

# **CAN CLOUD COMPUTING SYSTEMS OFFER HIGH ASSURANCE WITHOUT LOSING KEY CLOUD PROPERTIES?**

# High Assurance in Cloud Settings

2

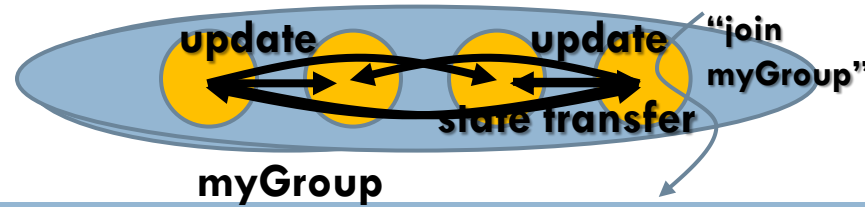
- A wave of applications that need high assurance is fast approaching

- 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

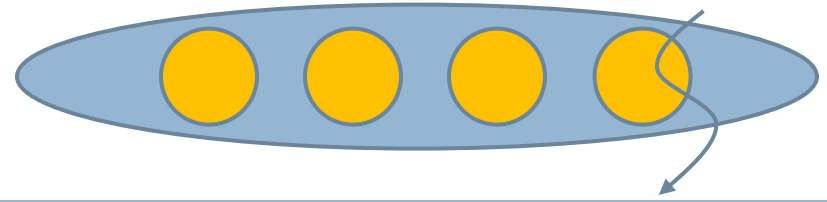
# Isis<sup>2</sup> System



3

- 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 gateway
  - ▣ Read-only operations performed on local state
  - ▣ Update operations update all the replicas

# Isis<sup>2</sup> Functionality



4

- 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

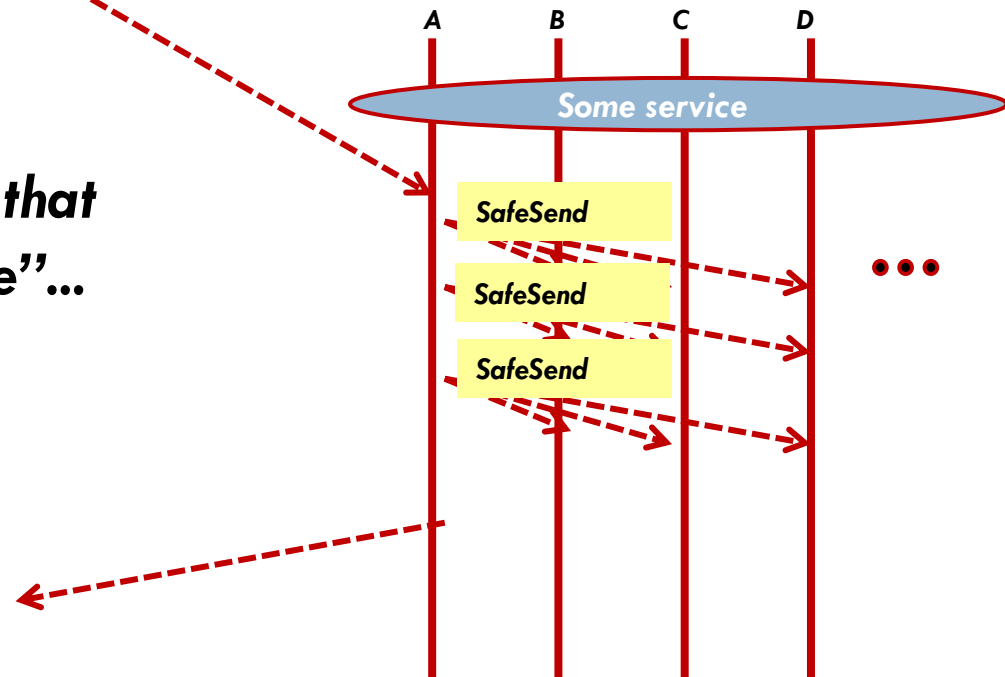
5



*Standard Web-Services method invocation*

*A distributed request that updates group “state”...*

*... and the response*



SafeSend is a version of Paxos.

# Isis<sup>2</sup> System

6

- 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<sup>2</sup> makes developer's life easier

7

## Benefits of Using Formal model

- Formal model permits us to achieve correctness
- Think of Isis<sup>2</sup> as a collection of modules, each with rigorously stated properties
- These help in debugging (model checking)

## Importance of Sound Engineering

- Isis<sup>2</sup> implementation needs to be fast, lean, easy to use, in many ways
- Developer must see it as easier to use Isis<sup>2</sup> than to build from scratch
- Need great performance under “cloudy conditions”

# Isis<sup>2</sup> makes developer's life easier

8

```
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 as seen for event upcalls and the assumptions user can make



# Isis<sup>2</sup> makes developer's life easier

9

```
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<sup>2</sup> makes developer's life easier

10

```
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<sup>2</sup> makes developer's life easier

11

```
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

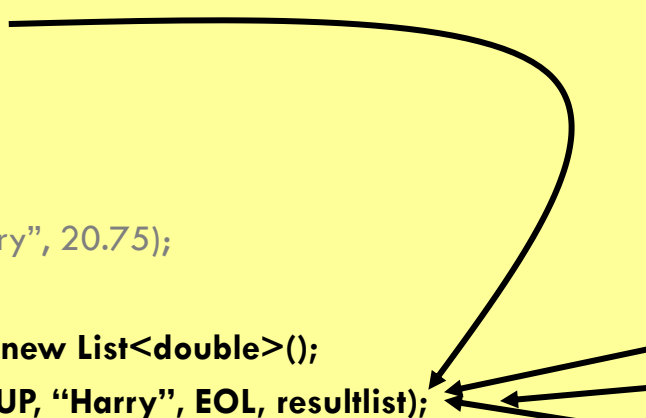
# Isis<sup>2</sup> makes developer's life easier

12

```
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<sup>2</sup> makes developer's life easier

13

```
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);
```

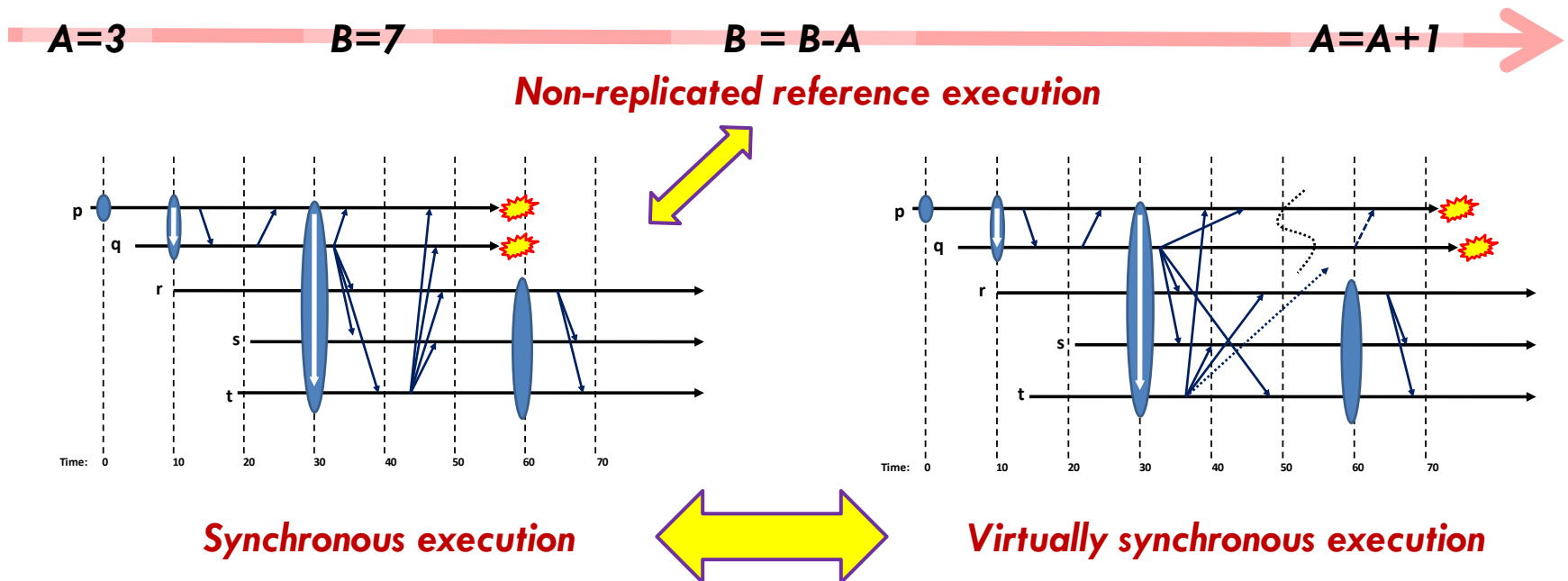
- First sets up group
- Join makes this entity a member. State transfer isn't shown
- Then can multicast, query. Runtime callbacks to the "delegates" as events arrive
- **Easy to request security, persistence, tunnelling on TCP...**
- **"Consistency" model dictates the ordering seen for event upcalls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

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

14

## □ 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



# Formalizing the model

15

- Must express the picture in temporal logic equations
- Closely related to state machine replication, but optimistic early delivery of multicasts (optional!) is tricky.
- What can one say about the guarantees in that case?
  - ▣ Either I'm going to be allowed to stay in the system, in which case all the properties hold
  - ▣ ... or the majority will kick me out. Then some properties are still guaranteed, but others might actually not hold for those optimistic early delivery events
  - ▣ User is expected to combine optimistic actions with *Flush* to mask speculative lines of execution that could turn out to be risky

# Core issue: How is replicated data used?

16

- 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?

17

- 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

# But why complicate it with optimism?

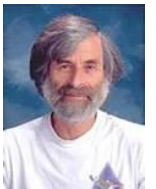
18

- Optimistic early delivery kind of breaks the model, although Flush allows us to hide the effects
- To reason about a system must (more or less) erase speculative events not covered by Flush. Then you are left with a more standard state machine model
- Yet this standard model, while simpler to analyze, is actually too slow for demanding use cases

# Roles for formal methods

19

- *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.

# The resulting theory is of limited value

20

- If we apply it only to Isis<sup>2</sup> itself, we can generally get quite far. The model is valuable for debugging the system code because we can detect bad runs.
- If we apply it to a user's application, the theory is often “incomplete” because the theory would typically omit any model for what it means for the application to achieve its end-user goals

# The fundamental issue...

21

- *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 DB)?*
- 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

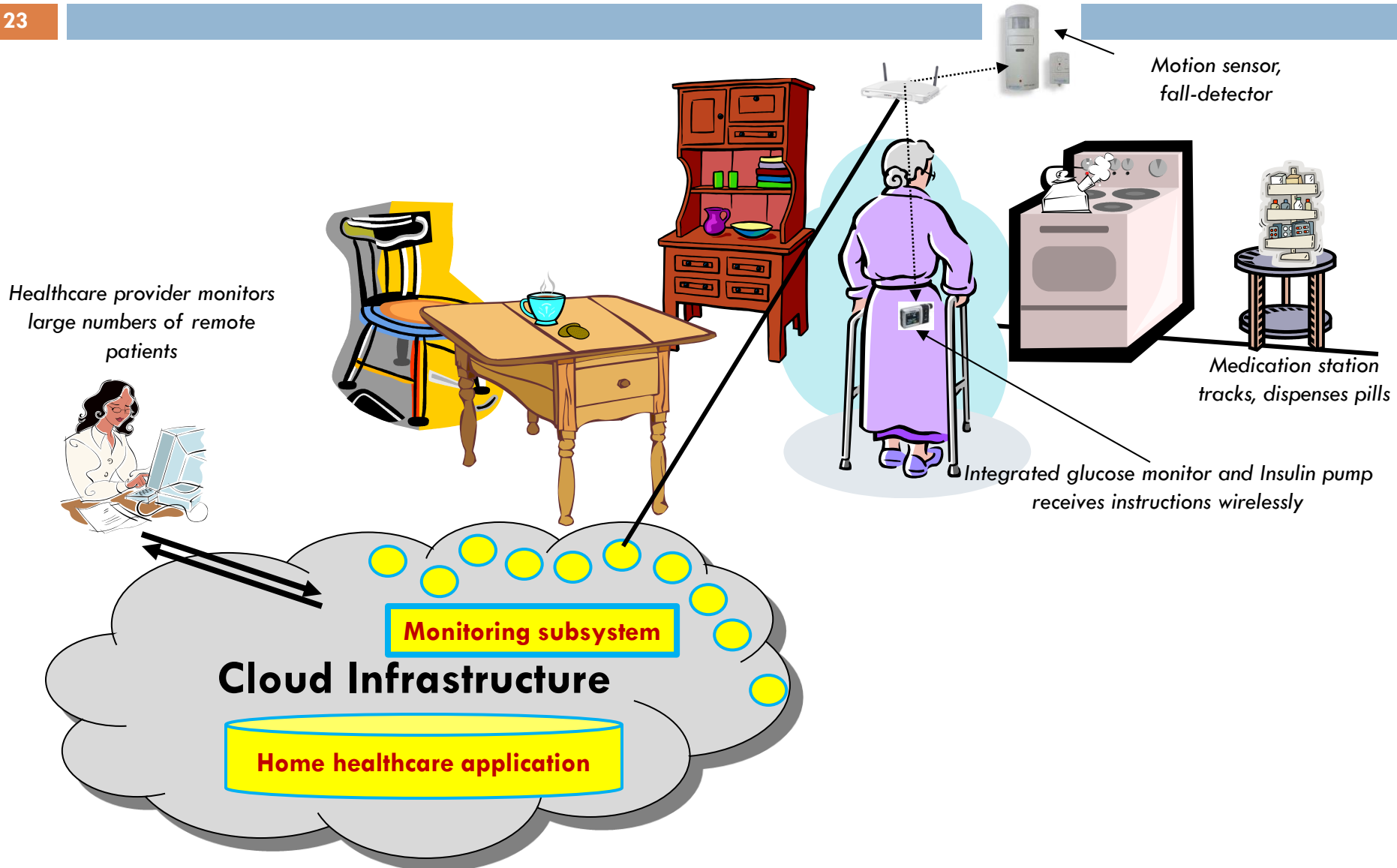
# A further issue: Performance causes complexity

22

- 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

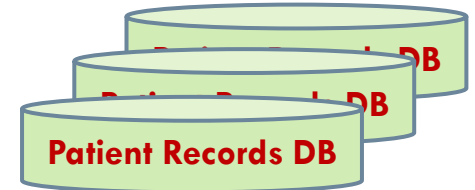
23



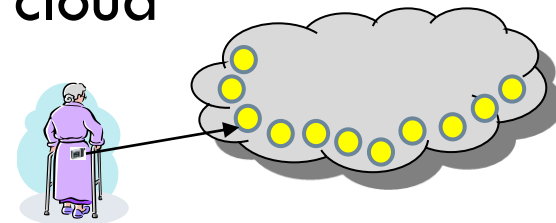
# Two replication cases that arise

24

- 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





# Real systems demand tradeoffs

25

- 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?

26

- 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.**



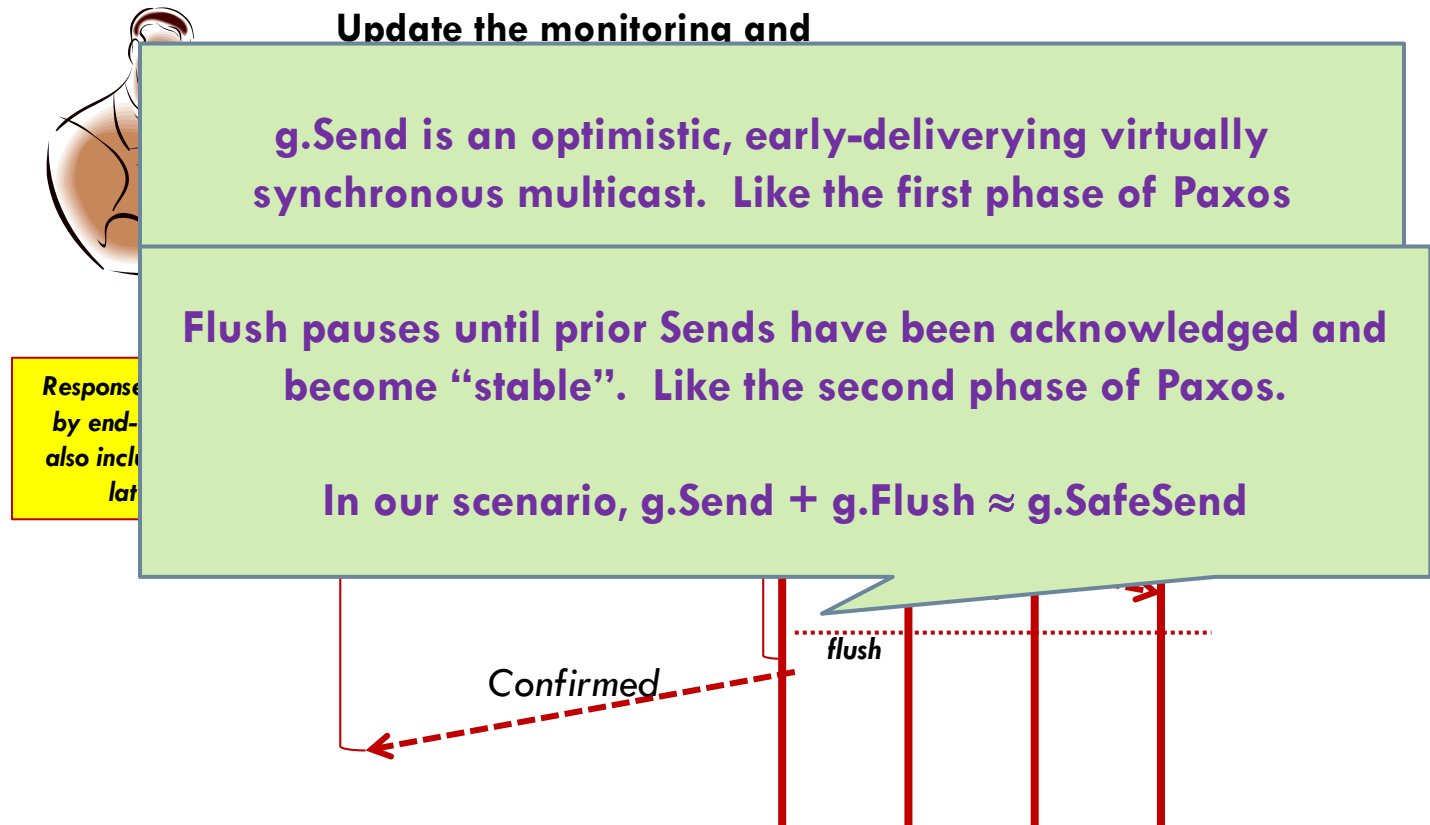
# Weakening properties in Isis<sup>2</sup>

27

- 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

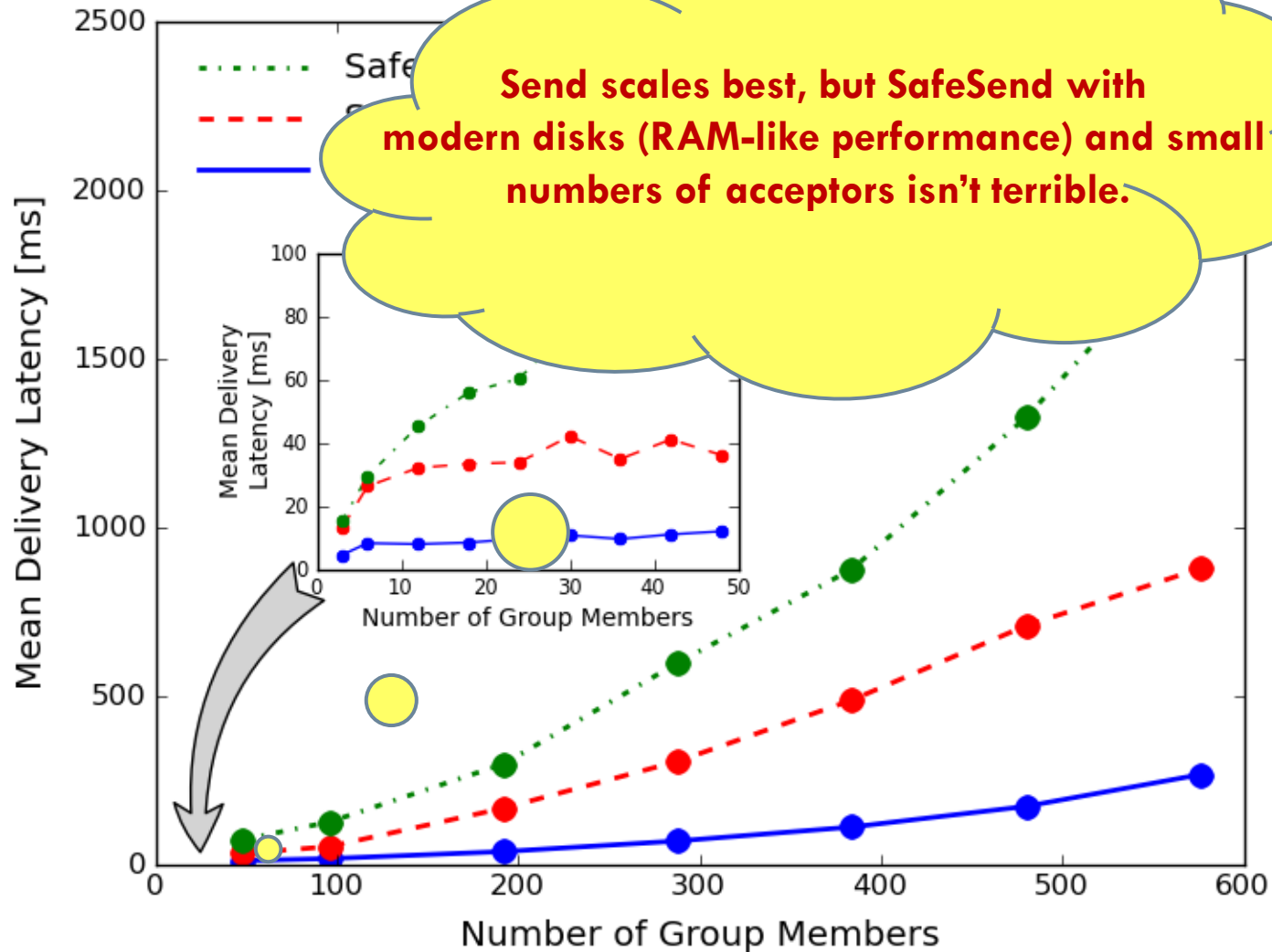
28



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

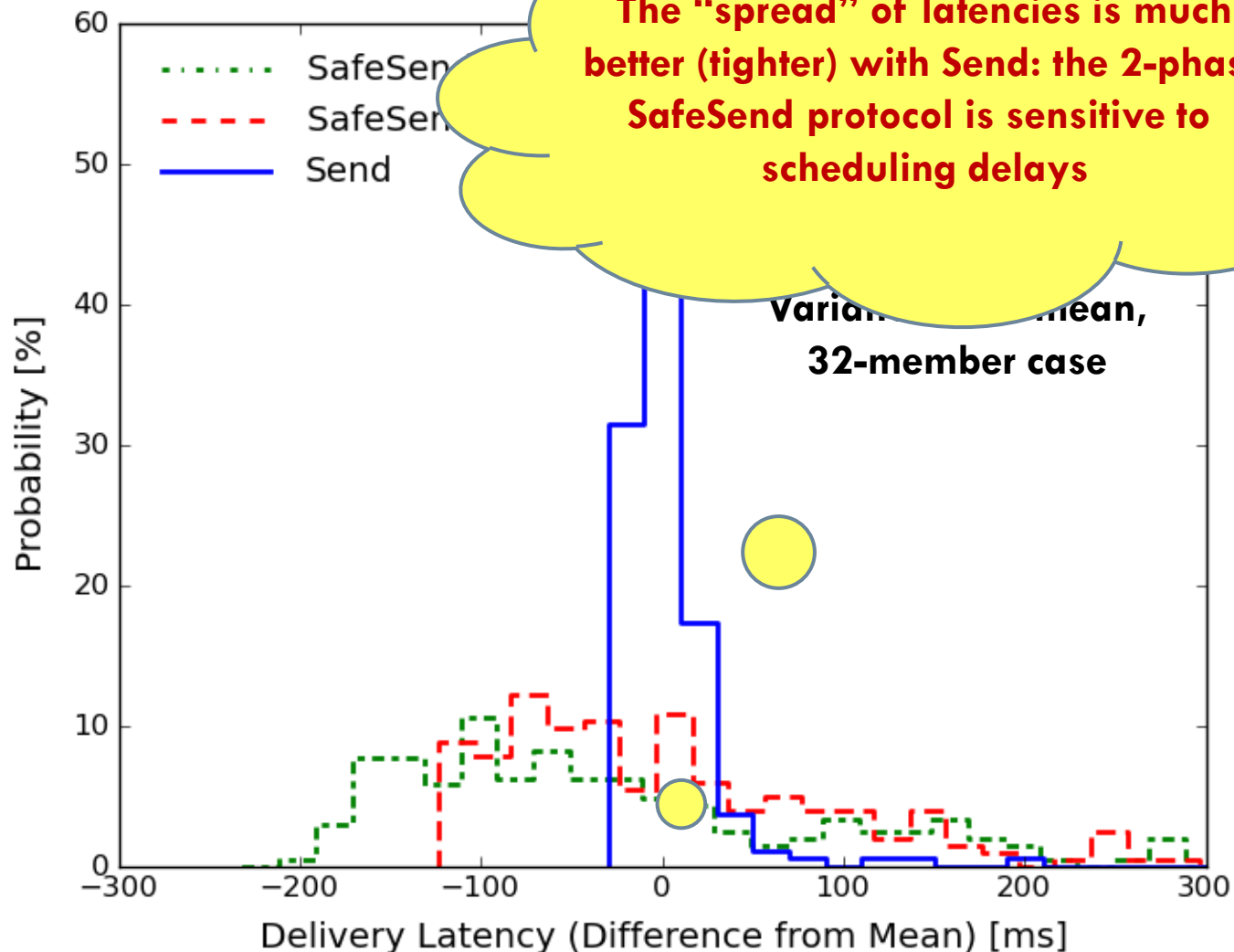
# Isis<sup>2</sup>: Send v.s. SafeSend

29



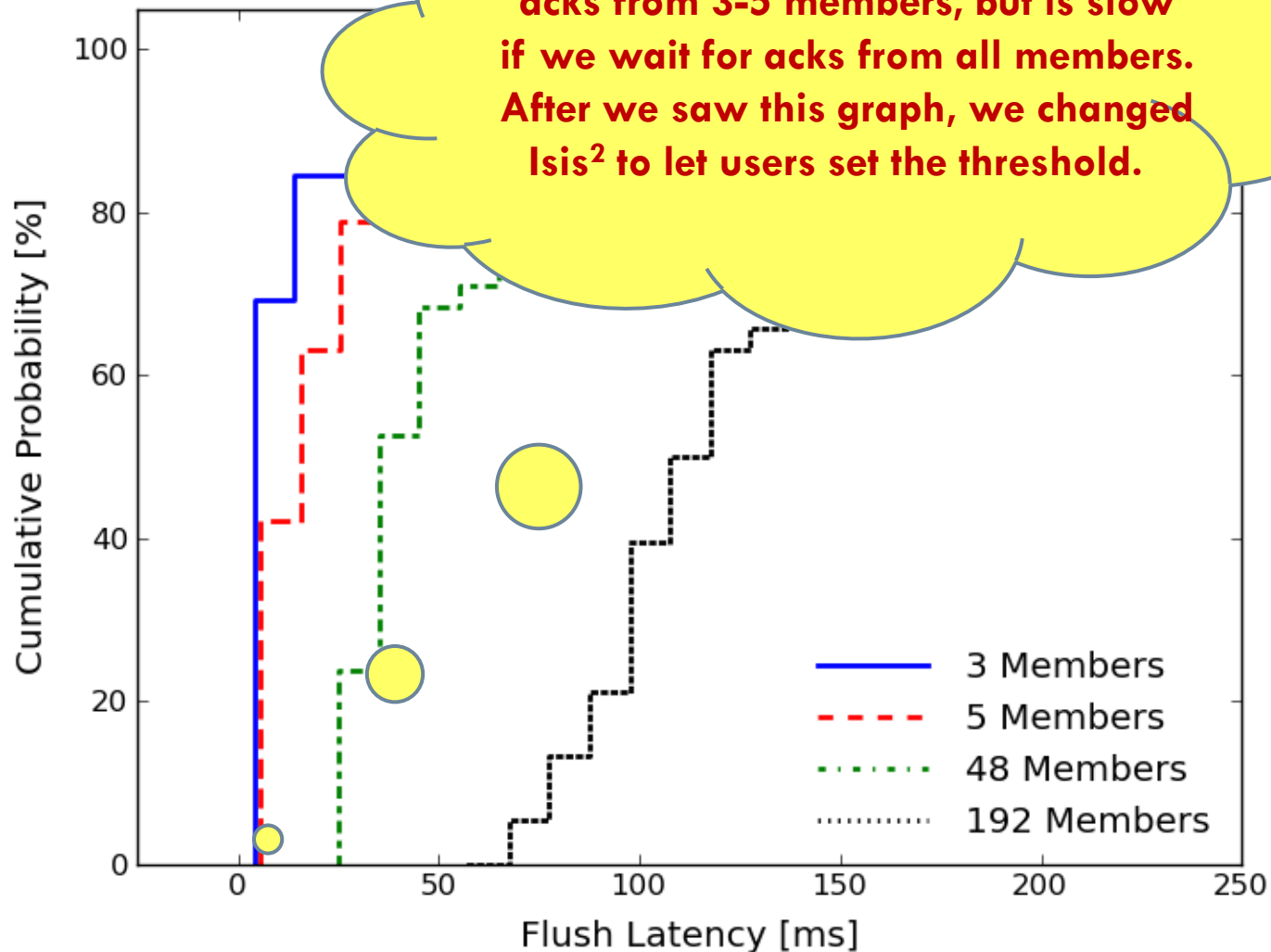
# Jitter: how “steady” are latencies?

30



# Flush delay as function of shard size

31



# What does the data tell us?

32

- 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*



# The challenge...

33



- 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?

34

## □ Which road leads forward?

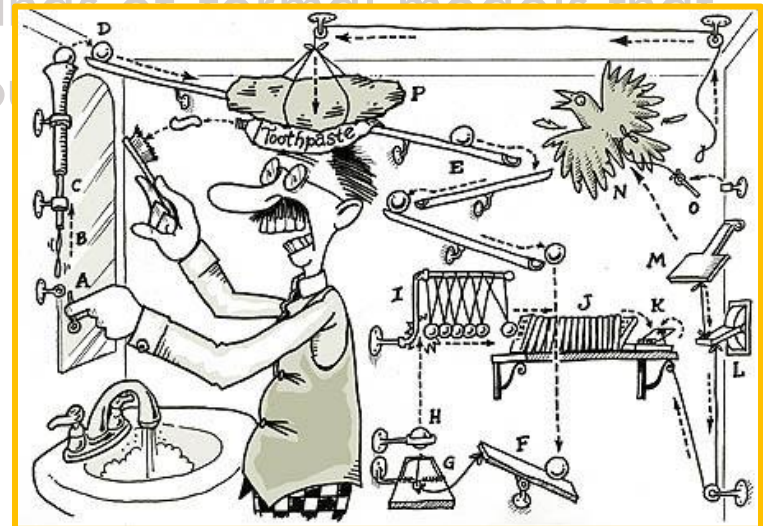
1. Extend our formal execution model to cover all elements of the desired solution: a “formal system”



### **Doubtful:**

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

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



# The challenge?

35



## □ 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<sup>2</sup>**

# The challenge?

36

## □ 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

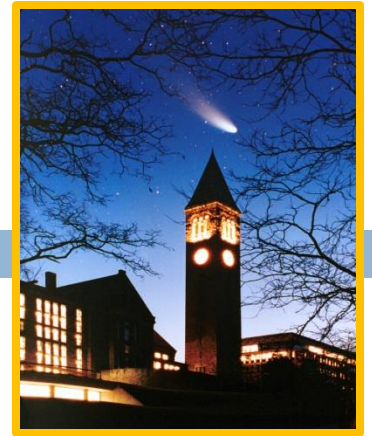


### **Intriguing future topic:**

- **Generating provably correct code**
- **In some ways much easier than writing the code, then analyzing it**

# Summary?

37



- We set out to bring formal assurance guarantees to the cloud
  - ▣ And succeeded: Isis<sup>2</sup> works ([isis2.codeplex.com](http://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