WHAT ARE THE RIGHT ROLES FOR FORMAL METHODS IN HIGH ASSURANCE CLOUD COMPUTING?

DARPA MRC
May 2012

Ken Birman, Cornell Universty

Isis^2 System

- C# library (but callable from any .NET language) offering replication techniques for cloud computing developers
- Based on a model that fuses virtual synchrony and state machine replication models
- Research challenges center on creating protocols that function well despite cloud "events"

- Elasticity (sudden scale changes)
- Potentially heavily loads
- High node failure rates
- Concurrent (multithreaded) apps

- Long scheduling delays, resource contention
- Bursts of message loss
- Need for very rapid response times
- Community skeptical of "assurance properties"

Isis^2 makes developer's life easier

Benefits of Using Formal model

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

Importance of 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) {
         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 aseen for event upcalls and the assumptions user can make

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) {
       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");
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) {
        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”);
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) {
      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");
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) {
       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");
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) {
        Reply(Values[s]);
};
g.SetSecure(myKey);
g.Join();

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

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

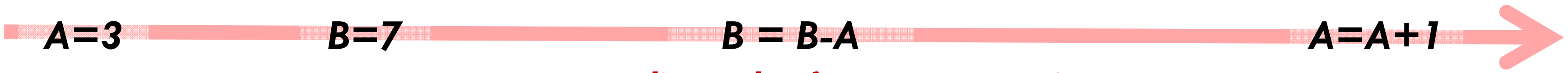
- First sets up group
- Join makes this entity a member. State transfer isn't shown
- Then can multicast, query. Runtime callbacks to the "delegates" as events arrive
- **Easy to request security, persistence, tunnelling on TCP...**
- **"Consistency" model dictates the ordering seen for event upcalls and the assumptions user can make**

Possible roles for formal methods

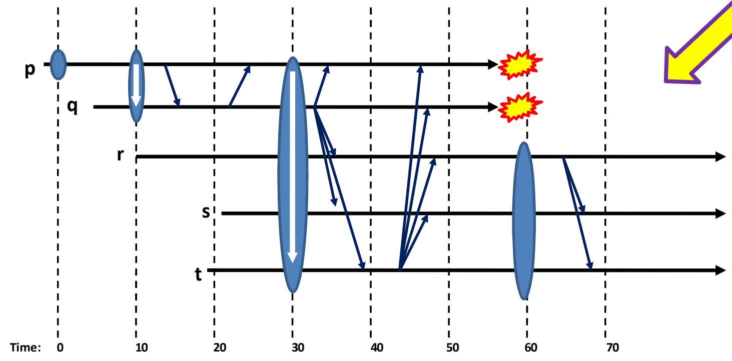
- Use theorem proving tools to formalize the execution model
- Work with CRASH tools to derive versions of protocols such as Paxos that are optimized for this model
- Use those proofs as the basis for better specifications
- Use the specifications to help the developer use the resulting replication solutions correctly
- Prove to the owner of a data center that when deployed on 1000's of nodes, the technology won't disrupt other users
- Prove resilience to failures/attacks

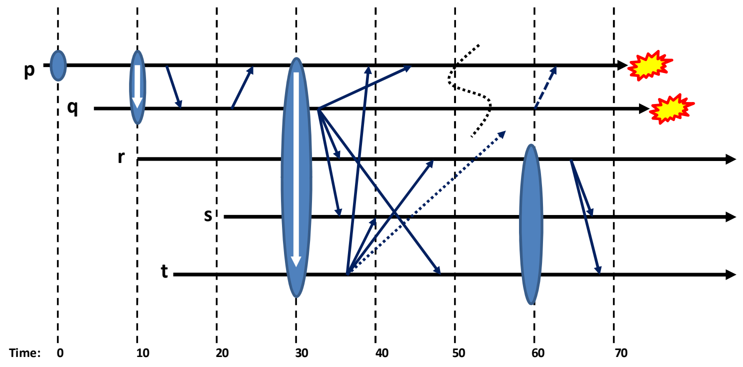
Consitency model: Virtual synchrony meets Paxos (and they live happily ever after…)

- **Virtual synchrony is a "consistency" model:**
  - ***Membership epochs:*** ***begin when a new configuration is installed and reported by delivery of a new "view" and associated state***
  - ***Protocols run "during" a single epoch:*** ***rather than overcome failure, we reconfigure when a failure occurs***



Non-replicated reference execution





Synchronous execution

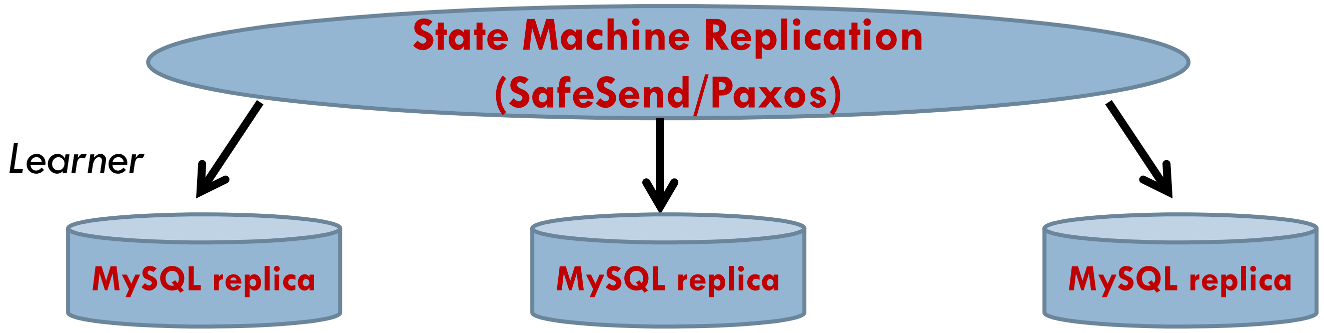
Virtually synchronous execution

How can we formalize such a model?

- We've done so in work with Dahlia Malkhi that builds on work by Lamport, Malkhi and Zhou
- We arrive at a way to formalize the picture we just saw and to reason about protocols that run in such an environment
- This becomes the basis for using NuPRL as a formal tool to solve our protocol challenges

Does our formal model actually help?

- Let's consider a use case
  - Take the replicated key/value code from 3 slides back
  - Turn it into a "replicated MySQL database" using state machine replication (Lamport's model)



Steps

- Modify the “view handler” to bind to a replica
  - A full-blown version would handle arbitrary membership changes
  - We’ll oversimplify and just do a database “import” using the rank of the member to select the replica
- Modify the “update handler” to to a DB update
- Modify the “lookup handler” to do a query
- Change g.Send into g.SafeSend (a version of Paxos)

Start with our old code...

```
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) {
         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);
```

How wo

```
Group g = new G
g.ViewHandle        dele
    IMPORT "db-replica:"+v.GetMyRank();
};
g.Handlers[UPDATE] += delegate(string s, double v)
{
    START TRANSACTION;
    UPDATE salary = v WHERE SET
    COMMIT;
};
...

g.SafeSend(UPDATE, "Harry", "85
```

We build the group as the system runs. Each participant just adds itself.

The leader monitors membership. This particular version doesn't handle failures but the "full" version is easy.

We can trust the membership. Even failure notifications reflect a system-wide consensus.

1. **Modify the view handler to bind to the appropriate**

Paxos guarantees agreement on message set, the order in which to perform actions and durability: if any

This code requires that mySQL is deterministic and that the serialization order won't be changed by QUERY operations (read-only, but they might get locks). As it happens, those assumptions are valid.

Some puzzles

- *How can we be sure that SafeSend is a correct "virtually synchronous" implementation of Paxos?*
    - We worked with Dahlia Malkhi to develop a version of Paxos optimized to exploit the virtual synchrony model. Leslie Lamport was involved at the outset of this effort.
    - Robbert later wrote this in 60 lines of Erlang. His code can be analyzed using NuPRL (CRASH-funded)
        - The protocol has optimizations relating to the virtual synchrony model that yield speedups for us
        - More or less what we actually used in Isis^2

Some puzzles

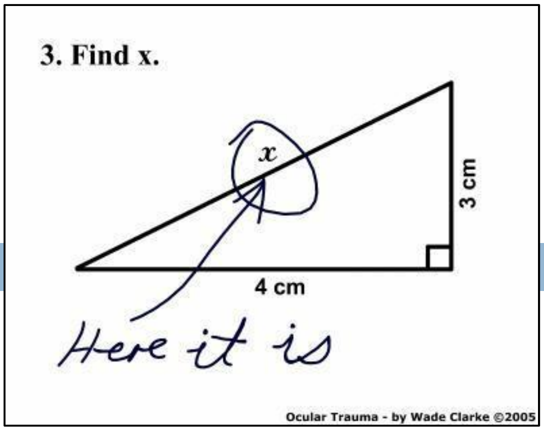
- *How do we know we're using SafeSend correctly?*
  - Raises an issue often overlooked with Paxos: the specification is self-contained but overlooks *composition* of the protocol with other components, like MySQL!
  - Surprise! This particular code sample was <u>incorrect</u>
    - It lacks recovery code needed if the replicated application is external to the replicated service
    - Learner must restart by checking for updates that MySQL lacks (doing this(in the view handler) before going online.
      - Requires either *an update log* or that operations be *idempotent*
    - Paxos correctness proofs won't help us find this bug
- Suggests that somehow the NuPRL proof was "incomplete"

The issues...

- Paxos didn't "document" the obligations of the learner, and those depend on where the state is

- With external services like MySQL Paxos requires
  - Determinism (not as trivial as you might think)
  - A way to sense which requests have completed
  - Logic to resync a recovering replica
    - This is because Paxos maintains its state in *quorums spread over* the replicas, but MySQL *needs each replica to be correct*

The issues…



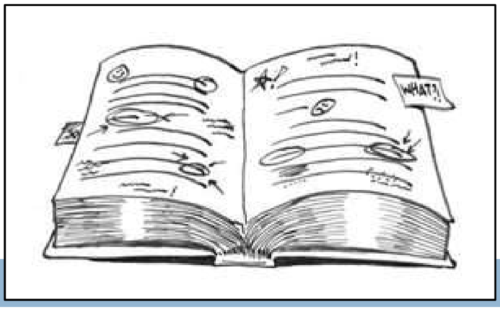
- *Where's the application state?*
  - In sample code v1, state was replicated in an in-memory data structure: each replica had a copy
  - In v2, the state is in an external service: the MySQL replicas. The replicas are just stubs
  - Yet the code looks "the same" to Isis^2

Making intent explicit

- To help the system differentiate we're adding a new "annotation" feature to Isis[2]
  - The developer indicates intent: this lets us distinguish the in-memory case from the MySQL case
  - Annotations on the view and update handlers could let us warn if code is incorrect
  - These annotations also select the fastest protocol that has the needed durability, ordering

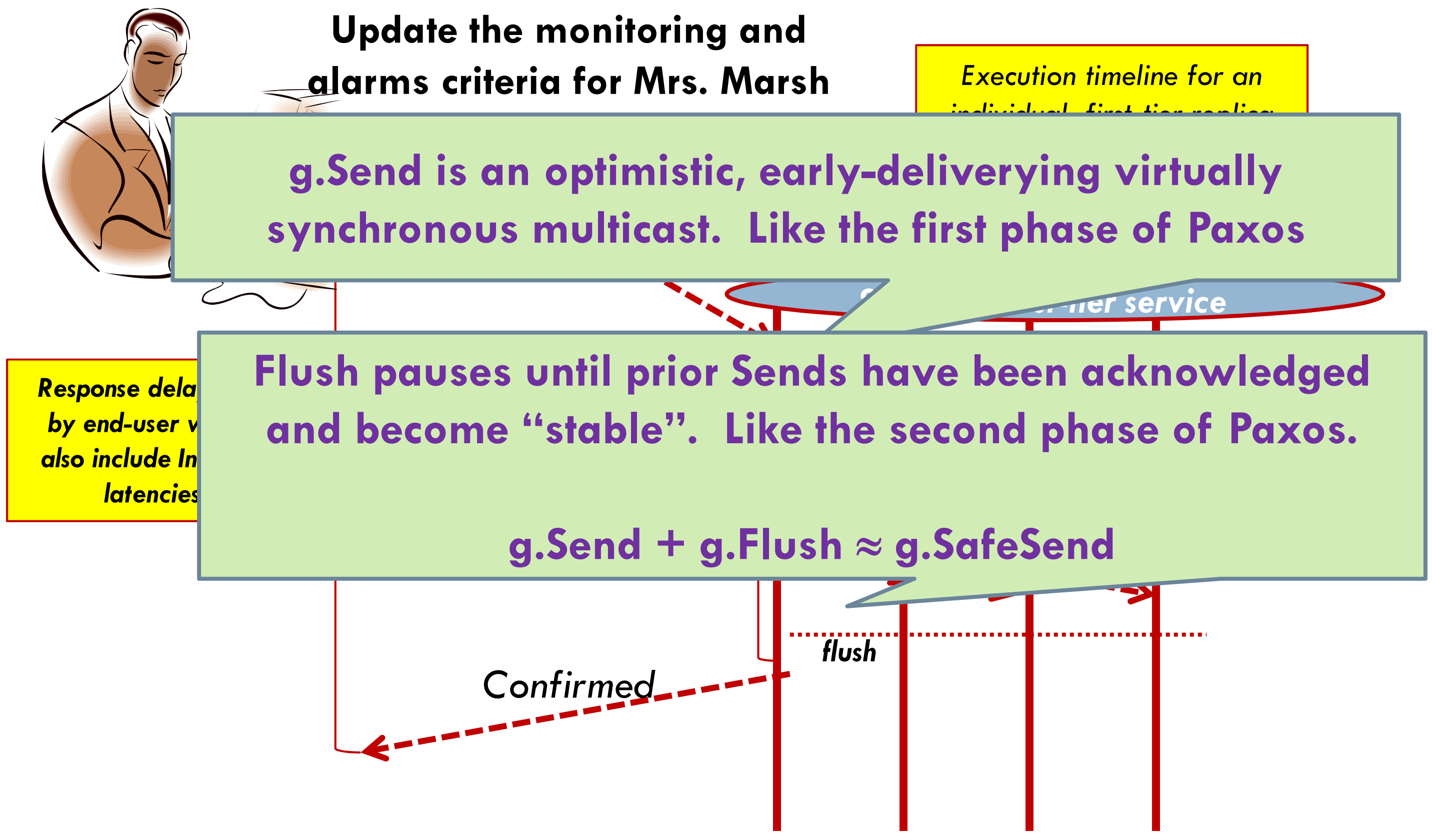
```
[FailureIndependent,HasLeader]
public class myGroup: Replicated { .... }
```

Annotation opportunities



- Annotate a whole class: can tell us whether the class is self-contained or talks to external entities
- Annotate a view handler: lets us understand how the class deals with a joining or recovering member
- Annotate an update handler: user can tell us about patterns of updates, locking
- Annotate a query handler: lets us distinguish read-only actions from those with side-effects
- … then pull all of this together via reflection when a multicast or query is issued to a group method

But a more elaborate example reveals that we'll need to do much more


Update the monitoring and alarms criteria for Mrs. Marsh
Execution timeline for an
Response dela
by end-user v
also include In
latencies
ier service
flush
Confirmed

g.Send is an optimistic, early-deliverying virtually synchronous multicast. Like the first phase of Paxos

Flush pauses until prior Sends have been acknowledged and become "stable". Like the second phase of Paxos.

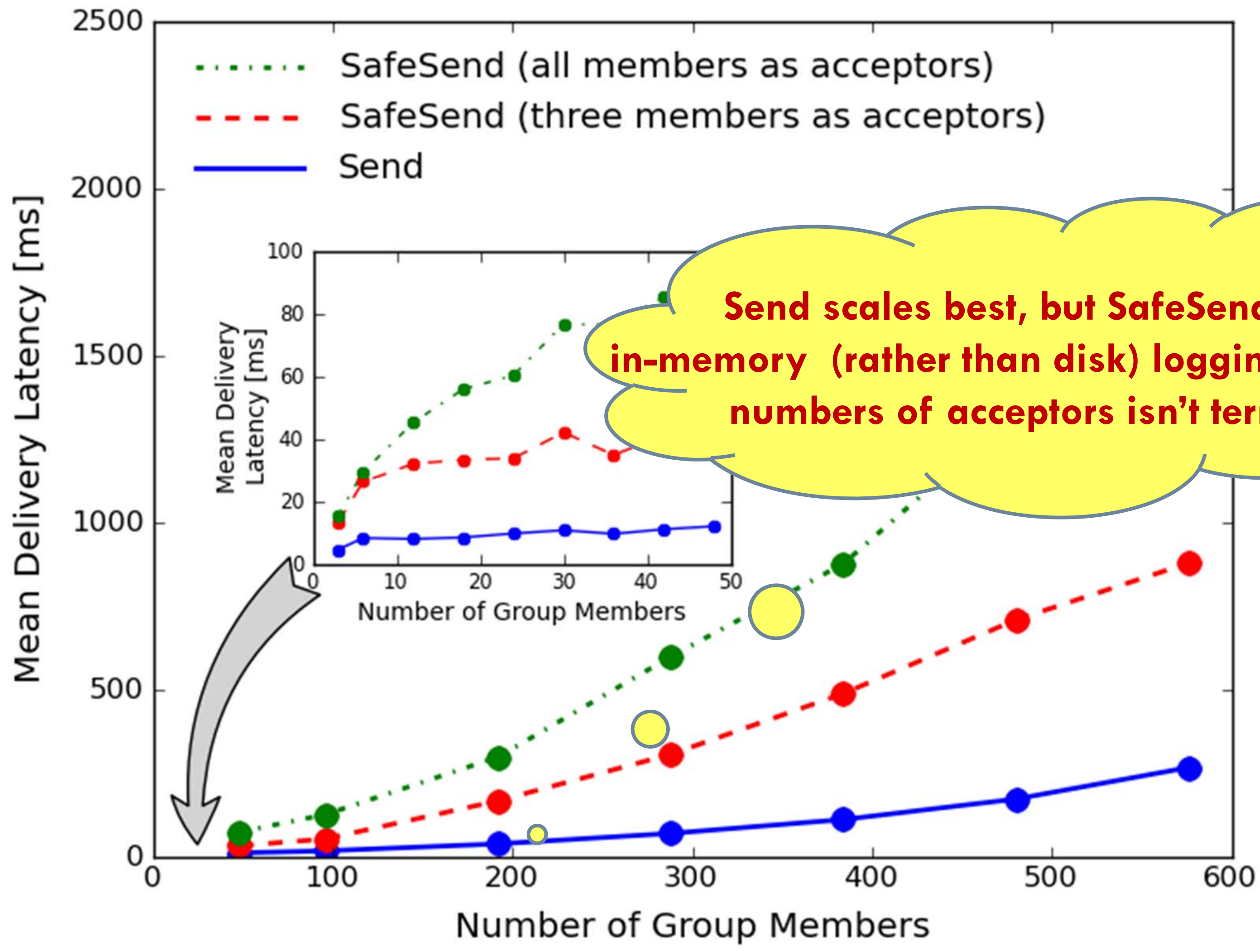
g.Send + g.Flush ≈ g.SafeSend

- Tradeoffs arise: how durable must the Send operations be? How ordered should they be?

Observations

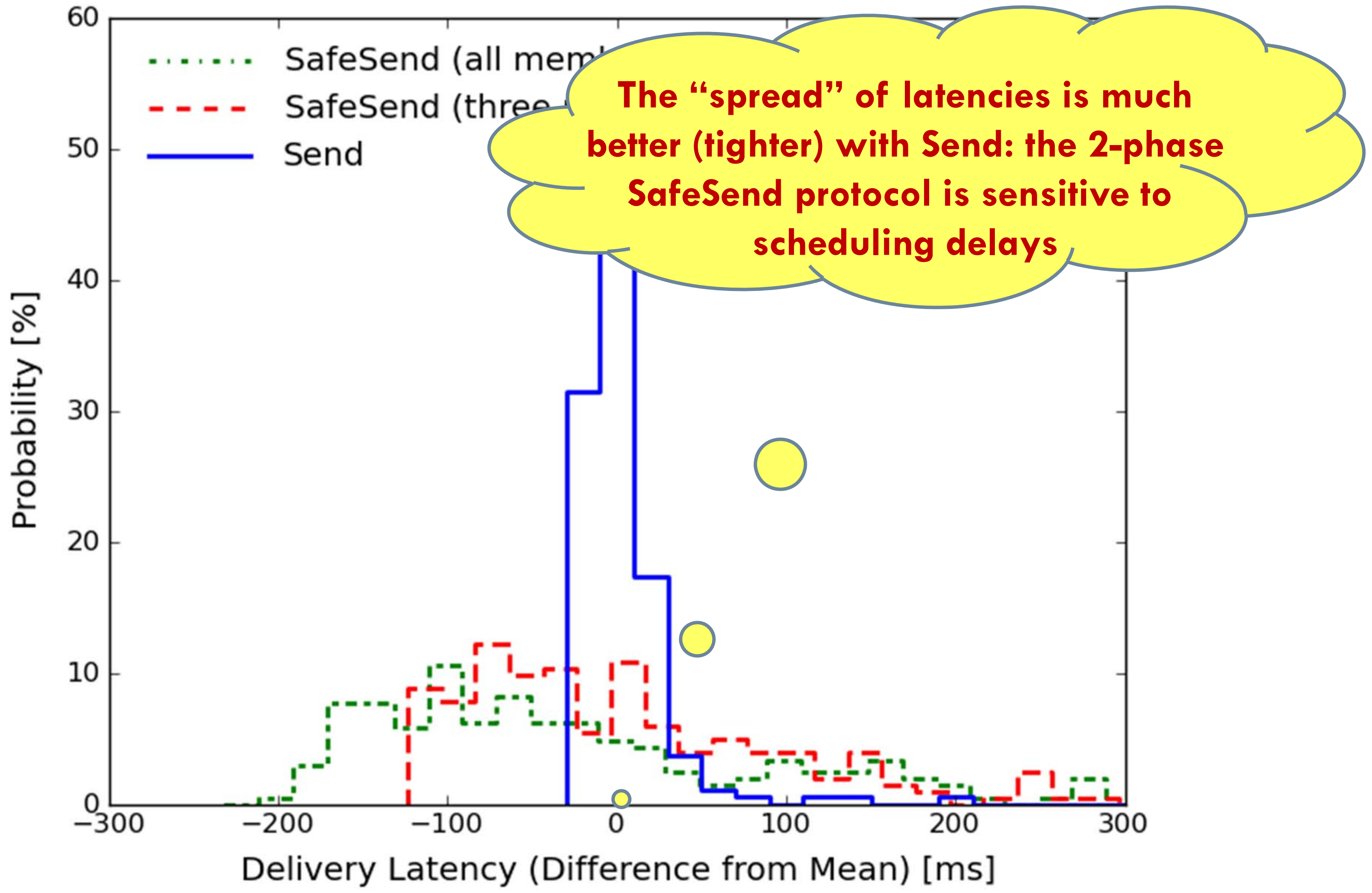
- The state being replicated here is an in-memory representation of the monitoring criteria in use now
  - Doesn’t preclude also maintaining a log for later audit
  - Our focus is on the data the group replicas maintain, so if we had a log, the question would be: where does it live, how do we interact with it, how “synchronously”?
- This particular code has a “flush barrier” before interacting with external clients: a chance to pause to make sure prior optimistic actions have finished

Isis^2: Send v.s. in-memory SafeSend



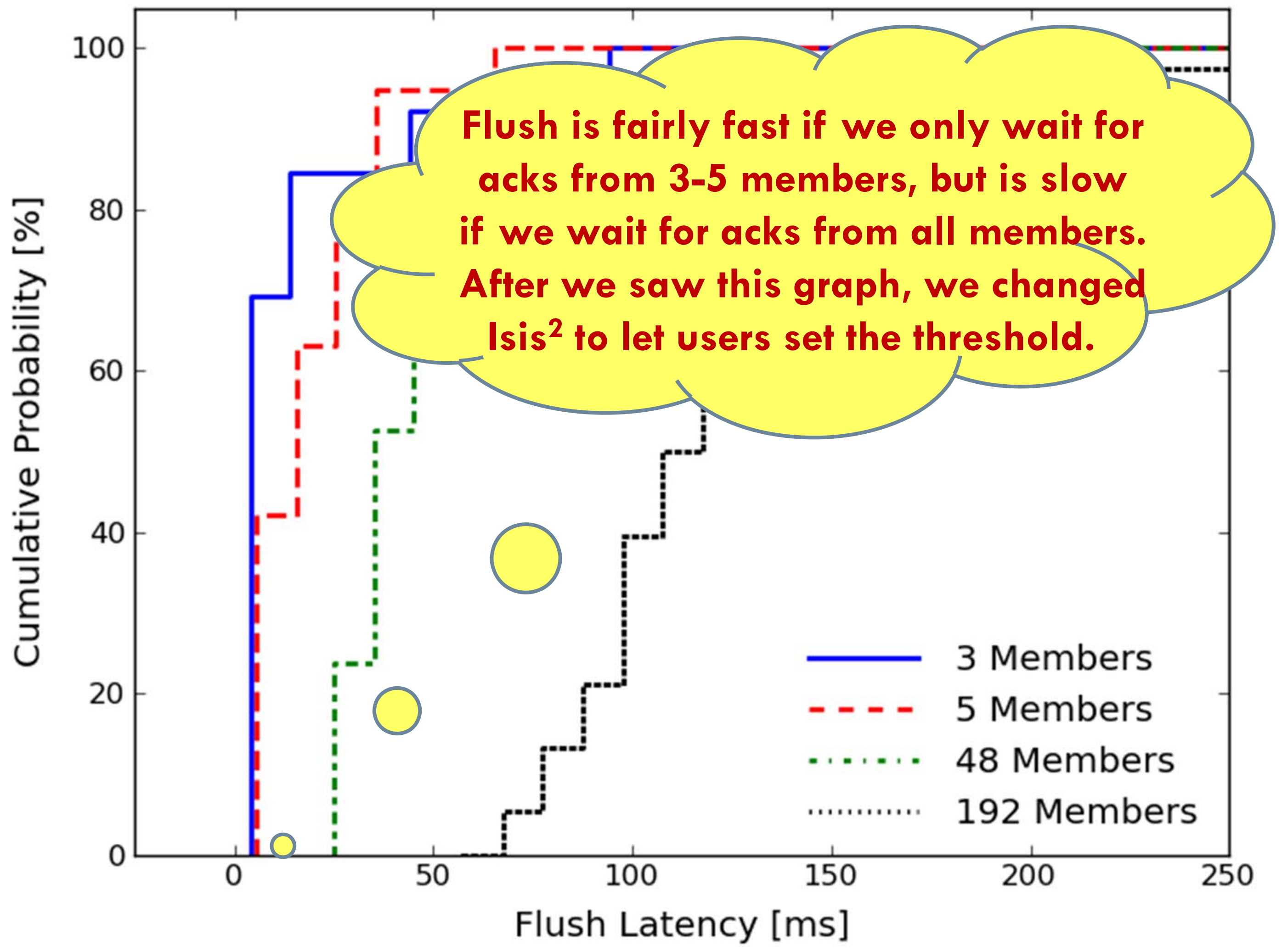
Send scales best, but SafeSend with in-memory (rather than disk) logging and small numbers of acceptors isn't terrible.

Jitter: how "steady" are latencies?



The "spread" of latencies is much better (tighter) with Send: the 2-phase SafeSend protocol is sensitive to scheduling delays

Flush delay as function of shard size



Flush is fairly fast if we only wait for acks from 3-5 members, but is slow if we wait for acks from all members. After we saw this graph, we changed Isis^2 to let users set the threshold.

How do annotations help?

- Rather than asking the developer to hard-code "SafeSend with 3 acceptors" (which involves subtle reasoning), the developer might tell us:
  - My application has durable state: [Durable/Ephemeral]
  - Updates are done from a "leader": [HasPrimary]
  - Members have uncorrelated failure behavior [FailureIndependent]
  - [FlushesBeforeExternalActions], etc…
- From this we can warn on errors, and if none is sensed, can pick the fastest applicable option.

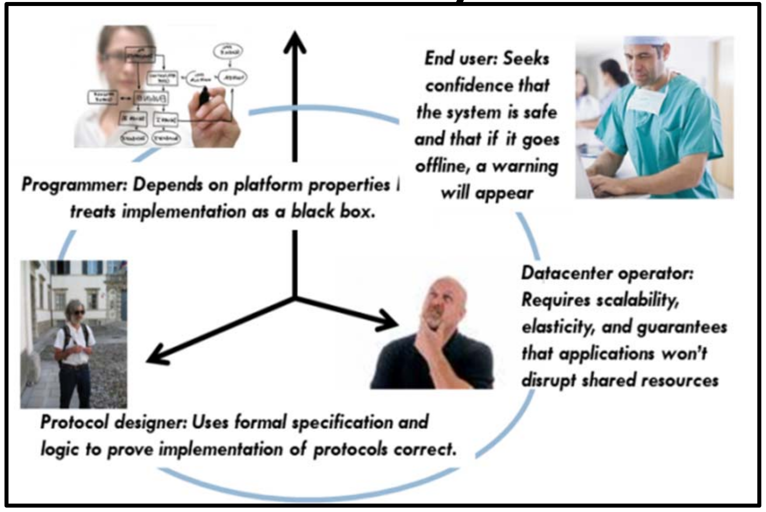
Are we there yet?

On the family trip to nirvana

- Do annotations tell the whole story?
  - What about flow control… use of g.SetSecure if required… fragmentation of massive messages… tunnelling over TCP if IPMC isn't available… stability at scale when under stress… managing IPMC address space if IPMC is permitted?
- More annotations could help for some of these
- But other aspects point to a new insight
  - There are many "perspectives" on what a system should do
  - Our high assurance story up to now focuses purely on the developer getting it right!

Who's asking the question?

- A single system needs to tell multiple kinds of assurance stories and not all in the same way
- Current High Assurance Formalisms try to reduce a complex story to a single specification and a single proof

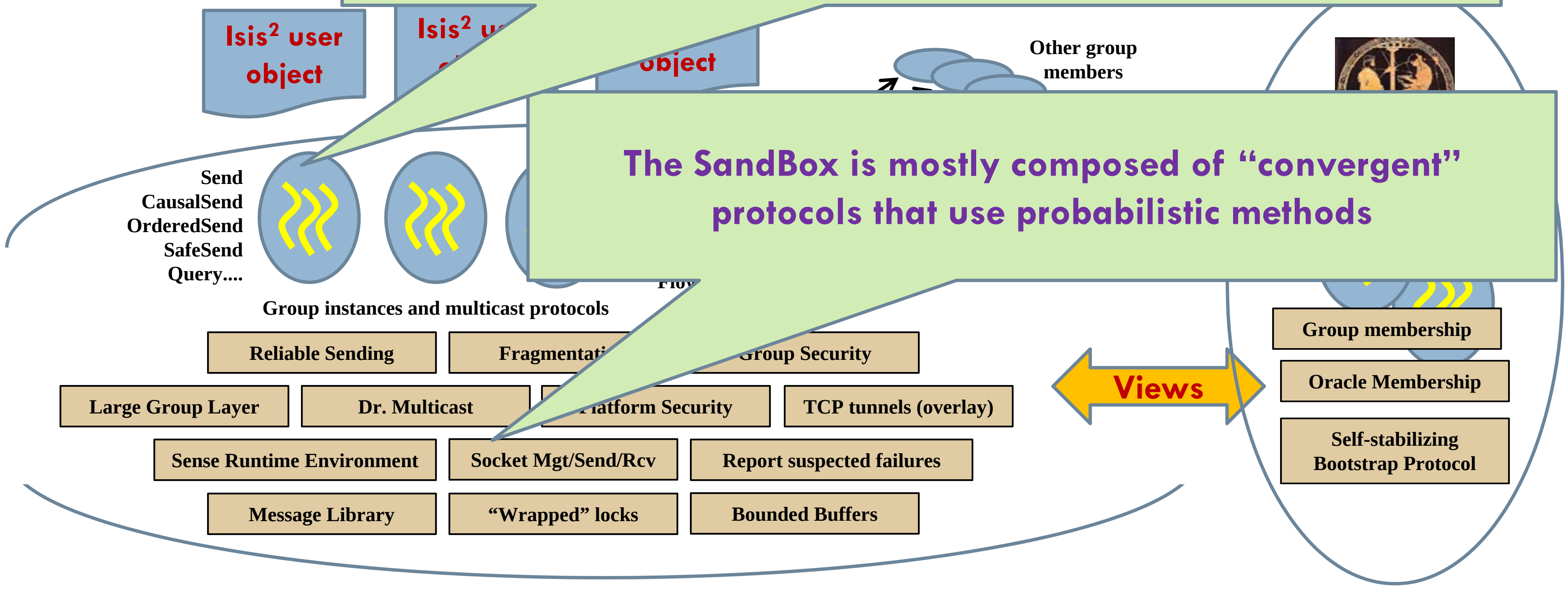

End user: Seeks confidence that the system is safe and that if it goes offline, a warning will appear
Programmer: Depends on platform properties I treats implementation as a black box.
Datacenter operator: Requires scalability, elasticity, and guarantees that applications won't disrupt shared resources
Protocol designer: Uses formal specification and logic to prove implementation of protocols correct.

- In fact we need to learn to think about high assurance in terms of distinct perspectives

Telling so many stories takes a "community"

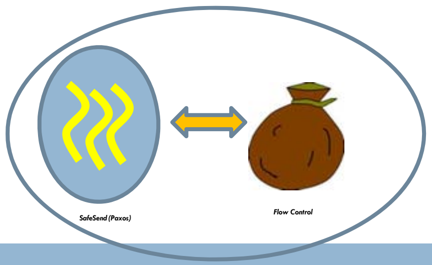
SafeSend and Send are two of the protocol components hosted over the sandbox. These share flow control, security, etc

The SandBox is mostly composed of "convergent" protocols that use probabilistic methods



- We structured Isis² as a sandbox containing protocol objects
- The sandbox provides system-wide properties; protocols like SafeSend "refine" it with additional behaviors.
- Like aspect-oriented programming applied to high assurance

Drill down: Flow control

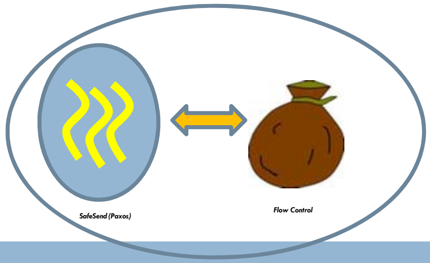


- Consider SafeSend (Paxos) within Isis^2
  - Basic protocol looks very elegant
  - Not so different from Robbert's 60 lines of Erlang

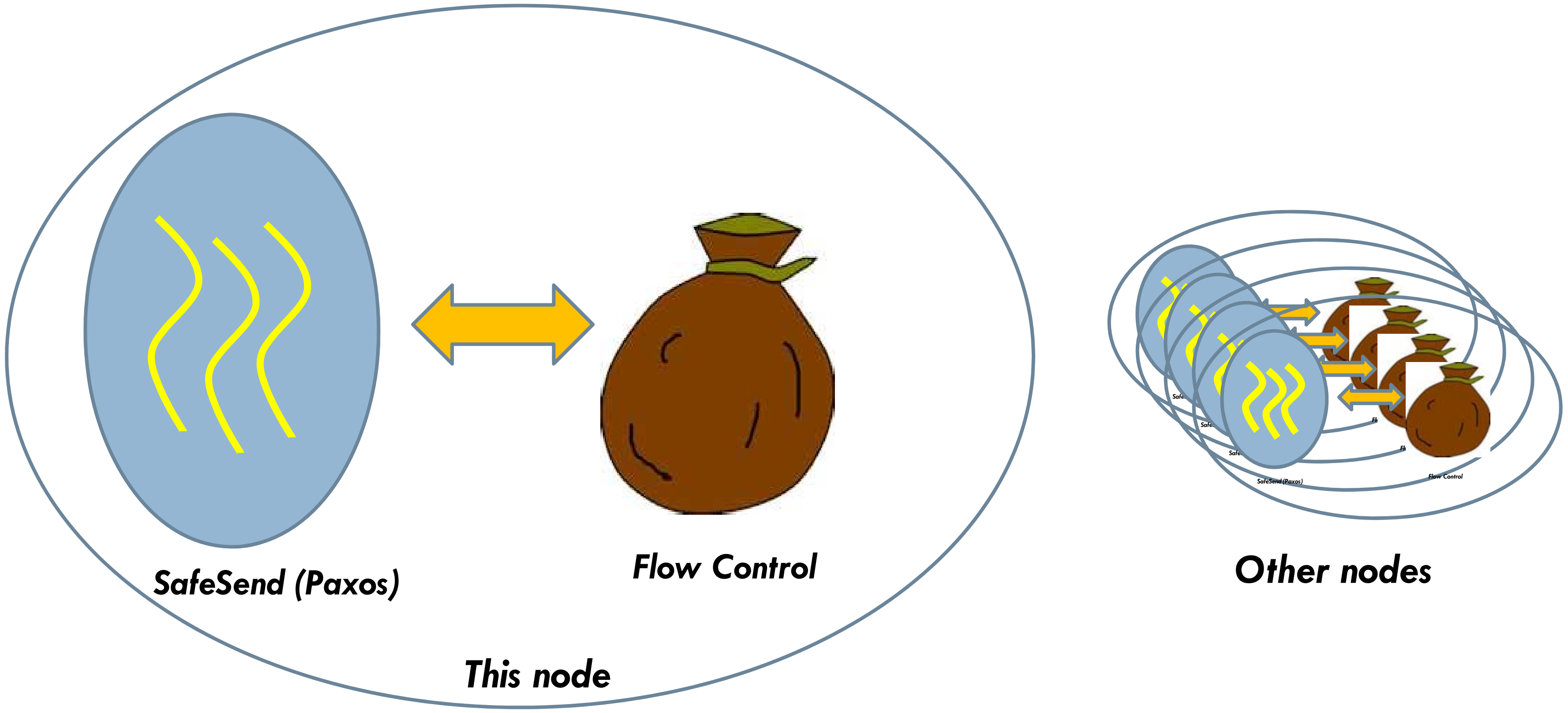
- But pragmatic details clutter this elegant solution
  - E.g.: Need "permission to send" from flow-control module
  - ... later tell flow-control that we've finished

- Flow control is needed to prevent overload
  - Illustrates a sense in which Paxos is "underspecified"

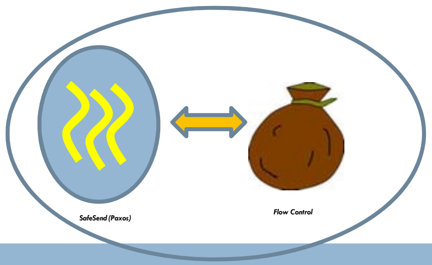
Pictoral representation



- "Paxos" state depends on "flow control state"
- Modules are concurrent. "State" spans whole group



… flow control isn’t local



- One often thinks of flow control as if the task is a local one: “don’t send if my backlog is large”
- But actual requirement turns out to be distributed
  - “Don’t send if the *system as a whole* is congested”
    - Permission to initiate a SafeSend obtains a “token” representing a unit of backlog at this process
    - Completed SafeSend must return the token
- Flow Control is a non-trivial distributed protocol!
  - Our version uses a gossip mechanism

Other elements of the Sandbox

- Vigfusson’s “Dr. Multicast” algorithm for IPMC address space management
- Fragmentation for large messages
- The platform level and per-group security key management layer
- The TCP tunnelling logic
- … and a few more

- Several use gossip-based algorithms

Lessons one learns… and challenges

- Formal models are powerful conceptual tools
  - Impossible to build a system like Isis^2 without them
  - But we know more about using them for modules than we do about composition of those modules

- The need seems to be for a form of aspect-oriented high assurance "proof"
  - Some way to prove things module by module
  - But also to think about what happens when they interact

The challenge?

- Which road leads forward?
  1. Extend our formal execution model to cover all elements of the desired solution: a "formal system"
  2. Develop new formal tools for dealing with complexities of systems built as communities of models
  3. Explore completely new kinds of formal models that might let us step entirely out of the box

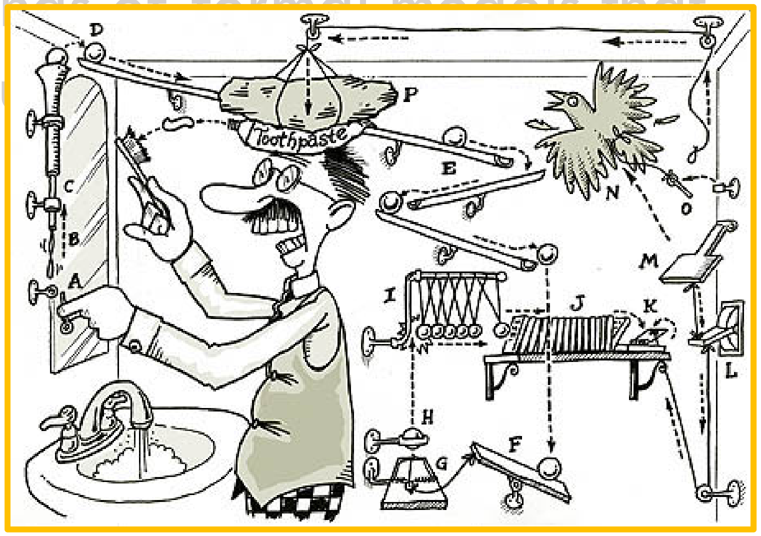
The challenge?

- Which road leads forward?
  1. Extend our formal execution model to cover all elements of the desired solution: a “formal system”

Doubtful:
- **The resulting formal model would be unwieldy**
- **Theorem proving obligations rise more than linearly in model size**

els

  3. Explore completely new kinds of formal models that might let us step entirely o



The challenge?

- Which road leads forward?
  1. Extend our formal execution model to cover all elements of the desired solution: a "formal system"
  2. Develop new formal tools for dealing with complexities of systems built as communities of models
  3. Explore completely new kinds of formal models that

Our current focus:

- **Need to abstract behaviors of these complex "modules"**
- **On the other hand, this is how one debugs platforms like Isis^2**

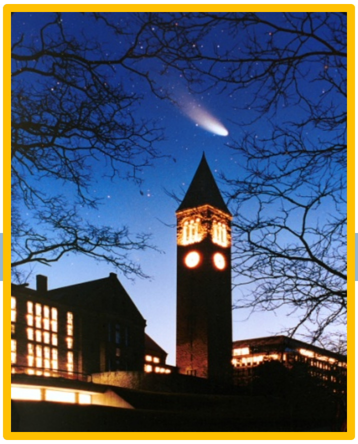
The challenge?

- Which road leads forward?
  1. Extend our formal execution model to cover all elements of the desired solution: a "formal system"
  2. Develop new formal tools for dealing with complexities of systems built as communities of models
  3. Explore completely new kinds of formal models that might let us step entirely out of the box

Intriguing future topic:

- **All of this was predicated on a style of deterministic, agreement-based model**
- **Could self-stabilizing protocols be composed in ways that permit us to tackle equally complex applications but in an inherently simpler manner?**

Summary?



- The word on the street is that cloud computing will rule but that the cloud can't do high assurance because assurance "methodologies" oversimplify & are non-scalable
- At Cornell, my group just doesn't accept either limitation
  - Isis^2 is our proof-by-demonstration that it can be done
  - Approach revolves around use of formal models and (we hope) elegant language embeddings but also demands overcoming big software engineering challenges
- Genuinely significant formal advances will require an enlarged perspective on the roles and scope of models