CS5412: VIRTUAL SYNCHRONY
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
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
**Isis**² is a *library* for group communication

**It Uses a Formal model**
- Formal model permits us to achieve correctness
- Isis² is too complex to use formal methods as a development tool, but does facilitate debugging (model checking)
- Think of Isis² as a collection of modules, each with rigorously stated properties

**It Reflects Sound Engineering**
- Isis² implementation needs to be fast, lean, easy to use
- Developer must see it as easier to use Isis² than to build from scratch
- Seek great performance under “cloudy conditions”
- Forced to anticipate many styles of use

CS5412 Spring 2012 (Cloud Computing: Birman)
Isis$^2$ makes developer’s life easier

```csharp
Group g = new Group("myGroup");
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) {
    Reply(Values[s]);
};
g.Join();
g.Send(UPDATE, "Harry", 20.75);

List<double> resultList = new List<double>;
nr = g.Query(LOOKUP, ALL, "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 as seen for event upcalls and the assumptions user can make
Isis² makes developer’s life easier

```csharp
Group g = new Group("myGroup");
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) {
    Reply(Values[s]);
};
g.Join();
g.Send(UPDATE, "Harry", 20.75);
List<double> resultlist = new List<double>;
nr = g.Query(LOOKUP, ALL, "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
Isis\(^2\) makes developer’s life easier

Group g = new Group(“myGroup”);
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) {
    Reply(Values[s]);
};

g.Join();

g.Send(UPDATE, “Harry”, 20.75);

List<double> resultlist = new List<double>;

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
Isis\(^2\) makes developer’s life easier

```csharp
Group g = new Group("myGroup");
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) {
    Reply(Values[s]);
};
g.Join();
g.Send(UPDATE, "Harry", 20.75);
```

- 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
Isis\(^2\) makes developer’s life easier

```csharp
Group g = new Group("myGroup");
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) {
    Reply(Values[s]);
};
g.Join();
g.Send(UPDATE, "Harry", 20.75);

List<double> resultlist = new List<double>;
```

- 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
Isis$^2$ makes developer’s life easier

```csharp
Group g = new Group("myGroup");
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) {
    Reply(Values[s]);
};
g.Join();
g.Send(UPDATE, "Harry", 20.75);
List<double> resultlist = new List<double>();
nr = g.Query(LOOKUP, ALL, "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
It takes a “community”

- A lot of complexity lurks behind those simple APIs
- Building one of your own would be hard
- Isis\(^2\) took Ken 3 years to implement & debug

**Isis\(^2\) library**

- Group instances and multicast protocols
- Reliable Sending
- Fragmentation
- Security
- TCP tunnels (overlay)

**Views**

- Membership Oracle
- Group membership
- Oracle Membership
- Self-stabilizing Bootstrap Protocol

**Isis\(^2\) user object**

- Send
- CausalSend
- OrderedSend
- SafeSend
- Query....

**Flow Control**

**Other group members**

**Message Library**

- "Wrapped" locks
- Bounded Buffers

**Large Group Layer**

- Sense Runtime Environment
- Socket Mgt/Send/Rcv
- Report suspected failures

**Dr. Multicast**

- Security

**Self-stabilizing**

- Bootstrap Protocol
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

CS5412 Spring 2012 (Cloud Computing: Birman)
You build your program and link with Isis²

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

- 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…
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.
P requests: I wish to join or create group "X".

GMS responds: Group X created with you as the only member.

T to GMS: What is current membership for group X?

GMS to T: X = \{p\}

Q joins, now X = \{p,q\}. Since p is the oldest prior member, it does a state transfer to q.

GMS notices that q has failed (or q decides to leave).

Q joins...
**Group membership service**

- Runs on some sensible place, like the first few machines that start up when you launch Isis²

- **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
Let’s start by focusing on how GMS tracks its own membership. Since it can’t just ask the GMS to do this purpose. The GMS is a group too. We’ll build it first and then will use it when building reliable multicast protocols. In fact it will end up using those reliable multicast to replicate membership information for other groups that rely on it.
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 GMP.
  - Others can’t run the GMP; they report events to the leader.
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
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
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 $\alpha$)

- 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

- Runs with a well-defined leader; Paxos didn’t need one (in Paxos we often prefer to have a leader but correctness is ensured with multiple coordinators)

- Very similar guarantees of ordering and durability

- Isis GMS protocol predates Paxos
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 any 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

CS5412 Spring 2012 (Cloud Computing: Birman)
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 **all** 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
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 “fbcast”
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
Does Isis\(^2\) offer Paxos?

- Yes! Via the SafeSend API mentioned last time
  - SafeSend is a genuine Paxos implementation
  - But it does have some optimizations

- 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
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
How about the “gotcha” from last time?

- Recall 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

- Isis²: 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?
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
With SafeSend can do better

- Isis$^2$ has the “DiskLogger” mentioned last time
  - It deals with catching a database up if it was out of the group for a while and missed updates
  - Each update gets delivered at least once
  - DB must filter duplicates

- Another option is to build a fancier state transfer
  - E.g. get it almost caught up “offline”
  - Then do the last small delta of state as a final step
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