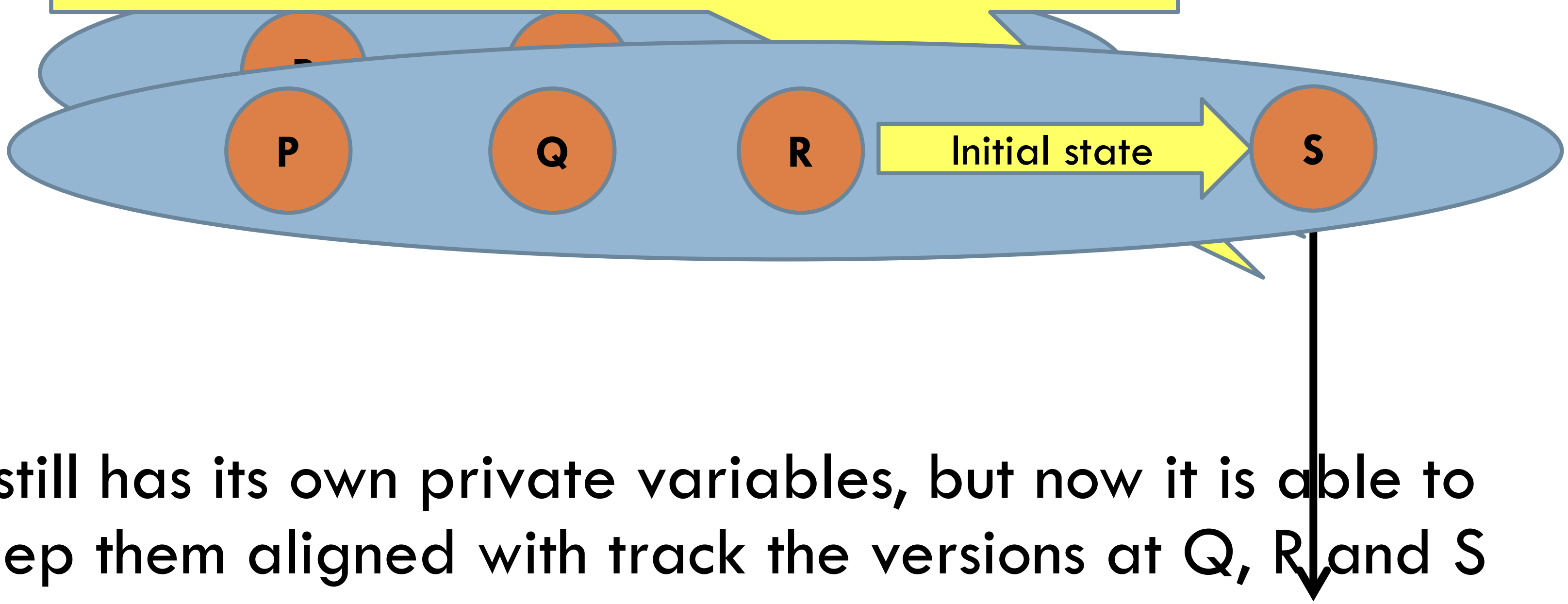
- Example: The Isis^2 system (but there are many such systems)

Animation: A process joins a group

- ... purely local

S starts by creating an endpoint object, attaching upcall
han...

Once the group endpoint is properly configured, S issues a "join" request. Isis[2] checks to see if the group already exists. If not, p can create a new instance, but in this case, the group is already active.



- P still has its own private variables, but now it is able to keep them aligned with track the versions at Q, R and S

P's endpoint

- Just an object that is P's "portal" for operations involving the group

- The endpoint lets P see events occurring in the group such as members joining, failing (detected slowly via timeout) or leaving (very fast notification), multicasts reporting updates or other events, queries, etc

- *But no data is automatically replicated.* P provides logic to maintain the data it associates with the group.

Isis^2 is a *library* for group communication

It Uses a Formal model

- Formal model permits us to achieve correctness
- Isis^2 is too complex to use formal methods as a development tool, but does facilitate debugging (model checking)
- Think of Isis^2 as a collection of modules, each with rigorously stated properties

It Reflects Sound Engineering

- Isis^2 implementation needs to be fast, lean, easy to use
- Developer must see it as easier to use Isis^2 than to build from scratch
- Seek great performance under "cloudy conditions"
- Forced to anticipate many styles of use

Isis^2 makes developer's life easier


```
Group g = new Group("myGroup");
Dictionary <string,double> Values = new Dictionary<string,double>();
g.ViewHandlers += delegate(View v) {
        Console.Title = "myGroup members: "+v.members;
};
g.Handlers[UPDATE] += delegate(string s, double v) {
        Values[s] = v;
};
g.Handlers[LOOKUP] += delegate(string s) {
        g.Reply(Values[s]);
};
g.Join();

g.OrderedSend(UPDATE, "Harry", 20.75);

List<double> resultlist = new List<double>();
nr = g.Query(ALL, LOOKUP, "Harry", EOL, resultlist);
```

- First sets up group
- Join makes this entity a member. State transfer isn't shown
- Then can multicast, query. Runtime callbacks to the "delegates" as events arrive
- Easy to request security (g.SetSecure), persistence
- "Consistency" model dictates the ordering seen for event upcalls and the assumptions user can make. User can tell Isis2 how strong ordering needs to be.

Isis^2 makes developer's life easier


```
Group g = new Group("myGroup");
Dictionary <string,double> Values = new Dictionary<string,double>();
g.ViewHandlers += delegate(View v) {
        Console.Title = "myGroup members: "+v.members;
};
g.Handlers[UPDATE] += delegate(string s, double v) {
        Values[s] = v;
};
g.Handlers[LOOKUP] += delegate(string s) {
        g.Reply(Values[s]);
};
g.Join();

g.OrderedSend(UPDATE, "Harry", 20.75);

List<double> resultlist = new List<double>();
nr = g.Query(ALL, LOOKUP, "Harry", EOL, resultlist);
```

First sets up group

Join makes this entity a member. State transfer isn't shown

Then can multicast, query. Runtime callbacks to the "delegates" as events arrive

Easy to request security (g.SetSecure), persistence

"Consistency" model dictates the ordering seen for event upcalls and the assumptions user can make. User can tell Isis2 how strong ordering needs to be.

Isis^2 makes developer's life easier


```
Group g = new Group("myGroup");
Dictionary <string,double> Values = new Dictionary<string,double>();
g.ViewHandlers += delegate(View v) {
        Console.Title = "myGroup members: "+v.members;
};
g.Handlers[UPDATE] += delegate(string s, double v) {
        Values[s] = v;
};
g.Handlers[LOOKUP] += delegate(string s) {
        g.Reply(Values[s]);
};
g.Join();

g.OrderedSend(UPDATE, "Harry", 20.75);

List<double> resultlist = new List<double>();
nr = g.Query(ALL, LOOKUP, "Harry", EOL, resultlist);
```

First sets up group

Join makes this a member. State transfer isn't shown

Then can multicast, query. Runtime callbacks to the "delegates" as events arrive

Easy to request security (g.SetSecure), persistence

"Consistency" model dictates the ordering seen for event upcalls and the assumptions user can make. User can tell Isis2 how strong ordering needs to be.

Isis^2 makes developer's life easier


```
Group g = new Group("myGroup");
Dictionary <string,double> Values = new Dictionary<string,double>();
g.ViewHandlers += delegate(View v) {
        Console.Title = "myGroup members: "+v.members;
};
g.Handlers[UPDATE] += delegate(string s, double v) {
        Values[s] = v;
};
g.Handlers[LOOKUP] += delegate(string s) {
        g.Reply(Values[s]);
};
g.Join();

g.OrderedSend(UPDATE, "Harry", 20.75);

List<double> resultlist = new List<double>();
nr = g.Query(ALL, LOOKUP, "Harry", EOL, resultlist);
```

- First sets up group
- Join makes this entity a member. State transfer isn't shown
- **Then can multicast, query. Runtime callbacks to the "delegates" as events arrive**
- Easy to request security (g.SetSecure), persistence
- "Consistency" model dictates the ordering seen for event upcalls and the assumptions user can make. User can tell Isis2 how strong ordering needs to be.

Isis^2 makes developer's life easier


```
Group g = new Group("myGroup");
Dictionary <string,double> Values = new Dictionary<string,double>();
g.ViewHandlers += delegate(View v) {
        Console.Title = "myGroup members: "+v.members;
};
g.Handlers[UPDATE] += delegate(string s, double v) {
        Values[s] = v;
};
g.Handlers[LOOKUP] += delegate(string s) {
        g.Reply(Values[s]);
};
g.Join();

g.OrderedSend(UPDATE, "Harry", 20.75);

List<double> resultlist = new List<double>();
nr = g.Query(ALL, LOOKUP, "Harry", EOL, resultlist);
```

- First sets up group
- Join makes this entity a member. State transfer isn't shown
- **Then can multicast, query. Runtime callbacks to the "delegates" as events arrive**
- Easy to request security (g.SetSecure), persistence
- "Consistency" model dictates the ordering seen for event upcalls and the assumptions user can make. User can tell Isis2 how strong ordering needs to be.

Isis^2 makes developer's life easier


```
Group g = new Group("myGroup");
Dictionary <string,double> Values = new Dictionary<string,double>();
g.ViewHandlers += delegate(View v) {
        Console.Title = "myGroup members: "+v.members;
};
g.Handlers[UPDATE] += delegate(string s, double v) {
        Values[s] = v;
};
g.Handlers[LOOKUP] += delegate(string s) {
        g.Reply(Values[s]);
};
g.Join();
g.SetSecure(key);
g.OrderedSend(UPDATE, "Harry", 20.75);

List<double> resultlist = new List<double>();
nr = g.Query(ALL, LOOKUP, "Harry", EOL, resultlist);
```

First sets up group

Join makes this entity a member. State transfer isn't shown

Then can multicast, query. Runtime callbacks to the "delegates" as events arrive

Easy to request security (g.SetSecure), persistence

"Consistency" model dictates the ordering seen for event upcalls and the assumptions user can make. User can tell Isis^2 how strong ordering needs to be.

Isis^2 makes developer's life easier


```
Group g = new Group("myGroup");
Dictionary <string,double> Values = new Dictionary<string,double>();
g.ViewHandlers += delegate(View v) {
        Console.Title = "myGroup members: "+v.members;
};
g.Handlers[UPDATE] += delegate(string s, double v) {
        Values[s] = v;
};
g.Handlers[LOOKUP] += delegate(string s) {
        g.Reply(Values[s]);
};
g.Join();
g.SetSecure(key);
g.Send(UPDATE, "Harry", 20.75);

List<double> resultlist = new List<double>();
nr = g.Query(ALL, LOOKUP, "Harry", EOL, resultlist);
```

First sets up group

Join makes this entity a member. State transfer isn't shown

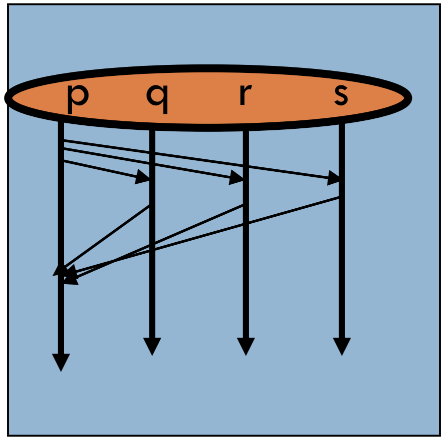
Then can multicast, query. Runtime callbacks to the "delegates" as events arrive

Easy to request security (g.SetSecure), persistence

"Consistency" model dictates the ordering seen for event upcalls and the assumptions user can make. User can tell Isis^2 how strong ordering needs to be.

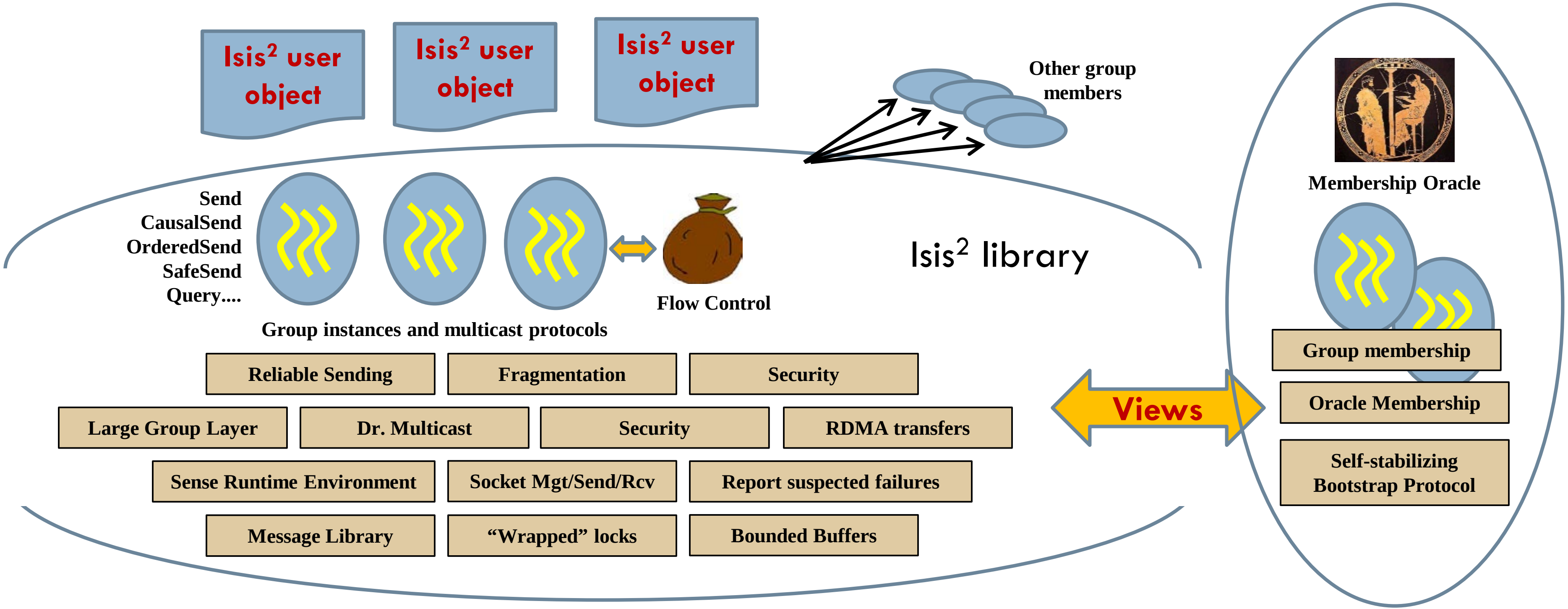
Concept: Query as a "multi-RPC"

- One member asks multiple group members to perform some action
- It could be doing this on behalf of an external client, and it might participate too
- Often group members subdivide the task (but there could be a fault-tolerance benefit to asking 2 or more to do the same work)

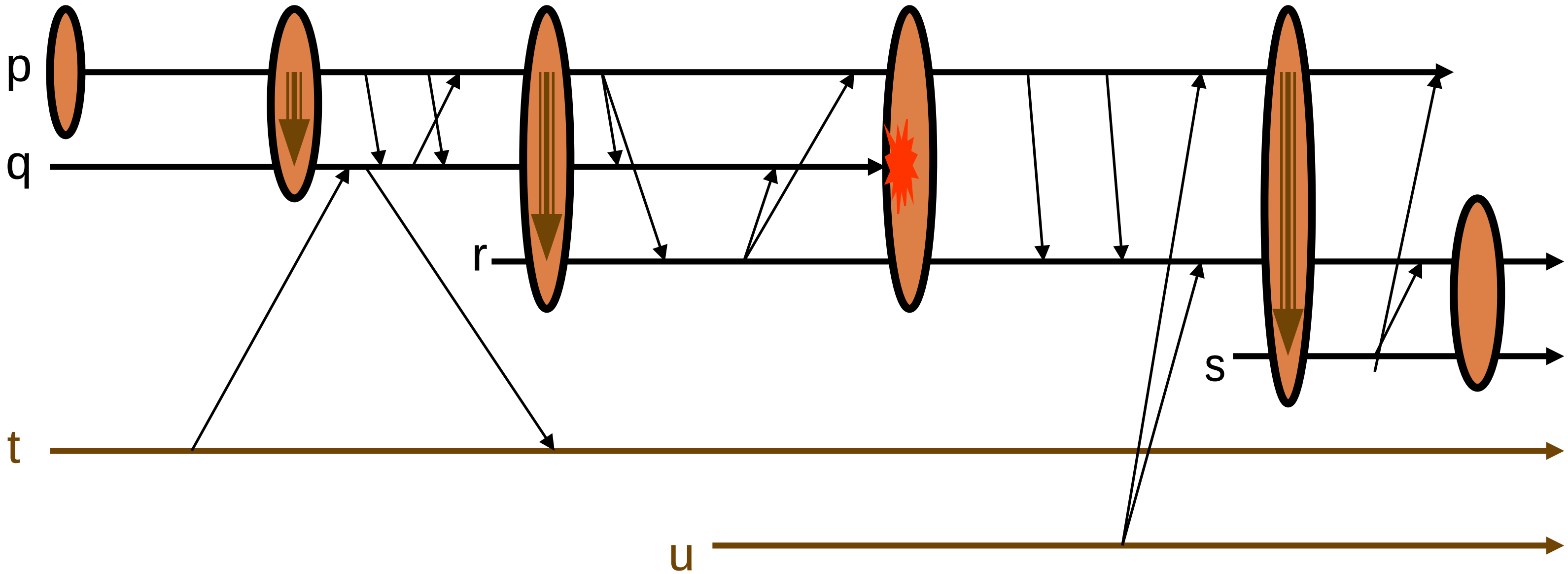


It takes a "community"

- A lot of complexity lurks behind those simple APIs
- Building one of your own would be hard
- Isis² took Ken >3 years to implement & debug



What goes on down there?



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

Clients of a group

- Applications linked to $Isis^2$ can access a group by joining it as a member, but can also issue requests as a "client" in RPC style
- One can also build a group that uses a web service standard (SOAP, WCF, REST) to accept requests from web clients that don't use $Isis^2$ at all. Many cloud services can automatically load balance such requests over the set of group members.
- The representative acts as a "proxy" for the client and can issue multicasts or queries on its behalf

Concepts

- You build your program and link with Isis[2]
- It starts the library (the new guy tracks down any active existing members)
- Then you can create and join groups, receive a "state transfer" to catch up, cooperate with others
- All kinds of events are reported via upcalls
  - New view: View object tells members what happened
  - Incoming message: data fields extracted and passed as values to your handler method

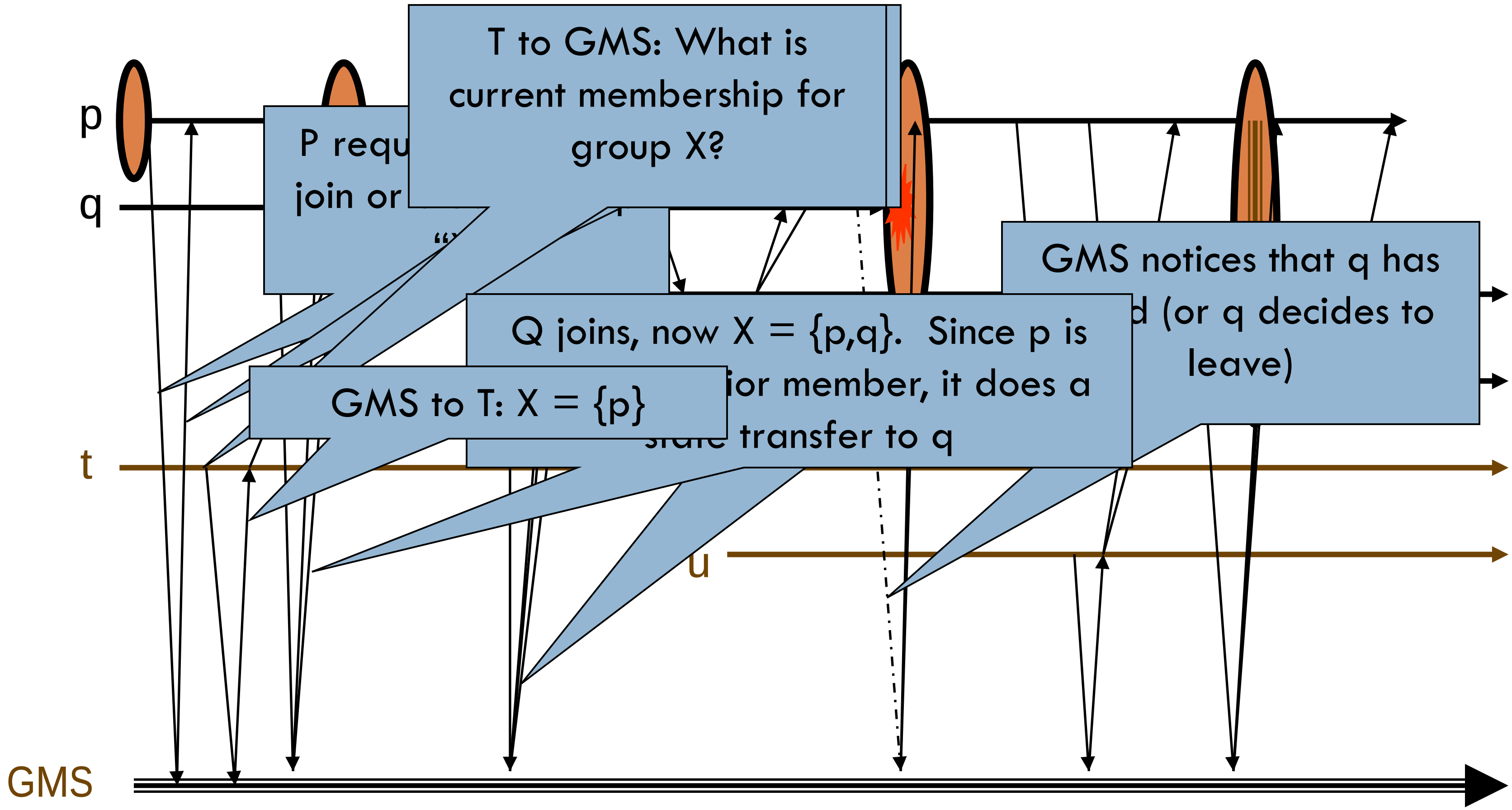
Recipe for a group communication system

- Bake 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



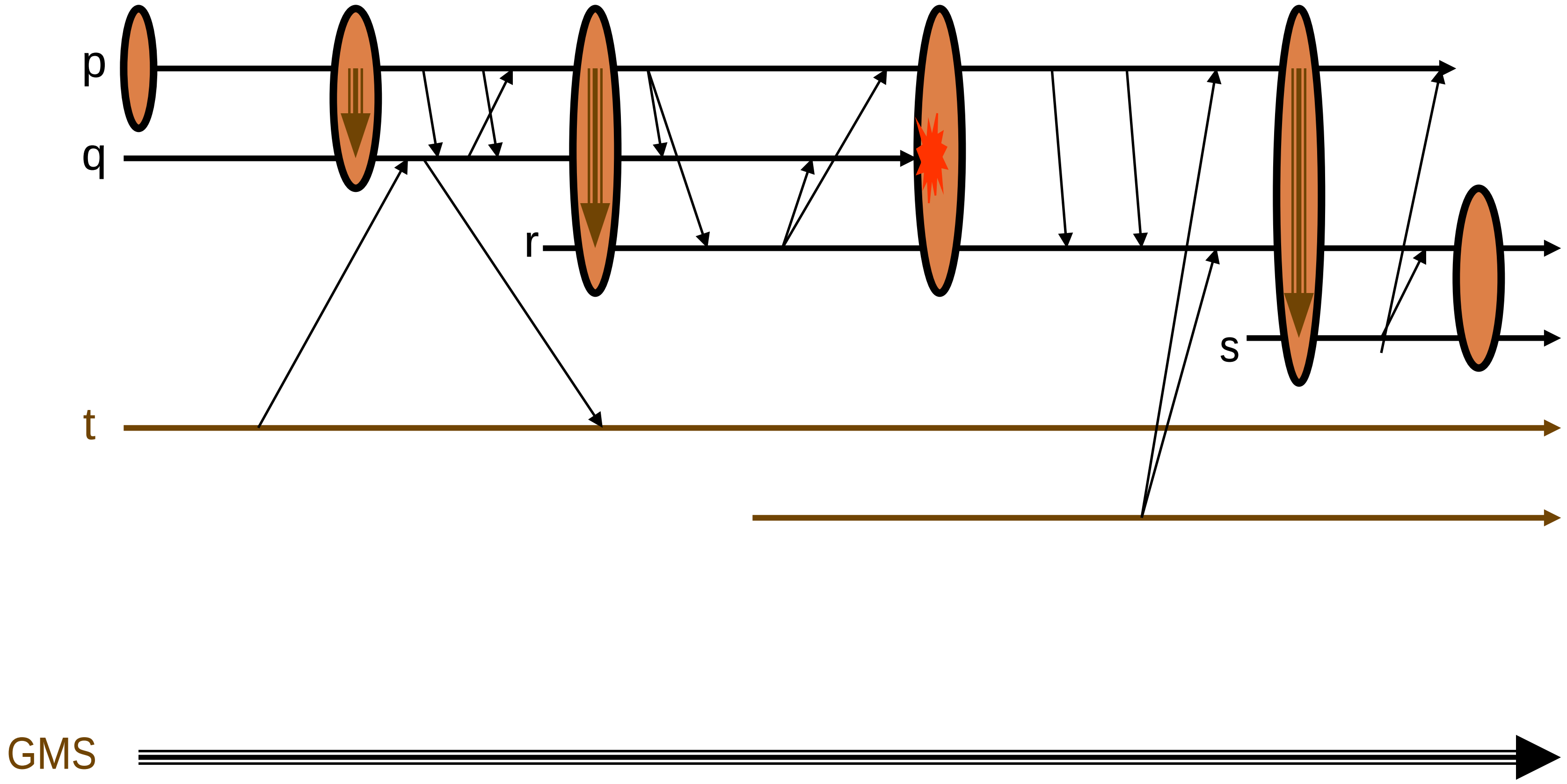
Group membership service

- Runs on some sensible place, like the first few machines that start up when you launch Isis[2]
- 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

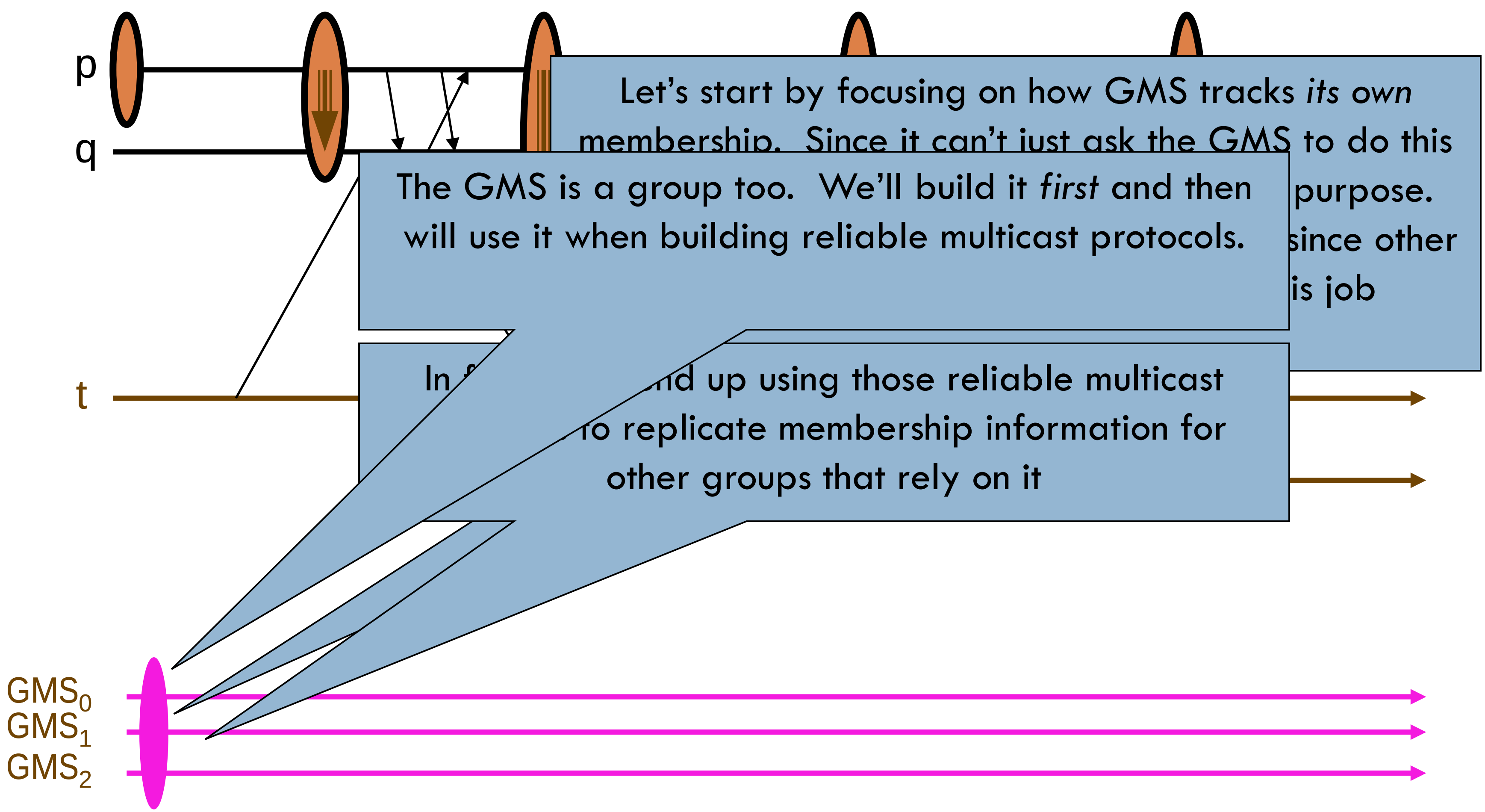
Issues?

- 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

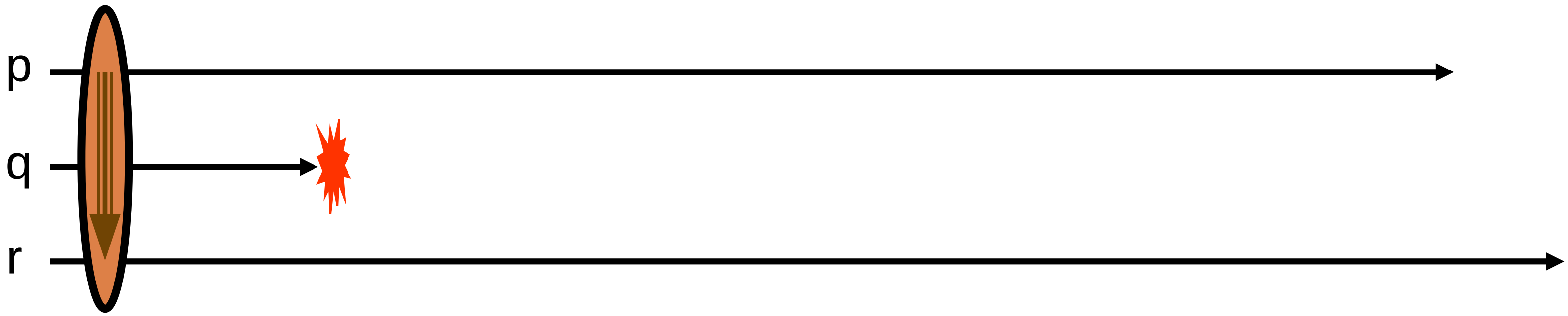


Approach

- 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 GMS
  - Others can't run the GMS; they report events to the leader

GMS example

The GMS group



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

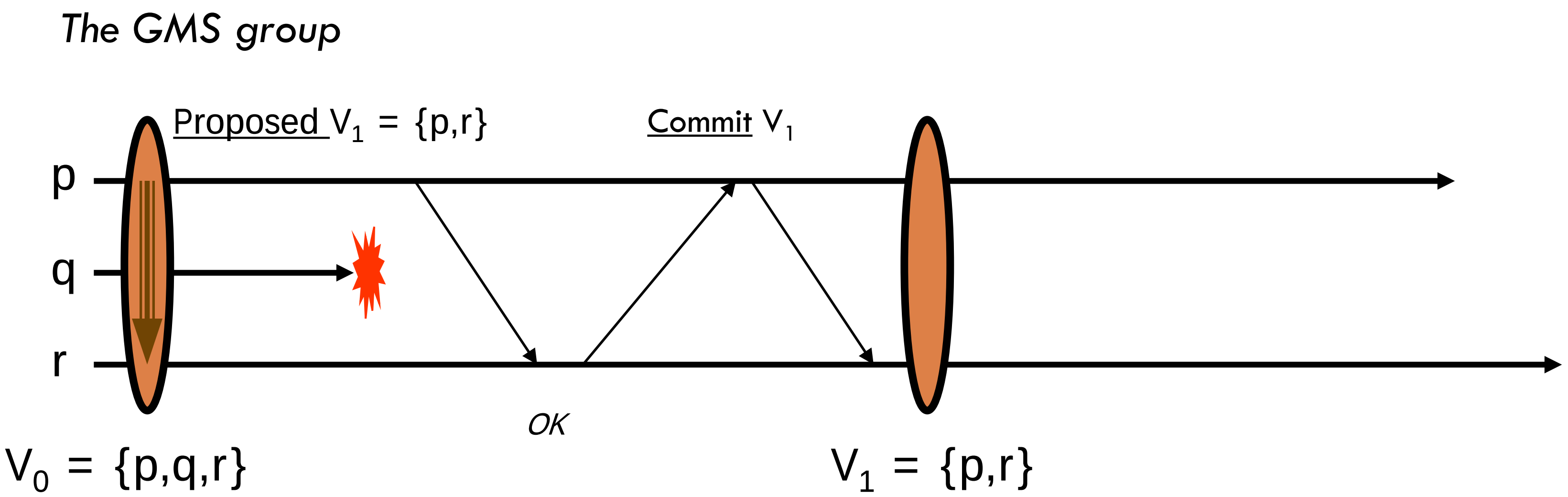
Failure detection: may make mistakes

- 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 GMS

- 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

GMS example



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

Special concerns?

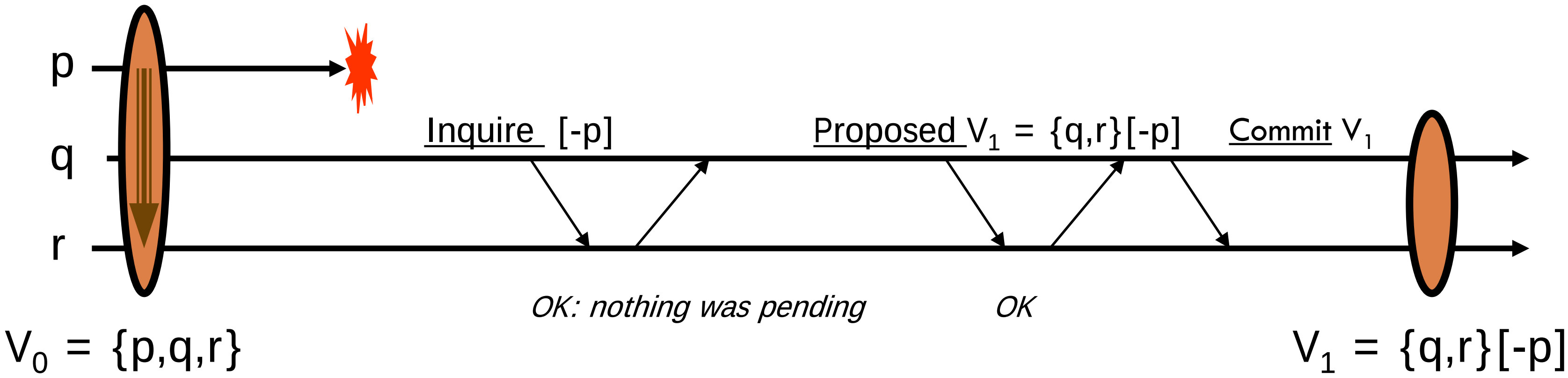
- What if someone doesn't respond?
  - P can tolerate failures of a minority of members of the current view
    - New first-round "overlaps" its commit:
      - "Commit that q has left. Propose add s and drop r"
  - P must wait if it can't contact a majority
    - Avoids risk of partitioning

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

GMS example

The GMS group



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

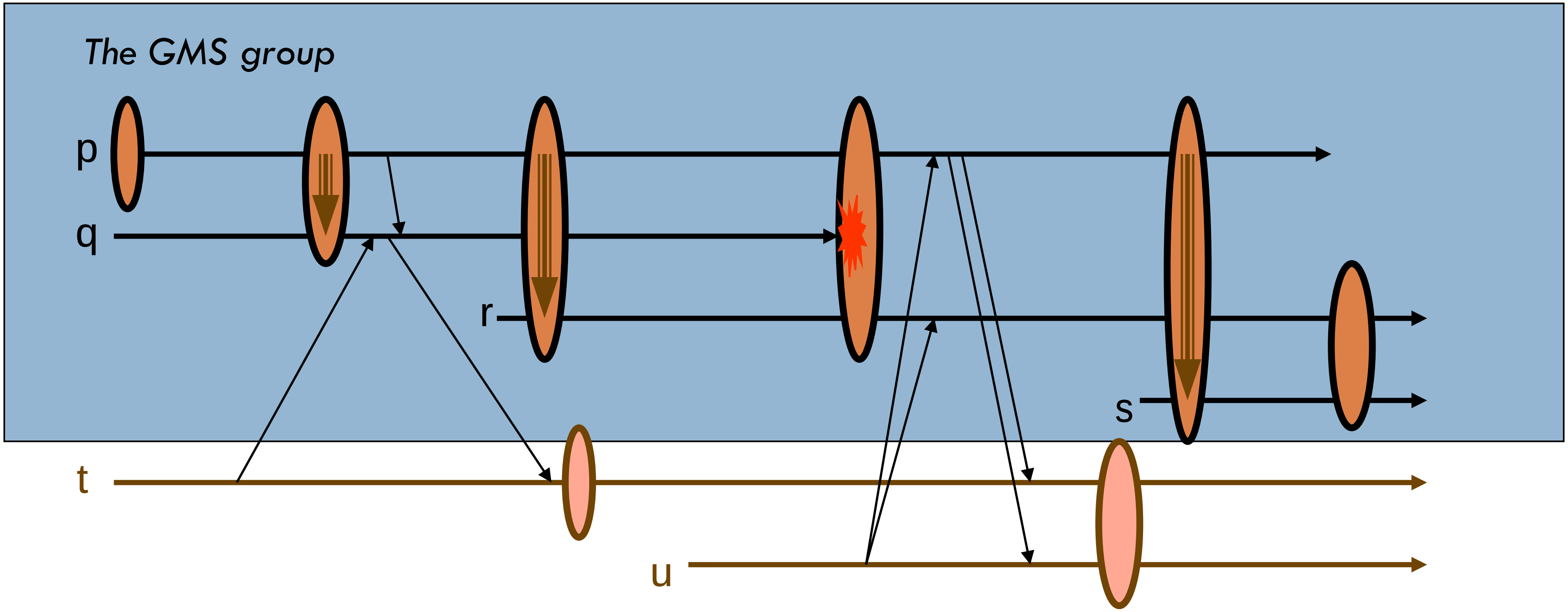
Properties of GMS

- 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 GMS
- 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

Relate to Paxos

- In fact we're doing something *very* similar to Paxos
  - The "slot number" is the "view number"
  - And the "ballot" is the current proposal for what the next view should be
  - With Paxos proposers can actually talk about multiple future slots/commands (concurrency parameter α)
  - With GMS, we do that too!
    - A single proposal can actually propose multiple changes
    - First [add X], then [drop Y and Z], then [add A, B and C]…
    - In order… eventually 2PC succeeds and they all commit

How does this differ from Paxos?

- Details are clearly not identical, and GMS state isn't durable
- Runs with a well-defined leader; Paxos didn't need one (in Paxos we often prefer to have a single leader but correctness is ensured with multiple coordinators)
- Very similar guarantees of ordering and if we added logging, durability too. (Isis^2 SafeSend adds this logging)
- Isis GMS protocol predates Paxos. It "bisimulates" Paxos, meaning that each can simulate the other.

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?

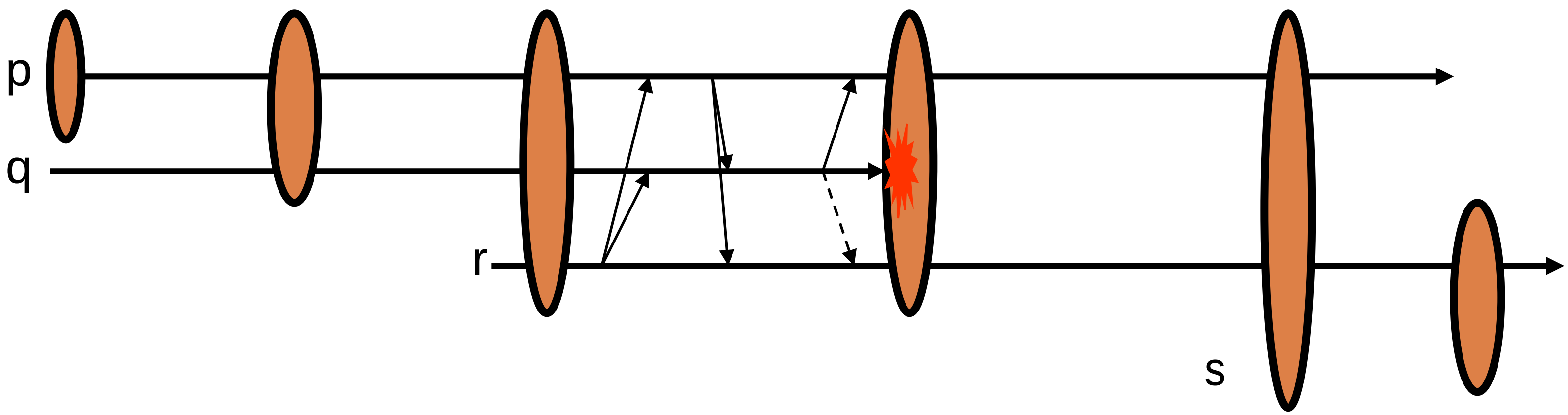
Unreliable multicast

- 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

Terminating an interrupted multicast

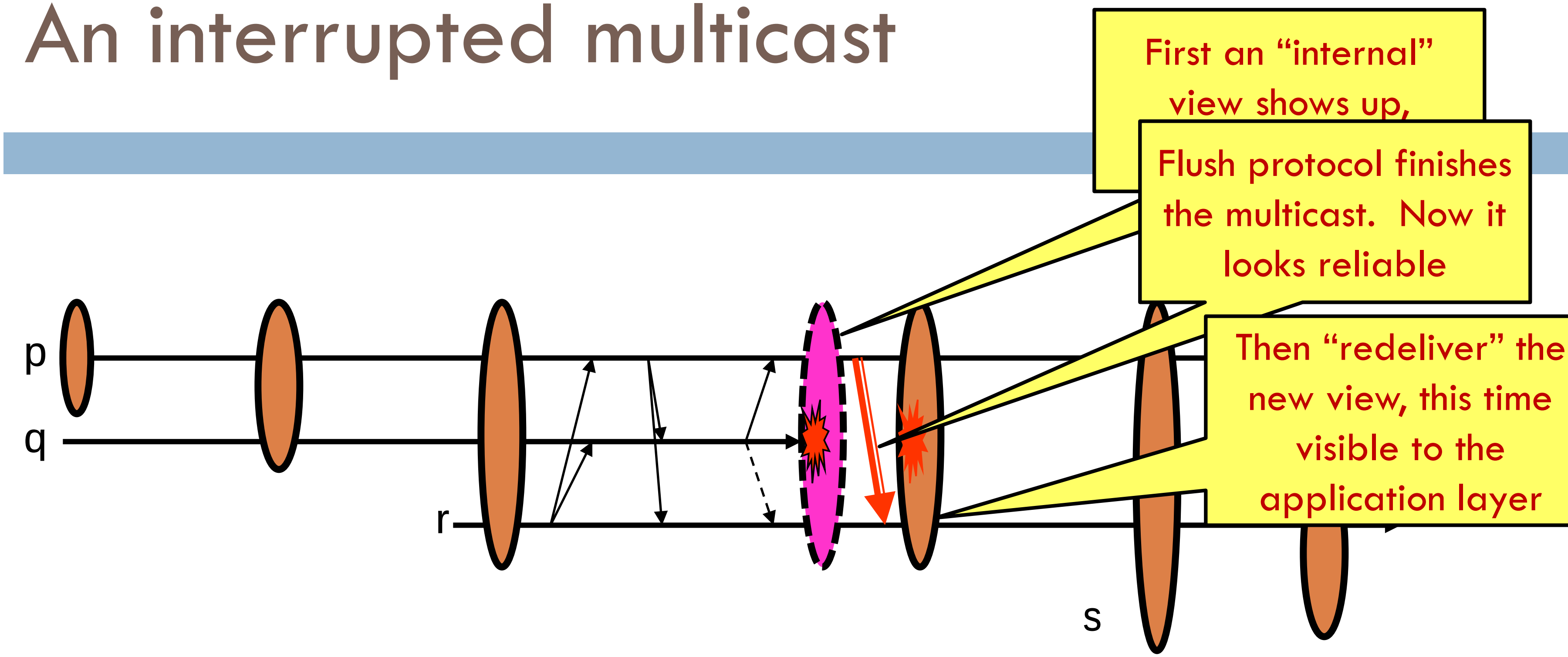
- We say that a message is *unstable* if some receiver has it but (perhaps) others don't
  - For example, q's message is unstable at process r
- If q fails we want to terminate unstable messages
  - Finish delivering them (without duplicate deliveries)
  - Masks the fact that the multicast wasn't reliable and that the leader crashed before finishing up

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

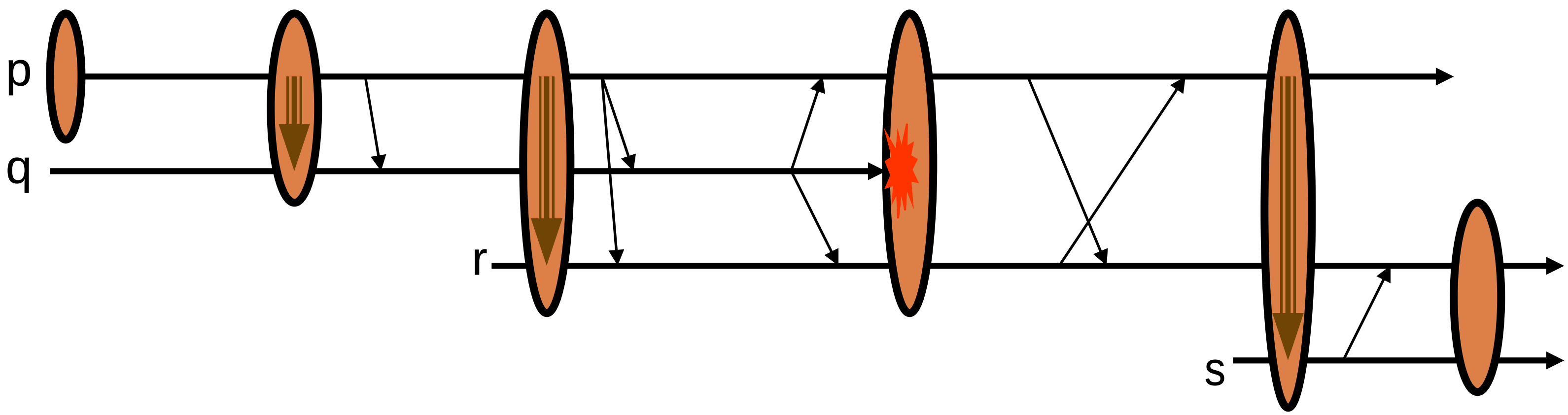
Event ordering

- 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 in the same view 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 "Send"

When is Send used?

- The protocol is very fast
  - Useful if ordering really doesn't matter
  - Or if all the updates to some object are sent by the same process. In this case FIFO is what we need

- Send is not the right choice if multiple members send concurrent, conflicting updates
  - In that case use g.OrderedSend()

Other options?

- OrderedSend: used if there might be concurrent sends

- SafeSend: Most conservative but also quite costly. A version of Paxos (topic of next lecture)

What does this give us?

- A second way to implement state machine replication in which each member has a complete and correct state
  - Notice contrast with Paxos where to learn the state you need to run a decision process that reads Q_R copies
  - Isis^2 replica is just a local object and you use it like any other object (with locking to prevent concurrent update)
  - Paxos has replicated state but you need to read multiple process states to figure out the value
- This makes Isis^2 faster and cheaper

Isis^2 versus Paxos

- Isis^2 offers control over message ordering and durability. Paxos has just one option.
- By default, Isis^2 is a multicast layer that just delivers messages and doesn't log them
- But you can log group states in various ways, including exactly what Paxos does.

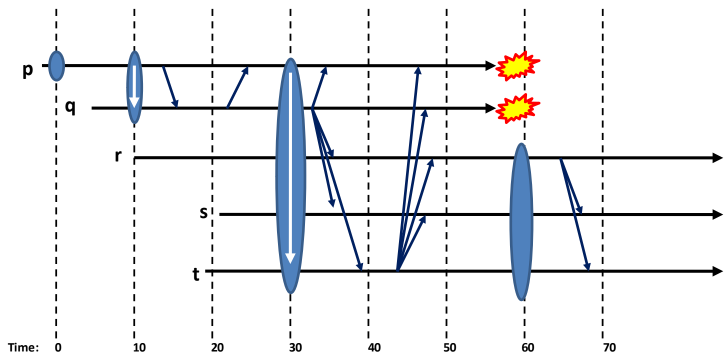
How can $Isis^2$ offer Paxos?

- Via the SafeSend API mentioned last time
  - SafeSend is a genuine Paxos implementation
  - But it does have some optimizations
  - And it has an unlogged mode. For Paxos durability you need to enable the logged feature.

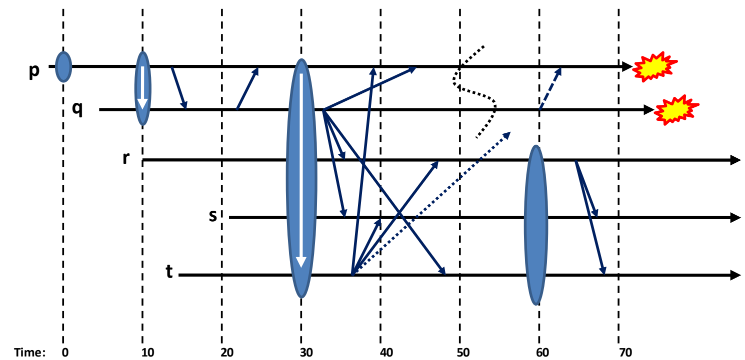
- In normal Paxos we don't have a GMS
  - With a GMS the protocol simplifies slightly and we can relax the quorum rules
  - SafeSend includes these performance enhancements but they don't impact the correctness or properties of sol'n

Consistency model: Virtual synchrony meets Paxos (and they live happily ever after…)

Non-replicated reference execution



Synchronous execution



Virtually synchronous execution

- **Virtual synchrony is a "consistency" model:**
  - *Synchronous runs: indistinguishable from non-replicated object that saw the same updates (like Paxos)*
  - *Virtually synchronous runs are indistinguishable from synchronous runs*

Is Isis^2 hard to use? Paxos was hard…

- We mentioned that just sticking Paxos in front of a set of file or database replicas is tempting, but a mistake
  - The protocol might "decide" something but this doesn't mean the database has the updates
  - Surprisingly tricky to ensure that we apply them all
- Isis^2: apply update when multicast delivered
  - This is safe and correct: all replicas do same thing
  - But it does require a state transfer to add members: we need to make a new DB copy for each new member
  - Can we do better?

Durability options

- Normal configuration of Isis^2 is optimized for "in-memory" applications.
  - State transfer: make a checkpoint, load it into a joining process, to initialize a joining group member
  - Checkpoint/reload can be used to make an entire group remember its state across shutdowns
- SafeSend, the Isis^2 version of Paxos, can be asked to log messages. This gives a stronger durability guarantee than with checkpoint/restart.

State transfer worry

- If my database is just a few Mbytes… just send it
- But in the cloud we often see databases with tens of Gbytes of content!
- Copying them will be a very costly undertaking

Out-of-Band (OOB) technology

- Allows copying big state by replication of memory-mapped files, very efficient
- There is a clever way to integrate OOB transfers with state transfer
- Effect is that with a bit more effort, Isis^2 won't need to send big objects through its multicast layer

Isis^2 DHT

- The system also has a fancy key-value store
  - Runs in a group and shards the data
  - One-hop get and put: no indirect routing needed!
  - Can even put or get multiple key-value pairs at a time, and there is a way to request totally ordered, consistent get and put: gives a form of atomicity

- Then you can do "aggregated query" operations to leverage the resulting parallel computing opportunity

GridCloud: Example Isis^2 application

GridCloud Cloud-hosted high-assurance system to monitor the electric power grid

sponsored by the Department of Energy ARPA-E program

Goal

Demonstrate a cloud-scale monitoring infrastructure able to host "smart grid" applications: the code that will make the power grid "smart"

Use cases

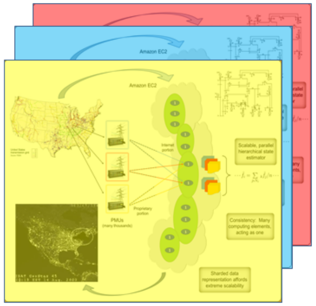
- Routine balancing of loads and generation
- Grid protection
- Analysis and adaptation after topology changes
- Integration of renewable energy

Challenges

Cloud lacks consistency, assurance, and timing guarantees. Industry demands very strong control over data flow with provable security.

Status

We're using Isis^2 to manage a structure in which replicated data permits high assurance reactive smart-grid monitoring and control. GridCloud features state estimation and GridStat software from Washington State University.

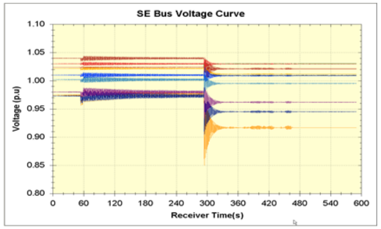


GridCloud enables

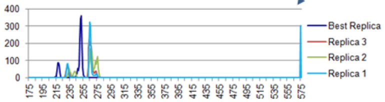
Large-scale, distributed power system applications in the cloud

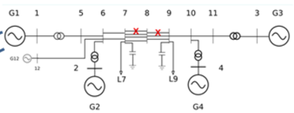
Low-latency and high-reliability using replicated data streams and application processes

- Key GridCloud technologies
  - ISIS and Dmake to manage a large number of processes running in the cloud
  - GridStat to provide multi-cast and rate filtering
  - Hierarchical Linear State Estimator (HLSE) as example application

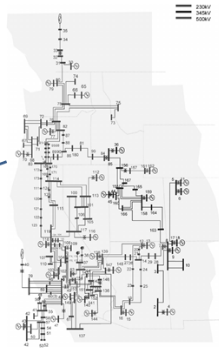


The HLSE uses EC2 Cloud resources to quickly and reliably do full-system SE 5 times per second with thousands of PMU data streams as input





	Aggregate Data Rate (Mb/s)	Total Connections	Substation SEs
Mar '13 12-bus (1 replica)	2.92	98	12
12-bus (3 replicas)	**8.75**	**294**	**36**
Oct '13 179-bus (1 replica)	46.93	1,577	127
179-bus (3 replicas)	**140.79**	**4,731**	**381**
Spring '14 6K-bus (3 replicas)	~570	~18,000	~3,000



- 2014 goal: Demonstrate GridCloud in a real-world setting, such as the regional transmission network for the North East USA

Definitions

PMU

A sensor (synchrophasor) used to measure voltage and phase angle of a power bus

Linear State Estimator (LSE)

Code that computes the state of a power grid using PMU data as input

Cornell University


Ken Birman


Robbert van Renesse


Zhiyuan Teo

Washington State University


Dave Bakken


Anjan Bose


Carl Hauser

www.cs.cornell.edu/Projects/Gridcontrol/

Cornell University

Washington State University
World Class. Face to Face.

Summary

- Group communication offers a nice way to replicate an application
  - Replicated data (without the cost of quorums)
  - Coordinated and replicated processing of requests
  - Automatic leader election, member ranking
  - Automated failure handling, help getting external database caught up after a crash
  - Tools for security and other aspects that can be pretty hard to implement by hand