CS5412: HOW DURABLE SHOULD IT BE?

Lecture XV

Ken Birman

Choices, choices…

- A system like Vsync lets you control message ordering, durability, while Paxos opts for strong guarantees.

- With Vsync, it works best to start with total order (g.OrderedSend) but then relax order to speed things up if no conflicts (inconsistency) would arise.

How much ordering does it need?

- Example: we have some group managing replicated data, using g.OrderedSend() for updates

- But perhaps only one group member is connected to the sensor that generates the updates
  - With just one source of updates, g.Send() is faster
  - Vsync will discover this simple case automatically, but in more complex situations, the application designer might need to explictly use g.Send() to be sure.

Question: Why?

- With one sender, everything is in "sender" or "FIFO" ordering: that sender sends $x_0\ x_1\ x_2\ \ldots$
- The g.Send multicast keeps updates in sender order
- So g.OrderedSend and g.Send actually promise the identical thing! g.OrderedSend has extra logic for a case that won't arise, namely two conflicting updates from two different senders.

How does Vsync optimize this case?

- Because this pattern is pretty common, Vsync has special logic for it.
  - If a group starts up, it initially tries to use g.Send when you request g.OrderedSend… this is invisible to you but each call to g.OrderedSend "maps" to g.Send.
  - Vsync automatically switches to the real g.OrderedSend mode automatically if a *second* sender issues a concurrent g.OrderedSend multicast.
- So g.OrderedSend is kind of a one-size-fits-all choice

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

Durability in real systems

- Database components normally offer durability
- Paxos also has durability.
  - Like a database of "messages" saved for replay into services that need consistent state
- Systems like Vsync focus on consistency for multicast and for these, durability is optional (and costly)

Should Consistency "require" Durability?

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

- Normally we run Paxos with the messages (the "list of commands") on disk, and hence Paxos can survive any crash
  - In Vsync, this is g.SafeSend with the "DiskLogger" active
  - But doing so slows the protocol down compared to not logging messages so durably

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?

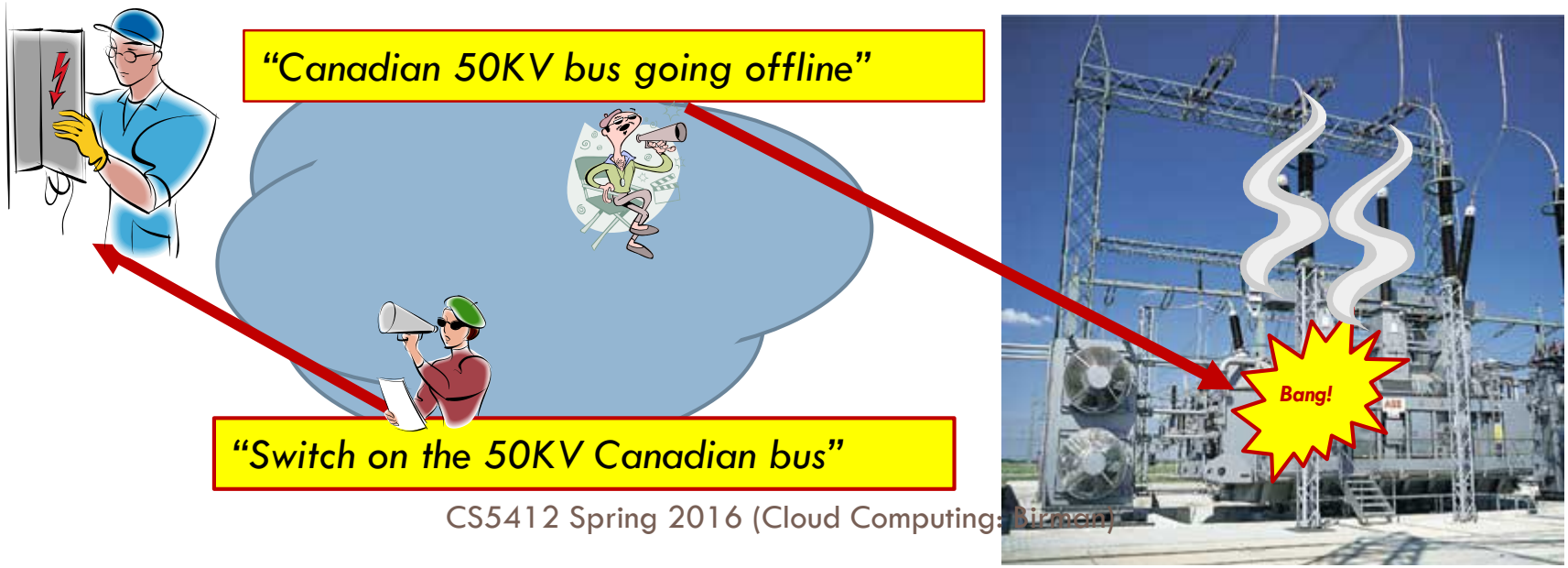
- Anything that was cached but "really" lives in a database or file server elsewhere in the cloud
  - If you wake up with a cold cache, you just need to reload it with fresh data
- Monitoring parameters, control data that you need to get "fresh" in any case
  - Includes data like "The current state of 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 definitely might want durability, but if applications are replicating data in tier1, Paxos is too costly: it works hard to provide a property that has no real meaning in tier1
- Any tier1 service that wants to persist data must do so by writing to files in a deeper layer of the cloud, like Amazon S3. Local files aren't persistent.
- Implication: no, you wouldn't want Paxos!

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

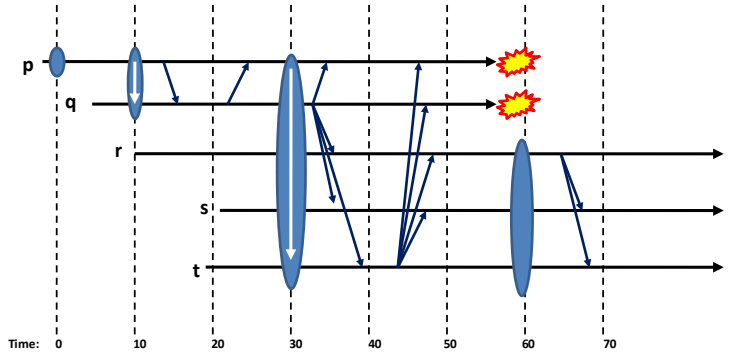


We do need consistency…

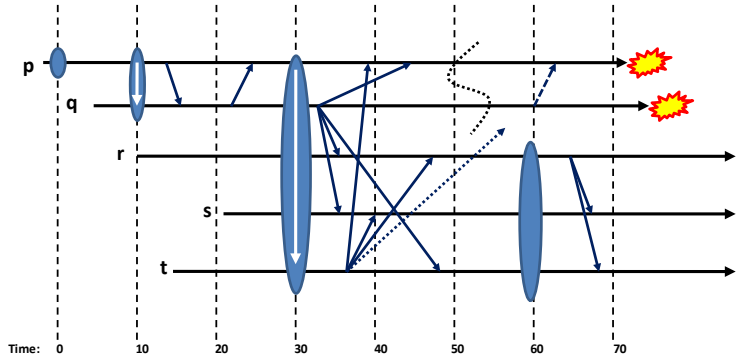
- But Vsync offers consistency even for g.OrderedSend
- For a purpose like this, there is no need for anything fancier.

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

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***

So why does Vsync include Paxos?

- Inside Vsync, Paxos is supported by g.SafeSend

- A more costly protocol that stores data into disk files
  - Not intended for tier1 use! This is for Vsync use deeper in the cloud, where a machine that restarts will still remember its files from before the crash
  - Vsync is trying to be universal: use it anywhere, make smart choices matched to your use case!

SafeSend vs OrderedSend vs Send

- SafeSend is durable and totally ordered and never has any form of odd behavior. Logs messages, replays them after a group shuts down and then later restarts. == Paxos.

- OrderedSend is much faster but doesn't log the messages (not durable) and also is "optimistic" in a sense we will discuss. Sometimes must combine with Flush.

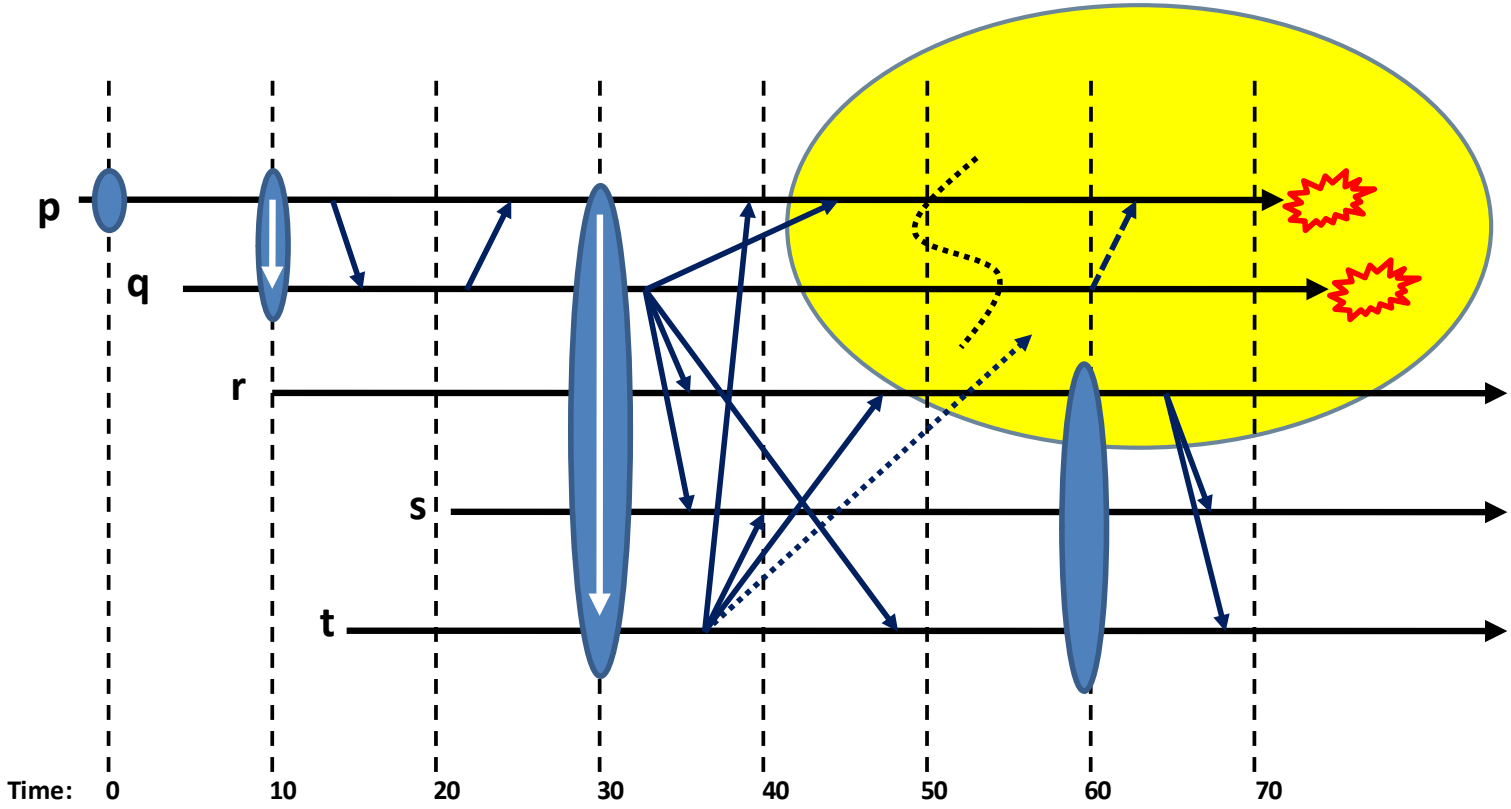
- Send is FIFO and optimistic, and also may need to be combined with Flush.

One oddity: a weird crash case

- There is one thing you need to be aware of with g.OrderedSend.

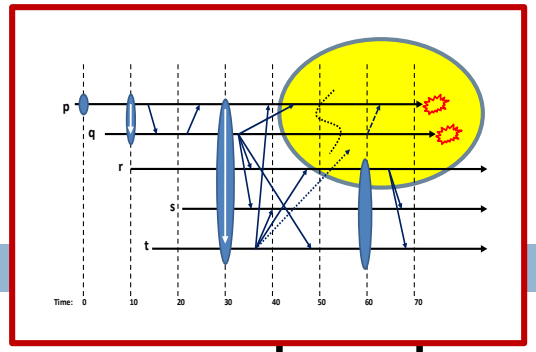
- To understand it, first think about writing data to files using printf in C or cout in C++.
  - Have you ever noticed that if a program crashes, the tail end of the file might not be written?
  - This is because data is buffered and written in *blocks*
  - With files, you need to call "flush" to be sure the data was output, and "fsync" to be sure it is on disk.

Analgous issue with g.OrderedSend



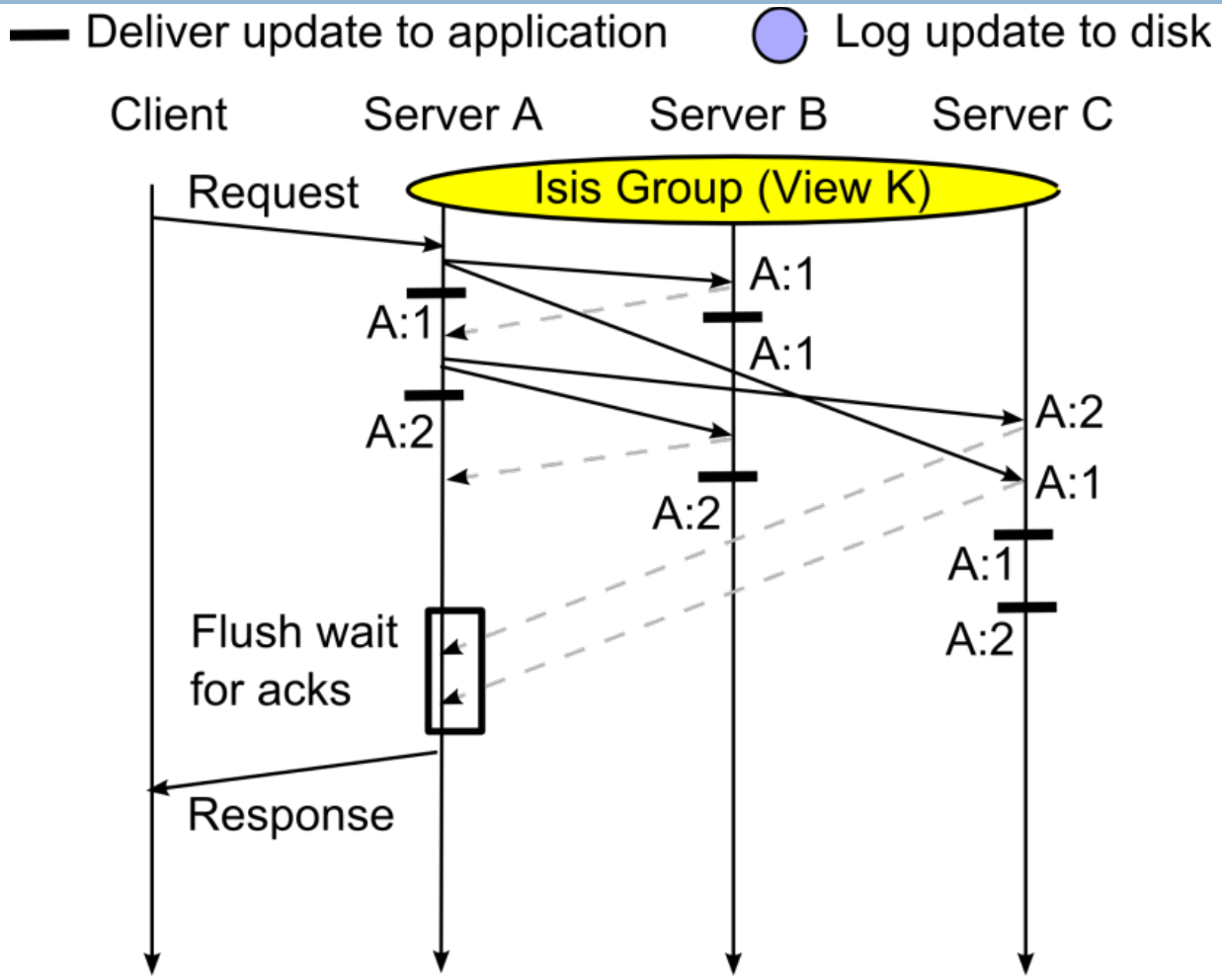
Virtually synchronous execution "amnesia" example (Send but without calling Flush)

What made it odd?

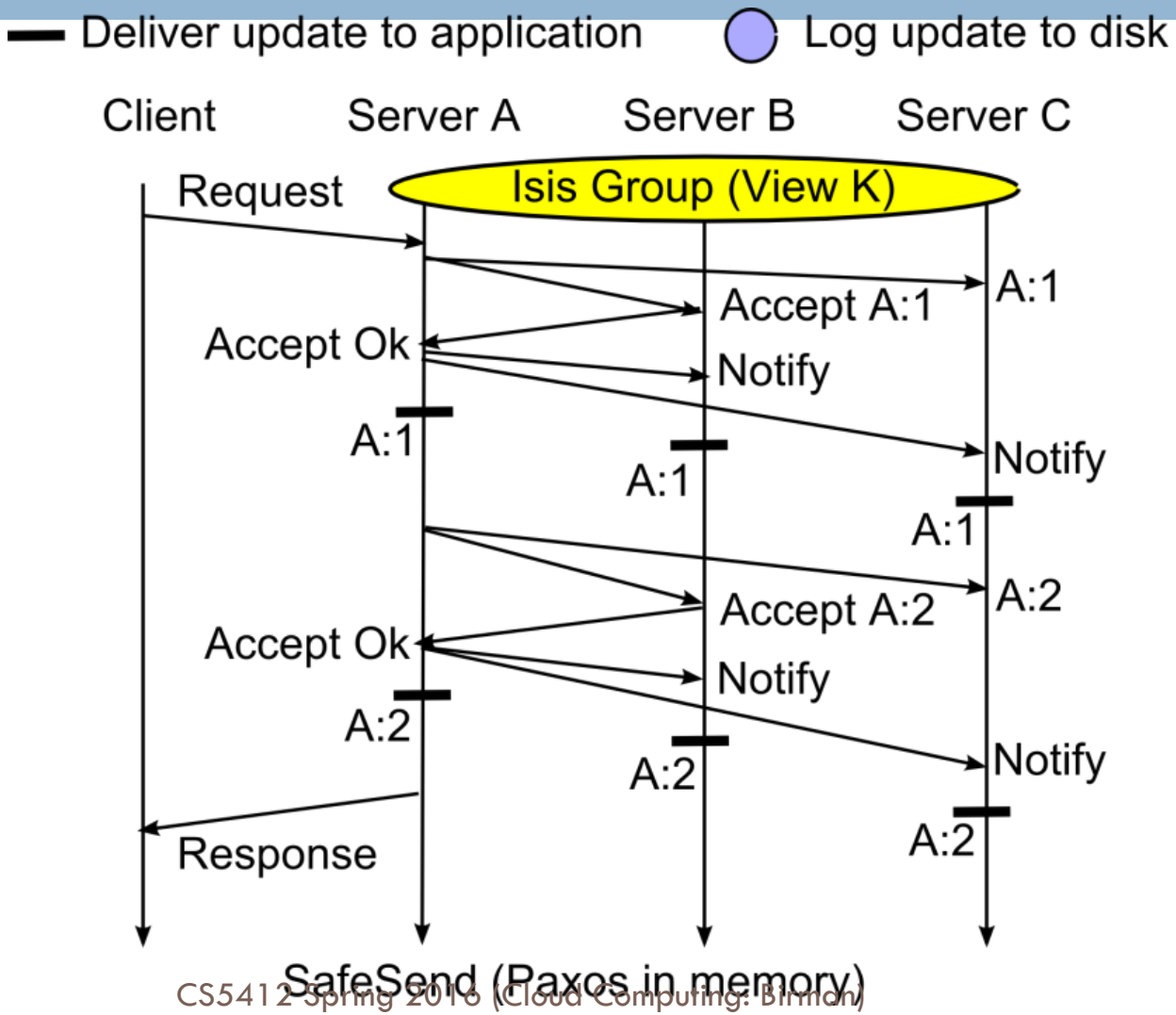


- In this example a network partition occurred and, before anyone noticed, some messages were sent and delivered
  - "Flush" would have blocked the caller, and SafeSend would not have delivered those messages
  - Then the failure *erases* the events in question: no evidence remains at all
  - So was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

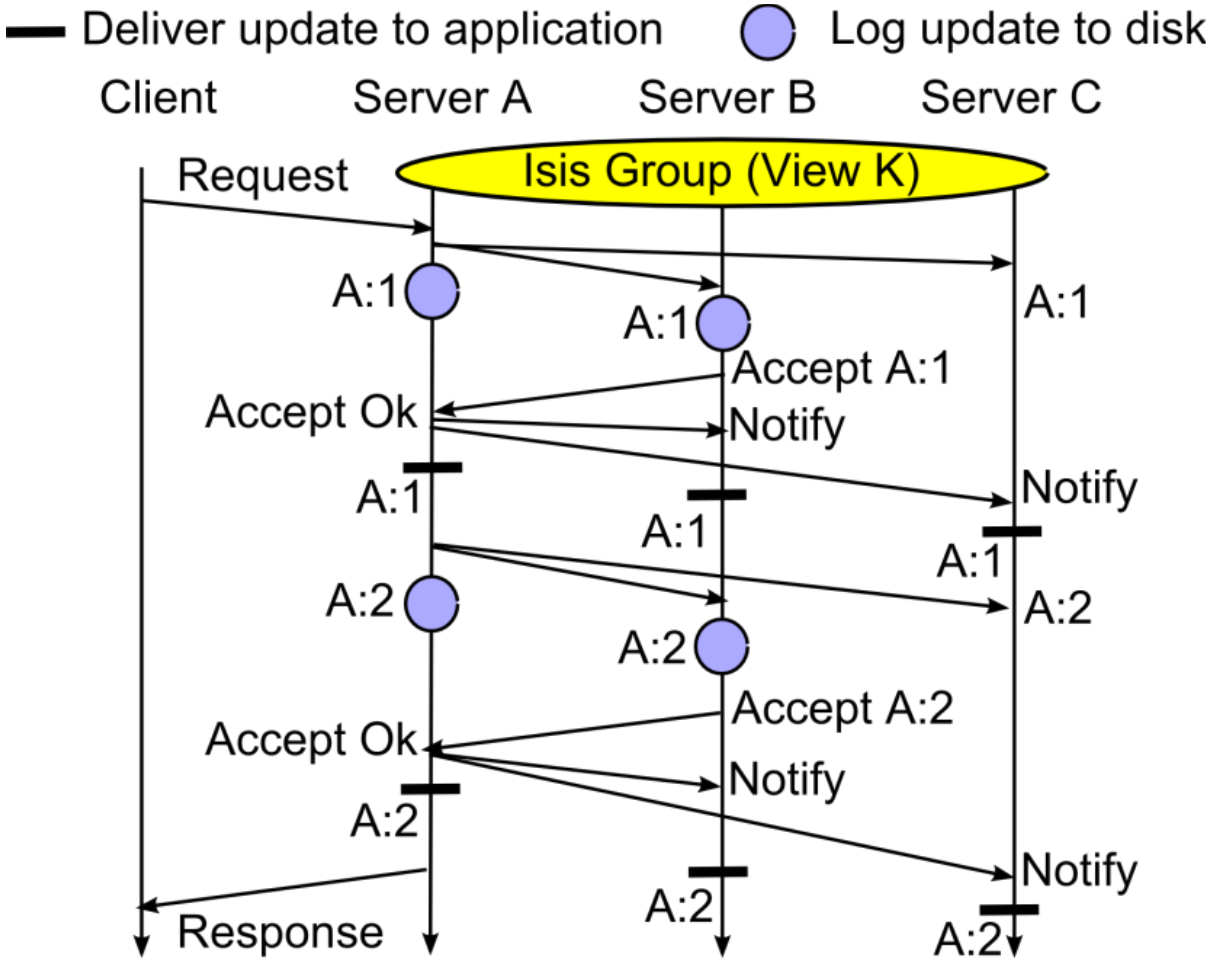
Looking closely at that "oddity"



Looking closely at that "oddity"



Looking closely at that "oddity"



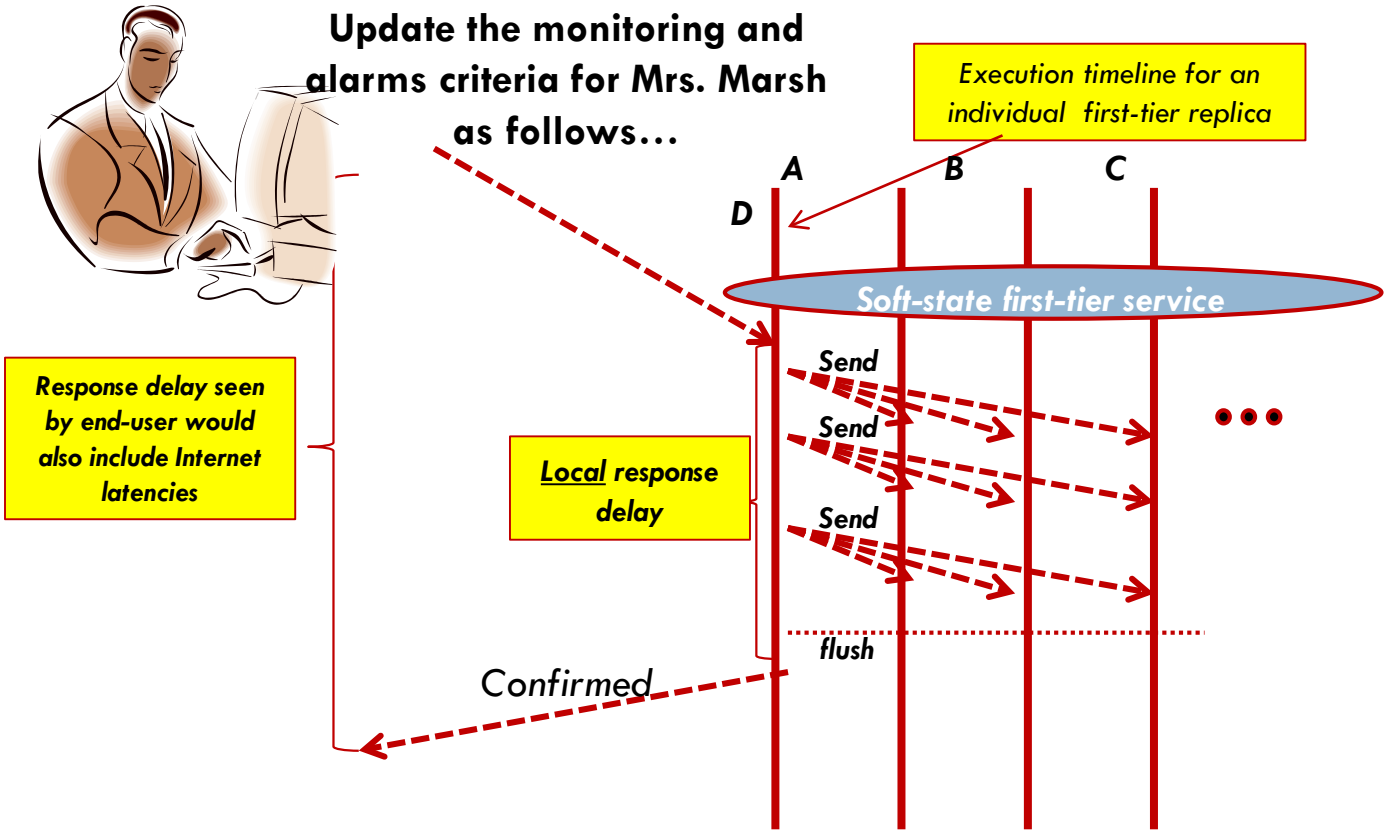
Paxos avoided the issue… at a price

- 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?

Do you recall our medical example?

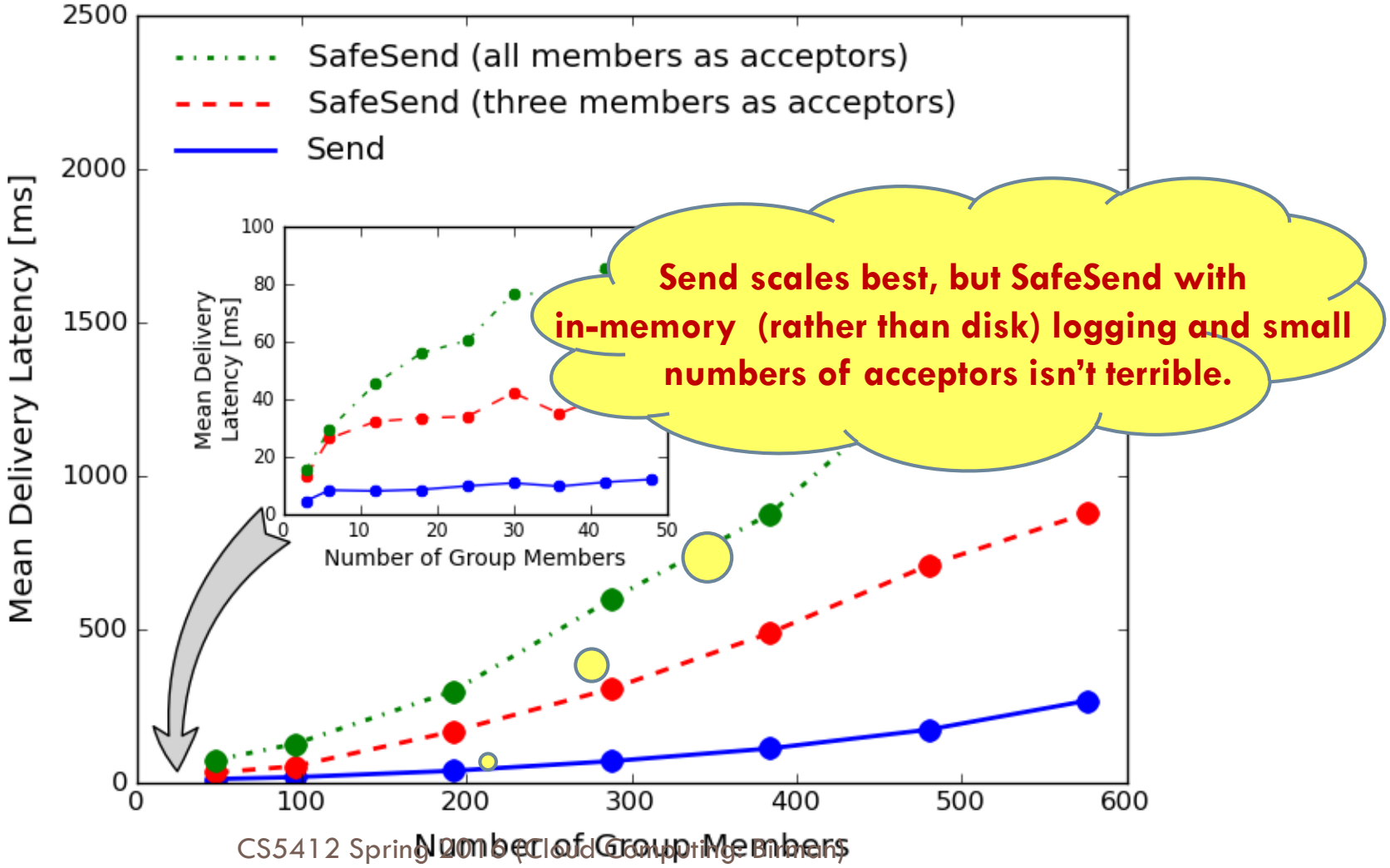
- Doctor updates the medical prescriptions for a patient: this is a kind of persistent database update
  - So it needs Paxos, implement via g.SafeSend
- Technician updates the online monitoring system
  - Configuration of that system changes all the time
  - If something crashes, on reboot it starts by asking "what should I be doing right now?"
  - So g.OrderedSend or g.Send will suffice

Medical monitoring scenario

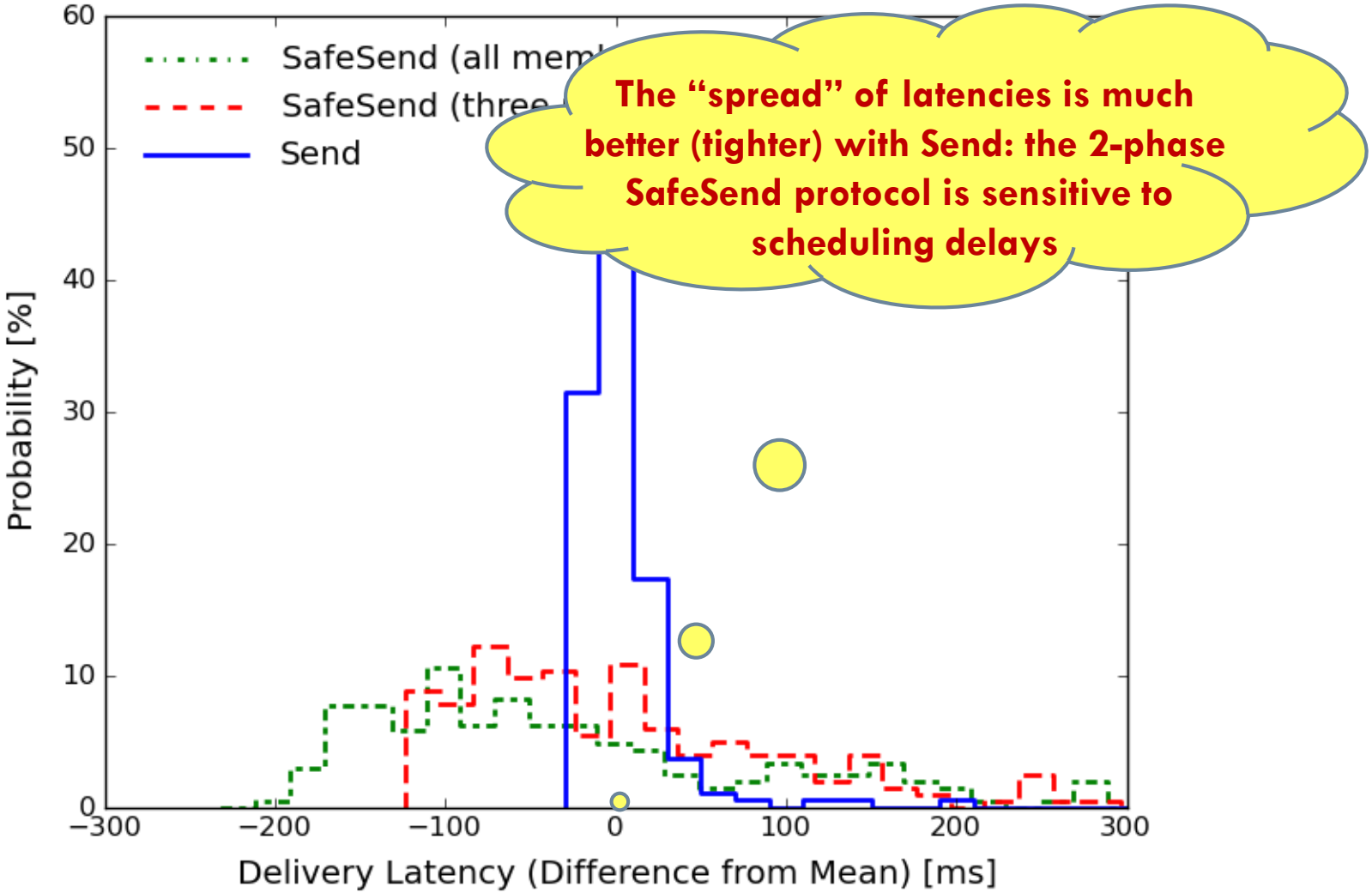


- An online monitoring system might focus on real-time response and be less concerned with data durability

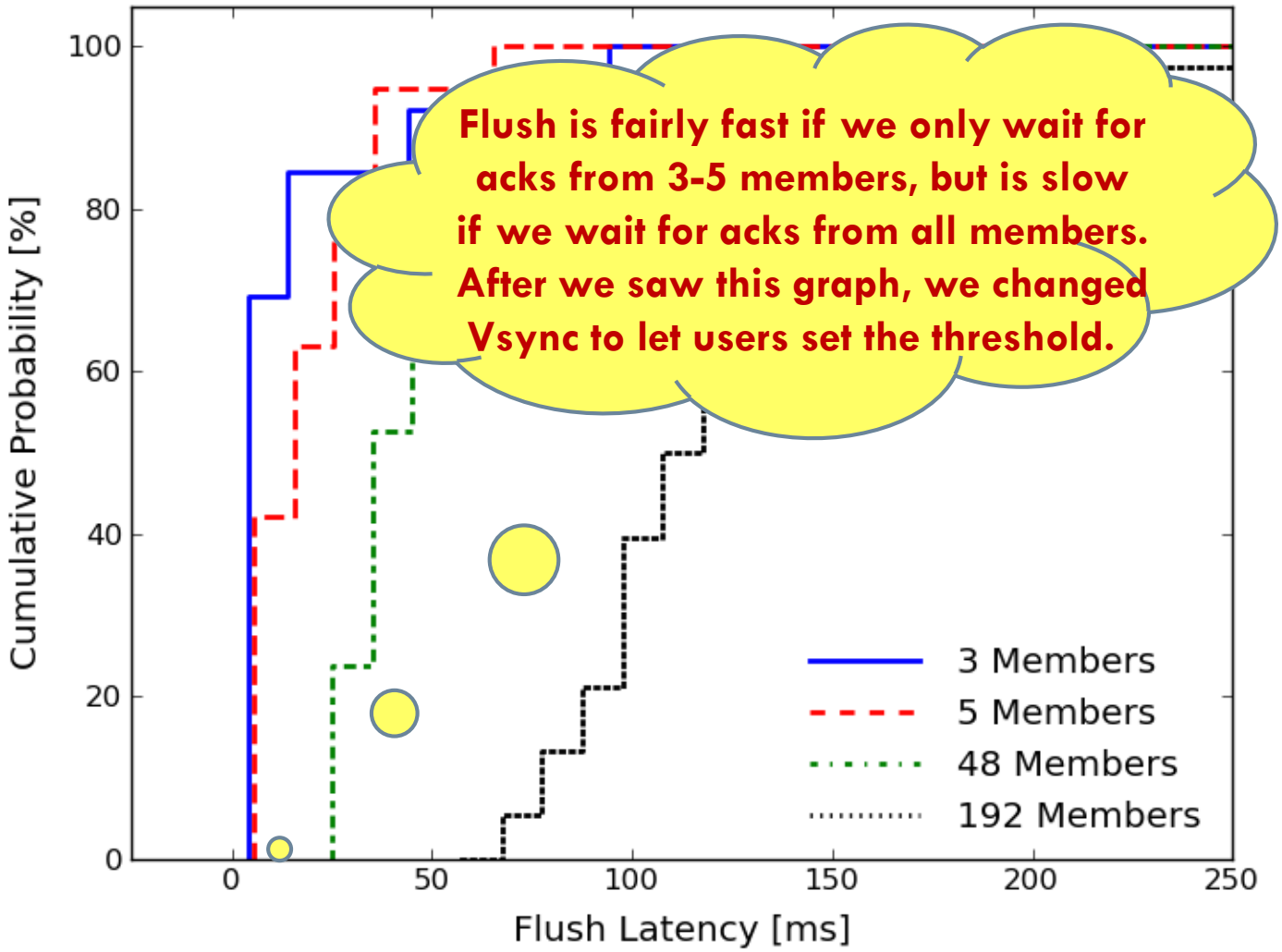
Vsync: Send v.s. in-memory SafeSend



Jitter: how "steady" are latencies?



Flush delay as function of shard size



Advantage: [Ordered]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

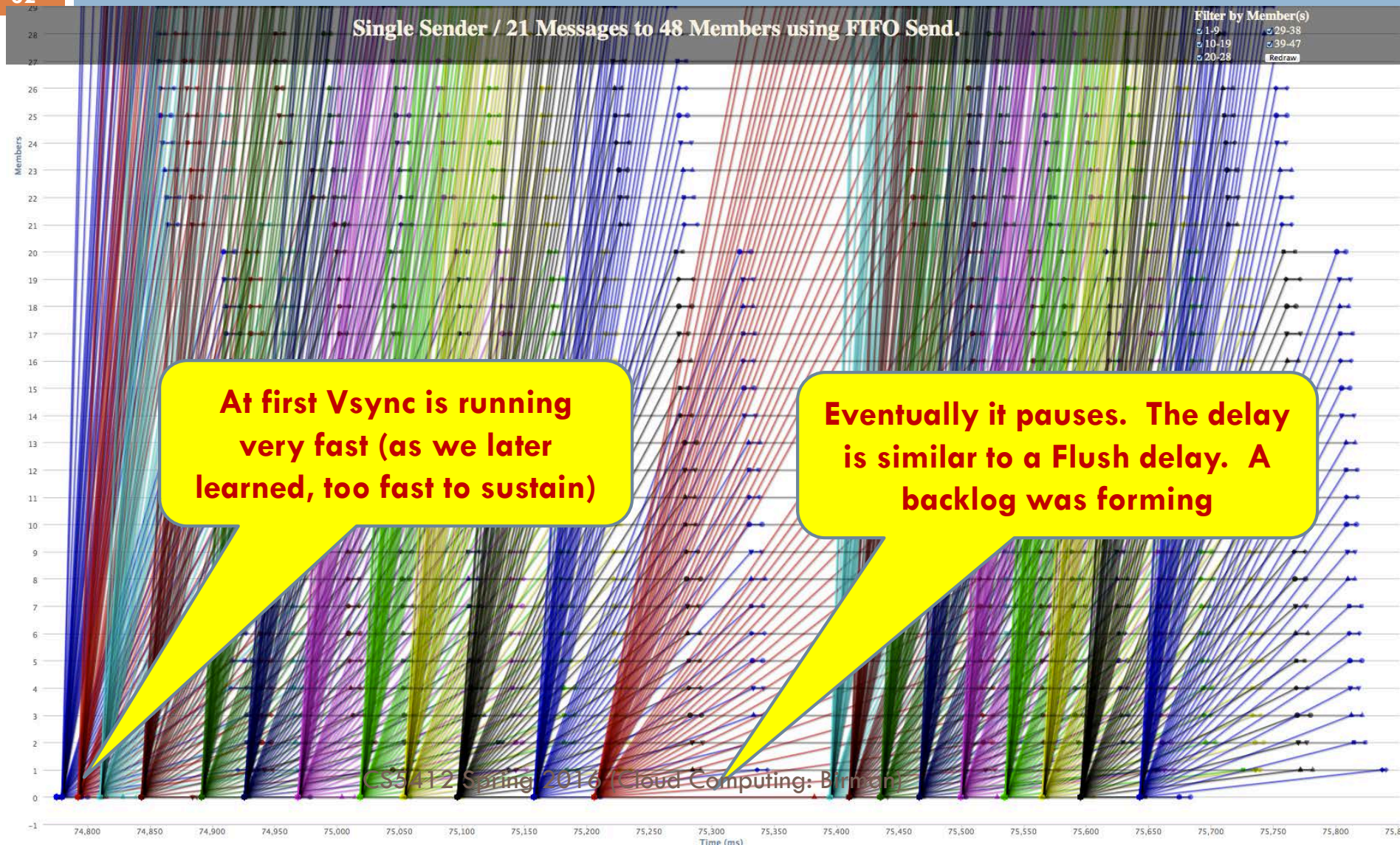
Only real option is to experiment

- 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 Vsync

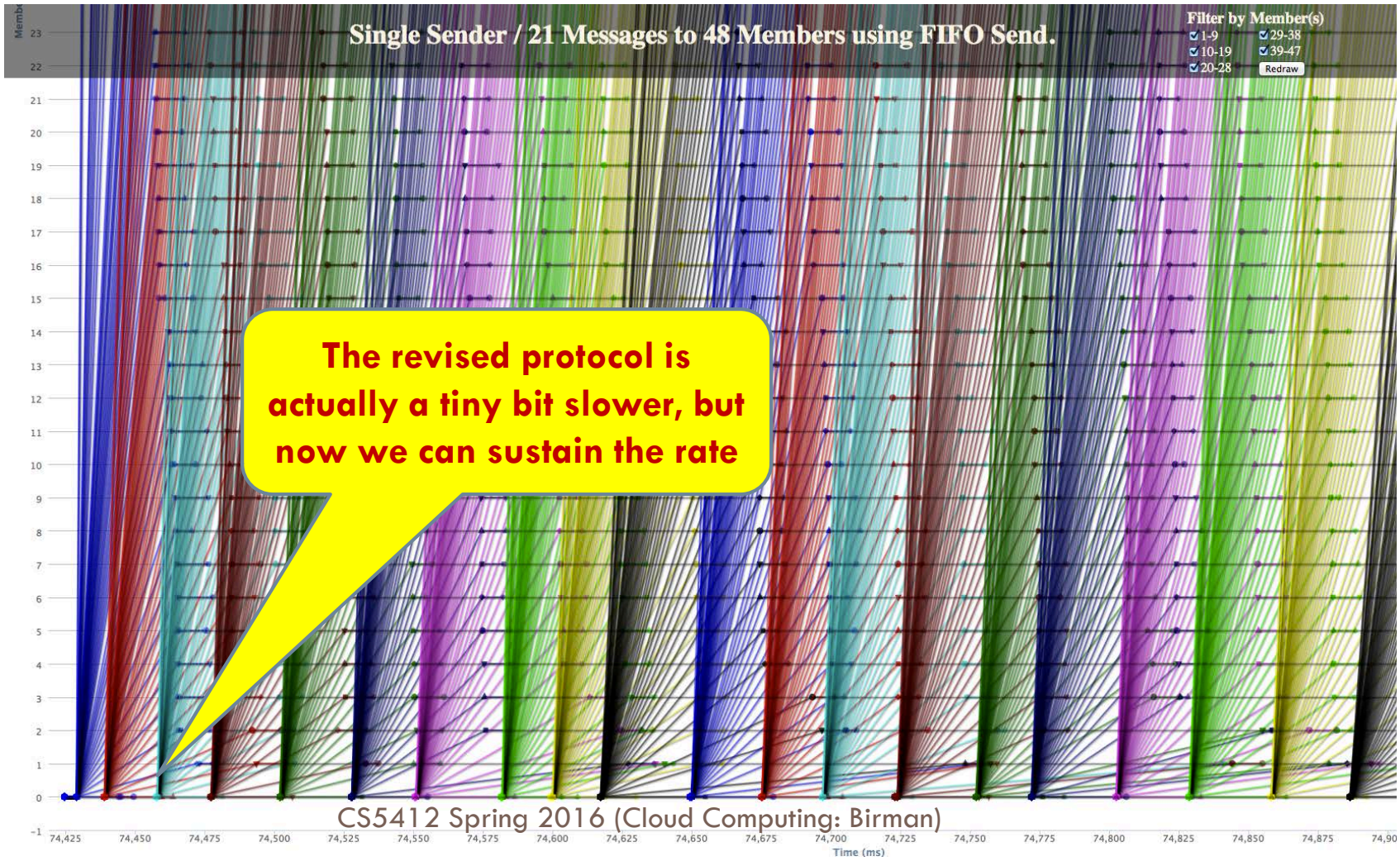
Something was… strangely slow

- 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

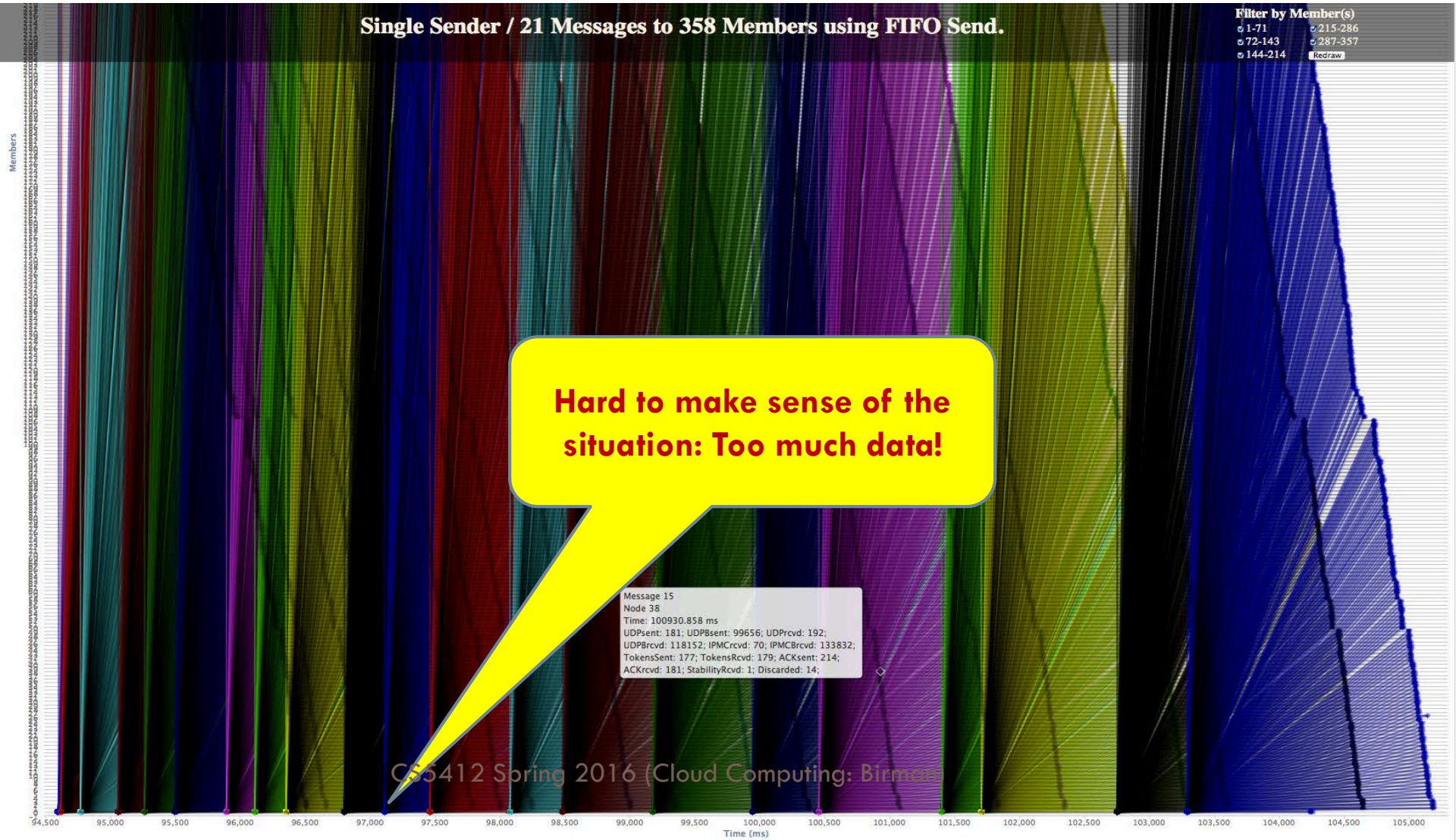


Debugging : Stabilization bug fixed

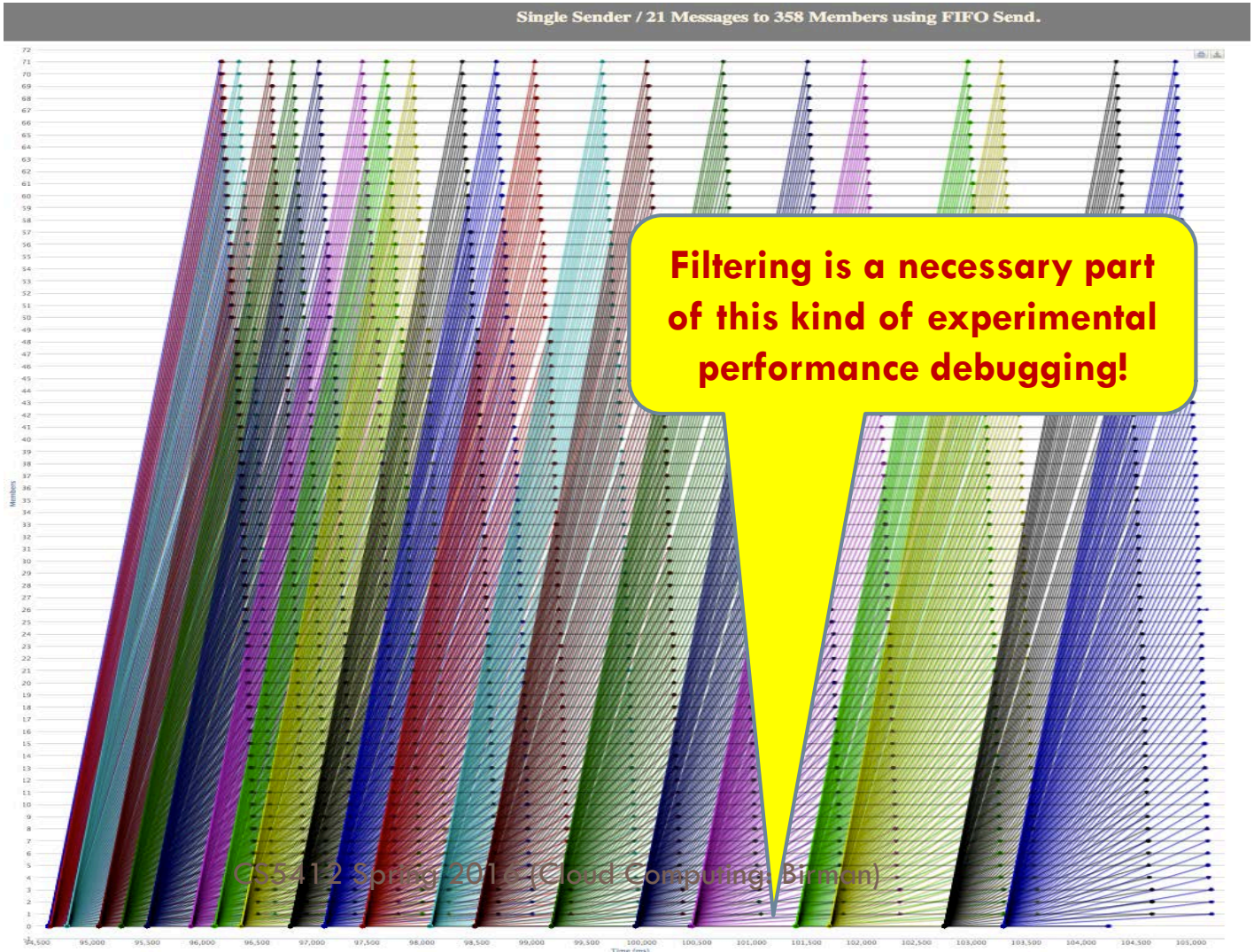


Debugging : 358-node run slowdown

358-node run slowdown: Zoom in



358-node run slowdown: Filter



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

Interesting insight…

- In fact, ***most versions of Paxos will tend to be bursty too. . .***
- 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
  - OrderedSend + Flush, or Send + Flush
    - In theory, OrderedSend will automatically notice that Send will suffice
    - But if you know for sure and want to be sure it will be used, just say so
  - SafeSend: Paxos, but this is overkill

- 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