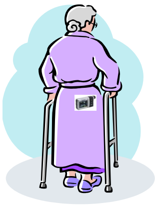
WHAT ARE THE RIGHT ROLES FOR FORMAL METHODS IN HIGH ASSURANCE CLOUD COMPUTING?

Princeton
Nov. 2012

Ken Birman, Cornell University

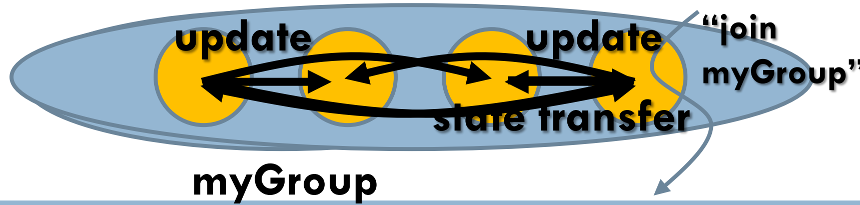
High Assurance in Cloud Settings

- A wave of applications that need high assurance is fast approaching
  - Control of the "smart" electric power grid
  - mHealth applications
  - Self-driving vehicles….



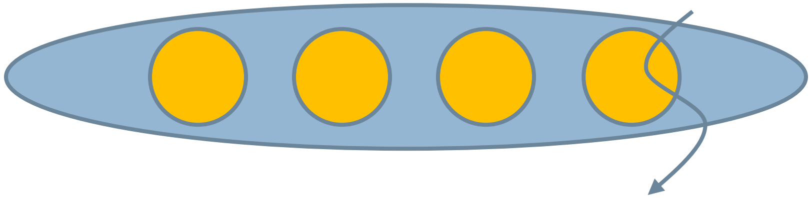
- To run these in the cloud, we'll need better tools
  - Today's cloud is inconsistent and insecure by design
  - Issues arise at every layer (client… Internet… data center) but we'll focus on the data center today

Isis^2 System



- Core functionality: *groups of objects*
  - … fault-tolerance, speed (parallelism), coordination
  - Intended for use in very large-scale settings
- The local object instance functions as a <u>*gateway*</u>
  - Read-only operations performed on local state
  - Update operations update all the replicas

Isis^2 Functionality

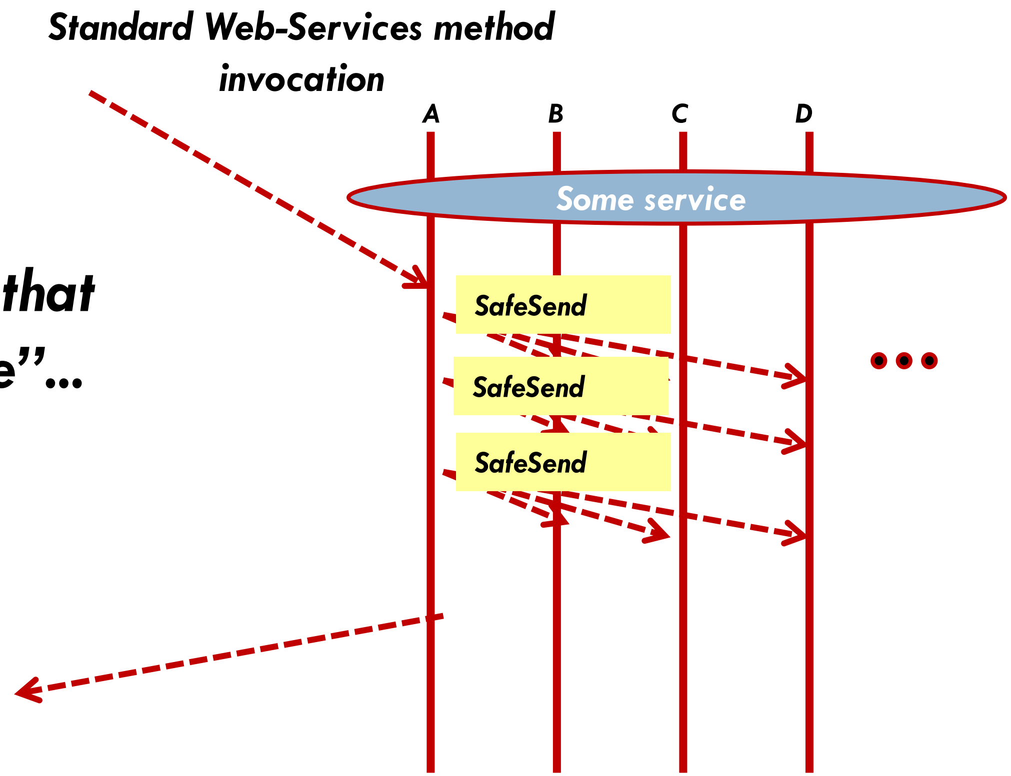


- We implement a wide range of basic functions
  - Multicast (many "flavors") to update replicated data
  - Multicast "query" to initiate parallel operations and collect the results
  - Lock-based synchronization
  - Distributed hash tables
  - Persistent storage…

- Easily integrated with application-specific logic

Example: Cloud-Hosted Service

A distributed request that updates group "state"...



... and the response

SafeSend is a version of Paxos.

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	Importance of Sound Engineering
• Formal model permits us to achieve correctness • Think of Isis^2 as a collection of modules, each with rigorously stated properties • These help in debugging (model checking)	• Isis^2 implementation needs to be fast, lean, easy to use, in many ways • Developer must see it as easier to use Isis^2 than to build from scratch • Need great performance under "cloudy conditions"

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.SafeSend(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 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) {
      g.Reply(Values[s]);
};
g.Join();

g.SafeSend(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

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.SafeSend(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

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.SafeSend(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

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.SafeSend(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

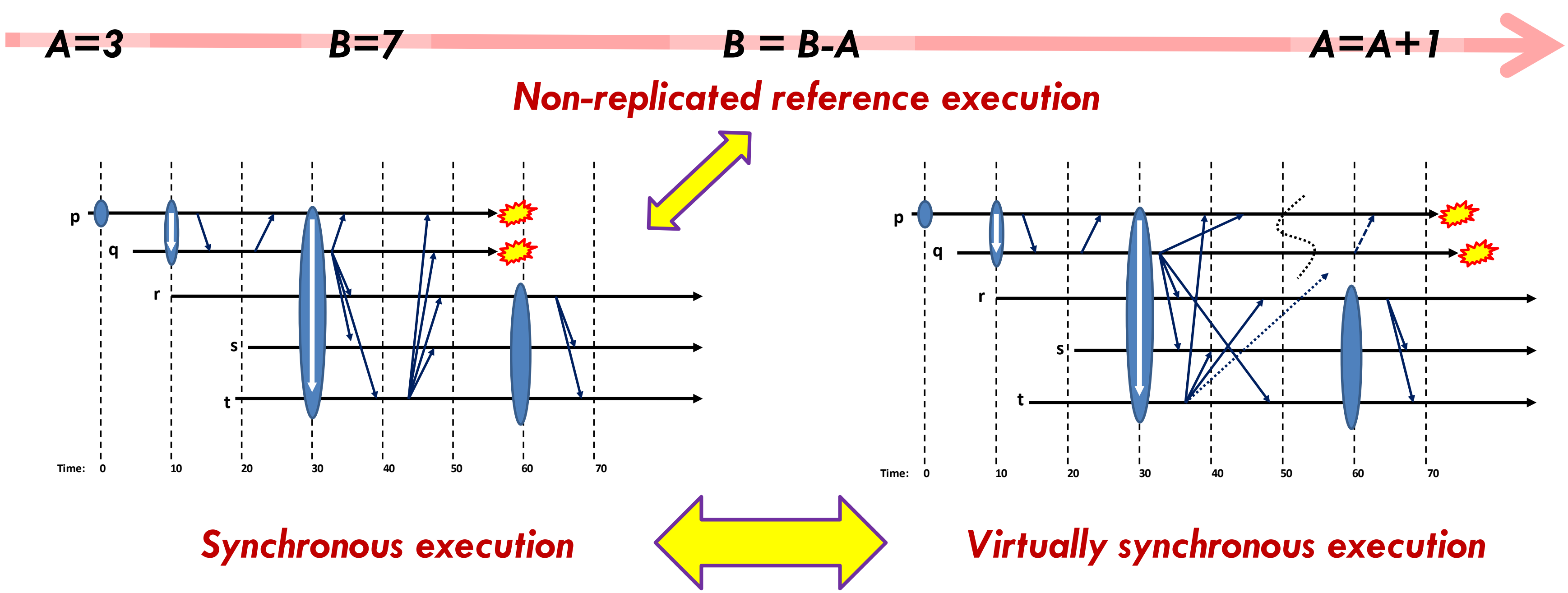
Isis^2 makes developer's life easier


```
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.SetSecure(myKey);
g.Join();

g.SafeSend(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**

Consistency 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***



Why replicate?

- High availability
- Better capacity through load-balanced read-only requests, which can be handled by a single replica
- Concurrent parallel computing on consistent data
- Fault-tolerance through "warm standby"

Do users find formal model useful?

- Developer keeps the model in mind, can easily visualize the possible executions that might arise
  - Each replica sees the same events
  - … in the same order
  - … and even sees the same membership when an event occurs. Failures or joins are reported just like multicasts

- All sorts of reasoning is dramatically simplified

… for example

- To build a locking service, like Google Chubby

  1. Define an API (commands like "lock", "unlock", ….)
  2. Service state: table of active locks & holders, wait-list
  3. On lock request/release events, invoke SafeSend
  4. Make state persistent by enabling Isis^2 checkpointing

- … and we're done! Run your service on your cluster, or rent from EC2, and you've created myChubby

Expressing virtual synchrony model in constructive logic

- Formalize notion of a reconfigurable state machine
  - It runs in epochs: a period of fixed membership
  - Simple protocols that don't need to tolerate failures.
  - If a failure occurs, or some other need arises, reconfigure by

    (1) stopping the current epoch,

    (2) forming a consensus on the new configuration, and

    (3) initializing new members as needed

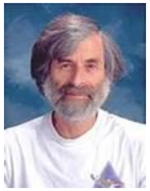
- The Isis^2 *membership Oracle* manages its own state and tracks membership for other groups

How is this formal model really used?

- We used it in developing Isis^2 itself…
  - One can use the model to reason about correctness
  - It is also possible to discover bugs by running the system under stress while watching runs for violation of the model – a form of model checking.
  - In fact we could go even further and <u>generate</u> the needed protocols from the model; more about this later

Roles for formal methods

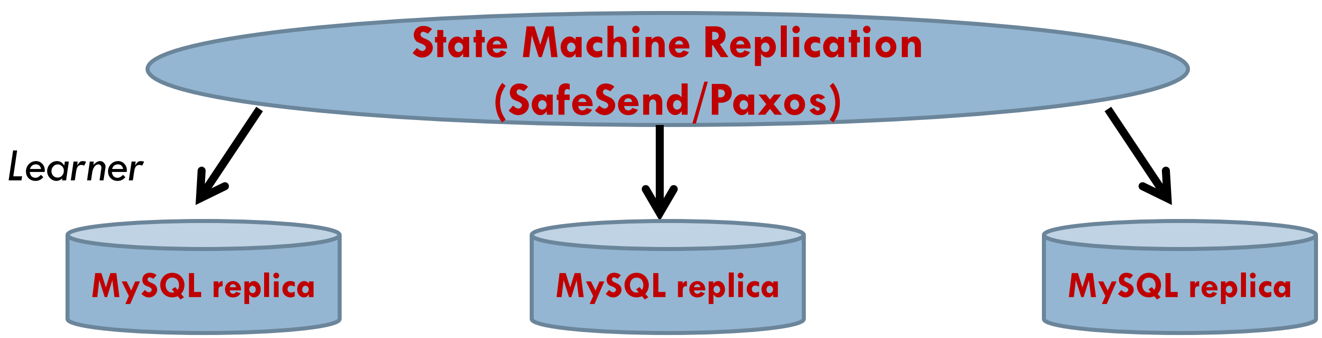
- *Proving that SafeSend is a correct "virtually synchronous" implementation of Paxos?*
  - I worked with Robbert van Renesse and Dahlia Malkhi to optimize Paxos for the virtual synchrony model.
    - Despite optimizations, protocol is still bisimulation equivalent
  - Robbert later coded it in 60 lines of Erlang. His version can be proved correct using NuPRL
  - Leslie Lamport was initially involved too. He suggested we call it "virtually synchronous Paxos".



Virtually Synchronous Methodology for Dynamic Service Replication. Ken Birman, Dahlia Malkhi, Robbert van Renesse. MSR-2010-151. November 18, 2010. Appears as Appendix A in **Guide to Reliable Distributed Systems. Building High-Assurance Applications and Cloud-Hosted Services.** Birman, K.P. 2012, XXII, 730p. 138 illus.

How would we replicate mySQL?

- We've said it simplifies reasoning… but does it?
- A more elaborate use case:
  - Take the replicated key/value code from 5 slides back (or the locking service: the code would look similar)
  - Turn it into a "replicated MySQL database" using state machine replication (Lamport's model)



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) {
      g.Values[s] = v;
};
g.Join();


g.SafeSend(UPDATE, "Harry", 20.75);
```

How wo

```
Group g = new G
g.ViewHandlers += 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
```

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

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.

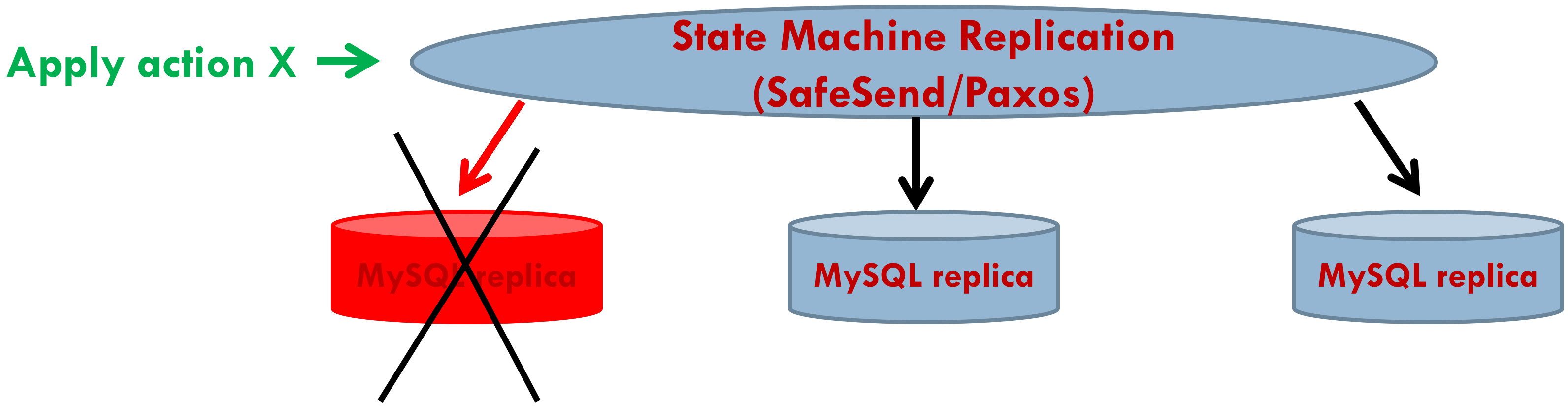
SafeSend (Paxos) guarantees agreement on message set, the order in which to perform actions and durability: if any member learns an action, every member will

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).

But now we run into a "gotcha"

- *Using SafeSend to replicate a service*
  - SafeSend is provably correct... yet our solution is wrong!
    - Our code lacked recovery logic needed because the application maintains state in an external database (file)
    - After crash, must restart by checking each replica for updates that it lacks, and applying them, before going online.

Illustration of the problem



- If one of the replicas is down, action X isn't applied to that replica. Paxos behaved correctly yet our replicas are not currently in sync
- On recovery, we're supposed to repair the damaged copy, by comparing Paxos "state" with replica "state"

A puzzle

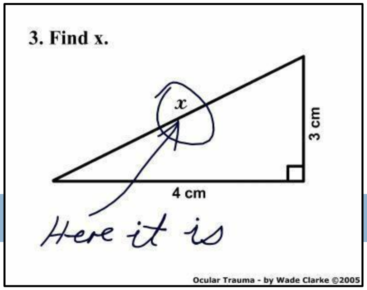
- What did the formal properties of SafeSend/Paxos "tell us" about the obligations imposed on mySQL?
  - Paxos is formalized today in an inward-looking manner.
  - Connecting Paxos to an external service is just not addressed

- Many researchers describe some service, than say "we use Paxos to replicate it", citing a public Paxos library
  - Isis^2/SafeSend can be understood as one of these
  - But one now starts to wonder: how many of those research systems are <u>incorrect</u> for the reasons just cited?

The issues…

- 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 the Paxos protocol state could be correct even if one of the mySQL replicas failed, then recovered by rolling back one of its operations, while other replicas didn't roll back that update operation
- Isis^2 includes a tool layered on SafeSend to address this… but doing so is surprisingly complicated

The fundamental issue…



- How is state maintained?
  - **Basic Paxos:** state is a replicated, durable, ordered *list* of updates, but individual replicas can have gaps.
  - **Virtually synchronous Paxos (SafeSend):** Protocol is *gap-free* hence can replicate the "state" not the list of updates.
  - **Replicated dictionary:** state was replicated in an in-memory data structure. Use checkpoint (or state transfer) at restart.
  - **Replicated MySQL:** the state is in the MySQL replicas, and survives even if the associated machine fails, then restarts.
- Our formalization of Paxos "ignores" the interaction of the Paxos protocol with the application using it

The fundamental issues...

- *How to formalize the notion of application state?*
- *How to formalize the composition of a protocol such as SafeSend with an application such as replicated mySQL?*
- No obvious answer… just (unsatisfying) options
  - A composition-based architecture: interface types (or perhaps phantom types) could signal user intentions. This is how our current tool works.
  - An annotation scheme: in-line pragmas (executable “comments”) would tell us what the user is doing
  - Some form of automated runtime code analysis

Is Isis^2 just "too general" to be useful?

- Hypothesis: object-group replication via multicast with virtual synchrony is too flexible
  - Claim: One size fits all would make developers happy
  - Leslie Lamport once believed this.. but not anymore

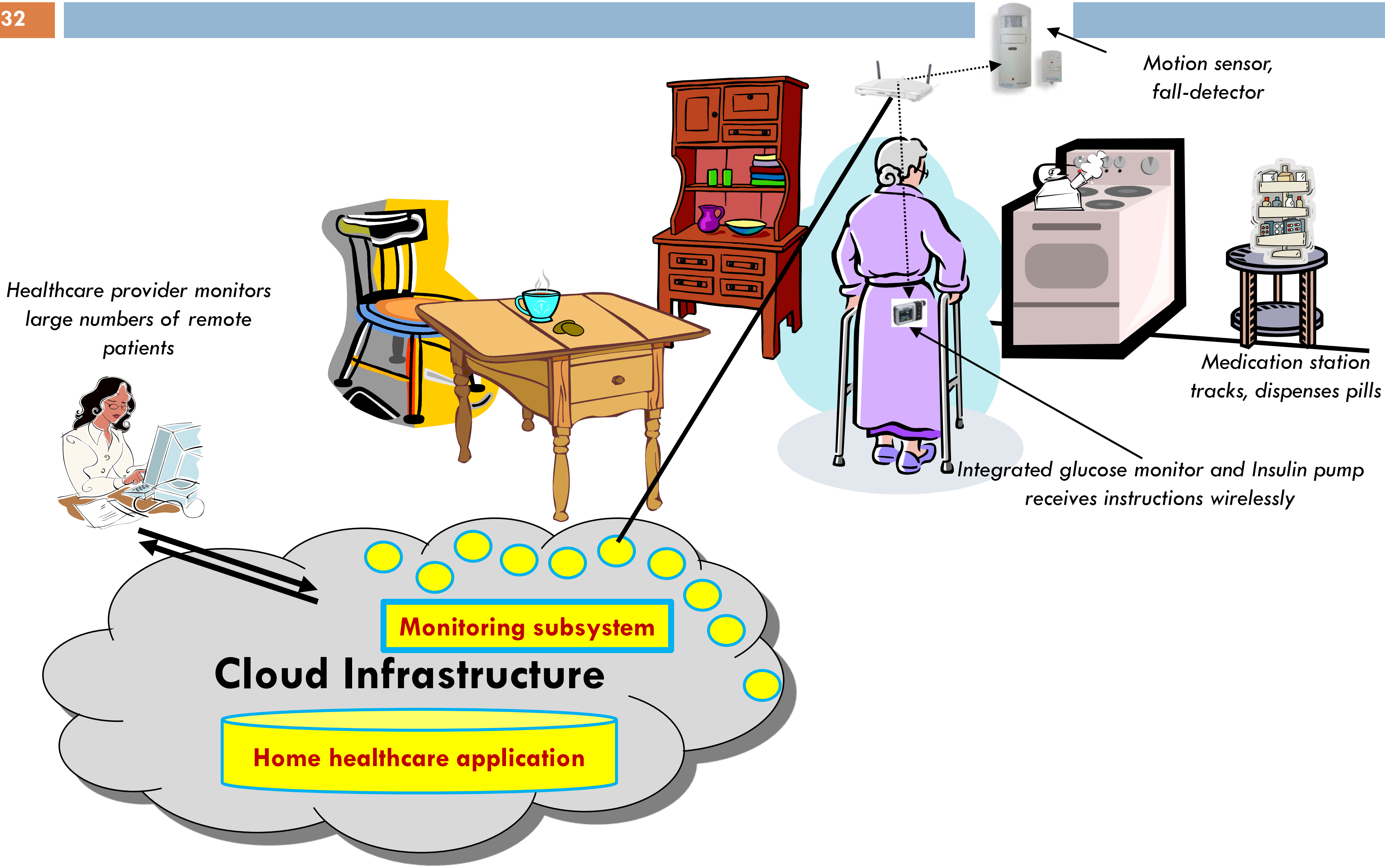
- Today there are a dozen Paxos varients!
  - Each is specialized for a particular setting
  - If a component has a performance or scale-critical role, optimizations can have huge importance

Performance demands flexibility!

- A one-size fits-all version of SafeSend wouldn't be popular with "real" cloud developers because it would lack necessary flexibility
  - Speed and elasticity are paramount
  - SafeSend is just too slow and too rigid: Basis of Brewer's famous CAP conjecture (and theorem)

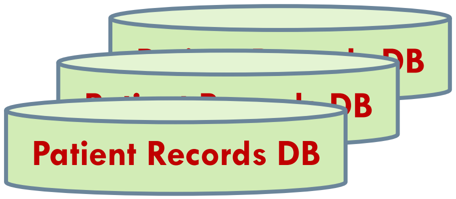
- Let's look at a use case in which being flexible is key to achieving performance and scalability

Building an online medical care system

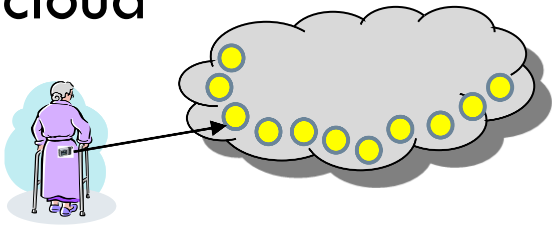


Two replication cases that arise

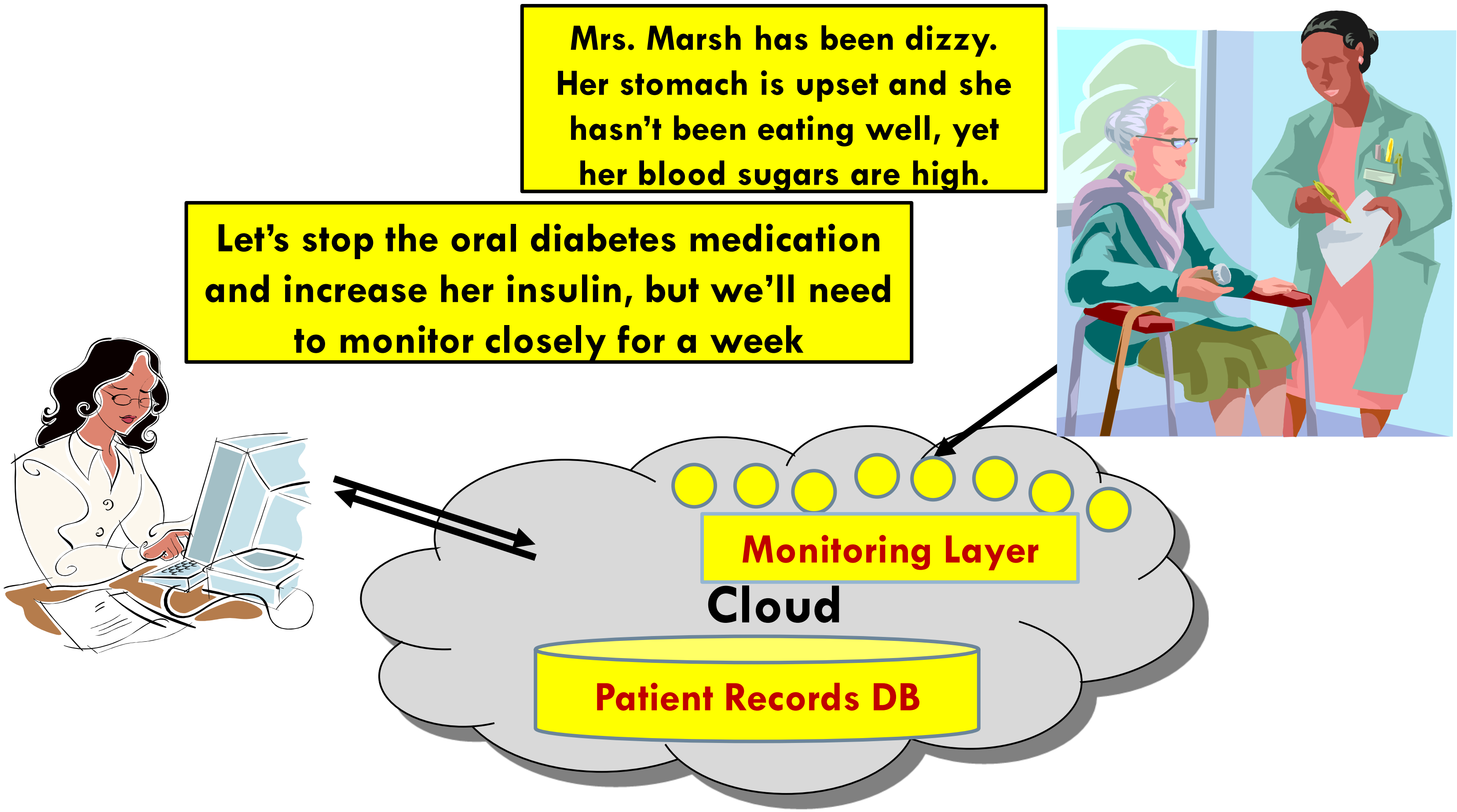
- Replicating the database of patient records
  - Goal: Availability despite crash failures, durability, consistency and security.
  - Runs in an “inner” layer of the cloud



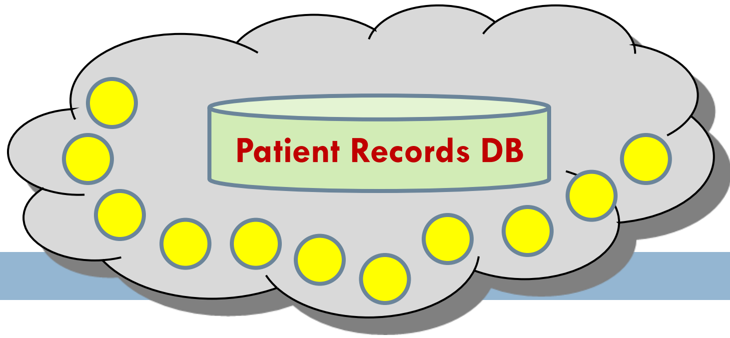
- Replicating the state of the “monitoring” framework
  - It monitors huge numbers of patients (cloud platform will monitor many, intervene rarely)
  - Goal is high availability, high capacity for “work”
  - Probably runs in the “outer tier” of the cloud



Which matters more: fast response, or durability of the data being updated?



Pay for what you use!



- Patient records database has stronger properties requirements but is under less load and needs a smaller degree of state replication

- The monitoring infrastructure needs to scale to a much larger degree, but has weaker requirements (and because it lives in the soft-state first tier of the cloud, some kinds of goals would make no sense)

- Similar pattern seen in the smart power grid, self-driving cars, many other high-assurance use cases

Real systems demand tradeoffs

- The database with medical prescription records needs strong replication with consistency and durability
  - The famous ACID properties. A good match for Paxos

- But what about the monitoring infrastructure?
  - A monitoring system is an *online* infrastructure
  - In the soft state tier of the cloud, durability isn't available
  - Paxos works hard to achieve durability. If we use Paxos, we'll pay for a property we can't really use

Why does this matter?

- We'll see that durability is expensive
  - Basic Paxos always provides durability
  - SafeSend is like Paxos and also has this guarantee

- If we weaken durability we get better performance and scalability, but we no longer mimic Paxos

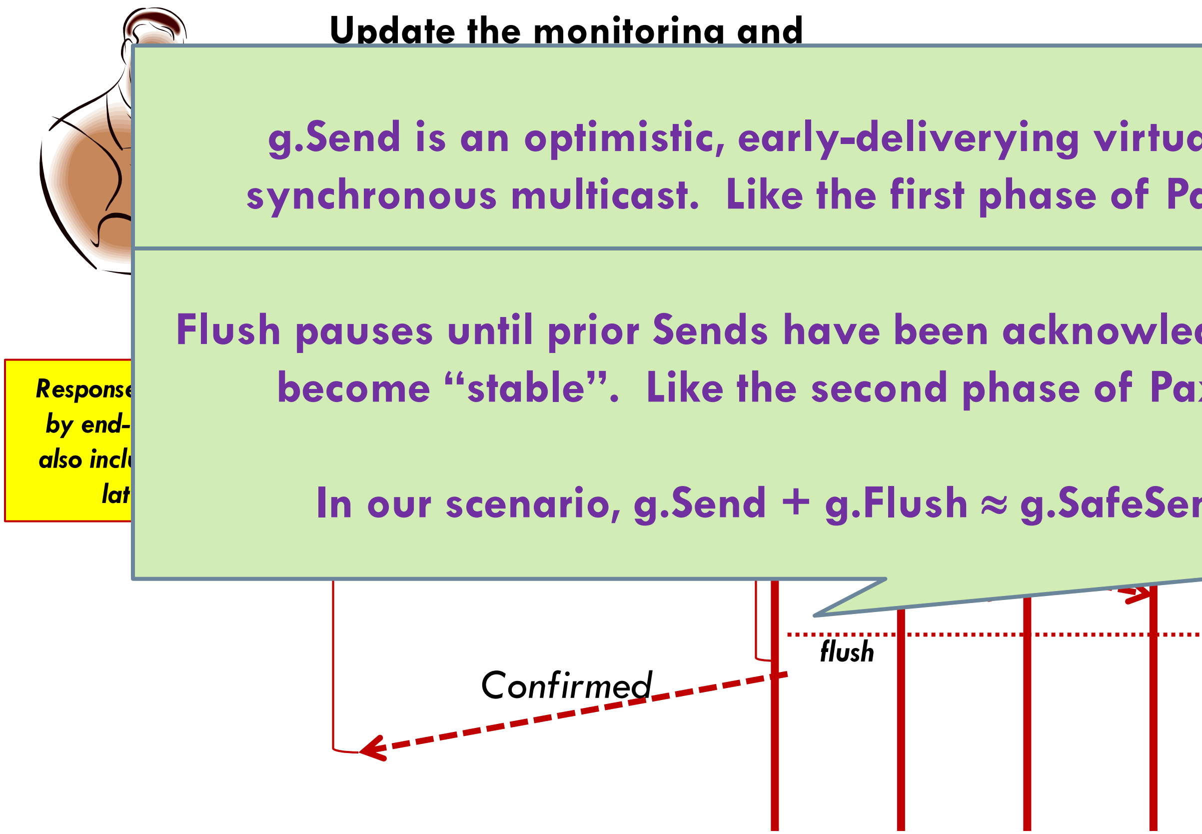
- **Generalization of Brewer's CAP conjecture: one-size-fits-all won't work in the cloud. You always confront tradeoffs.**

Weakening properties in Isis^2

- SafeSend: Ordered+Durable
- OrderedSend: Ordered but “optimistic” delivery
- Send, CausalSend: FIFO or Causal order
- RawSend: Unreliable, not virtually synchronous

- Flush: Useful after an optimistic delivery
  - Delays until any prior optimistic sends are finished.
  - Like “fsync” for a asynchronously updated disk file.

Monitoring in a soft-state service with a primary owner issuing the updates



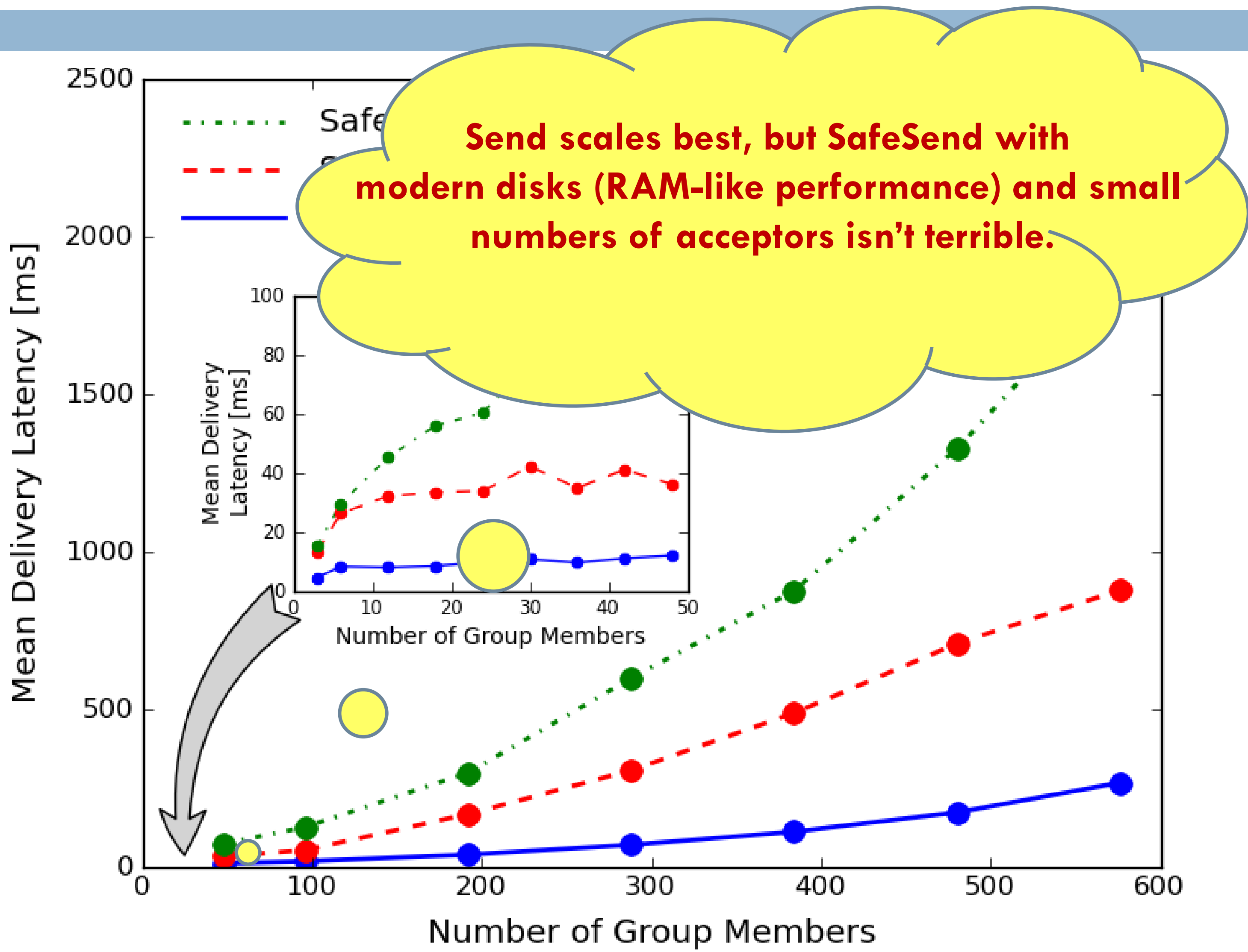
g.Send is an optimistic, early-deliverying virtually synchronous multicast. Like the first phase of Paxos

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

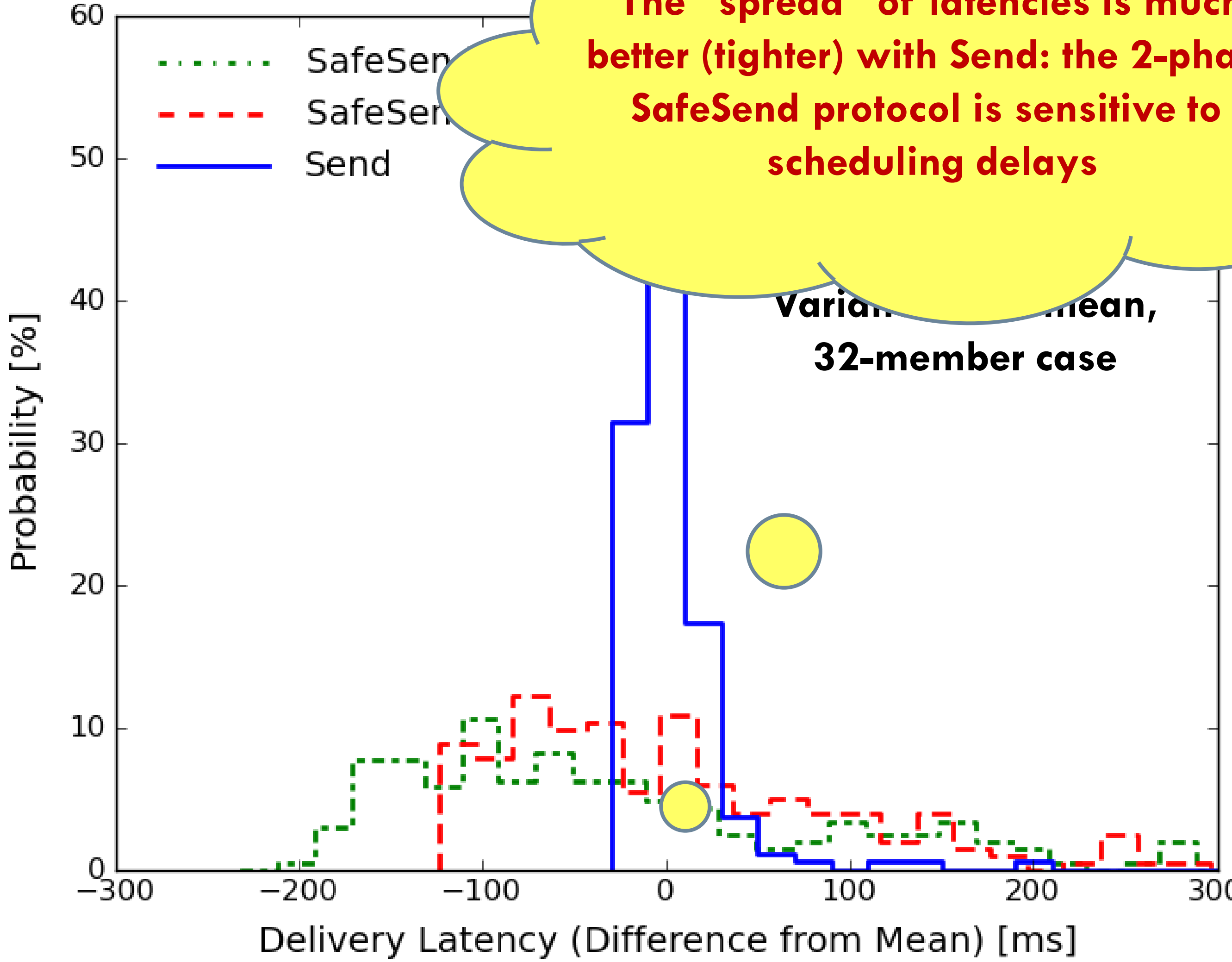
In our scenario, g.Send + g.Flush ≈ g.SafeSend

- In this situation we can replace SafeSend with Send+Flush.
- But how do we *prove* that this is really correct?

Isis^2: Send v.s. SafeSend

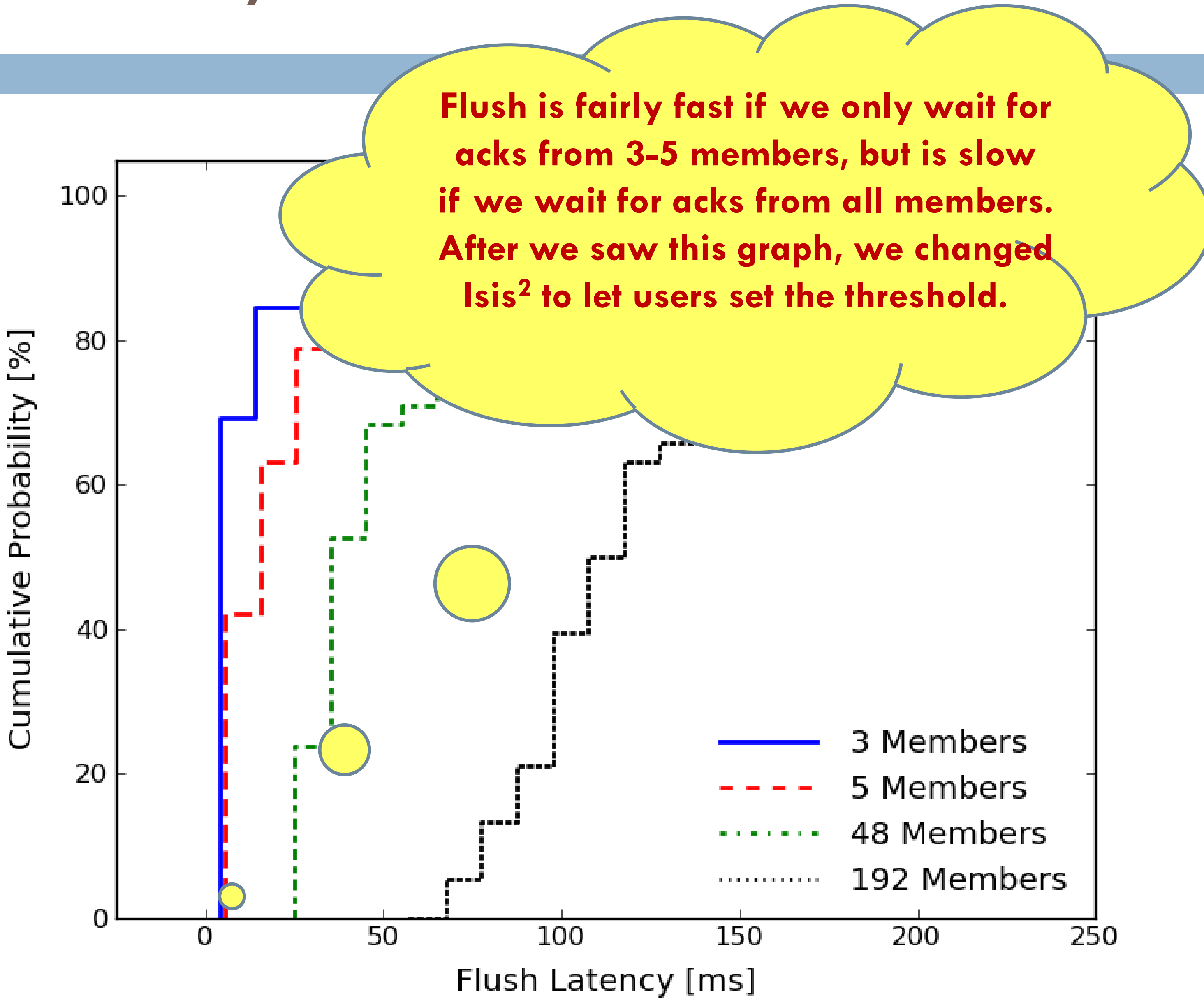


Jitter: how "steady" are latencies?



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



What does the data tell us?

- With g.Send+g.Flush we can have
  - Strong consistency, fault-tolerance, rapid responses
  - Similar guarantees to Paxos (but not identical)
  - Scales remarkably well, with high speed

- Had we insisted on Paxos
  - It wasn't as bad as one might have expected
  - But no matter how we configure it, we don't achieve adequate scalability, and latency is too variable
  - CAP community would conclude: *aim for BASE, not ACID*

Are we done?

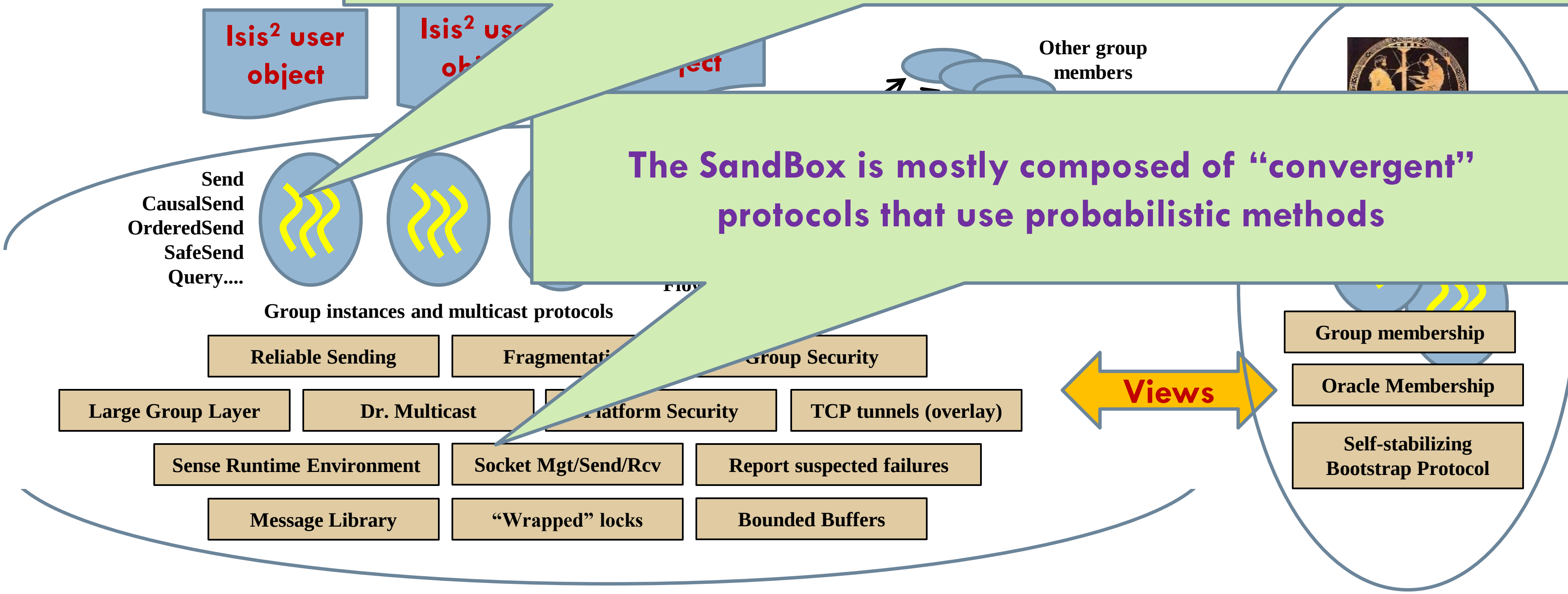
On the family trip to nirvana

- We've seen two issues so far
  - Composition of our SafeSend protocol with application revealed limitations of "formal specifications"
  - Need for speed and scalability forced us to make a non-trivial choice between SafeSend and Send+Flush
- But are these the only such issues?

Modular designs pose many such questions

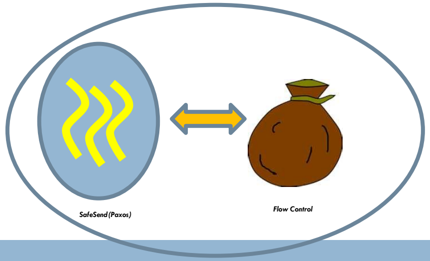
SafeSend and Send are two of the protocol components hosted over the sandbox. These share flow control, security, etc

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



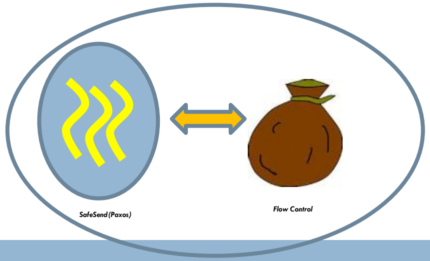
- We structured Isis² as a sandbox containing modular protocol objects
- The sandbox provides *system-wide properties*; protocols "refine" them.

Drill down: Flow control

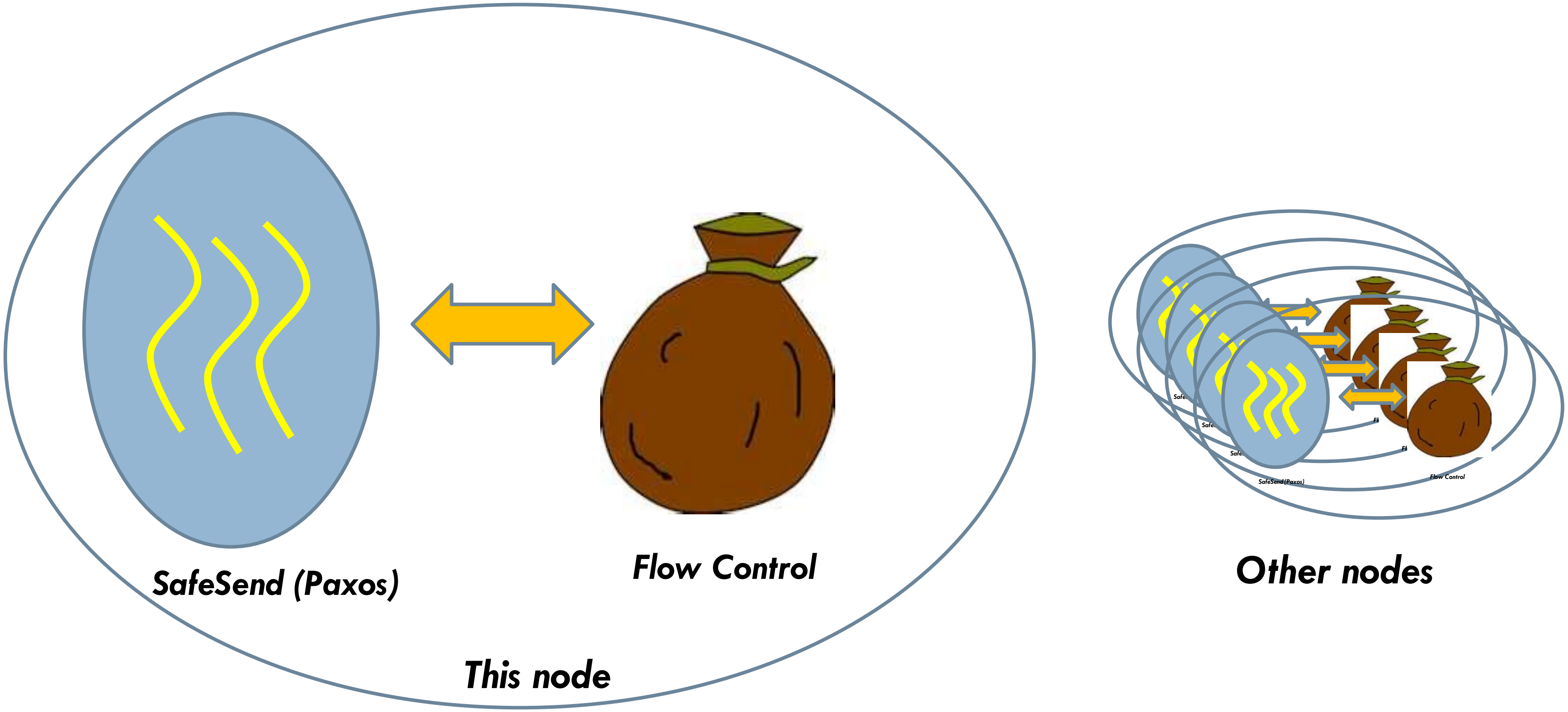


- Consider SafeSend (Paxos) within Isis^2
  - Basic protocol looks very elegant
  - Not so different from Robbert's 60 lines of Erlang
- But pragmatic details clutter this elegant solution
  - E.g.: Need "permission to send" from flow-control module
  - ... later tell flow-control that we've finished
- Flow control is needed to prevent overload
  - Illustrates a sense in which Paxos is "underspecified"

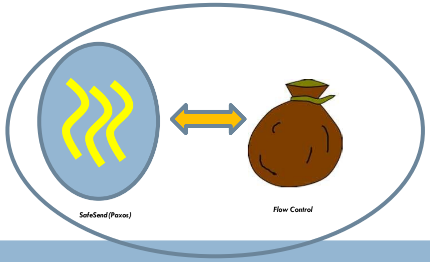
Pictoral representation



- “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 <u>my</u> backlog is large"
- But <u>actual requirement</u> 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

Sandbox design principles

- A "lesson learned" when building Isis[2]
  - Many components use gossip-based algorithms
  - These are robust, scalable, easy to reason about… but they are also sluggish
  - In practice they ended up in the "sandbox"

- The delicate, performance-critical protocols run on top of these more robust but slow components

… adding to our list!

- Earlier, we observed that
  - Today's formal methods look inward, not outward. They overlook the need to relate the correctness of the component with the correctness of the use case
  - They don't offer adequate support for developers who must reason about (important) optimization decisions

- … now we see that large-scale platforms confront us with a whole series of such issues

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?

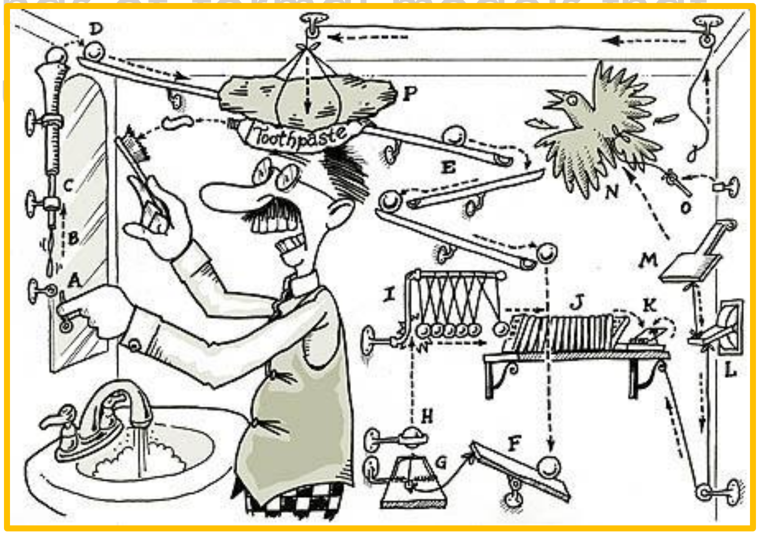
- Which road leads forward?
  1. Extend our formal execution model to cover all elements of the desired solution: a "formal system"

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

els

3. Explore completely new ki

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]**

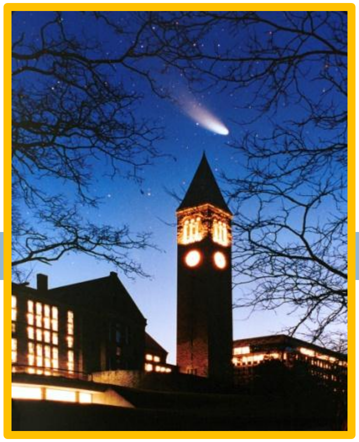
The challenge?

- 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?



- We set out to bring formal assurance guarantees to the cloud
  - And succeeded: Isis^2 works (isis2.codeplex.com)!
  - Industry is also reporting successes (e.g. Google spanner)
  - But along the way, formal tools seem to break down!

- Can the cloud "do" high assurance?
  - At Cornell, we think the ultimate answer will be "yes"
  - … but clearly much research is still needed