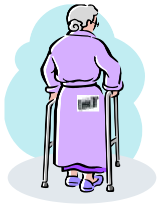
CAN CLOUD COMPUTING SYSTEMS OFFER HIGH ASSURANCE WITHOUT LOSING KEY CLOUD PROPERTIES?

CS6410

Ken Birman, Cornell University

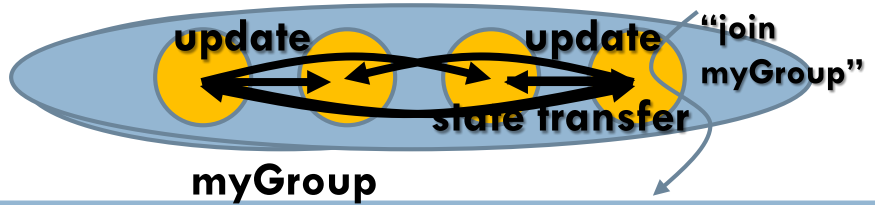
High Assurance in Cloud Settings

- 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

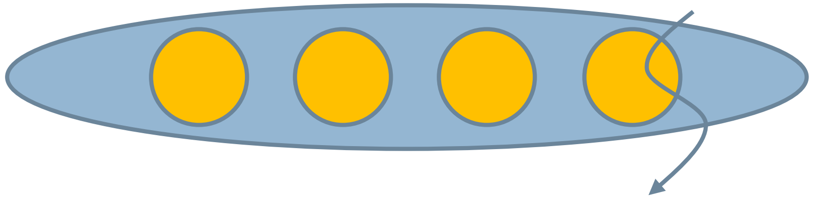
Isis^2 System



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

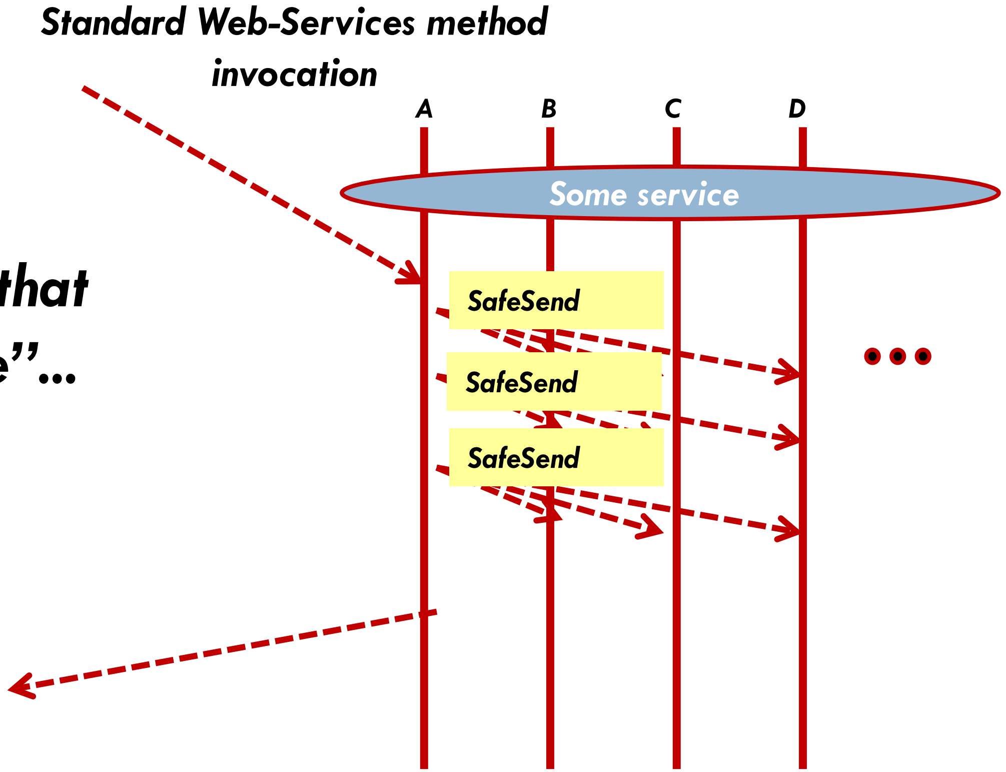
Isis^2 Functionality



- 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"

Isis2 makes developer's life easier

Benefits of Using Formal model

- Formal model permits us to achieve correctness
- Think of Isis2 as a collection of modules, each with rigorously stated properties
- These help in debugging (model checking)

Importance of Sound Engineering

- Isis2 implementation needs to be fast, lean, easy to use, in many ways
- Developer must see it as easier to use Isis2 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

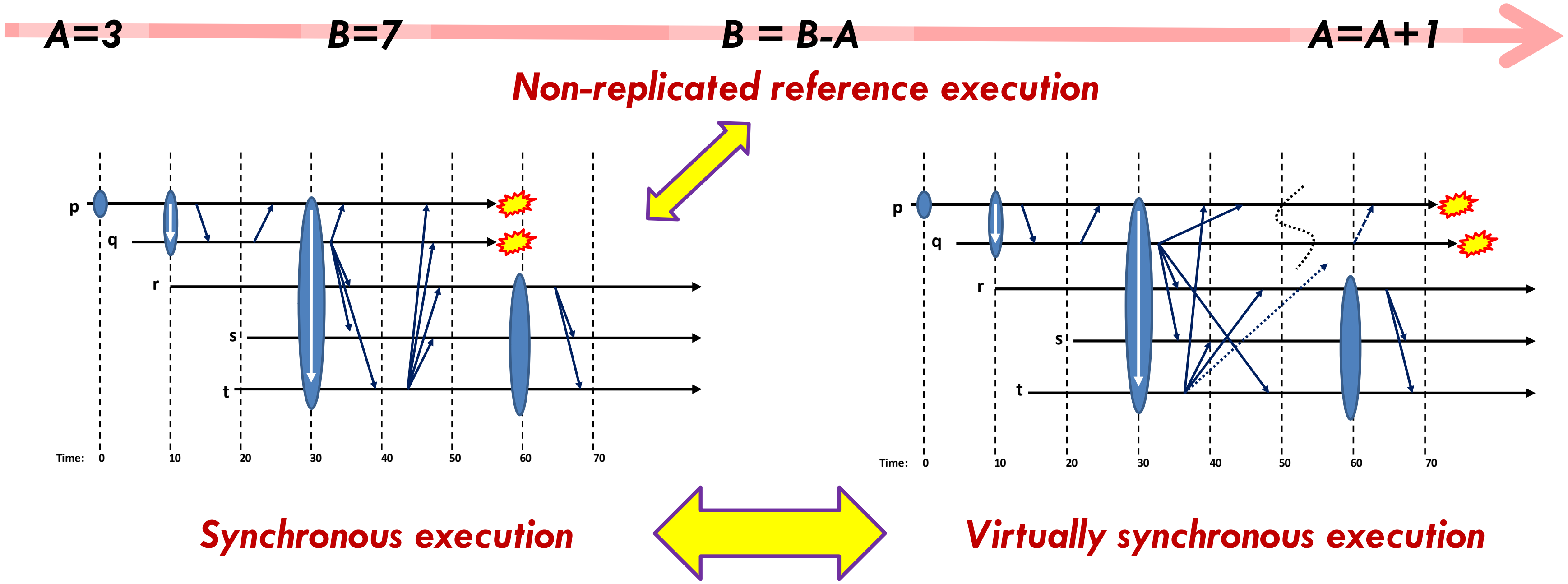
Isis^2 makes developer's life easier


```
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…)

- 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



Exact comparison

- What I am calling a synchronous (by which I mean "step by step") execution actually matches what Paxos offers, but Paxos, as we will see, uses quorum operations to implement this without group views

- Virtual synchrony has managed group membership, but also has some optimistic steps (early message delivery, which speeds things up, but it comes at the price of needing to do a "flush" to sync to the network)
  - Analogy: when you write to a file often the IO system buffers and until you do a file-sync, data might not yet be certain to have reached the disk

Formalizing the model

- 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?

- 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

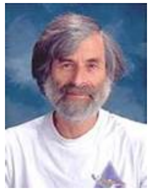
- All sorts of reasoning is dramatically simplified

But why complicate it with optimism?

- 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

- *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

- If we apply it only to Isis^2 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 plus Isis^2, 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
  - This pervasive tendency to ignore the user is a continuing issue throughout the community even today. It represents a major open research topic.

The fundamental issue...

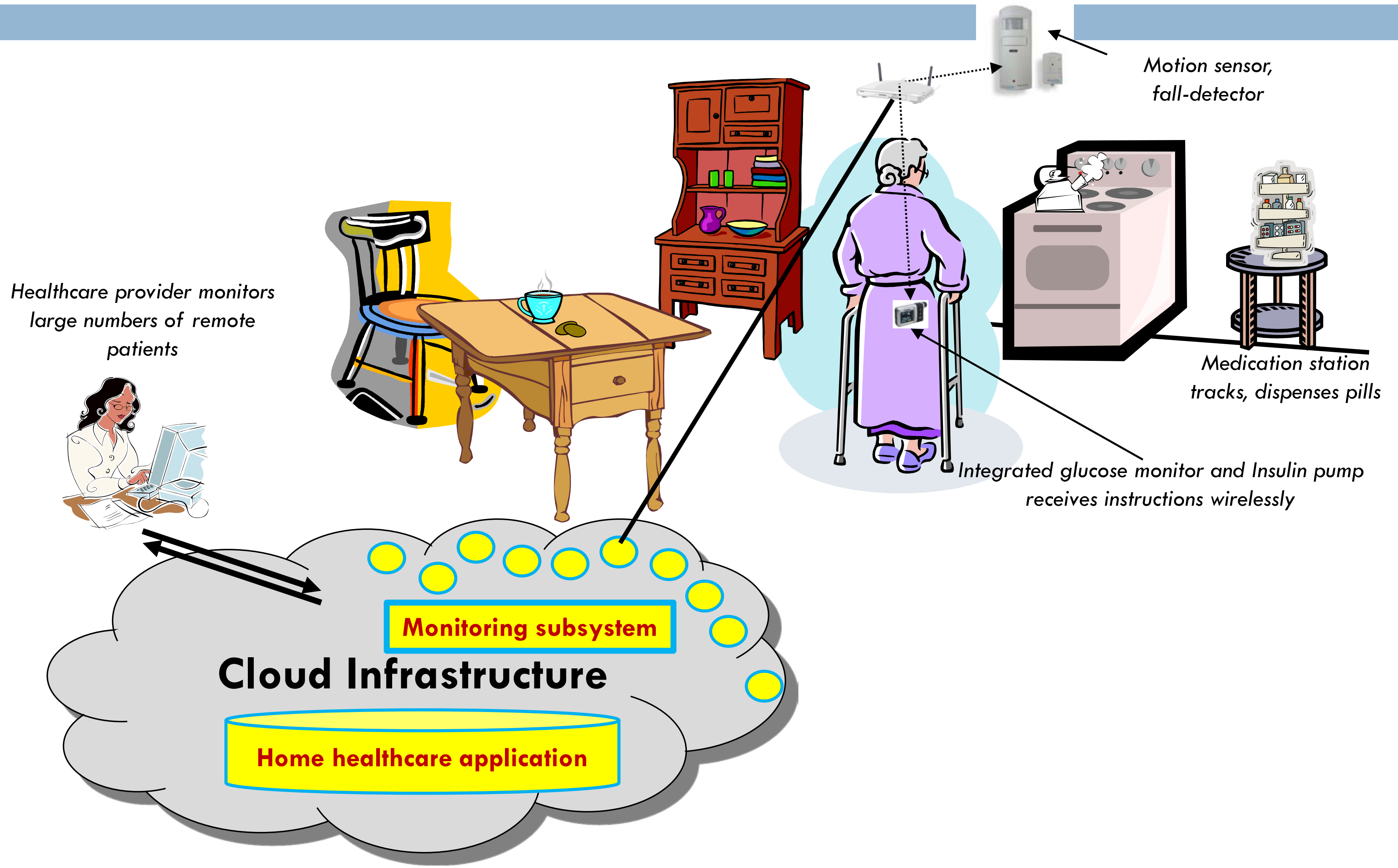
- *How to formalize the notion of application state?*
- *How to formalize the composition of a protocol such as SafeSend with an application (such as replicated DB)?*
- No obvious answer… just (unsatisfying) options
  - A composition-based architecture: interface types (or perhaps phantom types) could 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

- 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)

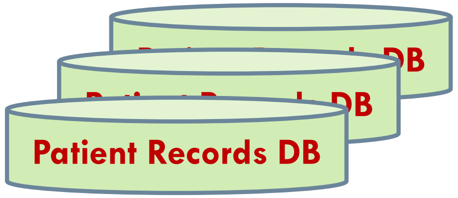
- Let's look at a use case in which being flexible is key to achieving performance and scalability

Building an online medical care system

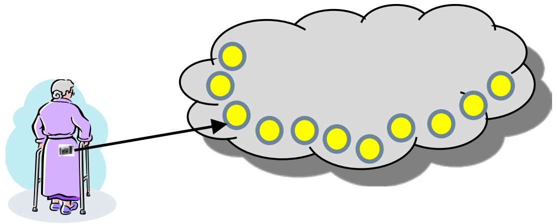


Two replication cases that arise

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



- 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

- 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?

- 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

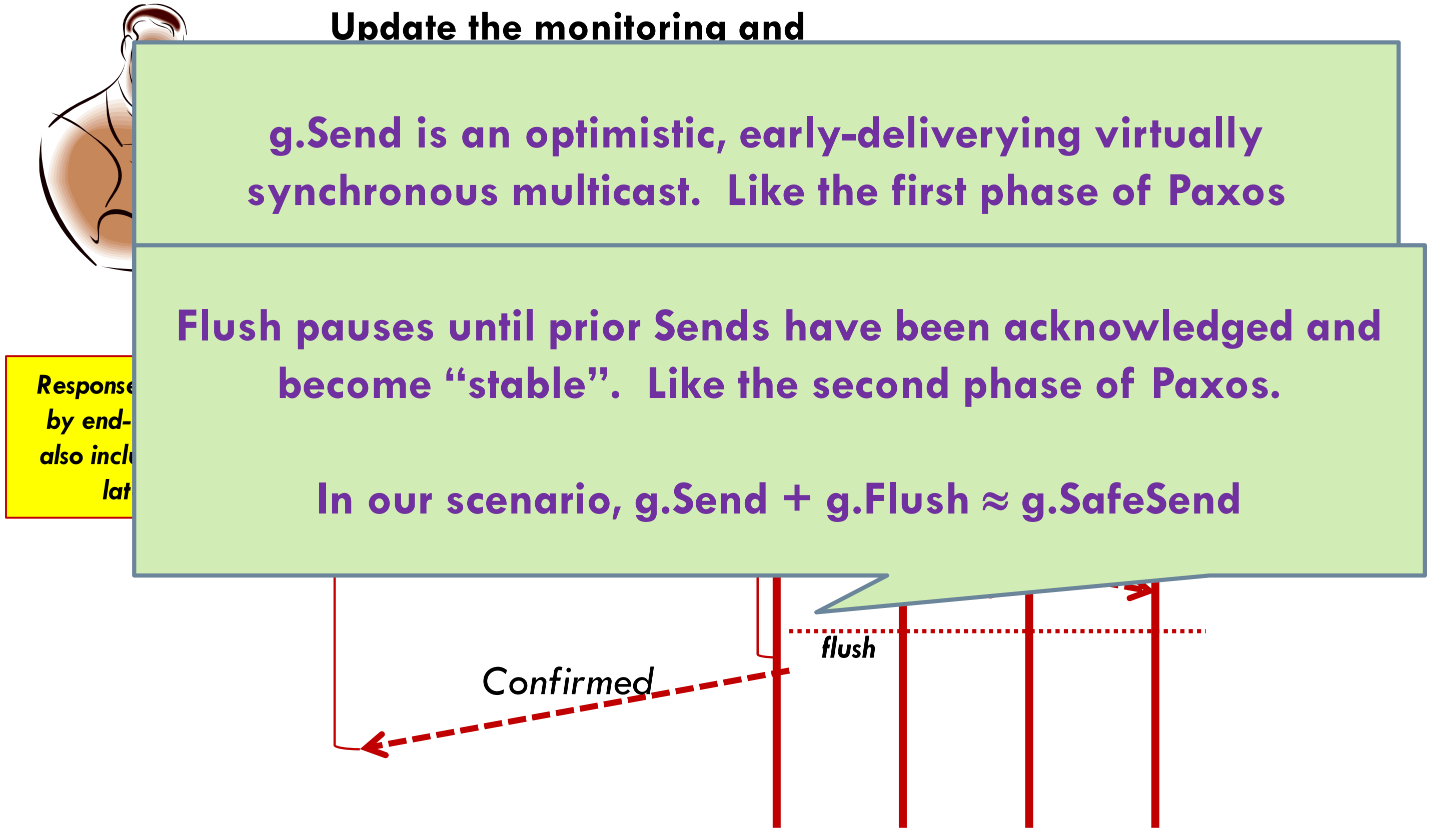
- **Generalization of Brewer's CAP conjecture: one-size-fits-all won't work in the cloud. You always confront tradeoffs.**

Weakening properties in Isis^2

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

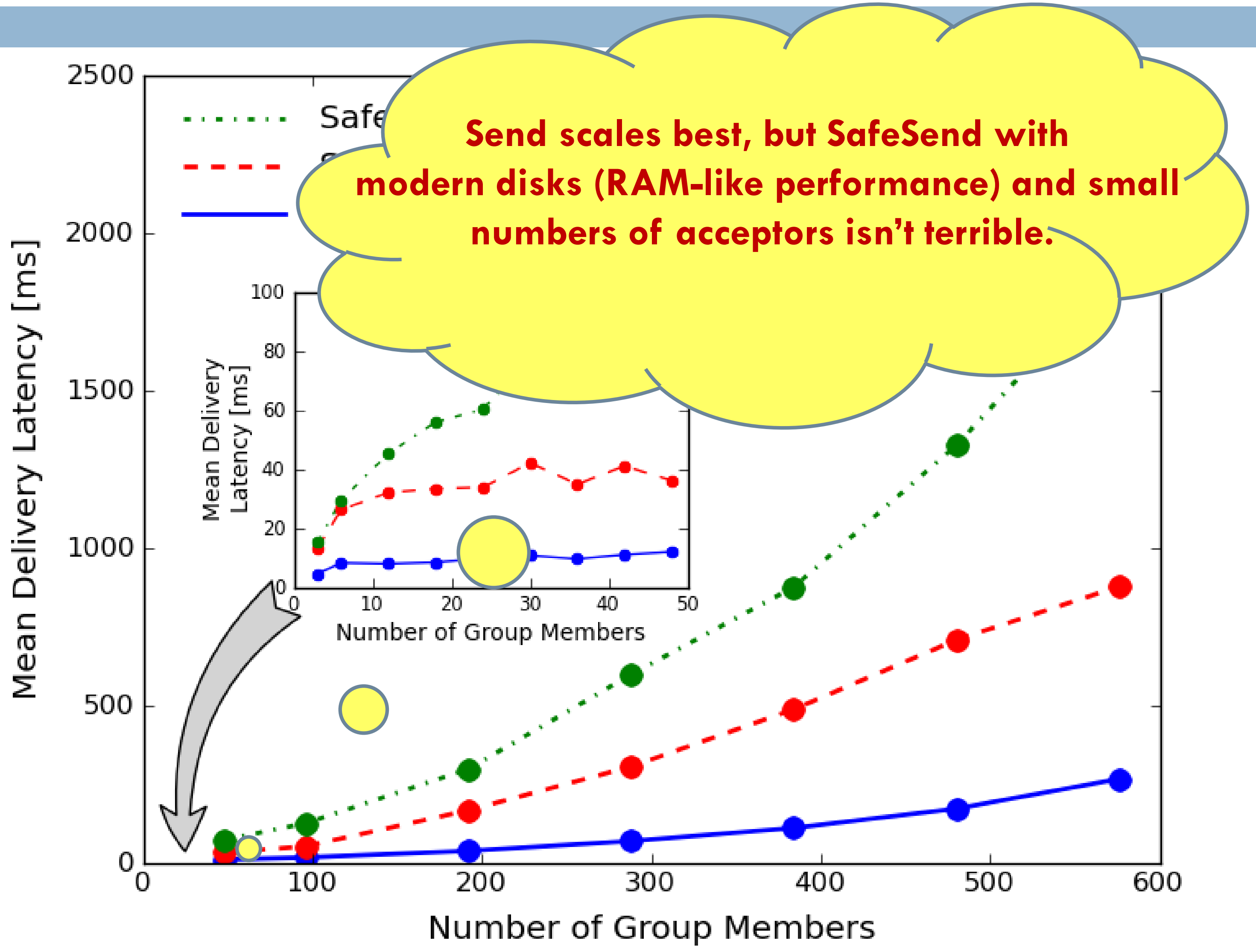
- Out of Band file transfer: Uses RDMA to asynchronously move big objects using RDMA network; Isis^2 application talks “about” these objects but doesn’t move the bytes (might not even touch the bytes)

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

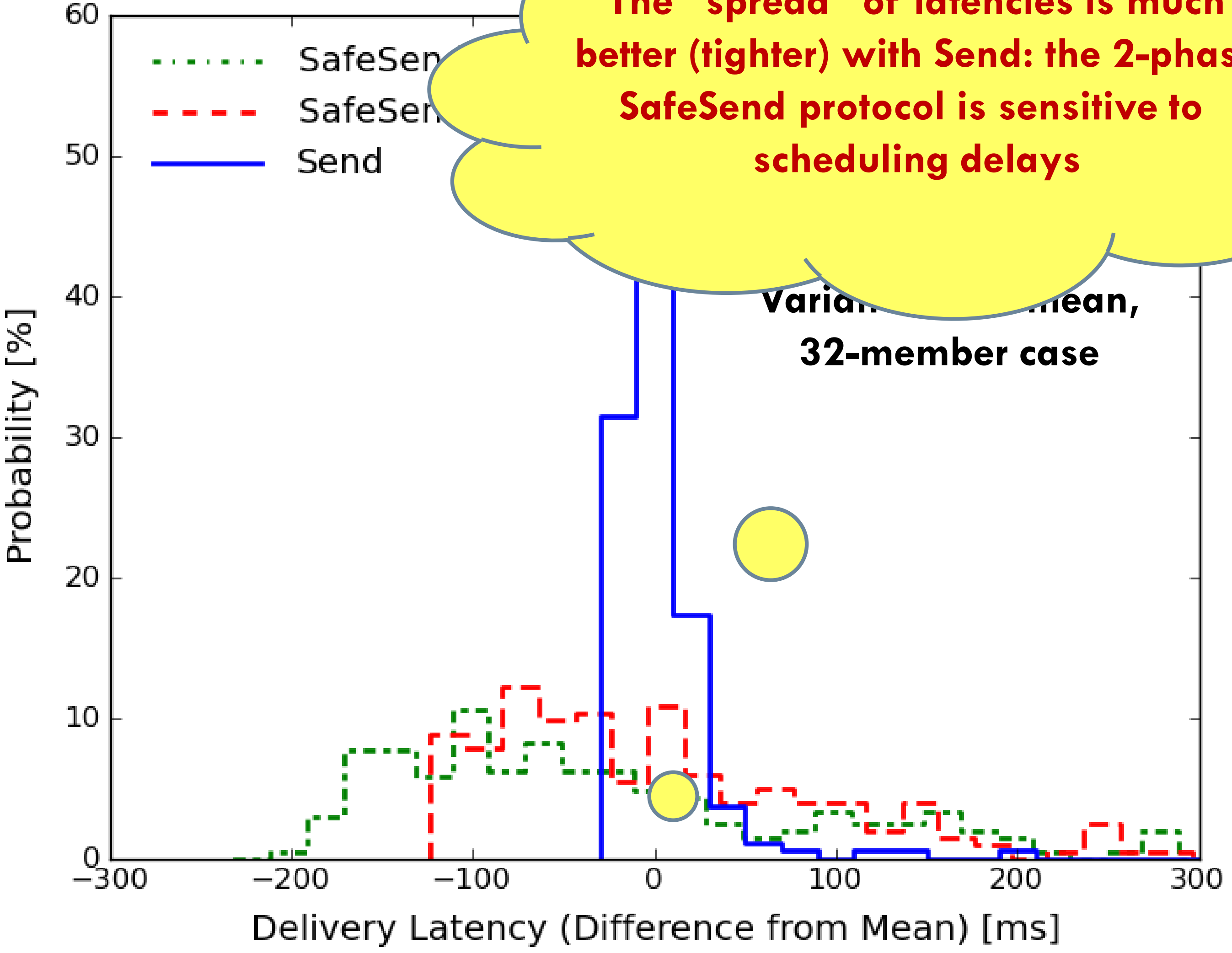


- In this situation we can replace SafeSend with Send+Flush.
- But how do we *prove* that this is really correct?

Isis^2: Send v.s. SafeSend

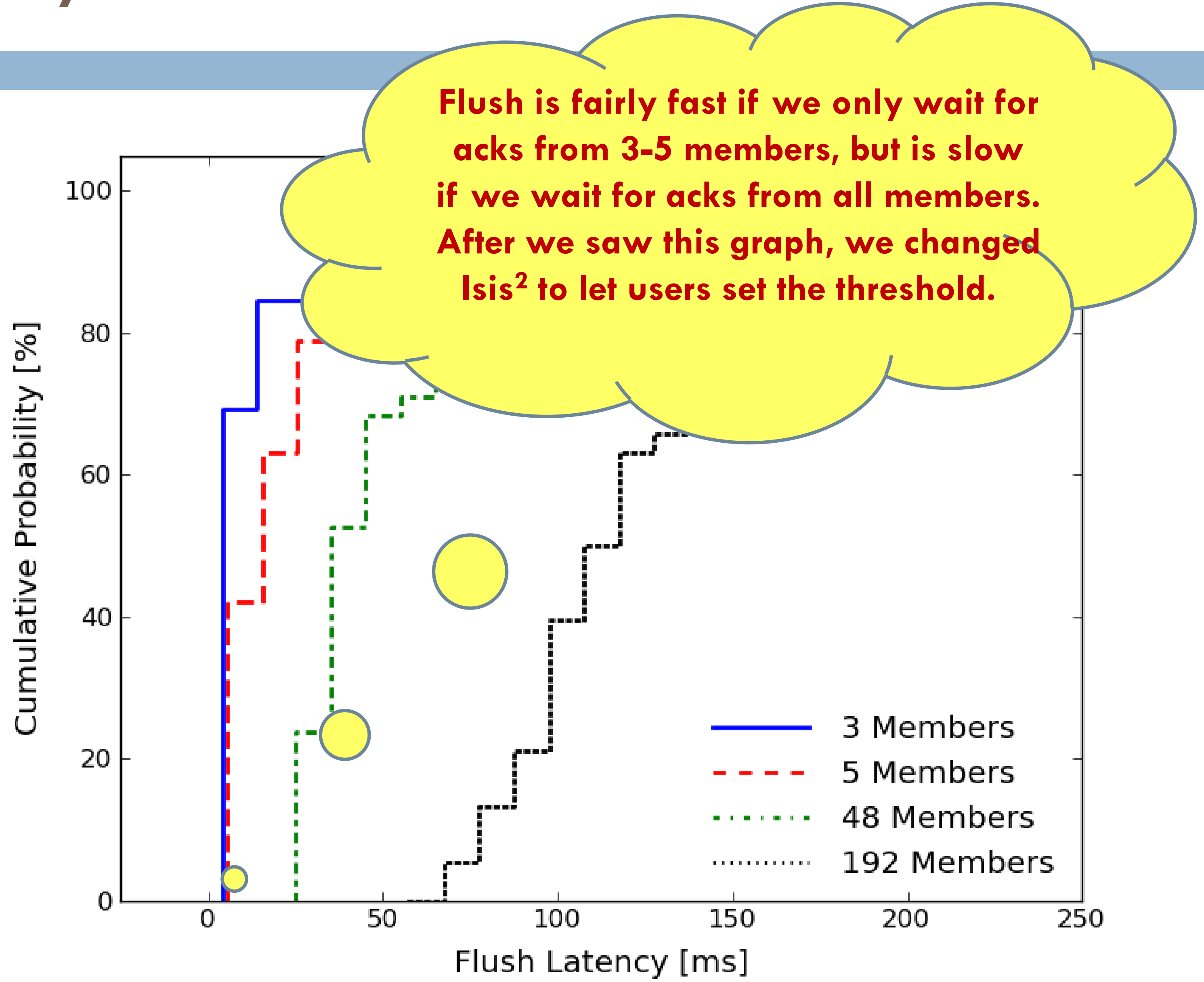


Jitter: how "steady" are latencies?



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

- The experiment isn't totally fair to Paxos
  - Even 5 years ago, hardware was actually quite different
  - With RDMA and NVRAM the numbers all get (much) better!

Sinfonia

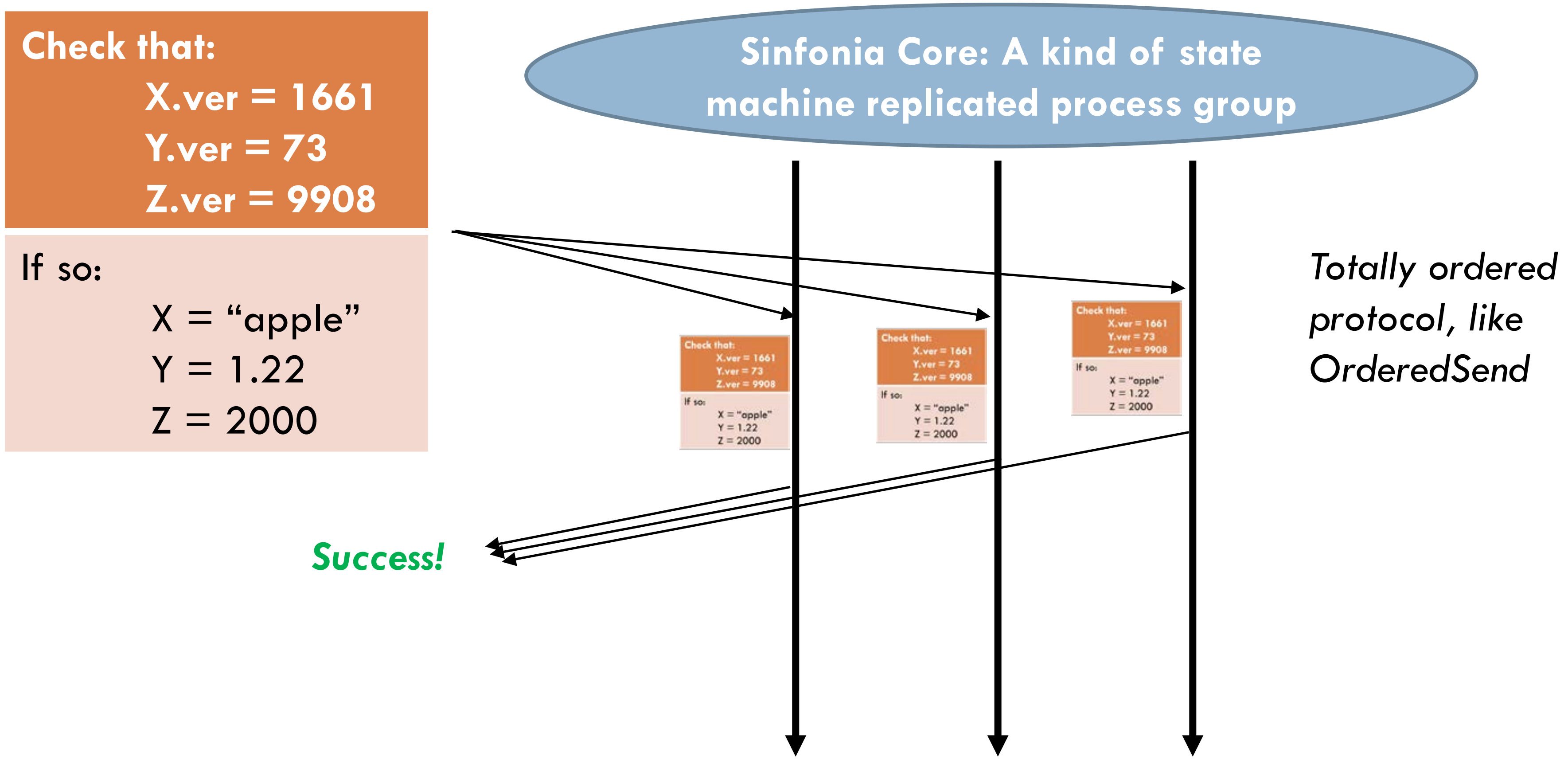
- A more recent system somewhat in the same style, but very different API and programming model
- Starts with a kind of atomic transaction model, which more recent work (this year's SOSP!) has made more explicit

Key Sinfonia idea

- Allow the application to submit "mini-transactions"
  - Not the full SQL + begin / commit / abort, but rather "RISC" in style

- They consist of:
  - **Precomputation:** Application prepares a mini-transaction however it likes
  - **Validation step**: objects and versions: the mini-transaction will not be performed (will abort) if any of these objects have been updated
  - **Action step:** If validation is successful, a series of updates to those objects, which will generate new versions. The actions are done atomically.

Illustration



- The server members are exact replicas, so all either perform the action or reject it. So the data replicas stay in the identical state

Precomputation step

- This gives Sinfonia remarkable scalability

- Idea is that we can keep cached copies of the system state, or even entire read-only replicas, and run any code we wish against it

- State = Any collection of data with some form of records we can identify and version numbers on each record
- Code = Database transaction, graph crawl, whatever…

Why does this give scalability?

- At the edge, we soak up the potentially slow, complex compute costs
  - Transactions can be very complex to carry out (joins, projections, aggregation operations, complex test logic…)
  - All of this can be done "offline" from the perspective of the core

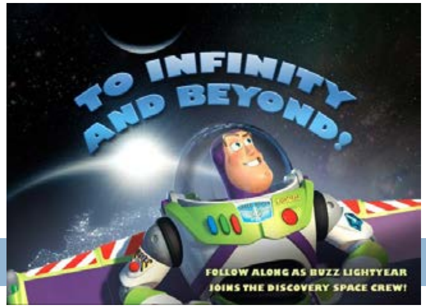
- Then we either commit the request all at once if the versions still match, or abort it all at once if not, so Sinfonia core stays in a consistent state
  - In fact, the edge can manage perfectly well with a slightly stale cache!

Generality?

- Paper explains how this model can support a great variety of use cases from the web, standard databases, financial settings (banking or stock trading), etc.
  - Basically, you just need an adaptor to “represent” your data in Sinfonia format with data records and version numbering

- And in recent work at Vmware, they add sharding (partitioning), automatic support for commutative actions, many other features, and get even more impressive performance

Summary?



- We set out to bring formal assurance guarantees to the cloud
  - And succeeded: Many systems like Isis[2] exist now and are in wider and wider use (Corfu, Zookeeper, Zab, Raft, libPaxos, Sinfonia, and the list goes on)
  - Industry is also reporting successes (e.g. *entire* SOSP program this year)
  - Formal tools are also finding a major role now (model checking and constructive logic used to prove these kinds of systems correct)

- Can the cloud "do" high assurance?
  - At Cornell, and in Silicon Valley, the evidence now is "yes"
  - … but even so, much more research is still needed because they are slow "on first try" and much optimization generally has to occur to make them fast