CS5412: HOW DURABLE SHOULD IT BE?

Lecture XV

Ken Birman

Durability

- When a system accepts an update and won't lose it, we say that event has become durable
- They say the cloud has a permanent memory
  - Once data enters a cloud system, they rarely discard it
  - More common to make lots of copies, index it…
- But loss of data due to a failure is an issue

Should Consistency "require" Durability?

- The Paxos protocol guarantees durability to the extent that its command lists are durable

- Normally we run Paxos with the command list on disk, and hence Paxos can survive any crash
  - In Isis^2, this is g.SafeSend with the "DiskLogger" active
  - But costly

Consider the first tier of the cloud

- Recall that applications in the first tier are limited to what Brewer calls “Soft State”
  - They are basically prepositioned virtual machines that the cloud can launch or shutdown very elastically
  - But when they shut down, lose their “state” including any temporary files
  - Always restart in the initial state that was wrapped up in the VM when it was built: no durable disk files

Examples of soft state?

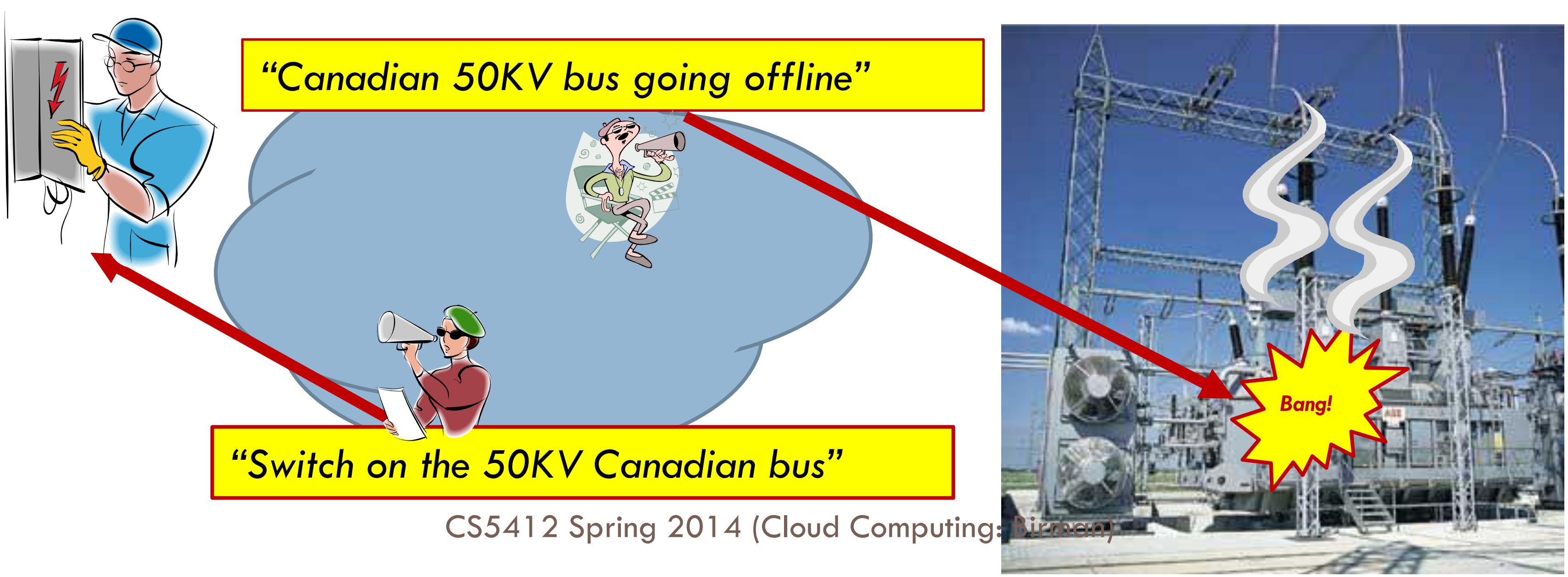
- Anything that was cached but "really" lives in a database or file server elsewhere in the cloud
  - If you wake up with a cold cache, you just need to reload it with fresh data
- Monitoring parameters, control data that you need to get "fresh" in any case
  - Includes data like "The current state of the air traffic control system" – for many applications, your old state is just not used when you resume after being offline
  - Getting fresh, current information guarantees that you'll be in sync with the other cloud components
- Information that gets reloaded in any case, e.g. sensor values

Would it make sense to use Paxos?

- We do maintain sharded data in the first tier and some requests certainly trigger updates
- So that argues in favor of a consistency mechanism
- In fact consistency can be important even in the first tier, for some cloud computing uses

Control of the smart power grid

- Suppose that a cloud control system speaks with "two voices"
- In physical infrastructure settings, consequences can be very costly



So… would we use Paxos here?

- In discussion of the CAP conjecture and their papers on the BASE methodology, authors generally assume that “C” in CAP is about ACID guarantees or Paxos
- Then argue that these bring too much delay to be used in settings where fast response is critical
- Hence they argue against Paxos

By now we've seen a second option

- Virtual synchrony Send is "like" Paxos yet different

- Paxos has a very strong form of durability
- Send has consistency but weak durability unless you use the "Flush" primitive. Send+Flush is amnesia-free

- Further complicating the issue, in Isis^2 Paxos is called SafeSend, and has several options
  - Can set the number of acceptors
  - Can also configure to run in-memory or with disk logging

How would we pick?

- The application code looks nearly identical!
  - g.Send(GRIDCONTROL, *action to take*)
  - g.SafeSend(GRIDCONTROL, *action to take*)

- Yet the behavior is very different!
  - SafeSend is slower
  - … and has stronger durability properties. ***Or does it?***

SafeSend in the first tier

- Observation: like it or not we just don't have a durable place for disk files in the first tier
- The *only* forms of durability are
  - In-memory replication within a shard
  - Inner-tier storage subsystems like databases or files
- Moreover, the first tier is expect to be rapidly responsive and to talk to inner tiers asynchronously

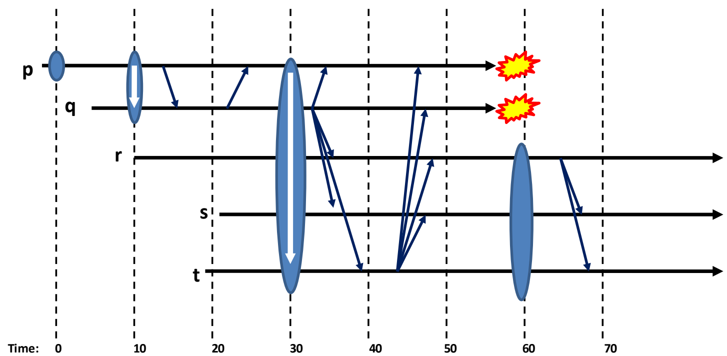
So our choice is simplified

- No matter what anyone might tell you, in fact the only real choices are between two options
  - Send + Flush: Before replying to the external customer, we know that the data is replicated in the shard
  - In-memory SafeSend: On an update by update basis, before each update is taken, we know that the update will be done at every replica in the shard

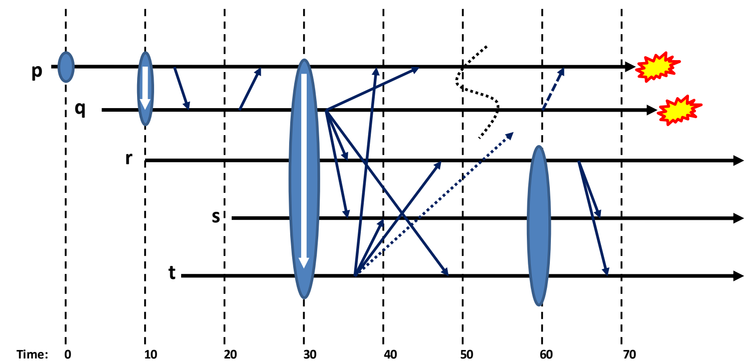
Consistency model: Virtual synchrony meets Paxos (and they live happily ever after…)

A=3 ***B=7*** ***B = B-A*** ***A=A+1***

Non-replicated reference execution



Synchronous execution



Virtually synchronous execution

- **Virtual synchrony is a "consistency" model:**
  - ***Synchronous runs:*** ***indistinguishable from non-replicated object that saw the same updates (like Paxos)***
  - ***Virtually synchronous runs*** ***are indistinguishable from synchronous runs***

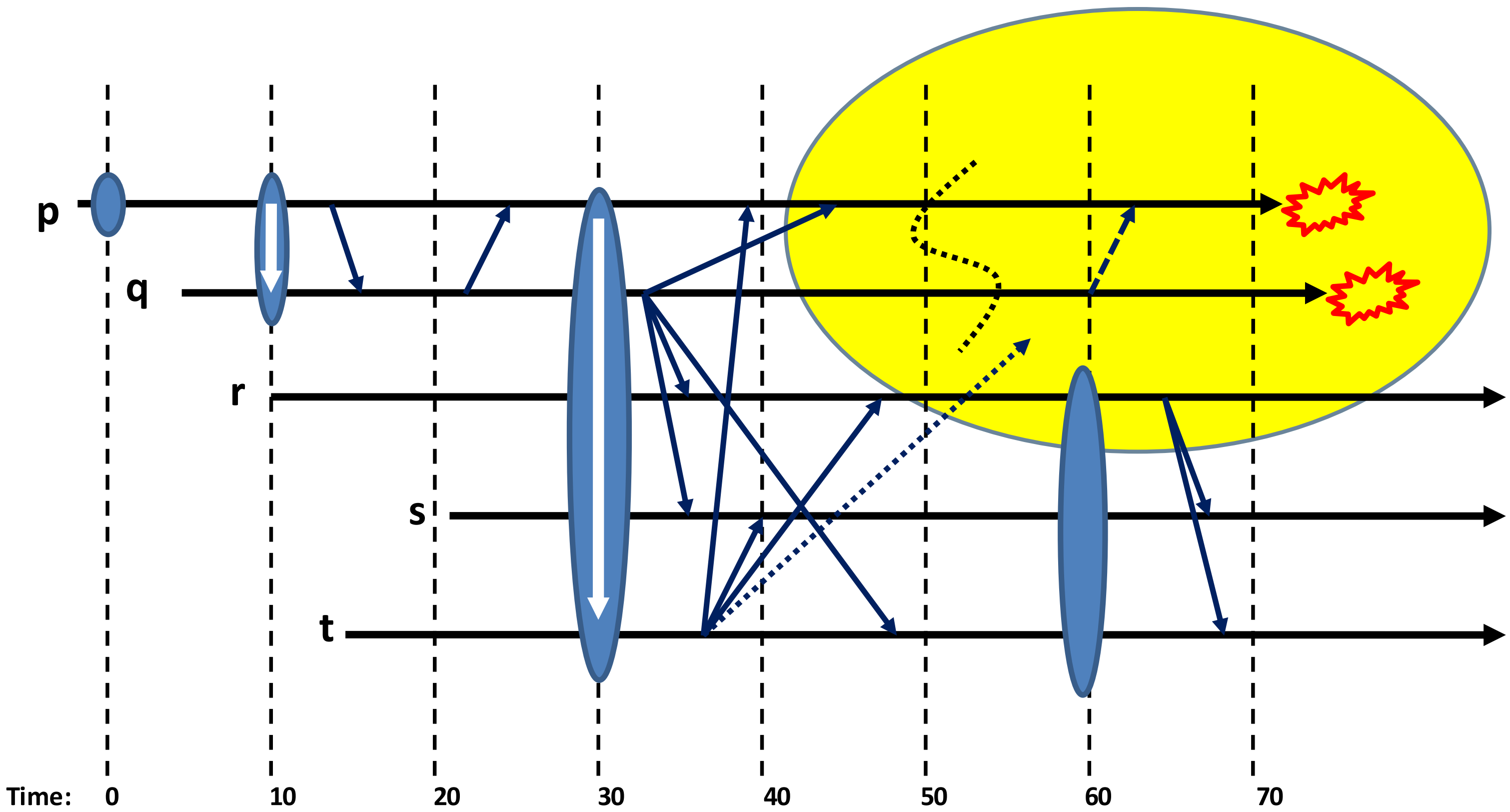
SafeSend versus Send

- Send can have different delivery orders if there are different senders
  - In fact Isis^2 offers other options, we'll discuss them next time.

- SafeSend can't have the strange amnesia problem see in the top right corner on the timeline picture

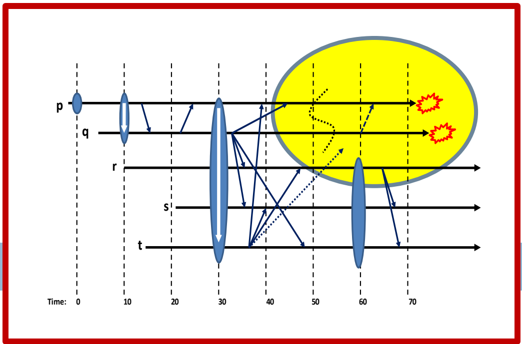
- But these guarantees are pretty costly!

Looking closely at that "oddity"



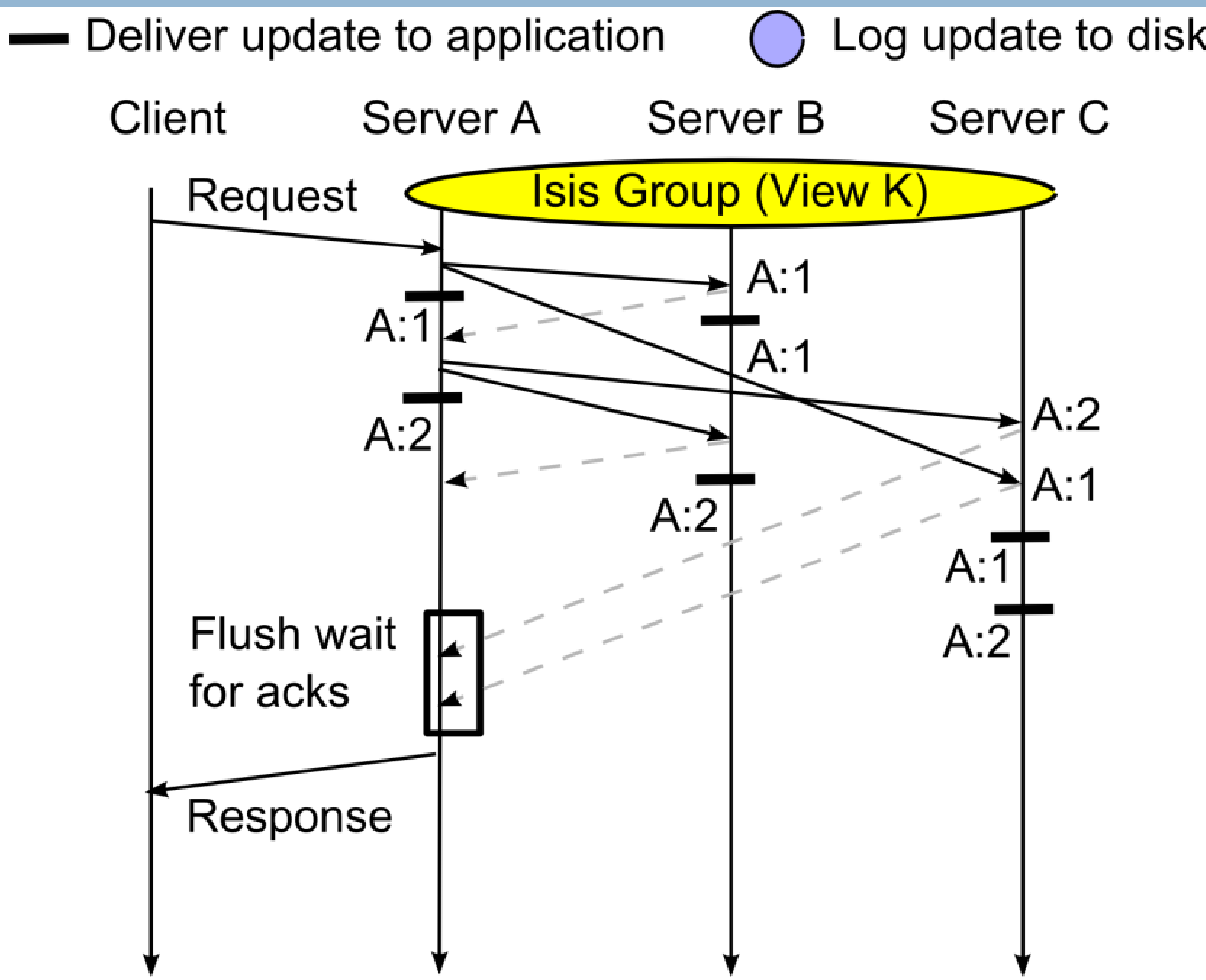
Virtually synchronous execution "amnesia" example (Send but without calling Flush)

What made it odd?



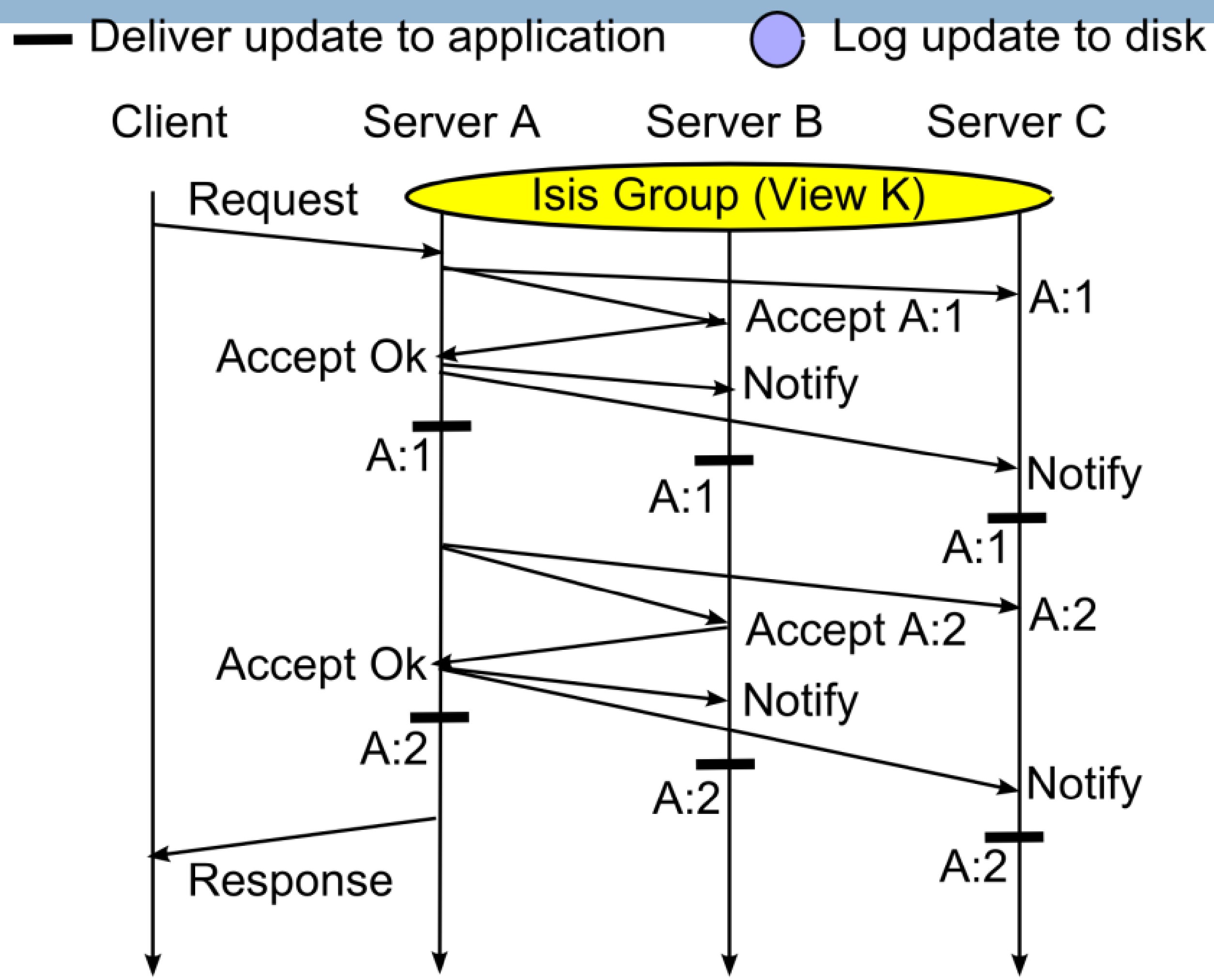
- In this example a network partition occurred and, before anyone noticed, some messages were sent and delivered
  - "Flush" would have blocked the caller, and SafeSend would not have delivered those messages
  - Then the failure <u>*erases*</u> the events in question: no evidence remains at all
  - So was this bad? OK? A kind of transient internal inconsistency that repaired itself?

Looking closely at that "oddity"



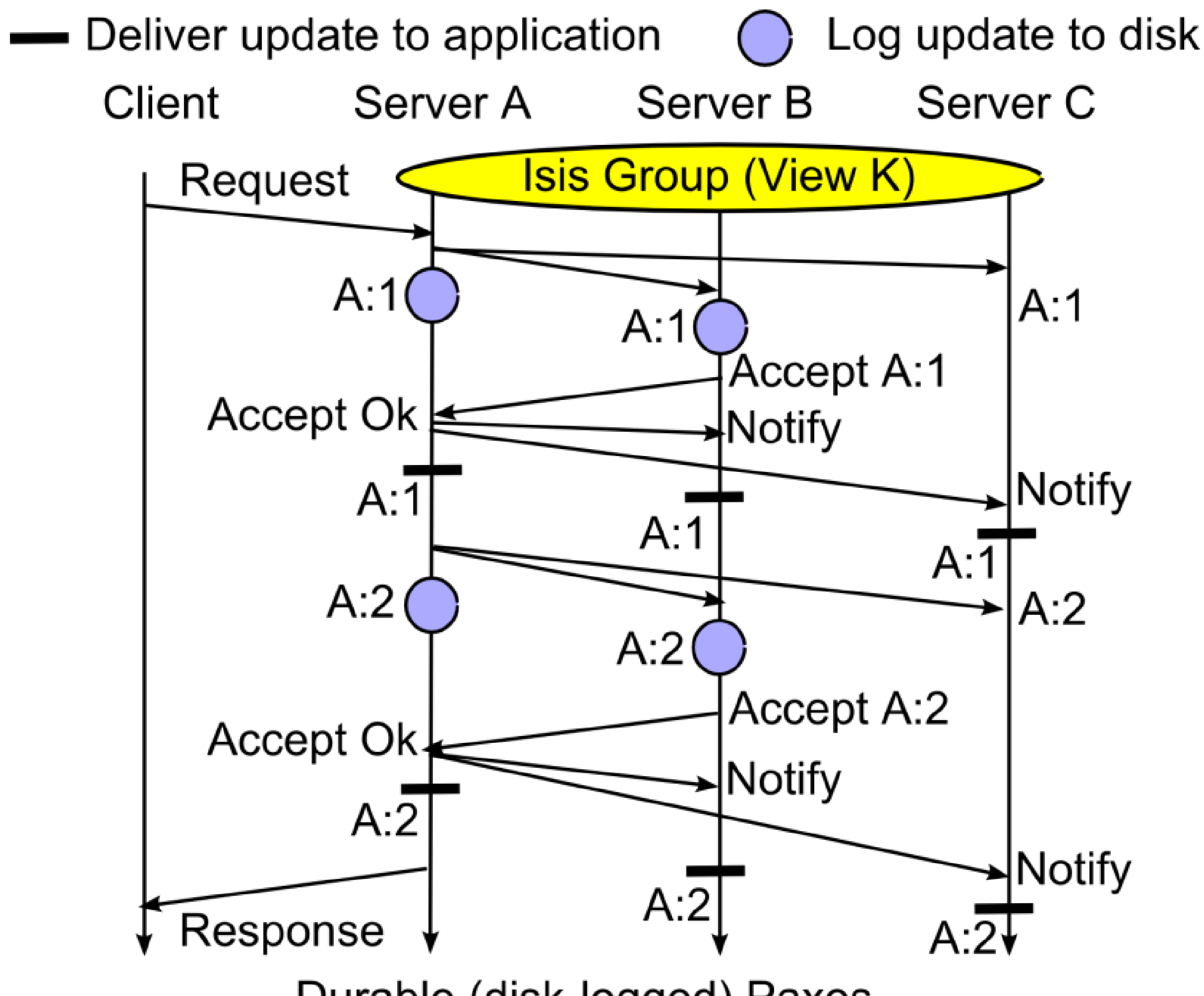
Send with Flush

Looking closely at that “oddity”



SafeSend (Paxos in memory)

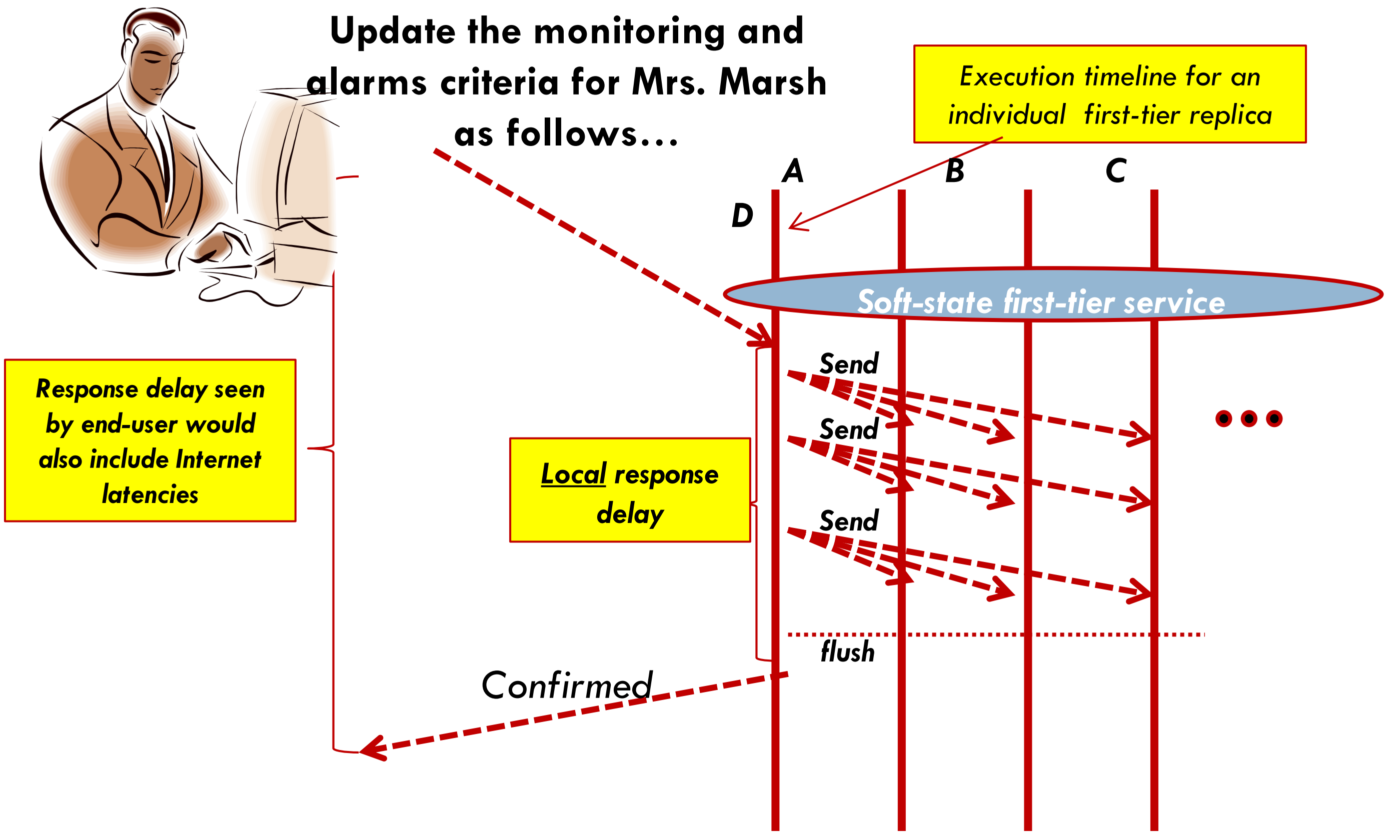
Looking closely at that "oddity"



Paxos avoided the issue… at a price

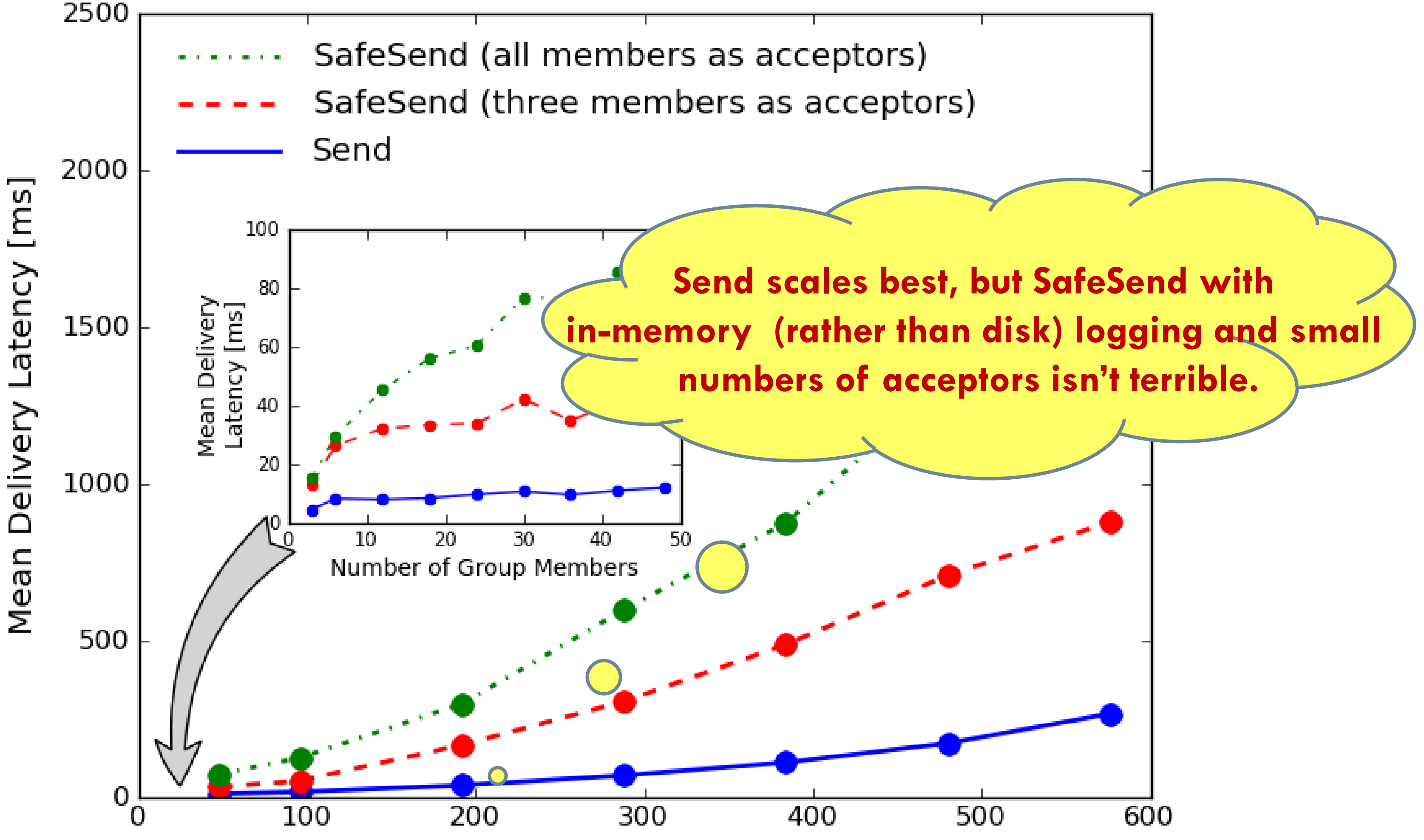
- SafeSend, Paxos and other multi-phase protocols don't deliver in the first round/phase
- This gives them stronger safety on a message by message basis, but also makes them slower and less scalable
- Is this a price we should pay for better speed?

Revisiting our medical scenario

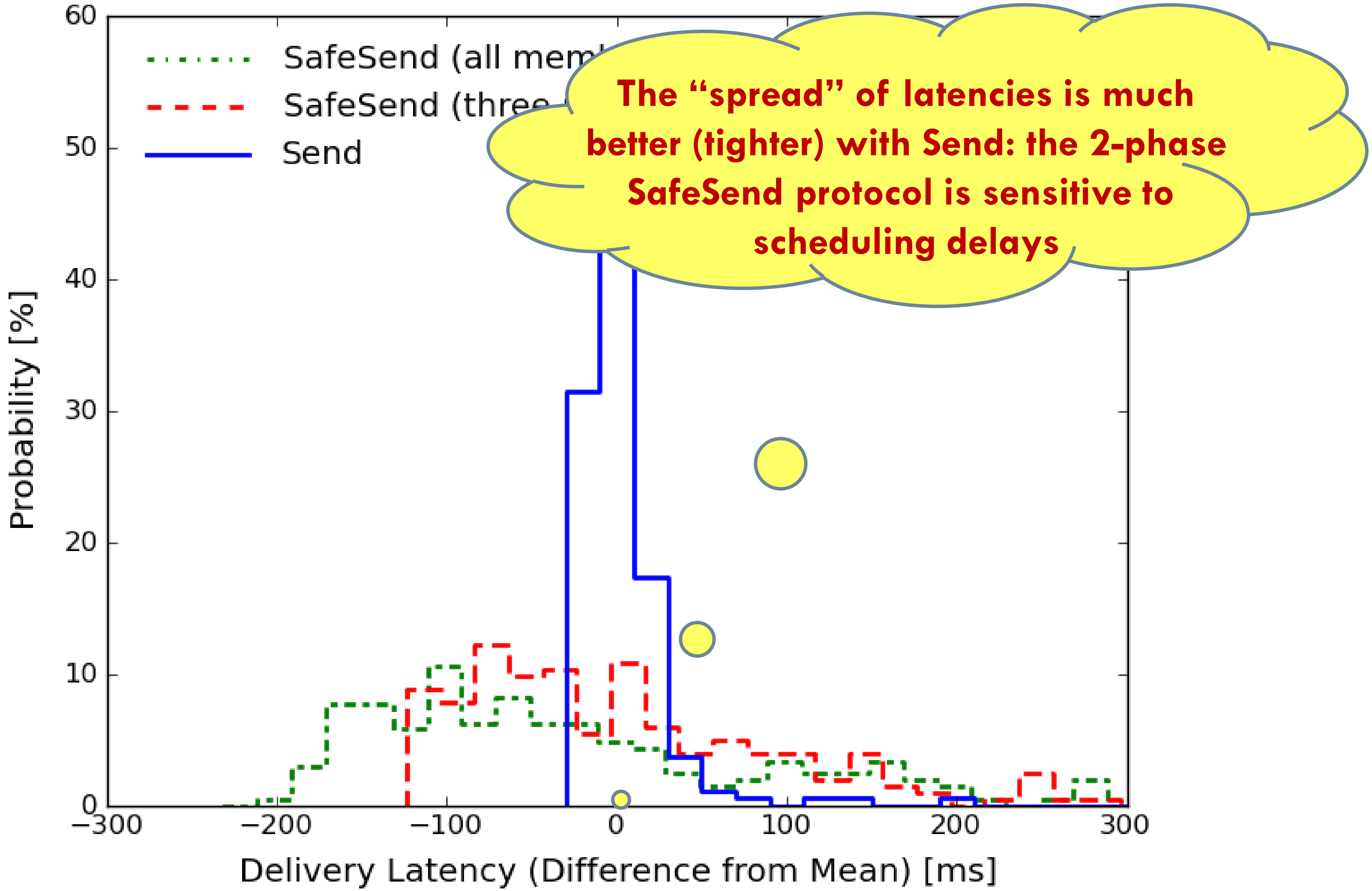


- An online monitoring system might focus on real-time response and be less concerned with data durability

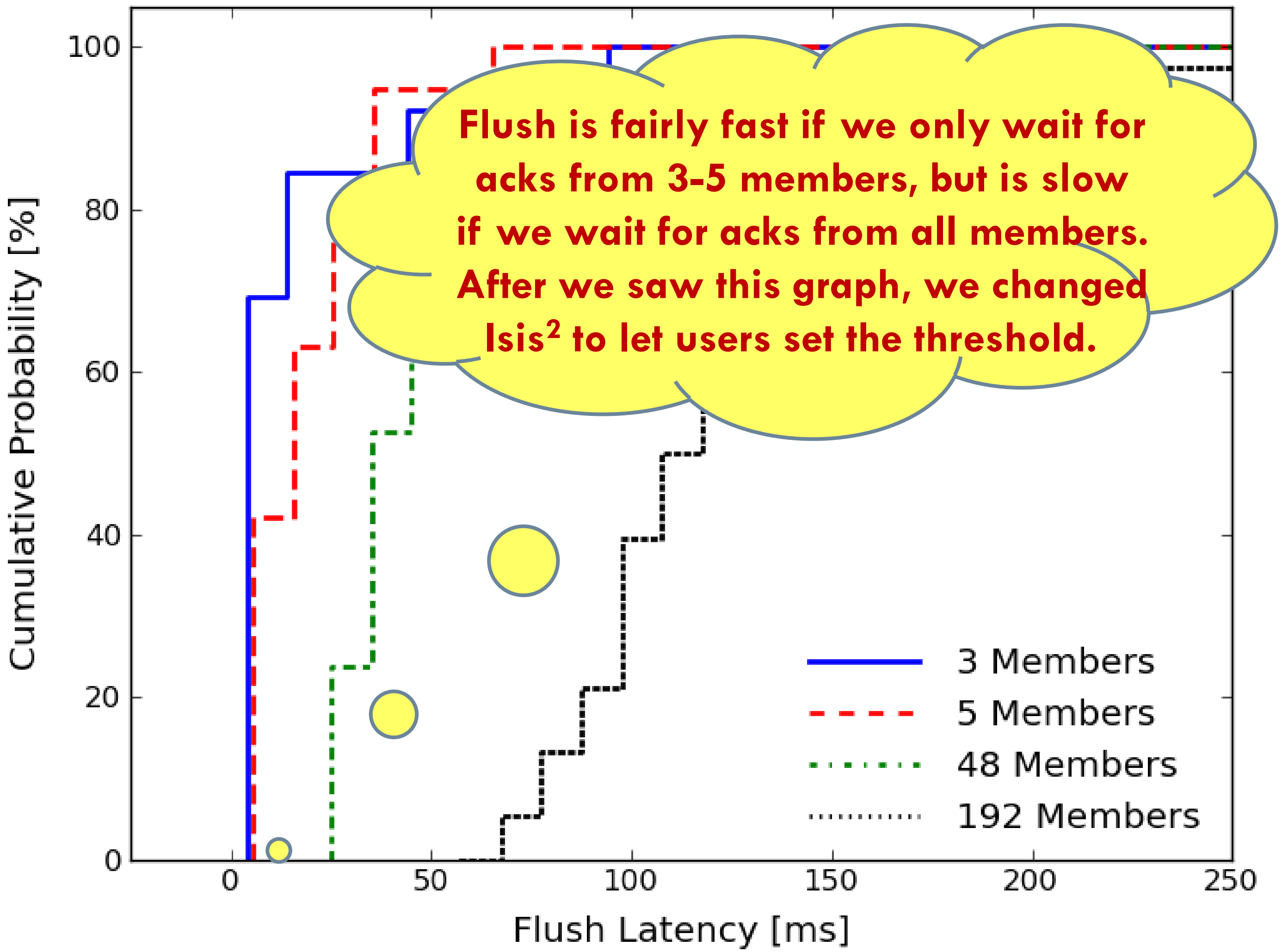
Isis²: Send v.s. in-memory SafeSend



Jitter: how "steady" are latencies?



Flush delay as function of shard size



First-tier "mindset" for tolerant *f* faults

- Suppose we do this:
  - Receive request
  - Compute locally using consistent data and perform updates on sharded replicated data, consistently
  - Asynchronously forward updates to services deeper in cloud but don't wait for them to be performed
  - Use the "flush" to make sure we have $f+1$ replicas

- Call this an "amnesia free" solution. Will it be fast enough? Durable enough?

Which replicas?

- One worry is this
  - If the first tier is totally under control of a cloud management infrastructure, elasticity could cause our shard to be entirely shut down "abruptly"

- Fortunately, most cloud platforms do have some ways to notify management system of shard membership
  - This allows the membership system to shut down members of multiple shards without ever depopulating any single shard
  - Now the odds of a sudden amnesia event become low

Advantage: Send+Flush?

- It seems that way, but there is a counter-argument

- The problem centers on the Flush delay
  - We pay it both on writes and on *some reads*
  - If a replica has been updated by an unstable multicast, it can't safely be read until a Flush occurs
  - Thus need to call Flush prior to replying to client even in a read-only procedure
    - Delay will occur *only* if there are pending unstable multicasts

We don't need this with SafeSend

- In effect, it does the work of Flush prior to the delivery ("learn") event
- So we have slower delivery, but now any replica is always safe to read and we can reply to the client instantly
- In effect the updater sees delay on his critical path, but the reader has no delays, ever

Advantage: SafeSend?

- Argument would be that with both protocols, there is a delay on the critical path where the update was initiated
- But only Send+Flush ever delays in a pure reader
- So SafeSend is faster!
  - But this argument is flawed…

Flaws in that argument

- The delays aren't of the same length (in fact the pure reader calls Flush but would rarely be delayed)

- Moreover, if a request does multiple updates, we delay on each of them for SafeSend, but delay just once if we do Send…Send…Send…Flush

- How to resolve?

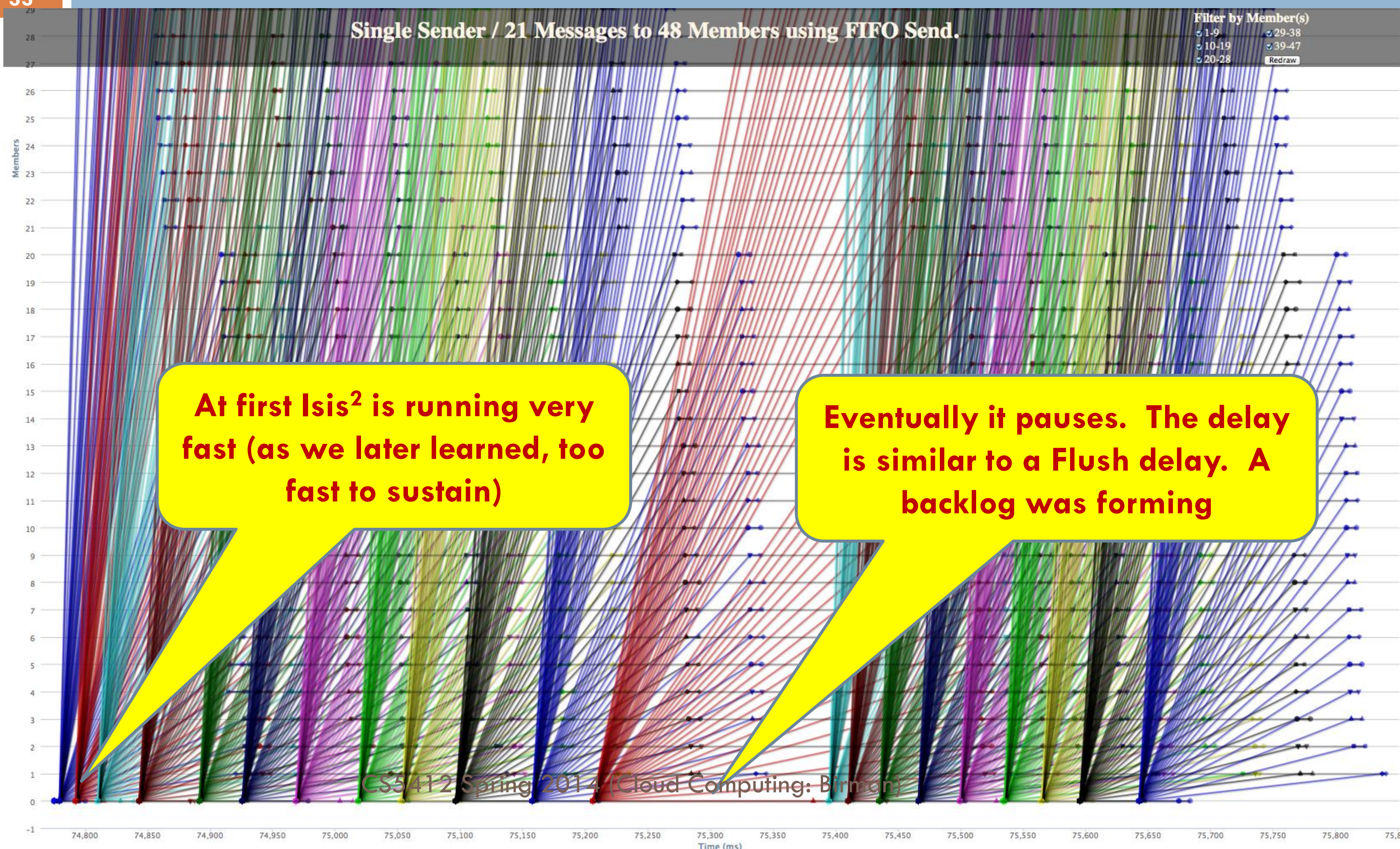
Only real option is to experiment

- In the cloud we often see questions that arise at
  - Large scale,
  - High event rates,
  - … and where millisecond timings matter
- Best to use tools to help visualize performance
- Let’s see how one was used in developing $Isis^2$

Something was… strangely slow

- We weren't sure why or where
- Only saw it at high data rates in big shards
- So we ended up creating a visualization tool just to see how long the system needed from when a message was sent until it was delivered
- Here's what we saw

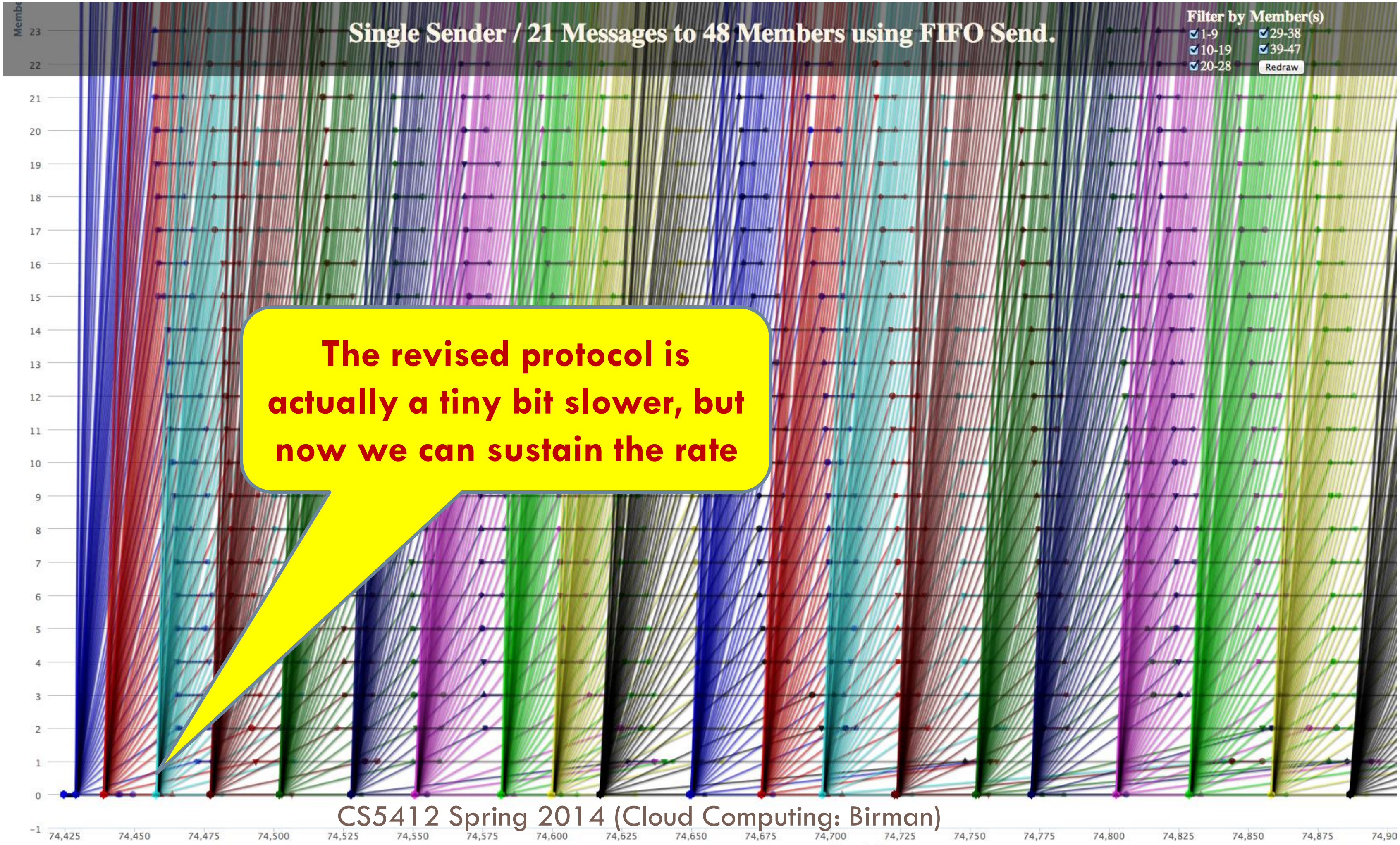
Debugging: Stabilization bug



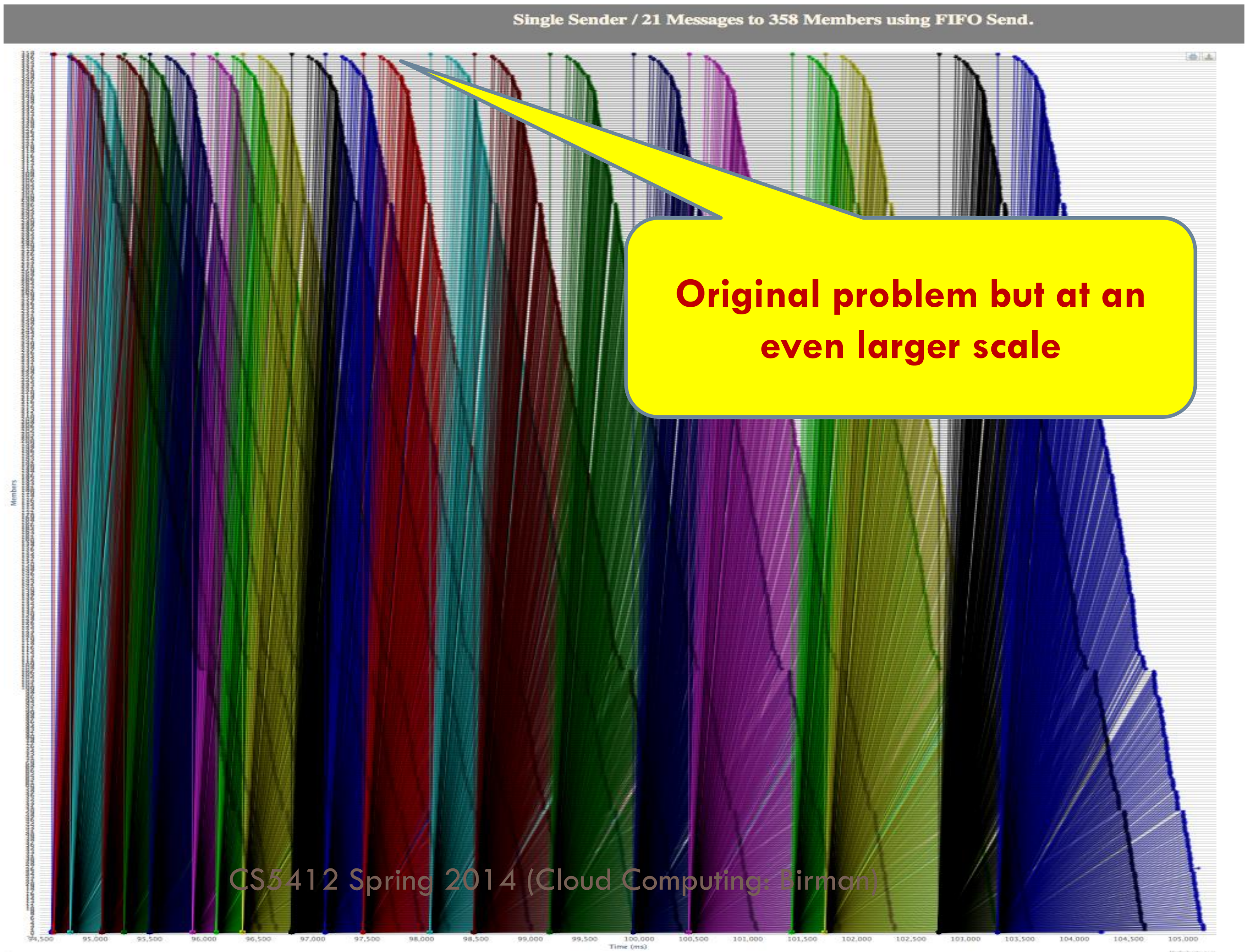
At first Isis[2] is running very fast (as we later learned, too fast to sustain)

Eventually it pauses. The delay is similar to a Flush delay. A backlog was forming

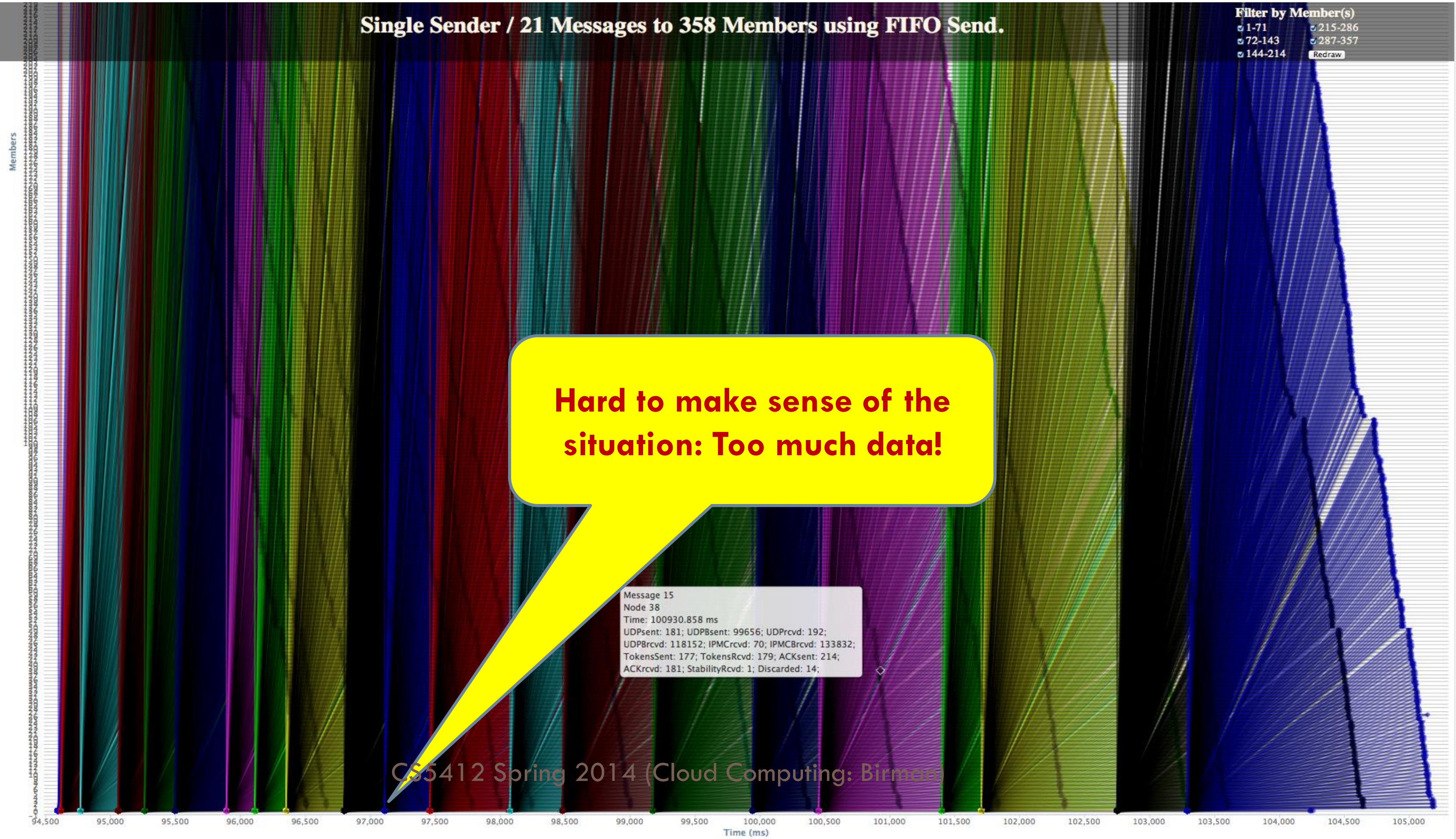
Debugging : Stabilization bug fixed



Debugging : 358-node run slowdown

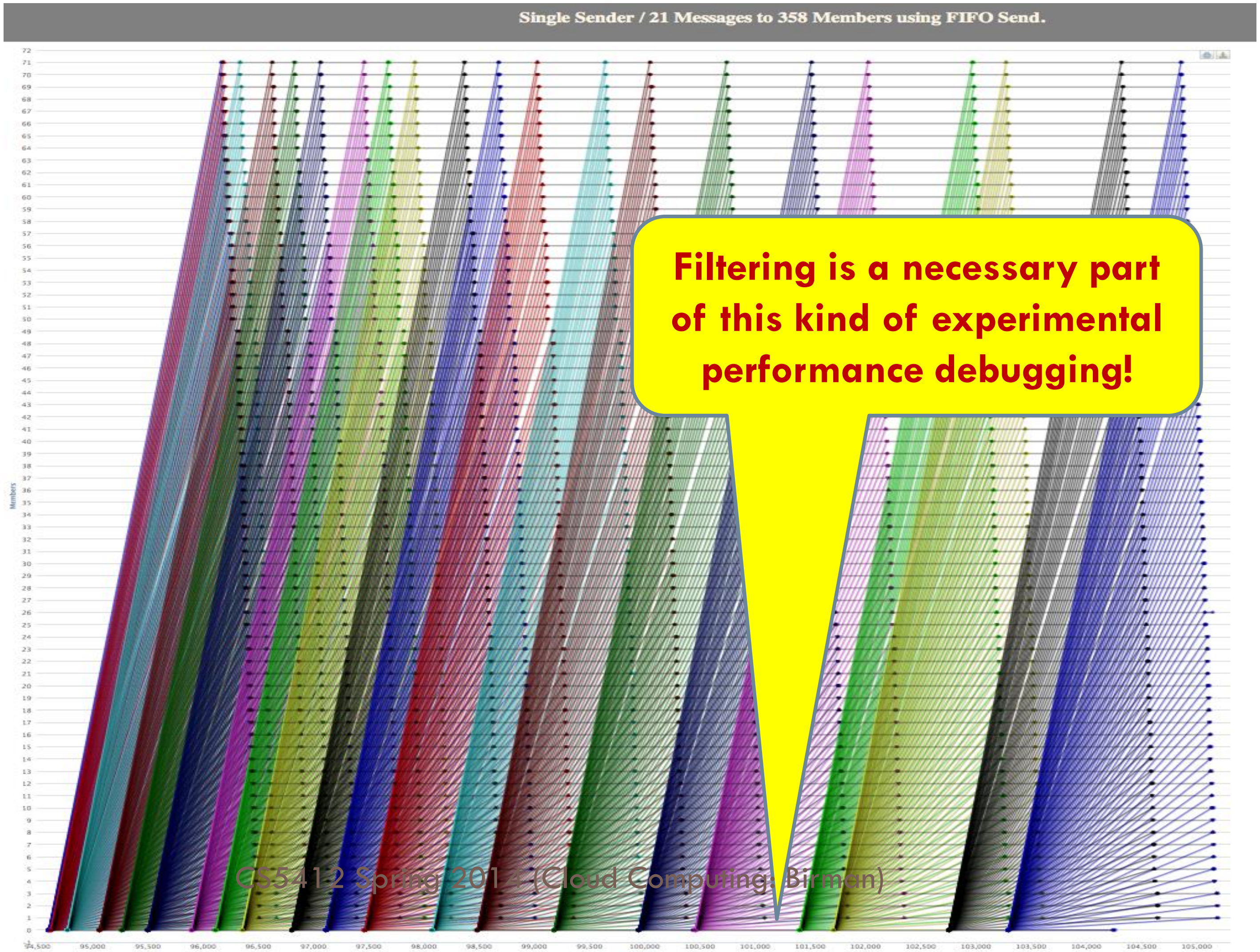


358-node run slowdown: Zoom in



Hard to make sense of the situation: Too much data!

358-node run slowdown: Filter



Filtering is a necessary part of this kind of experimental performance debugging!

What did we just see?

- Flow control is pretty important!
- With a good multicast flow control algorithm, we can garbage collect spare copies of our Send or OrderedSend messages before they pile up and stay in a kind of balance
  - *Why did we need spares?*
    … To resend if the sender fails.
  - *When can they be garbage collected?*
    … When they become stable
  - *How can the sender tell?*
    … Because it gets acknowledgements from recipients

What did we just see?

- … in effect, we saw that one can get a reliable virtually synchronous ordered multicast to deliver messages at a steady rate

Would this be true for Paxos too?

- Yes, for some versions of Paxos
  - The Isis[2] version of Paxos, SafeSend, works a bit like OrderedSend and is stable for a similar reason
  - There are also versions of Paxos such a ring Paxos that have a structure designed to make them stable and to give them a flow control property

- But not every version of Paxos is stable in this sense

Interesting insight…

- In fact, ***most versions of Paxos will tend to be bursty***….
  - The fastest Q_W group members respond to a request before the slowest N-Q_W, allowing them to advance while the laggards develop a backlog
  - This lets Paxos surge ahead, but suppose that conditions change (remember, the cloud is a world of strange scheduling delays and load shifts). One of those laggards will be needed to reestablish a quorum of size Q_W
  - … but it may take a while for them to deal with the backlog and join the group!
- Hence Paxos (as normally implemented) will exhibit long delays, triggered when cloud-computing conditions change

Conclusions?

- A question like "how much durability do I need in the first tier of the cloud" is easy to ask… harder to answer!

- Study of the choices reveals two basic options
  - Send + Flush
  - SafeSend, in-memory

- They actually are similar but SafeSend has an internal "flush" before any delivery occurs, on each request
  - SafeSend seems more costly
  - Steadiness of the underlying flow of messages favors optimistic early delivery protocols such as Send and OrderedSend. Classical versions of Paxos may be very bursty