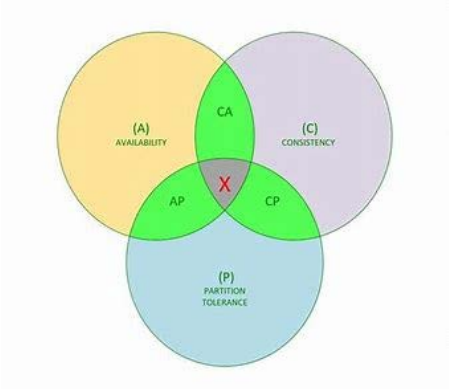
CS5412 / LECTURE 8
REPLICATION AND CONSISTENCY (PART II: PRACTICAL OPTIONS)

Ken Birman

Fall, 2022

RECAP



Eric Brewer told us that stale data and inconsistency is fine. Why worry? CAP and BASE work for many cloud computing web sites, like Amazon.com

But we realized that *if a μ-service does the whole job* – it holds its own state and also does the computing, and uses the primary-partition membership approach – then P won't matter. We can opt for C+A.

So we introduced state machine replication and virtual synchrony.

RECAP: STATE MACHINE REPLICATION

This is a model in which we have deterministic programs

They see one update at a time and apply the updates in the same order. There is a communications version of this (atomic multicast) and a log-append version (persistent replicated logging).

If our replicas start up in sync, they stay in sync.

RECAP: ATOMIC MULTICAST AND DURABLE REPLICATION

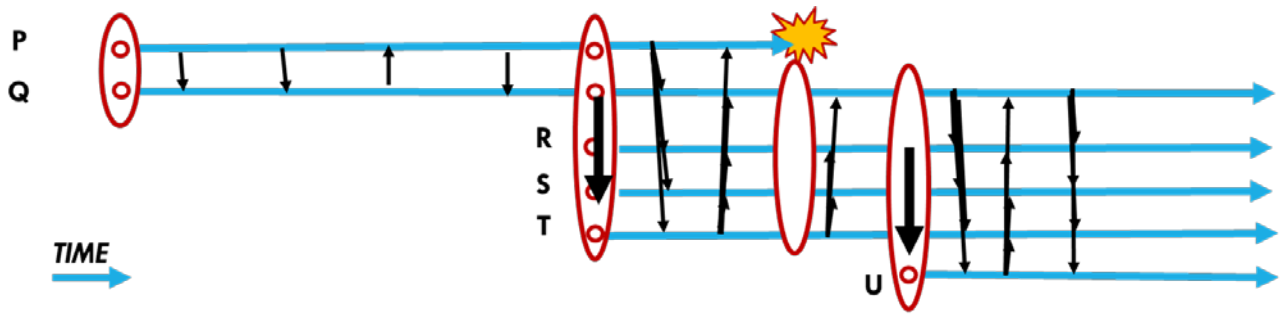
Both ideas center on applying the same updates in the same order

With atomic multicast updates are carried in messages, and delivered to all receivers (or none), in the identical order, even despite failures.

With durable replication each receiver also keeps a log of updates or of the result after applying them.

- In some solutions each replica (each log) is itself a complete history.
- In other solutions logs must be merged to know the full update sequence.

RECAP: VIRTUAL SYNCHRONY



The idea was to break down a distributed systems into a

- **Multicast or durable update protocol.** Sends only while membership is stable. Updates *every current member*. This differs from classic Paxos: no log merge is needed, because every member sees every update (like Paxos with QW=N, QR=1).
- **Membership service.** Tracks which processes are in the system, and what role each is playing (like which shard it is in). Automatically drops failed members. Never can experience a split-brain outage.
- **State transfer mechanism.** Uses checkpointing to initialize a joining process.

In virtual synchrony, membership changes only when updates are frozen and vice-versa. The multicast and state transfer logic becomes much simpler.

LESLIE LAMPORT: CLASSIC PAXOS

Greek philosophers debating a proposed law

The senate in the lovely town of Paxos debated laws. A majority is required to enact ("decide") a proposed new law. Laws were enacted in sequential order.

A senator who was present will never forget an enacted law

Periodically, a scribe would reconstruct the complete sequence of laws and copy any newly enacted ones to the last page in the great <u>scroll of the law</u> in the town square.

The scroll of the law

WE CAN FORMALIZE THIS MATHEMATICALLY

We have some set of processes, N of them in total. ***Fixed membership.***

But within this membership, some processes may be unresponsive

Each new proposal must be voted by the processes, and requires majority consent (but the processes in Paxos don't try to modify the proposals).

If the process making a proposal doesn't fail, it will eventually be adopted

NOTICE: PAXOS DOES NOT USE VIRTUAL SYNCHRONY

At the time, he wasn't familiar with my model. Much later he and Dahlia Malkhi suggested that virtual synchrony + Paxos would be the way to go.

In the original "classic" version of Paxos, because the processes can come and go, not every process will "know" about each decided proposal.

To actually learn the sequence of decided proposals, we need to merge the logs managed by the individual processes.

QUORUM POLICY: UPDATES (WRITES)

To achieve high availability, allow an update to make progress without waiting for all the copies to acknowledge it.

- Require that a "write quorum" (QW) must participate in the update
- Easy to implement this requirement using a 2-phase commit protocol

Basic approach: Leader asks the loggers ("acceptors") to log an update. But it won't commit unless QW respond "success". So we have a request phase and then a commit phase: a 2-phase commit.

ASSUME N MEMBERS, F POTENTIAL FAILURES

We want to be sure that any two write quorums overlap:

- $QW+QW > N$

We also want any read to "see" all the prior committed writes

- $QW+QR > N$

For fault-tolerance, we need $QW \leq N-F$ and $QR \leq N-F$.

For maximum speed, QR should be as small as possible

EXAMPLE WITH 5 SERVERS

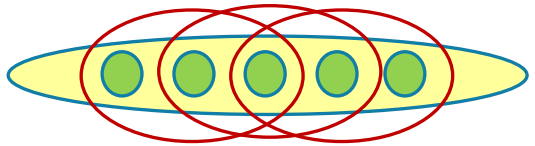
N = 5

Suppose F = 1: We want to tolerate 1 failure

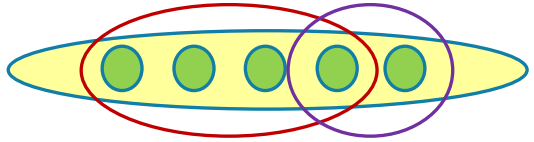
- QW must be 3 or 4.
- QR would be 3 if QW=3, and 2 if QW=4.

… cloud systems normally are read-intensive, so QR=2, QW=4 is best

N=5, QW=3

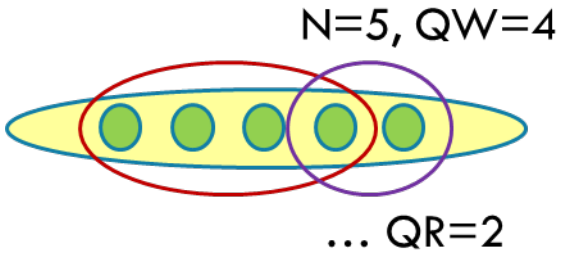


N=5, QW=4



… QR=2

THOUGHT QUESTIONS



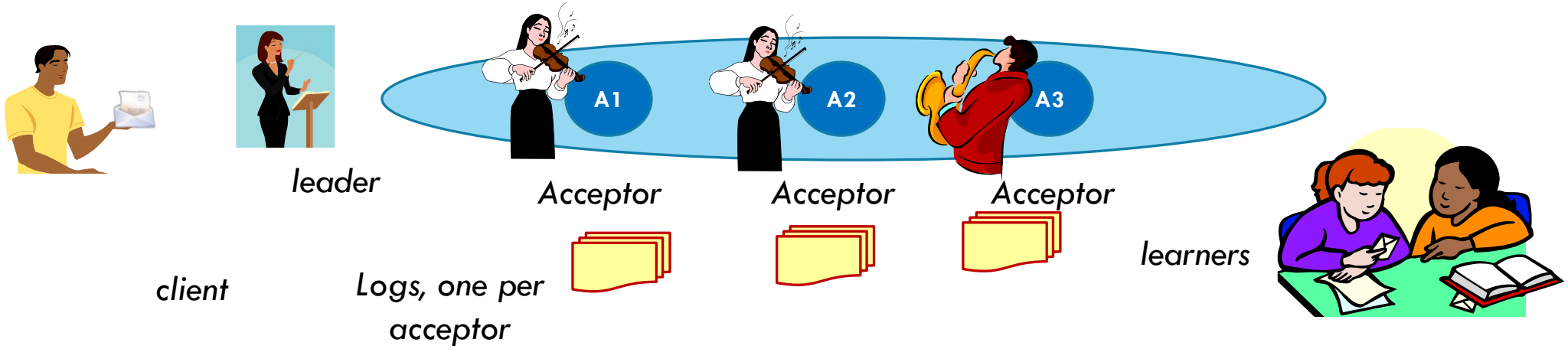
1. We happen to know that in cloud computing systems, almost all accesses to a DHT will be pure reads (queries). 99.999% is common!

   What does that statistic tell us about <u>picking values</u> for QW and QR?

2. Consider the case where N=5, QW=4. QR could be 2, 3 or 4.

   The illustration shows QR=2. Would QR=3 or 4 be <u>better</u> or <u>worse</u>?

VISUALIZING PAXOS SETUP

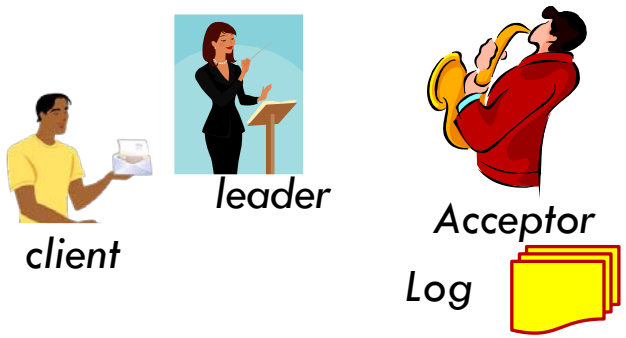


The client asks the leader to add a message to the Paxos logs. Paxos is like a "postal system". The leader will be in charge of this request.

The system "discusses" the letter for a while (the first phase, which picks the slot in the log, stores the letter in the log, and reaches QW acceptors).

Once the update is "committed" the learners can execute the command

LOGS



The log lives on a disk – Paxos is used for persistent updates, not atomic multicast. Each log holds a series of updates, like a stack of paper.

An acceptor has its own log. Updates are appended at the end of the log

Any single logs can have *gaps* because QW < N! This differs from atomic multicast, where every receiver gets every message.

NOTE: CLASSIC PAXOS ≠ VIRTUALLY SYNCHRONOUS PAXOS

With classic Paxos, individual logs can be incomplete because the acceptor might have been unavailable when the update committed.

… so this forces us to merge logs: Because QW < N, QR > 1.

With virtual synchrony, we dynamically adjust the membership. This allows us to set $QW=N_v$, where N_v is "The size of the current view **v**". With virtual synchrony no updates are missing, hence QR=1 becomes possible

CLASSIC PAXOS RUNS IN STAGES

In the first stage, the leader asks the acceptors to “reserve” a slot in their logs.

This can take a few tries because there could be multiple concurrent leaders running, and there could be some crashes as they run

PAXOS BALLOT NUMBERS

This leads to the idea of "ballots" – the first phase loops, with the leader trying to get QW successes on a series of proposals, with an increasing ballot numbers

Leader X to acceptor 4:

"This is my ballot number 1. Can you reserve slot 1 for me?"

Acceptor 4 to leader X:

"Ack: Slot 1 ballot 1 reserved for you."

or,

"Nack: Slot 1 ballot 1 has already been reserved for leader Y."

WHAT LEADER "LEARNS" FROM A QUORUM

Every write quorum will overlap with every other write quorum, so the leader will learn about any prior write that could have gained a quorum and begun its commit stage on the same slot/ballot.

Also, because acceptors only can vote once per slot/ballot, if one write proposal gains a quorum, no other proposal can gain a quorum in the same slot.

Finally, the leader learns that it has, in fact, reached a quorum.

WHAT ABOUT A SKIPPED ACCEPTOR?

The leader might give up on some acceptor. It will have a gap in its log

We usually just denote it with a ? or a symbol like $\perp$. The Paxos read protocol will merge Q_R logs, which is enough to ensure that if a slot contains a committed write, it learns about that write.

Note: There is a rare error case where a slot commits to ϕ

STAGE TWO: LEADER GETS A QUORUM

To move from stage one to stage two, a leader needs QW acks for some ballot. Perhaps, 4 out of the 5 acceptors agree: Leader X reserved slot 7 on ballot 1.

The way the Paxos rules are designed, no other leader can "win" once this condition is achieved, although more ballots may be required for the leader to discover that it has a quorum because messages could be lost or acceptors could crash.

Now leader X sends the actual data:

"In slot 1 (ballot 1), please write value v into your log"

STAGE THREE

Once QW acceptors have written v into the slot, they acknowledge and then the leader can commit (finalize) the new entry

Leader sends:

"Commit pending update v in slot 7, ballot 1."

TOLERATING FAULTS

The actual protocol has to tolerate leader crashes…

… and acceptor crashes

… and work correctly even with concurrent leaders fighting for some slot, even if crashes occur

PROVING THAT PAXOS "REALLY" WORKS?

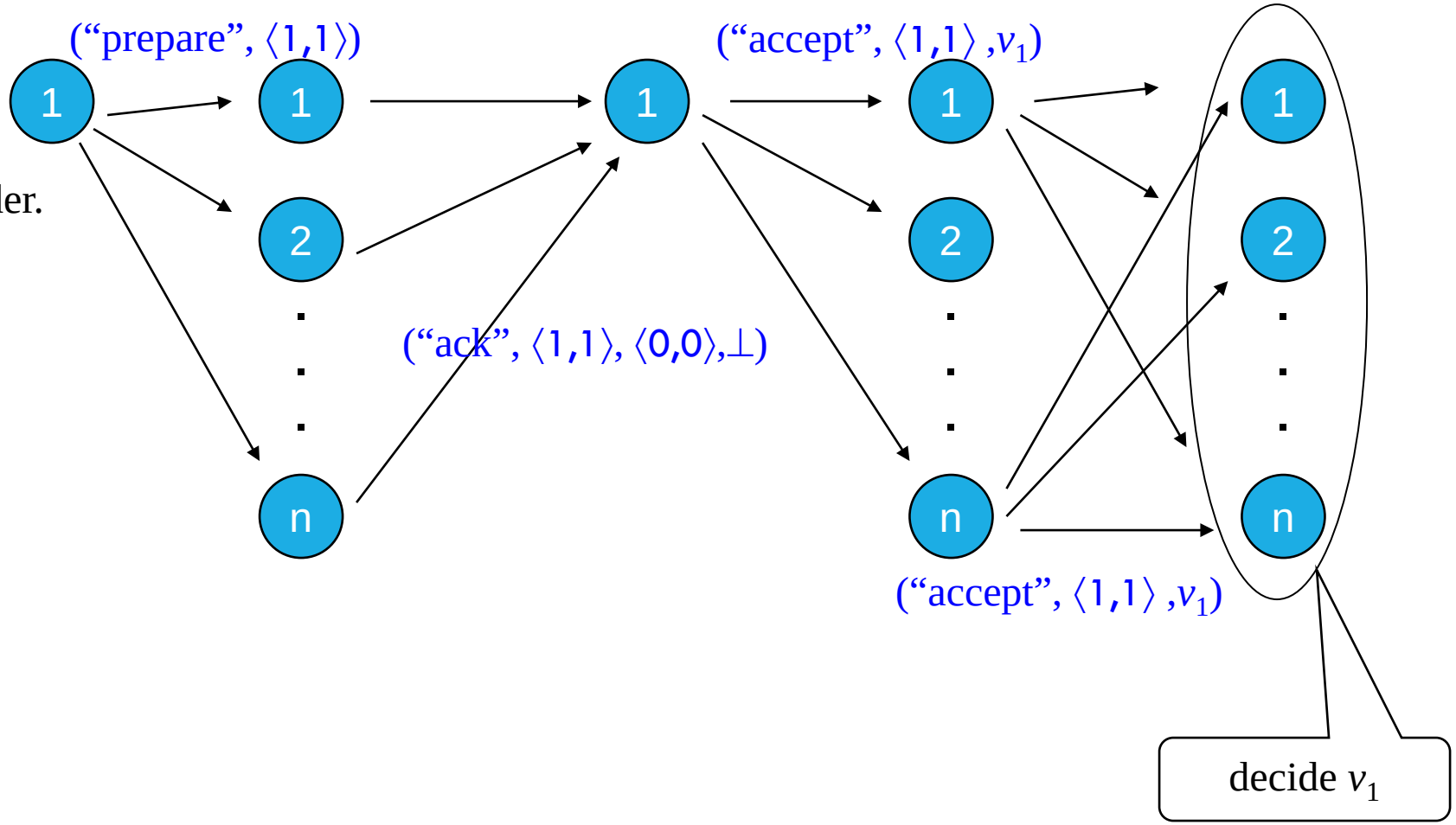
The idea wasn't super complex, but it is very hard to prove correct

You should review Robbert van Renesse's paper **"Paxos made moderately complex"** to understand all the possible failure cases and how they are addressed in Paxos.

The actual full protocol is very complicated!

EXAMPLE: PAXOS THREE-STAGE PROTOCOL

Process 1 is the leader.



See Robbert van Renesse's paper to understand the notation

CRITICISMS OF PAXOS

- The original paper was very hard for people to understand.
- The protocol is very slow in the original form. Later work improved it, but until 2019 we didn't have an "optimal" solution.
- The classic Paxos protocol wasn't even invented by Lamport! The same idea can be found in several papers from the 1985-1990 period. But Lamport's proof techniques were very original, and important.

LESLIE LAMPORT'S REFLECTIONS

"Inspired by my success at popularizing the consensus problem by describing it with Byzantine generals, I decided to cast the algorithm in terms of a parliament on an ancient Greek island.

"To carry the image further, I gave a few lectures in the persona of an Indiana-Jones-style archaeologist.

"My attempt at inserting some humor into the subject was a dismal failure.

DERECHO SYSTEM

GOALS FOR DERECHO

Project started in 2014. We wanted to build a fully useable virtually synchronous atomic multicast and Paxos durable logging feature for Linux.

- Embed it into a simple, practical, programming model that scales well.
- Achieve the highest possible performance using C++ 17 and modern hardware of the kind founded in data centers.
- Use the formal techniques to prove it correct and verify (check) the proof

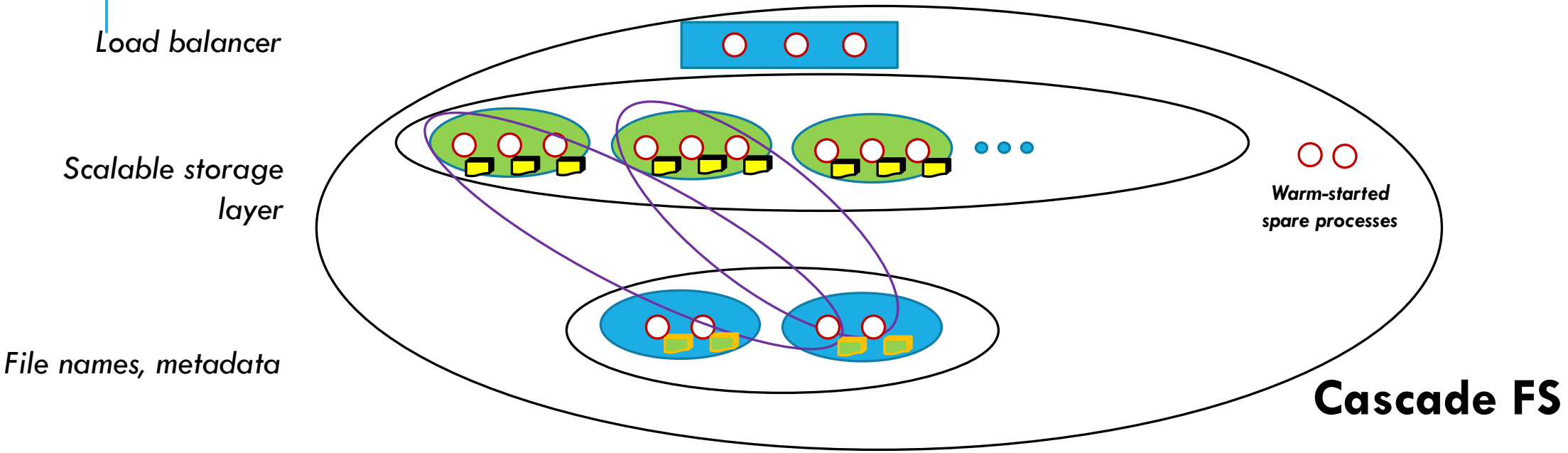
WHAT HAD CHANGED IN 2014?

The first cloud-style data centers were huge, but used identically the same hardware and software we had seen in small clusters

But by 2014 all the components had evolved. For example, there was a new way to "offload" TCP into optical ethernet hardware, called RDMA.

RDMA 100x faster than standard TCP, but when people took a Paxos library (LibPaxos) and ported it to RDMA, they saw little speedup.

DERECHO TARGET: FANCY μ-SERVICES



This example shows a file system constructed as one big μ-service that internally consists of three "sub-services". With Derecho we can use state machine replication to create a version that runs at high speeds yet is self-contained and offers strong consistency.

DERECHO VISION

A Derecho is a powerful wind

Derecho implements a new version of Paxos.

- It can support atomic multicast or durable (logged) Paxos.
- It runs on standard TCP but also supports RDMA. When setting up Derecho, a configuration file tells it which to use (not every system has RDMA hardware).
- Our goal was to run Paxos at "the speed of light"

DERECHO IS A SOFTWARE LIBRARY

Derecho is a C++ library that handles membership, atomic multicast and persistent logging. It uses a virtually synchronous Paxos model. *Every group member will see every update to the data replicated in that group.*

It is designed specifically to support sharded micro-services in modern datacenter settings.

The developer builds a new microservice by linking against the library.

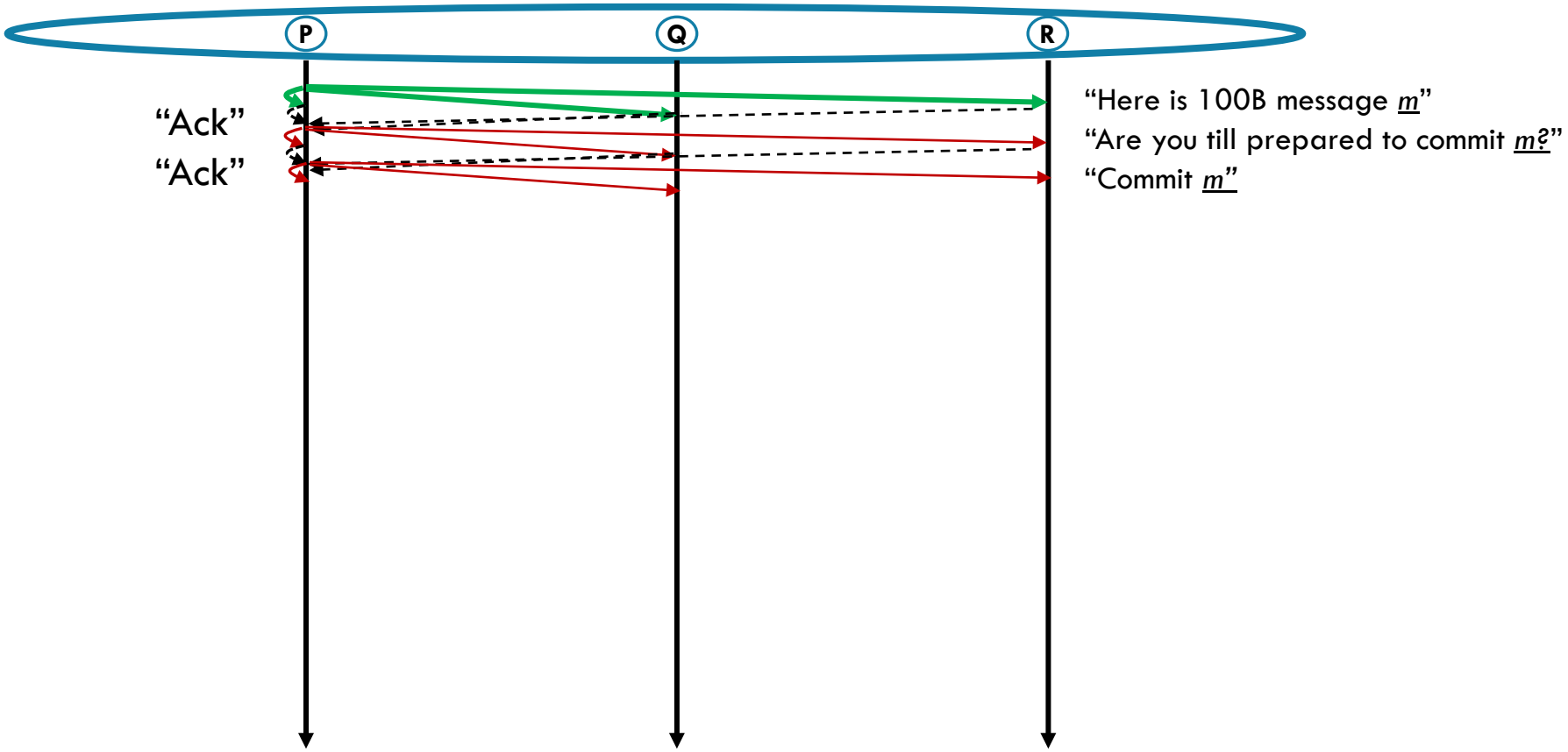
DERECHO WAS A REAL SUCCESS!

As much as 10,000x faster than standard Paxos protocols.

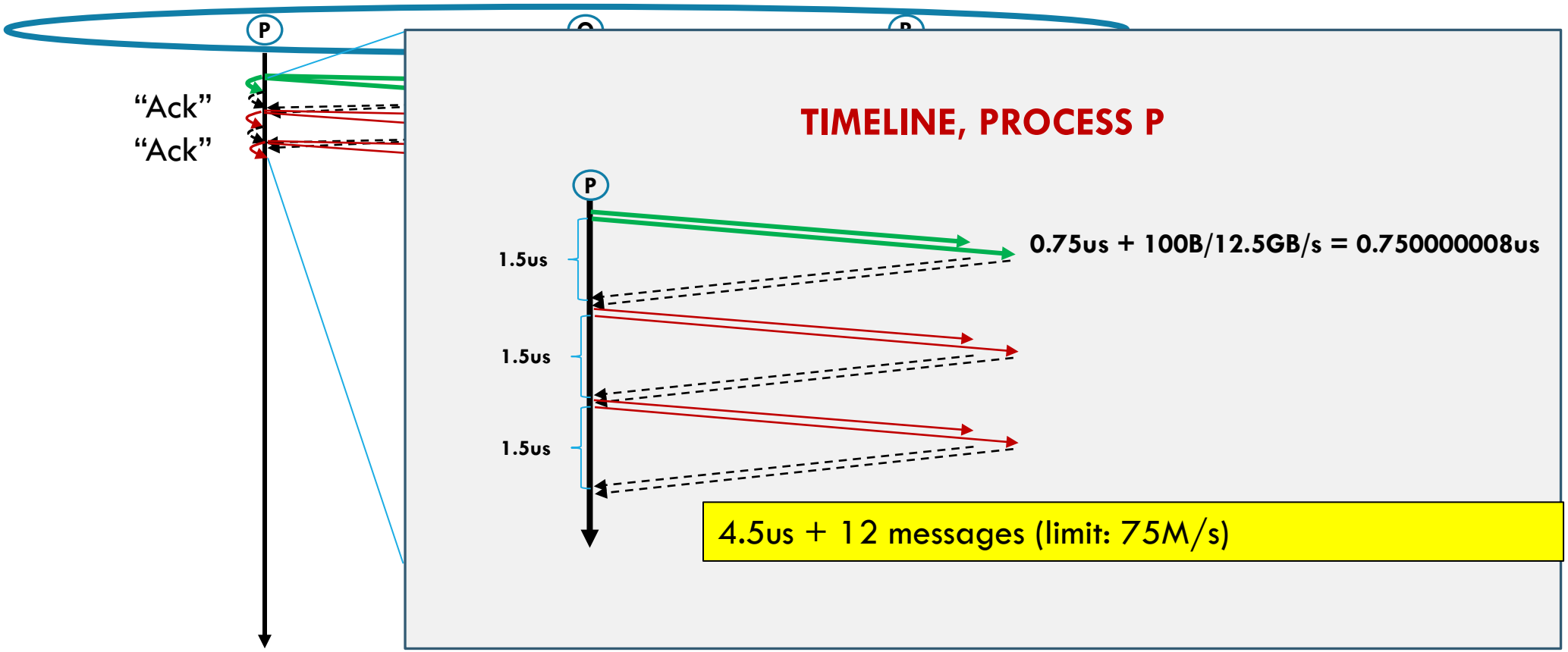
In fact we can even prove that Derecho is an optimal Paxos solution: no Paxos protocol can eliminate any delays from Derecho. Decisions occur as early as they safely can be performed.

DERECHO INTERNALS

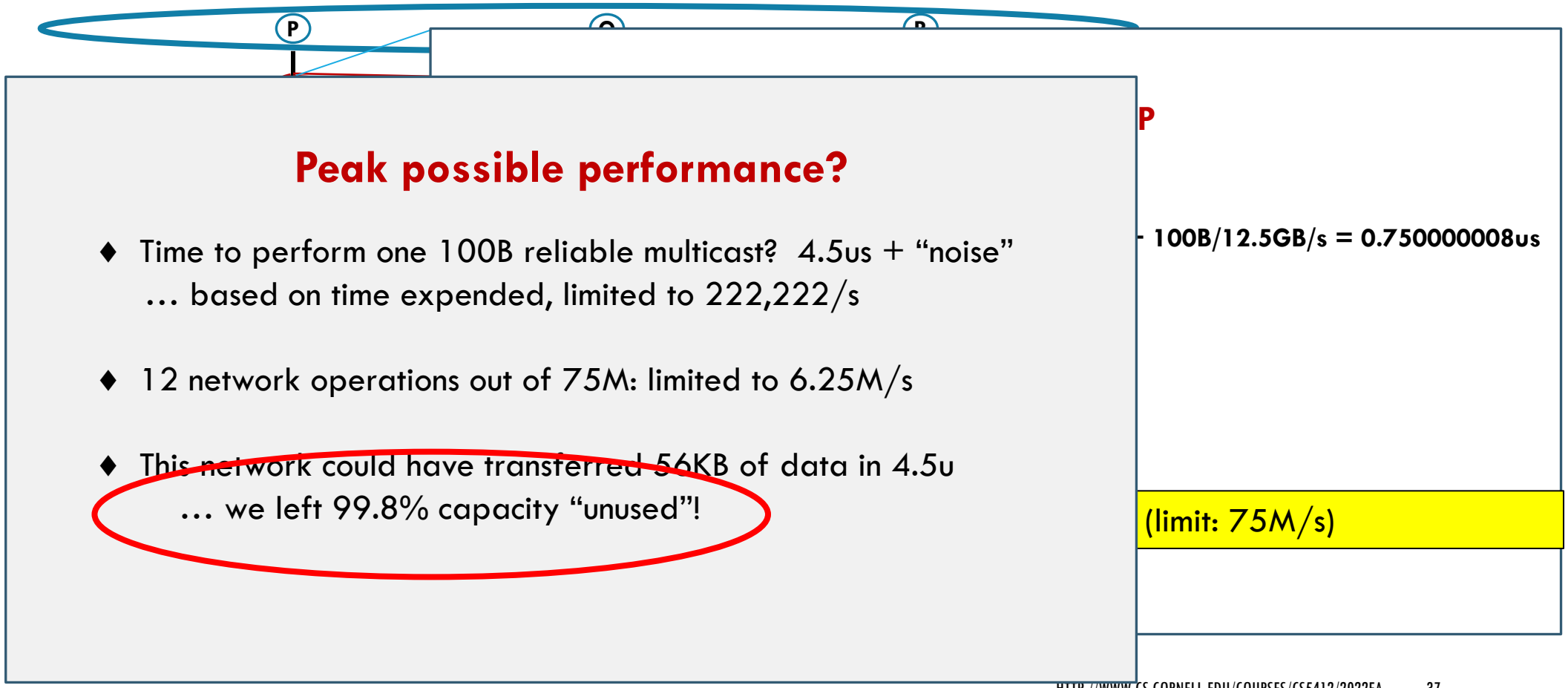
MOTIVATION: CONSIDER PAXOS ON A FAST NETWORK



MOTIVATION: CONSIDER PAXOS ON A FAST NETWORK



MOTIVATION: CONSIDER PAXOS ON A FAST NETWORK



Peak possible performance?

- Time to perform one 100B reliable multicast? 4.5us + "noise"
  … based on time expended, limited to 222,222/s
- 12 network operations out of 75M: limited to 6.25M/s
- This network could have transferred 56KB of data in 4.5u
  … we left 99.8% capacity "unused"!

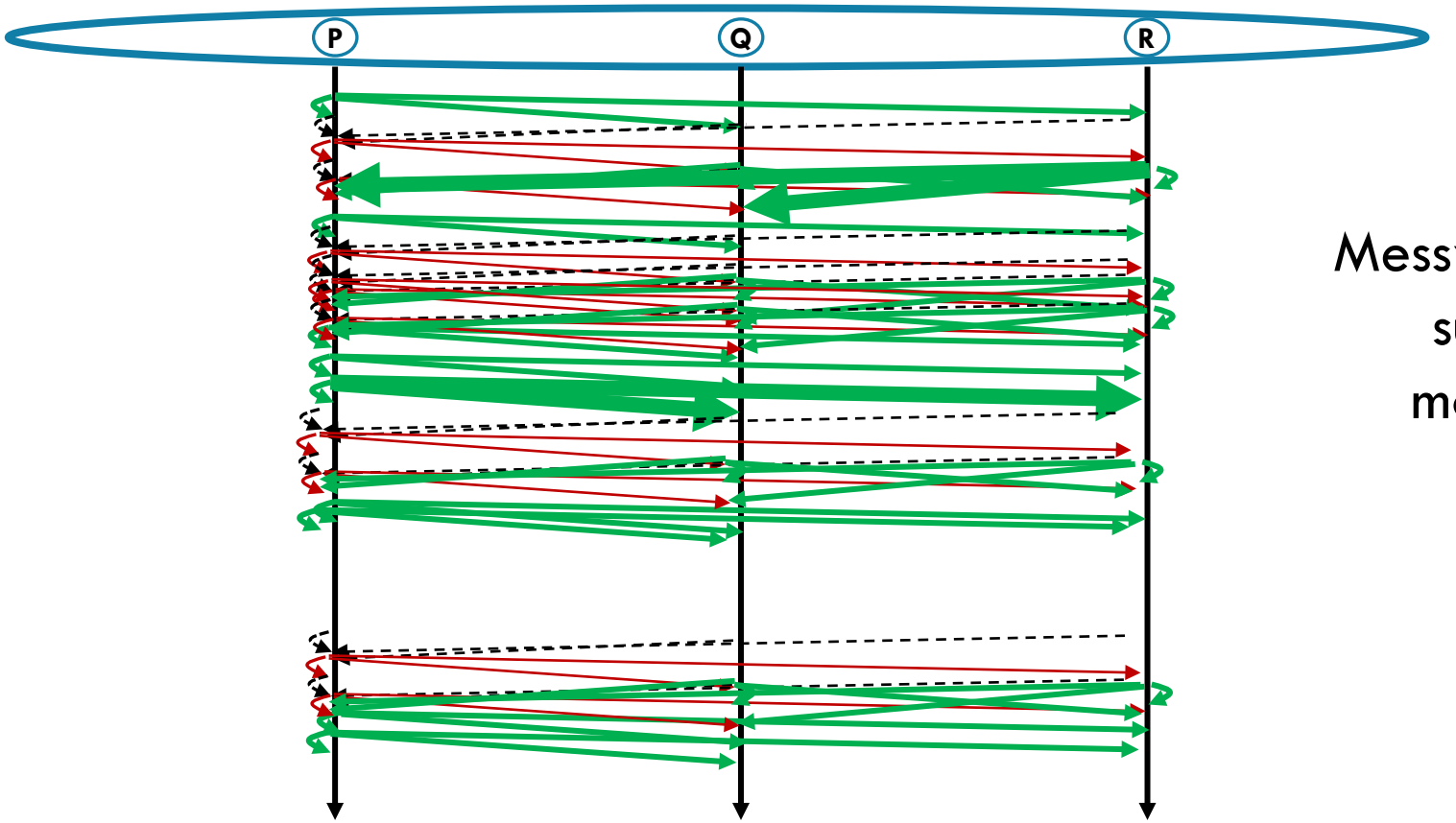
A FEW IDEAS

Have all the 3 members perform concurrent updates… now we might get some overlap and push our efficiency… to 0.6% ☹

Run lots of threads… maybe 10 per process. We aim for 6% efficiency (but locking and scheduling delays will cut this sharply) ☹

Batch 1000 messages at a time. ☺ But now the average message waits until 500 more have turned up. Latency soars to 2.25ms ☹

AT BEST, YOU GET SOMETHING LIKE THIS…



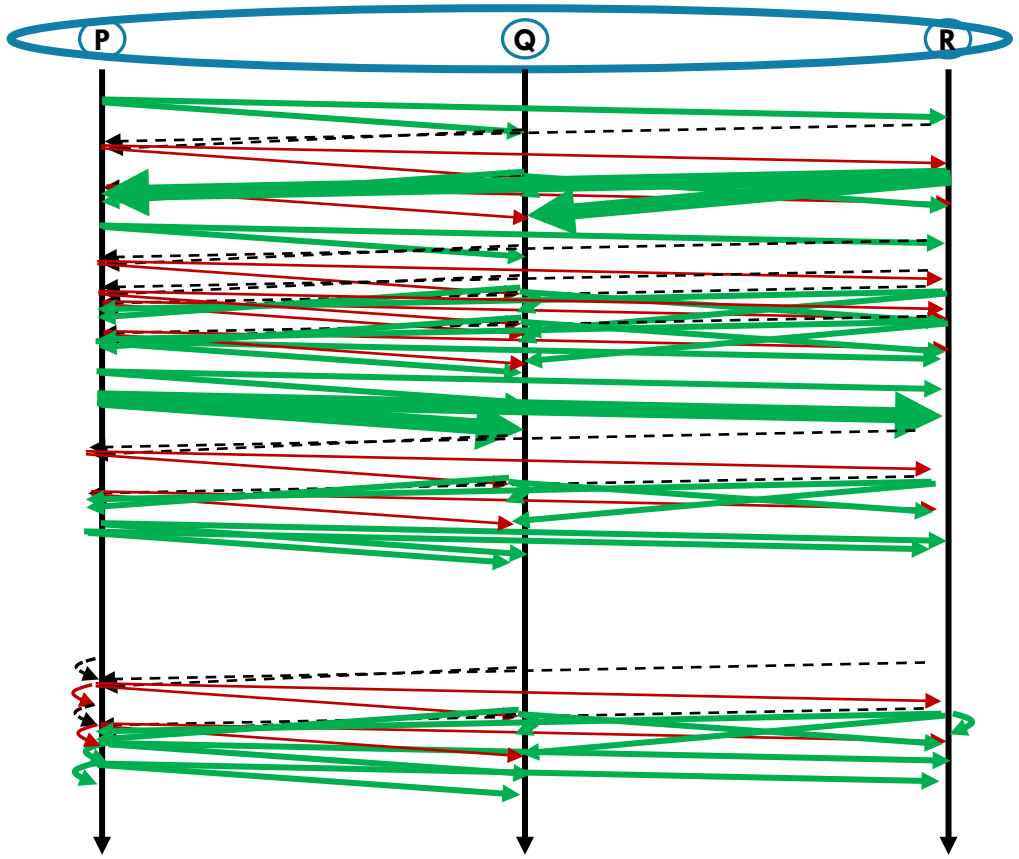
Messy and unpredictable with sudden bursts of data movement… Unlikely to perform well ☹

BETTER: SEPARATE DATA PLANE AND CONTROL PLANE, MAKE THEM LOCK-FREE

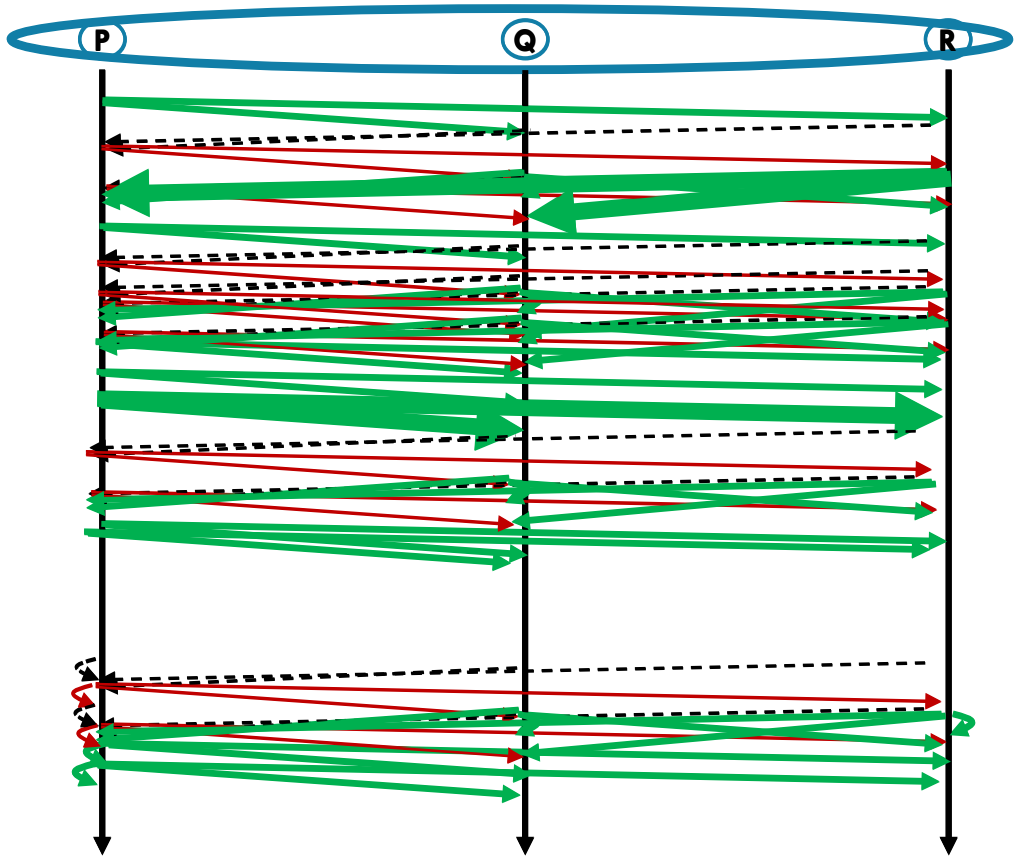
Data plane: The actual data messages. Send them continuously, as soon as new updates show up

Control plane: Responsible for deciding when it is safe to deliver ("commit"). Receivers continuously report their acks, in an all-to-all pattern. This way every process can deduce that messages are deliverable.

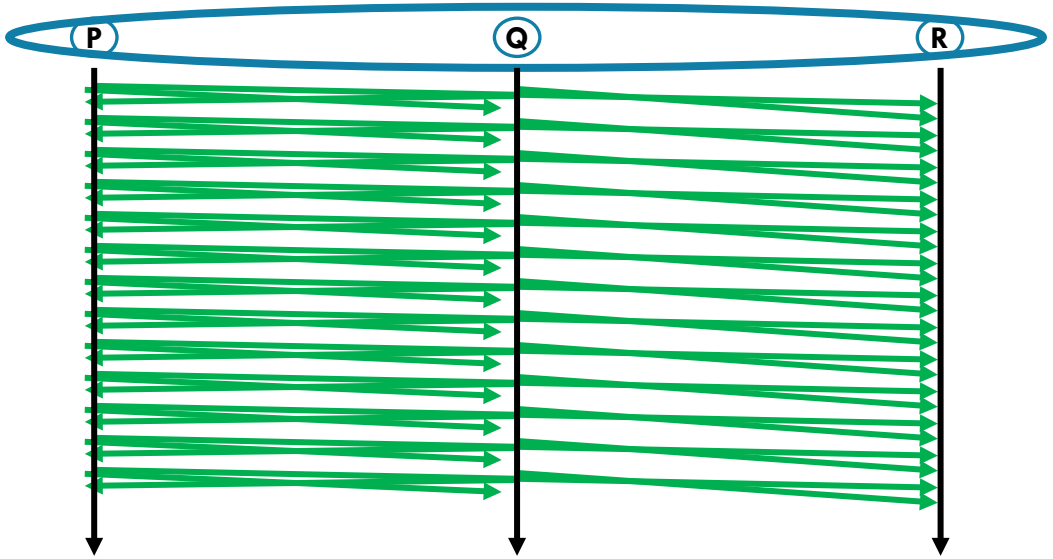
BETTER: SEPARATE DATA PLANE AND CONTROL PLANE, MAKE THEM LOCK-FREE



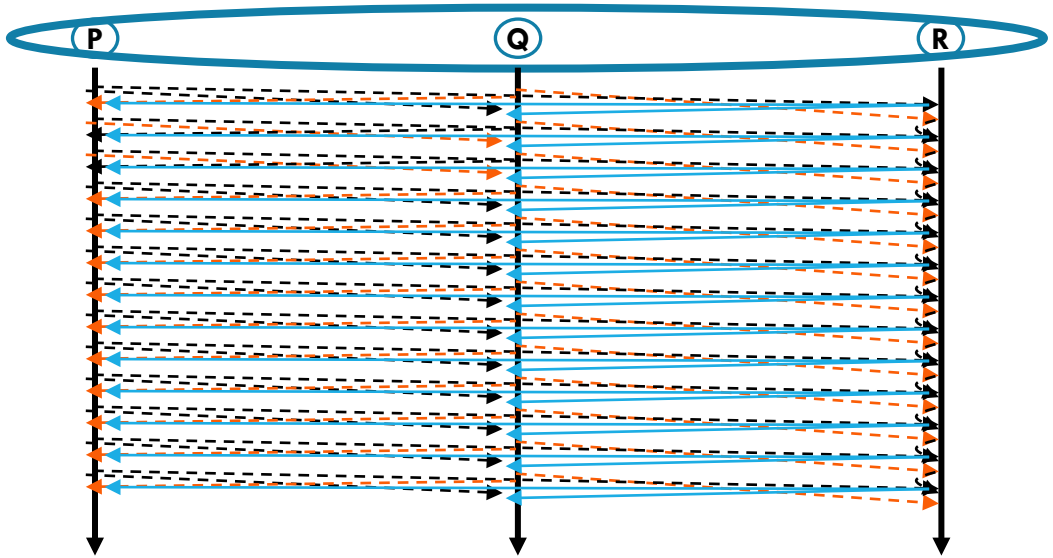
BETTER: SEPARATE DATA PLANE AND CONTROL PLANE, MAKE THEM LOCK-FREE



BETTER: SEPARATE DATA PLANE AND CONTROL PLANE, MAKE THEM LOCK-FREE



Data plane runs steadily



Control information exchanged continuously

DATA/CONTROL SPLIT

Data plane

Like TCP, but instead of 1-to-1, 1-to-many

Carries messages with user-provided data, such as "decrement the inventory of X-box units" or "help me understand this video snippet"

Some messages are small… some huge!

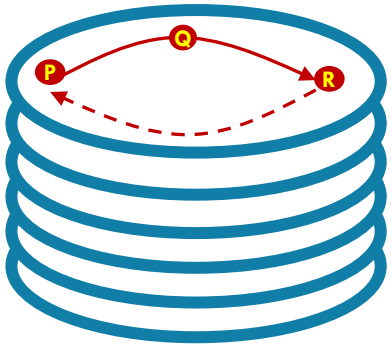
Control plane

All to all communication pattern

Control data is in the form of counters, like "process Q has received 22 multicasts from process P"

In Derecho the control plane does *not* use state machine replication. Data is asynchronously streamed, and there are no ordering guarantees.

HOW TO PUT THESE INTO ORDER?



Contention for "slots" is one factor that slows standard Paxos down. Derecho uses round-robin order: one message from P, then one from Q, etc.

If a process has nothing to send, Derecho generates a "null message" from it, so that the others won't have to pause.

This rule allows processes to stream data at high speeds without pausing.

HOW DERECHO GETS ITS SPEED

By “aligning” the flow of information with the network and not waiting for round-trip responses, it can run at the full network speed continuously.

Derecho never pauses unless the application no longer has data to send.

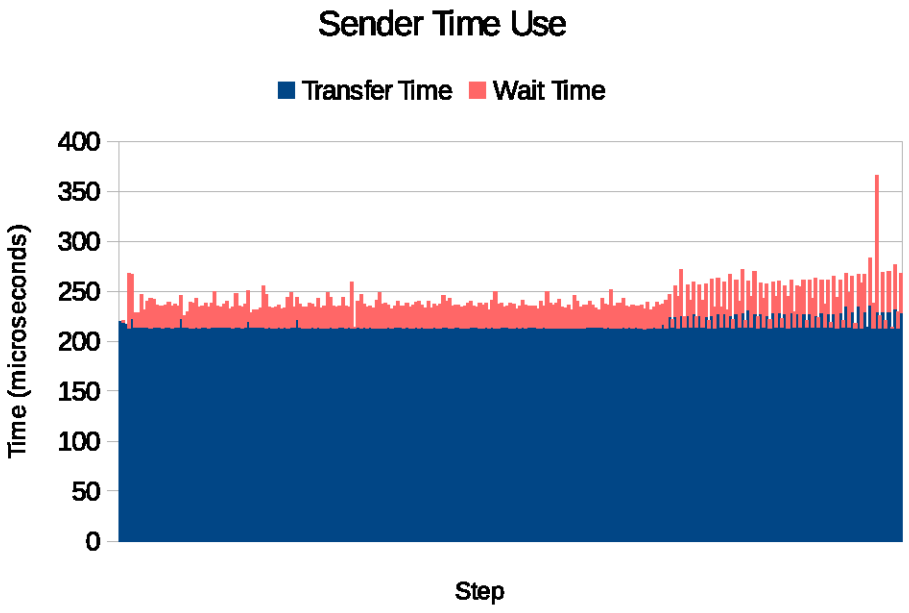
Analogous real-world situation: filling a series of buckets from a steady stream of water (Derecho), versus filling one cup at a time, then pausing to drink it before filling the next cup (Paxos).

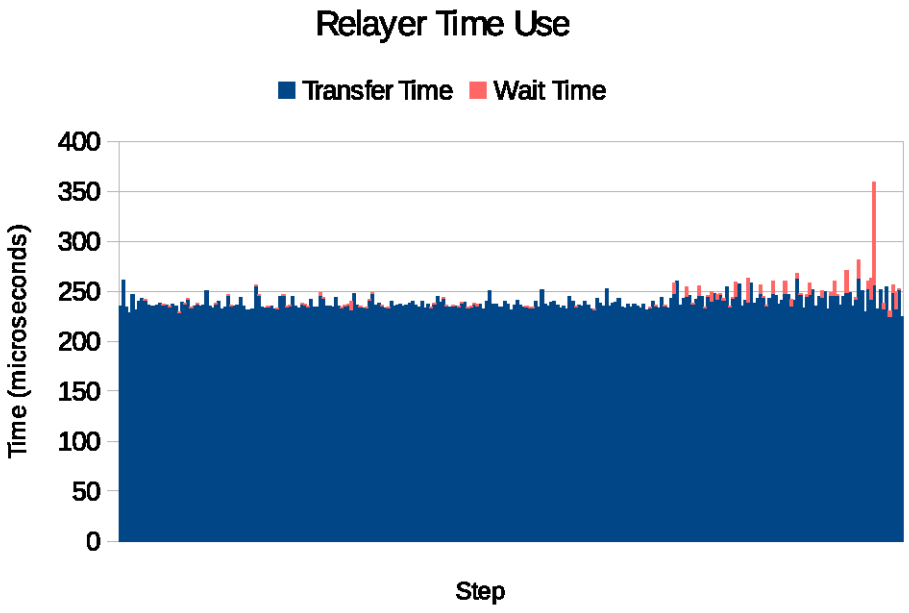
HOW FAST IS DERECHO?

On RDMA networking (100Gbits/s) and on a typical Intel server:

- Derecho can perform 8M "small" replicated updates per second in each shard. Delay from sending until delivery is as low as 3*us*
- Derecho can transport very large messages, like photos (1MB) or even videos (100MB or more) between 110Gbits/second for a small number of replicas and ~85 Gbits/second for a large number
- Scaling is almost flat. Whether you need 5 replicas or 25, the speed and delay are similar… and 3-5x faster than **memcpy** on our servers.
- This is thousands of times faster than LibPaxos or Zookeeper

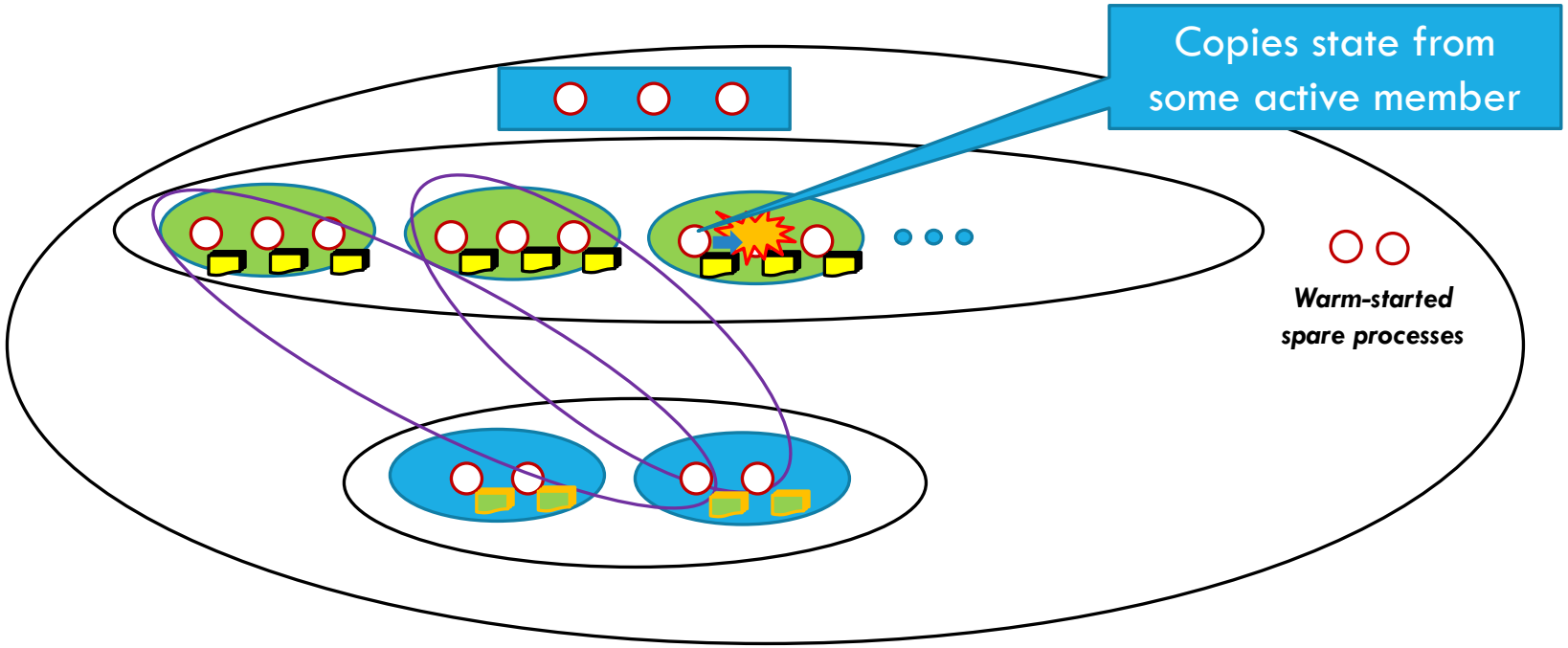
BY OFFLOADING WORK TO THE THE NETWORK DERECHO FREES HOST COMPUTERS TO DO OTHER USEFUL TASKS





This snapshots a heavy load scenario. Blue shows time ("work") being done by the network hardware. Pink is work done on the host computers running Derecho

DERECHO'S EPOCH MECHANISM IN ACTION



…. If a failure occurs, Derecho automatically repairs it.

IS THIS REVOLUTIONARY?

… kind of.

Companies were extremely interested in Derecho when it was first published. Some big cloud vendors discussed adopting it

But they concluded that really effective use of Derecho would force them to recode many core systems that work pretty well. Plus, Derecho was created at Cornell. As of today, none has decided to bet heavily on it.

CORNELL DECIDED TO USE DERECHO TO CREATE CASCADE

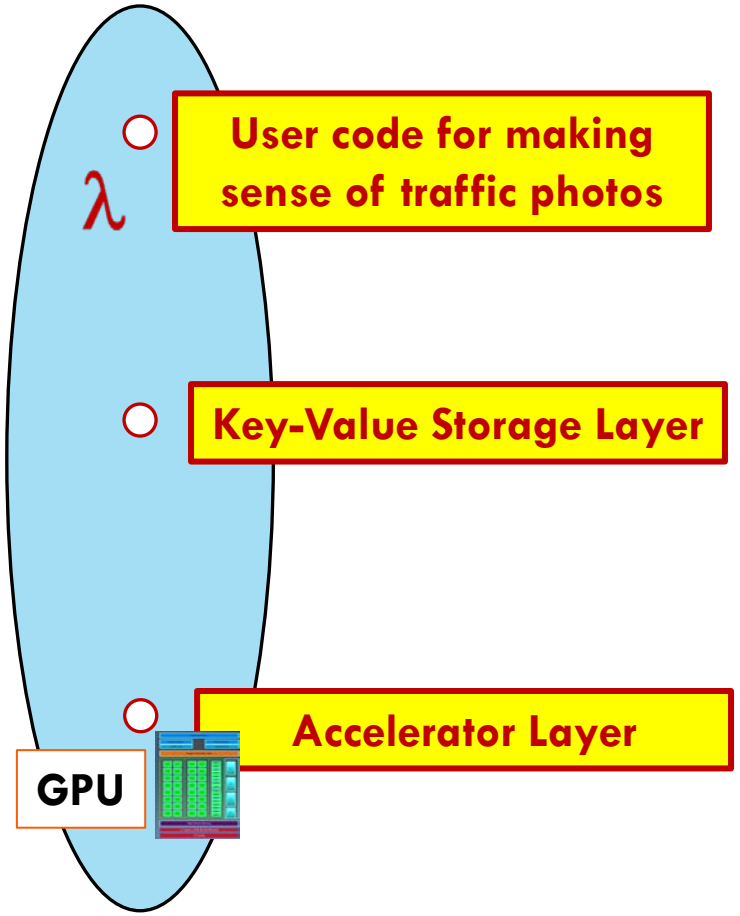
Cascade is an ML-hosting platform built on Derecho

The idea of Cascade is to support standard ML platforms but also host ML lambda methods (lecture 3). Your code lives right inside Cascade.

Cascade will be the topic of Lecture 10, on Sept. 26, but let's take a peek

THIS SHOWS ONE CASCADE INSTANCE WITH SOME USER-SUPPLIED LOGIC (λ)

Sample application: A smart traffic intersection



CASCADE USED TO CREATE A μ-SERVICE

Request for classification triggers a C++ lambda in the Cascade address space

Fast-path logic (DLL)

Ideally, these are cached on GPU

λ

Key-Value Storage Layer

RDMA directly into GPU memory

Image to classify

ML model, configuration, parameters

Accelerator Layer

Upload "instructions"

GPU

GPU-accelerated kernel initiated from the lambda

… SO, HOW IS CONSISTENT REPLICATION ACTUALLY USED?

Now that we have it, what good is it?

HISTORICALLY, CONSISTENT REPLICATION WAS USED TO CREATE ULTRA-RELIABLE SERVICES

Systems Ken worked on that actually require strong guarantees

- Air traffic control system (France, 1995-present)
- Stock exchange trading platforms and notification infrastructures
- Control systems for highly automated VLSI fabrication and other forms of industrial factories, like refineries
- US Navy AEGIS battle station information support
- Northeastern US electric power grid "smart" monitoring infrastructure

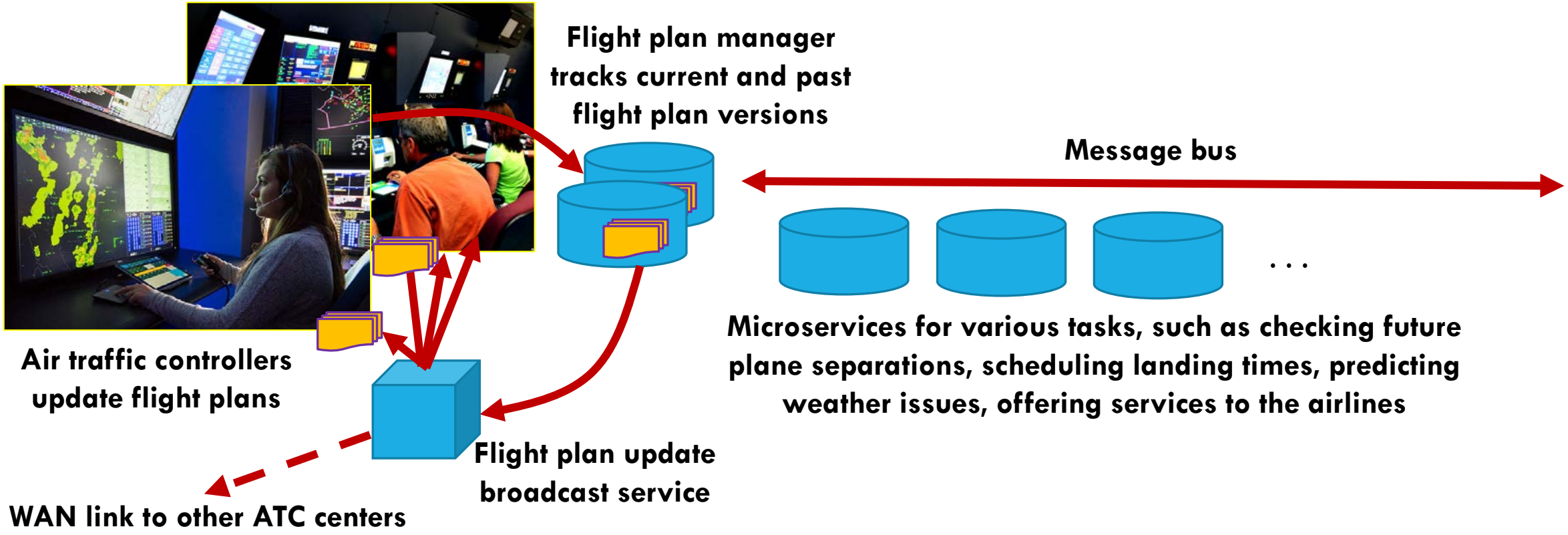
WHAT DO THESE HAVE IN COMMON?

Use of state machine replication as a root source of consistency

For the ATC example, this entails three important elements:

- There can only be on official version of the current flight plan
- There needs to be agreement on which consoles are active in the system to ensure that updates reach all the controllers who need to see them
- Planes and controllers need to agree on who is in charge of what

FRENCH AIR TRAFFIC CONTROL (1995)



… NOT SHOWN

There is also a wide-area architecture, for sharing between ATC centers, and several replicated databases for “detail” data

And there is a fail-safe mechanism so that if one center or even a whole country has a sudden outage, control can seamlessly shift to a neighboring system in some other region or country

There are also elaborate security mechanisms to protect against attack by hostile intruders who might want to disrupt the system in various ways

HAVING PLATFORM-LEVEL CONSISTENCY DOESN'T MAKE IT TRIVIAL

Building consistency mechanisms into a system doesn't ensure that your applications, layered over the consistent framework, will be correct.

You still need very careful specifications and detailed justifications (proofs) that each element satisfies its requirements!

Speed matters too: Air traffic systems aren't exactly "real-time" (deadlines are quite long, when you look closely), but do need quick actions!

FUTURE NEEDS FOR CONSISTENCY?

But any system that reacts to the outside world while controlling things needs consistency to avoid causing accidents or damage.

It is easier to build IoT systems that simply capture and monitor the outside world. Those can store data first, then process it much later.

Snappy but correct actions are more and more important as the edge of the cloud is integrated into the outside world of sensors and actuators

SUMMARY

We actually can have C+A if P isn't needed – the key is to have the microservice hold the needed state in replicated objects, and then to align the Paxos protocol with the properties of the hardware

The resulting performance is amazing, and important in many settings

Today, people work with higher level AI packages like Tensor Flow and Databricks, and those run on DHTs. This is why Cornell is now building Cascade: An extensible, customizable DHT that leverages Derecho.