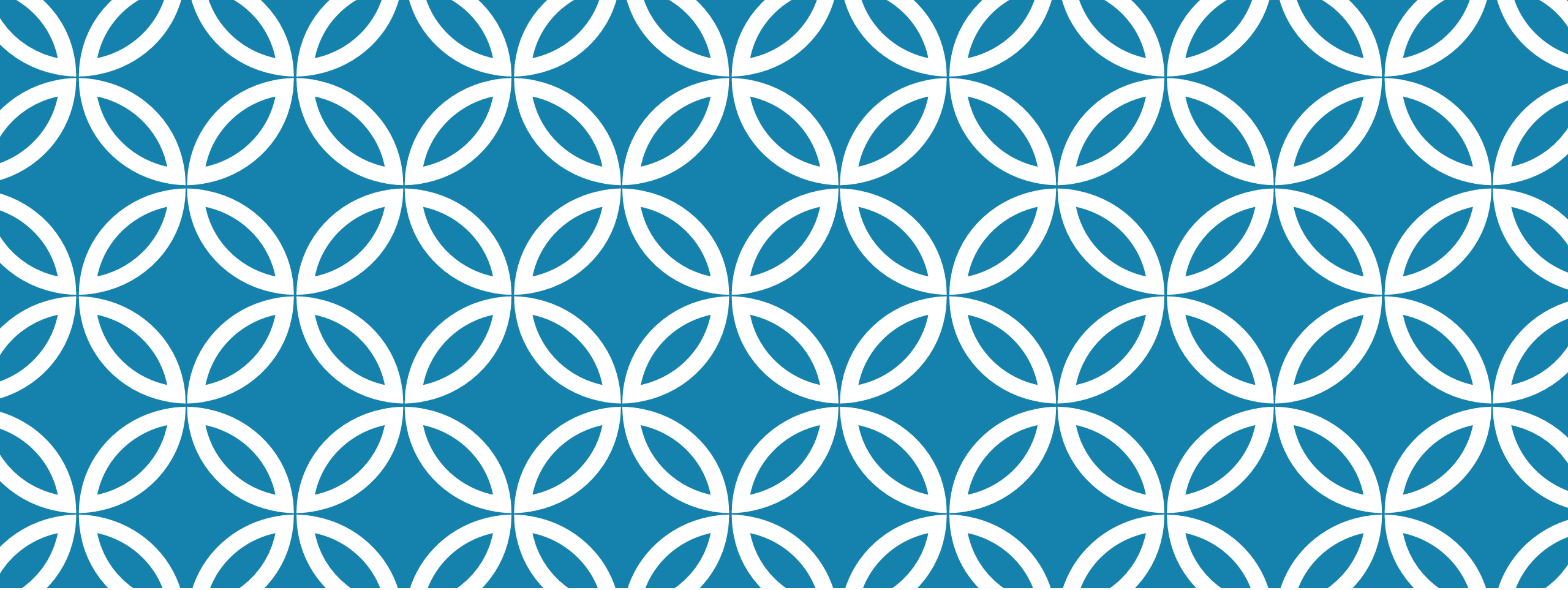


AUTOMATING CONSISTENCY

CS 4414/CS5416
Fall 2025

IDEA MAP

Lamport defined consistency and gave examples of how it can be achieved.

Inconsistency can cause havoc! Entire projects fail over it!

The assumption seems to be that consistency is expensive, especially when information needs to be shared at several locations (data replication).

Today our goal will be to figure out how to offer very inexpensive consistency guarantees and then how to hide them in standard framework components. This allows MLs to benefit from consistency without directly doing anything unusual.

REMINDER: MANY CLOUD AND ML SERVICES ARE SHARDED AND NEED REPLICATION

All of us use these services all the time, so if they were able to guarantee consistency, applications running at higher levels inherit consistency.

Sharding is also a universal pattern. Reads are kind of easy: if data isn't evolving, we just need to read from current data and avoid stale cached data items. But updates need to be mirrored on all replicas.

Lelie Lamport considered the sharding question to be an instance of state machine replication.

Leslie Lamport

NOT EVERYTHING NEEDS CONSISTENT REPLICATION

Data that is created **but never updated** might need replicas but we don't need anything fancy to make them.

Most web apps are designed so that if a piece of data is stale but in cache, it is ok to use that stale data anyhow. Inconsistency "by design"

Spawning ML jobs involves a form of coordination, but it isn't tricky to implement. You just select the machines and launch the jobs.

YET SOME TASKS DO REQUIRE CONSISTENT REPLICATION

Updating programs on machines that will run them, or entire virtual machines.

Replication of configuration parameters and input settings.

Real-world data updates.

Replication for fault-tolerance, within the datacenter or at geographic scale.

Replication of transient data like a checkpoint or backup as an ML computes a new model (which can take days)

Replication for parallel processing in the back-end layer.

Data exchanged in AllReduce/MapReduce

Interaction between members of a group of tasks that need to coordinate

- Locking
- Leader selection and disseminating decisions back to the other members
- Barrier coordination

FAULT-TOLERANCE MATTERS TOO

We heard in lecture 12 that failures are common in the cloud

- Hardware can crash, but bugs cause more crashes
- "Reboot, reimage, replace" philosophy
- So consistency has to include fault-tolerance.

James Hamilton (three R's)

We also learned in that lecture that the crash ("halting") failure model is widely accepted within cloud datacenters.

PAXOS: OVERARCHING "APPROACH" WHERE CONSISTENT REPLICATION IS NEEDED

Paxos is the name of a model used by a series of protocols that Lamport created to solve state machine replication. He also showed how to prove that they are correct and even how to verify an implementation.

Paxos (Greek Island)

There are many ways to implement Paxos protocols.

Leslie claimed they all were discovered as a side effect of the "part time" senate that ruled the city of Paxos

THE PART TIME SENATE

Senators would come and go, because on Paxos everyone had to farm, fish, and do house chores

So the decisions of the Senate were recorded on a ledger. And the Senate needed a way to update this ledger despite the turmoil of Senators showing up unexpectedly, or wandering off without notice.

Also, parchment is easily damaged, so they actually used three or more replicas. Sometimes one would be taken away to freshen the ink, briefly.

BASIC IDEA (WE WON'T DO THE DETAILS)

Leslie's central idea was to use a pattern seen in some of the earlier work on similar replication problems.

- Senators would propose a new rule to the Senate by writing it into the available ledgers.
- To pass, a proposed rule must get majority support (a "quorum")
- Once a vote succeeds, the adopted "decree" would be retained in a fixed order: new decrees added at the end.

WHERE IS THE "CLEVER TRICK"?

Lamport suggests a clever way of doing the vote that

- Includes a kind of logical clock, called a ballot counter
- Includes a kind of consistent cut mechanism, which comes from a rule he uses to put decrees into a single, permanent, total order that centers on the sequence of final vote rounds: ballots approved by a quorum of Senators.

His mathematical proofs of these complicated protocols are widely cited

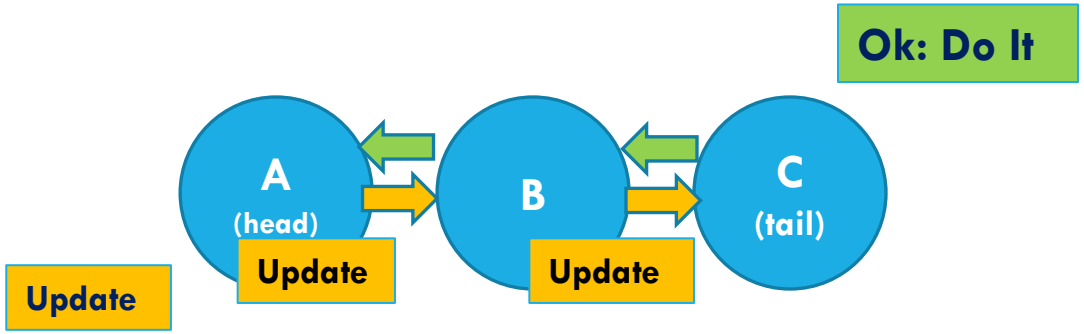
THE PAXOS PATTERN IS VALUABLE!

These days we know that Leslie's original protocols were hard to explain and even harder to understand, and also very inefficient.

Yet the pattern in them is extremely important! The main change is that rather than running Paxos for every update to replicated data in a shard, modern systems use Paxos (or something equivalent) in a service off to one side, tracking "membership" of the system.

This turns out to be sufficient to enable efficient update protocols

EXAMPLE: CHAIN REPLICATION



A common approach is "chain replication", used to make copies of application data in a small group. *It assumes that we know which processes participate.*

Once we have the group, we form a chain **and send updates to the head.**

The updates transit node by node to the tail, and only then are they applied: first at the tail, then node by node back to the head.

Queries are always sent to the tail of the chain: it is the most up to date.

DOES CHAIN REPLICATION NEED PAXOS?

Chain Replication is a good shard "update" option, but needs a Paxos-based membership protocol, to track the list of shard members.

Chain replication is provably correct with a sufficiently strong membership protocol, combined with a synchronization rule so that if membership changes and we need to modify the chain, pending updates finish first

This is actually why Lamport felt that a formal model (a mathematical one) and a methodology for proving things about protocols was needed.

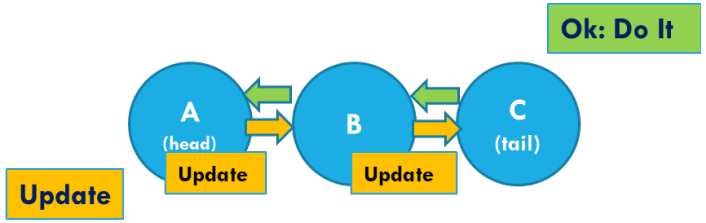
MEMBERSHIP AS A DIMENSION OF CONSISTENCY

When we replicate data, that means that some **set of processes** will each have a replica of the information.

So the membership of the set becomes critical to understanding whether they end up seeing the identical evolution of the data.

This suggests that membership-tracking is “more foundational” than replication, and that replication with managed membership is the right goal.

MEMBERSHIP CONCERNS FOR CHAIN REPLICATION



Where did the group come from? How will chain be managed? State machine replication doesn't turn out to provide a detailed solution for this.

How to initialize a restarted member? You need to copy state from some existing one, but the model itself doesn't provide a way to do this.

Why have K replicas and then send all the queries to just 1 of them? If we have K replicas, we would want to have K times the compute power!

MEMBERSHIP MANAGED BY A "LIBRARY"

Ideally, you want to link to a library that just solves the problem.

It would automate tasks such as tracking which computers are in the service, what roles have been assigned to them.

It would also be also be integrated with fault monitoring, management of configuration data (and ways to update the configuration). Probably, it will offer a notification mechanism to report on changes

With this, you could easily "toss together" your chain replication solution!

DERECHO IS A LIBRARY, EXACTLY FOR THESE KINDS OF ROLES!

Derecho: A powerful straight-line wind

You build one program, linked to the Derecho C++ library. Or, you could use a service built using Derecho that reports membership changes.

Now you can run N instances (replicas). They would read in a configuration file where this number N (and other parameters) is specified.

As the replicas start up, they ask Derecho to "manage the reboot" and the library handles rendezvous and other membership tasks. Once all N are running, it reports a *membership view* listing the N members (consistently!).

OTHER MEMBERSHIP MANAGEMENT ROLES

Derecho does much more, even at startup.

- It handles the "layout" role of mapping your N replicas to the various subgroups you might want in your application, and then tells each replica what role it is playing (by instantiating objects from classes you define, one class per role). It does "sharding" too.
- If an application manages persistent data in files or a database, it automatically repairs any damage caused by the crash. This takes advantage of replication: with multiple copies of all data, Derecho can always find any missing data to "fill gaps".
- It can initialize a "blank" new member joining for the first time.

SPLIT BRAIN CONCERNS

Tes.com

This worry arises if a service plays an important role in a system, and has a backup scheme: if the primary server crashes, everyone uses the backup

Suppose that some machines think the primary is down but others think it is still up. Now we have two servers both making decisions.

Our system could behave incorrectly!

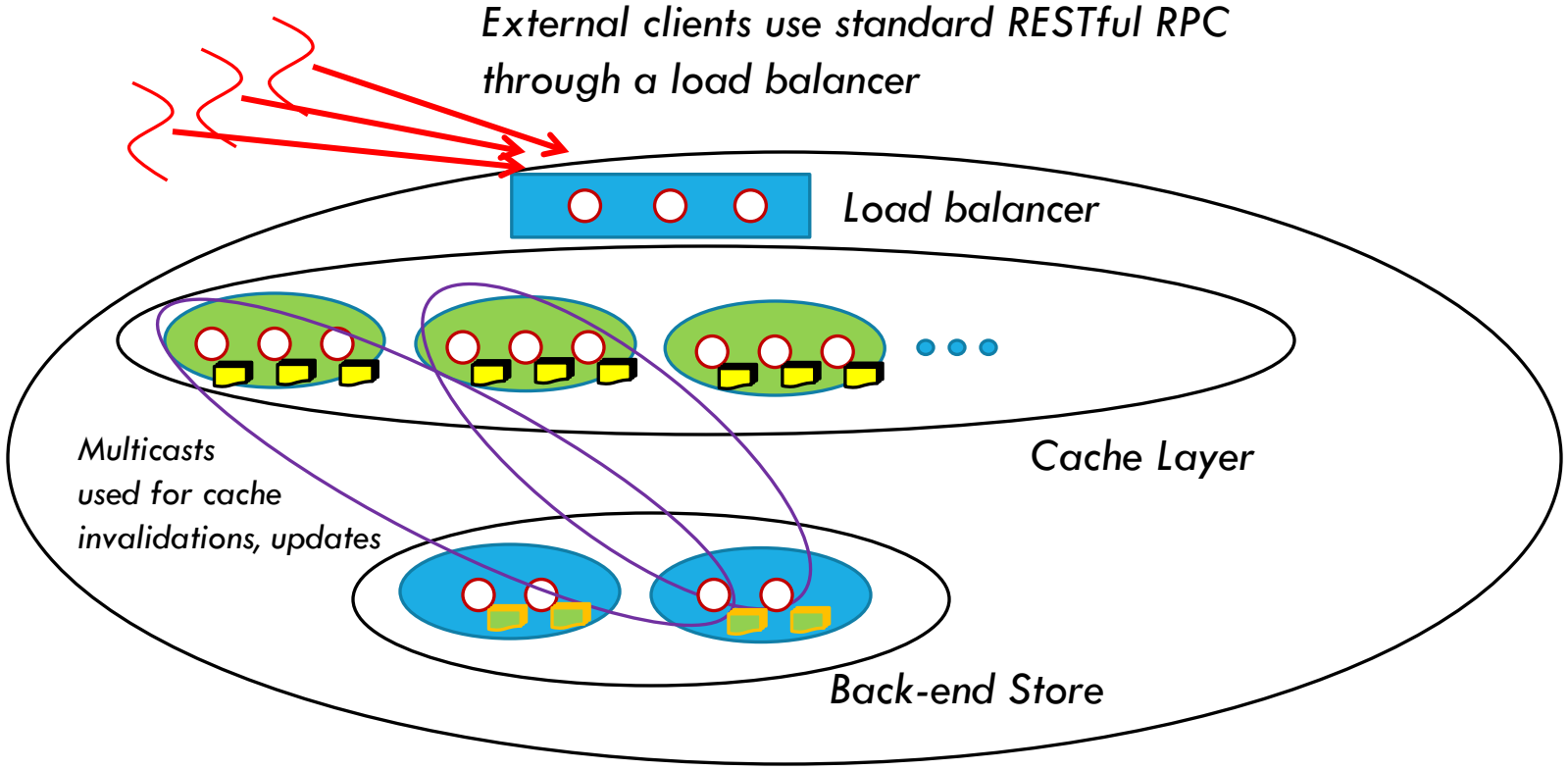
SOLVING THE SPLIT BRAIN PROBLEM

We use a "quorum" approach – and this is the tie-in to Paxos!

Our system has N processes and only allows progress if more than half agree on the next membership view. Example: if N=5, we say that after a failure, we need 3 or more of the original N to resume.

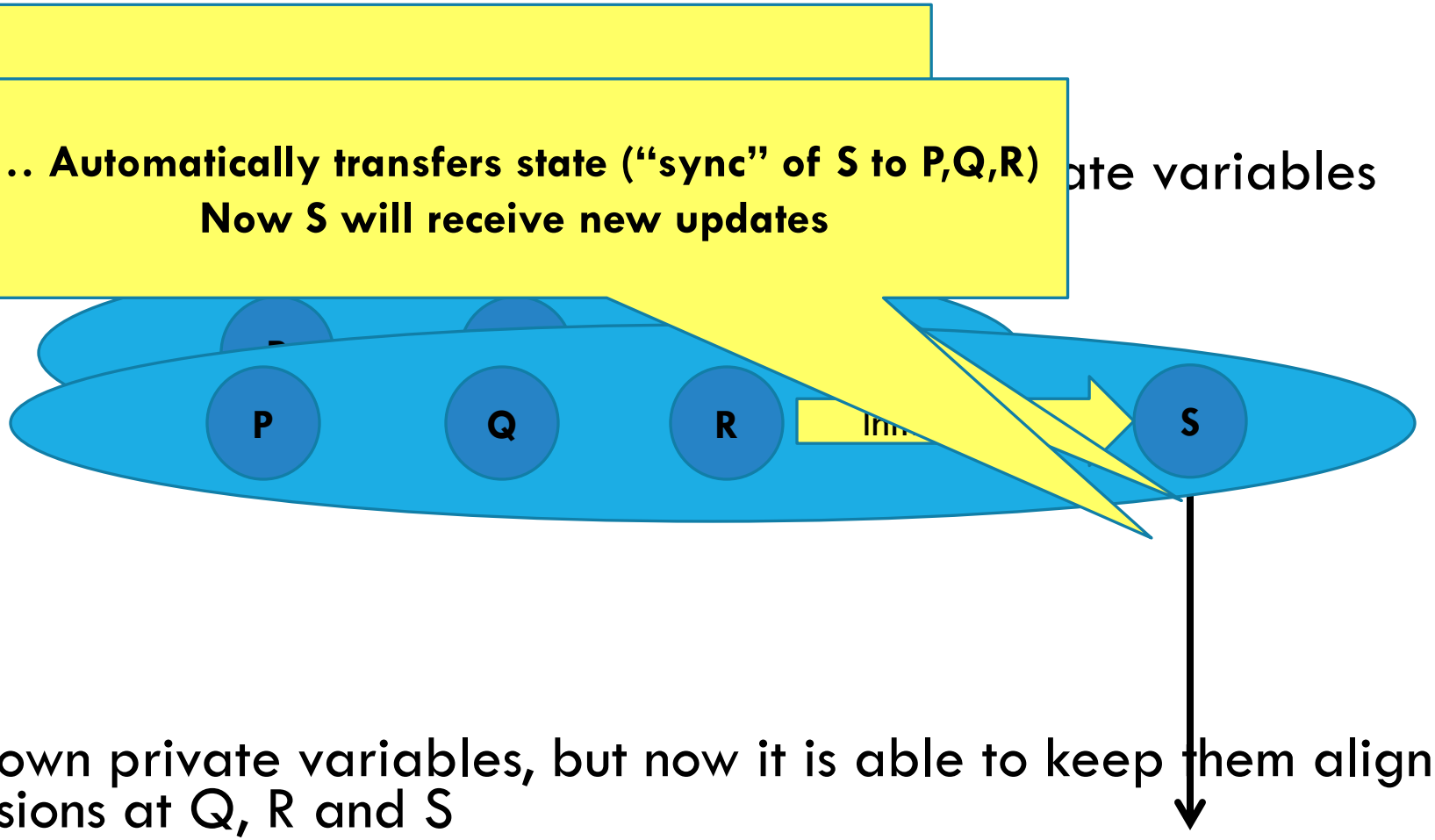
Since there can't be two subsets that both have more than half, it is impossible to see a split into two subservices.

… BEYOND SHARDING, DERECHO CAN EVEN SUPPORT STRUCTURES LIKE THIS!



A PROCESS JOINS A GROUP

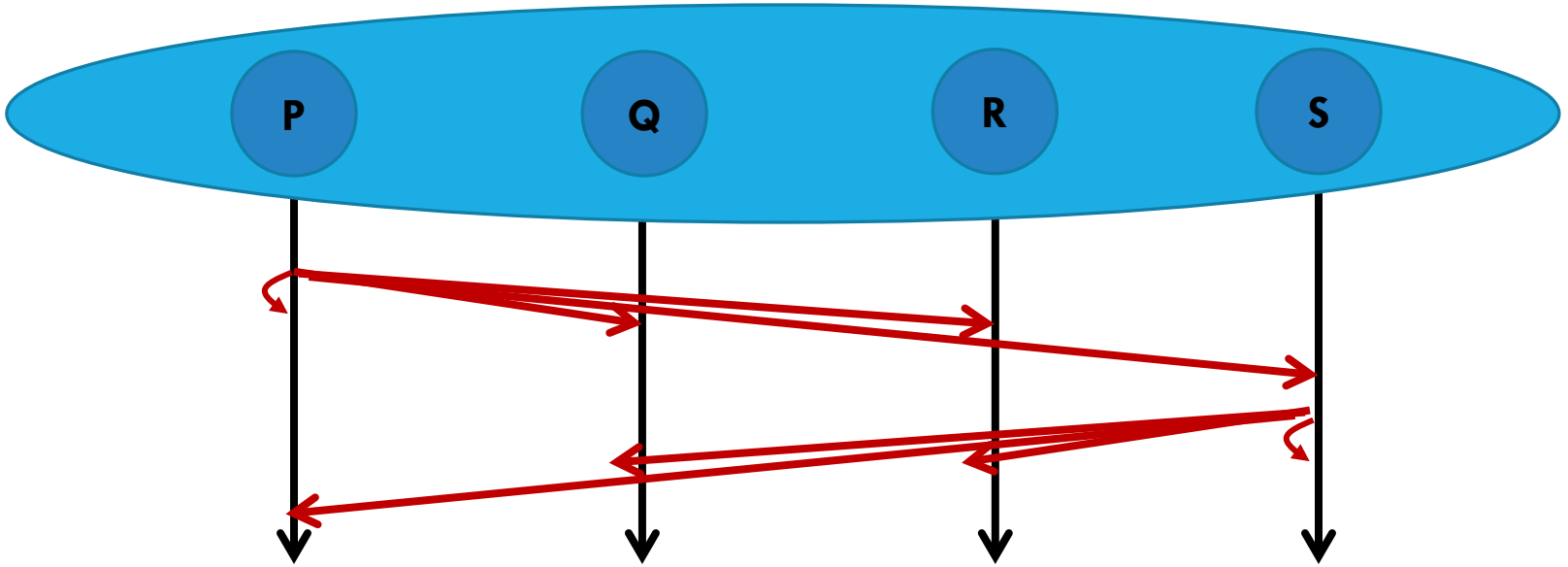
At first, P ate variables



P still has its own private variables, but now it is able to keep them aligned with track the versions at Q, R and S

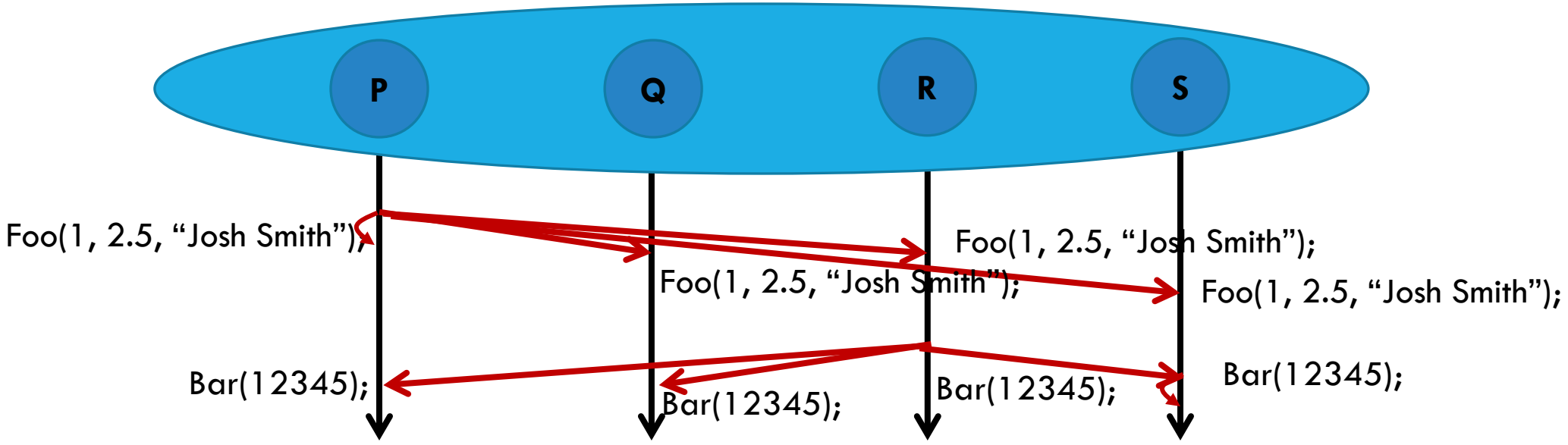
A PROCESS RECEIVING A MULTICAST

All members see the same "view" of the group, and see the multicasts in the identical order.



A PROCESS RECEIVING AN UPDATE

In this case the multicast invokes a method that changes data.



SO, SHOULD WE USE CHAIN REPLICATION IN THESE SUBGROUPS AND SHARDS?

It turns out that once we create a subgroup or shard, there are better ways to replicate data.

Derecho delivers ordered multicasts in a way that it extremely efficient, using the hardware in a smarter way than chain replication.

A common goal is to have every member be able to participate in handling work: this way with K replicas, we get K times more "power".

WHAT EXACTLY DOES STATE MACHINE REPLICATION GIVE US?

First, the Derecho version gives us membership tracking and also *layout* tracking: the mapping from members to subgroup/shard roles.

Next, it automates repair of damage after a crash.

Then, when active and healthy, it offers a way to send an "atomic multicast" or a "Paxos durable update" to all the members of a subgroup or a shard.

- If any process delivers such a multicast, or persists an updated state, all non-failed processes do, and they deliver in the same order.
- Data will be durable if desired: recovered after a crash.

THE "METHODS" PERFORM STATE MACHINE UPDATES. YOU GET TO CODE THESE IN C++.

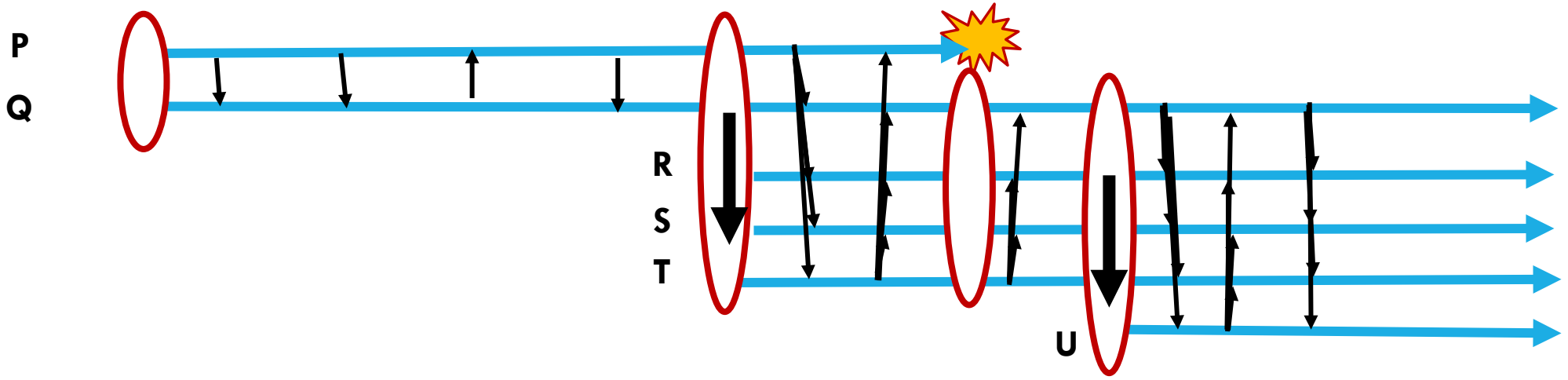
In these examples, we send an update by "calling" a method, Foo or Bar. The atomic multicast or Paxos is used to do the call, invisible to you.

Even with concurrent requests, every replica performs the identical sequence of Foo and Bar operations. We require that they be deterministic.

With an atomic multicast, everyone does the same method calls in the same order. So, our replicas will evolve through the same sequence of values.

VIRTUAL SYNCHRONY: MANAGED GROUPS

Epoch: A period from one membership view until the next one.

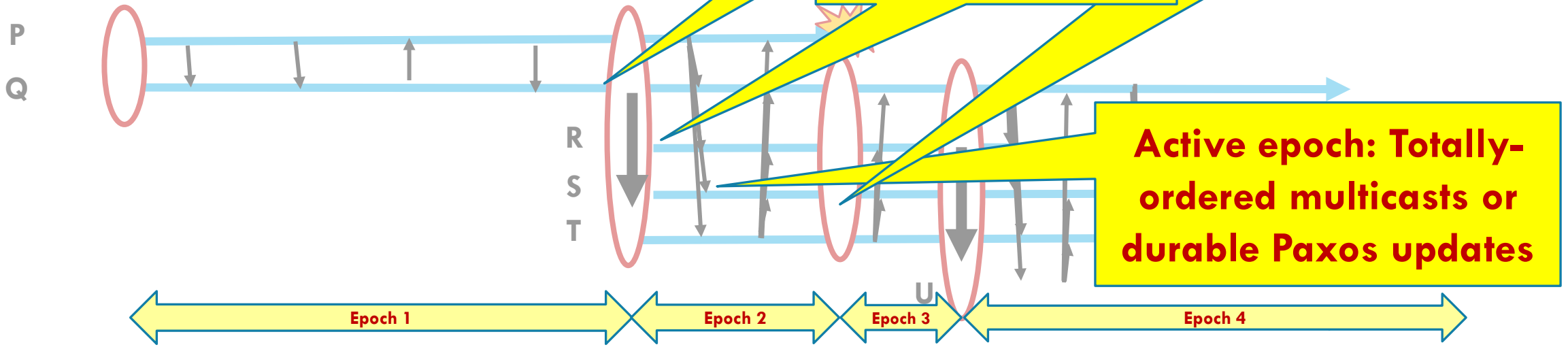


Joins, failures are “clean”, state is transferred to joining members

Multicasts reach all members, delay is minimal, and order is identical…

VIRTUAL SYNCHRONY: MA S

Epoch: A period from one membersh e.



Joins, failures are "clean", state is transferred to joining members

Multicasts reach all members, delay is minimal, and order is identical…

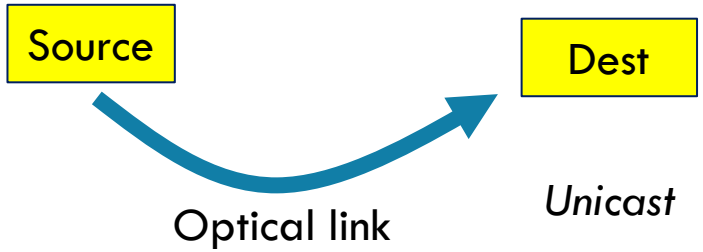
DERECHO'S VERSION OF PAXOS

Derecho splits its Paxos protocol into two sides.

One side handles message delivery within an epoch: a group with unchanging membership.

The other is more complex and worries about membership changes (joins, failures, and processes that leave for other reasons).

HOW DOES DERECHO TRANSFER DATA? IT USES "RDMA".

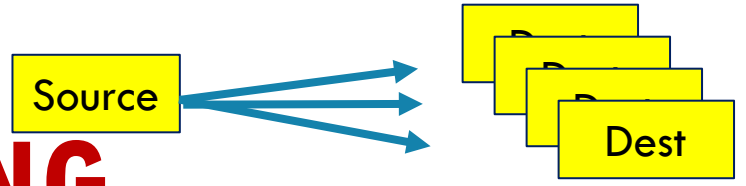


RDMA: Direct **zero copy** from source memory to destination memory. But it is like TCP: a one-to-one transfer, not a one-to-many transfer.

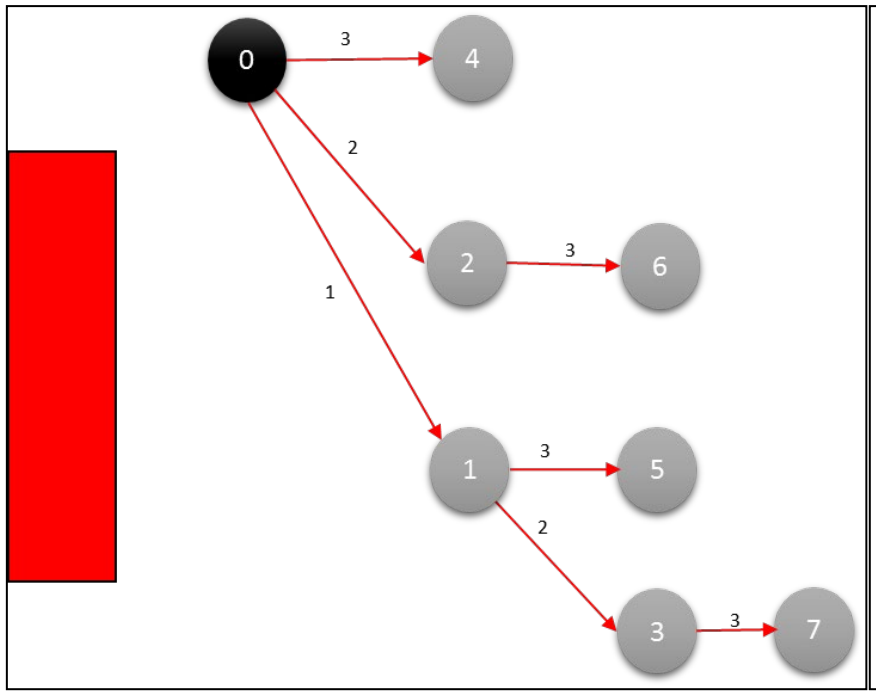
RDMA can actually transfer data to a remote machine faster than a local machine can do local copying.

Like TCP, RDMA is reliable: if something goes wrong, the sender or receiver gets an exception. This only happens if one end crashes

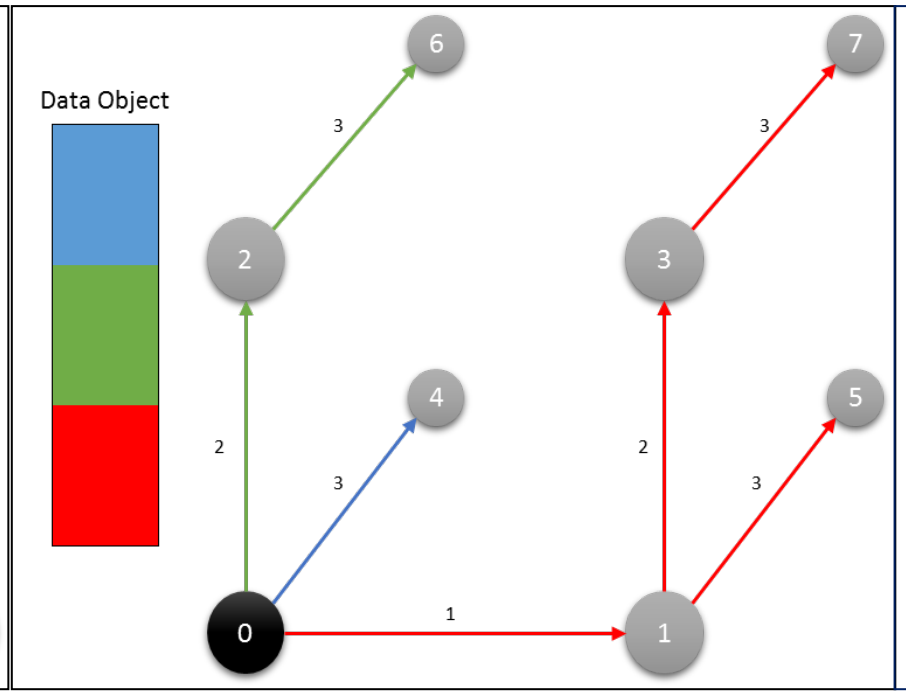
LARGE MESSAGES USE A RELAYING METHOD WE CALL RDMC



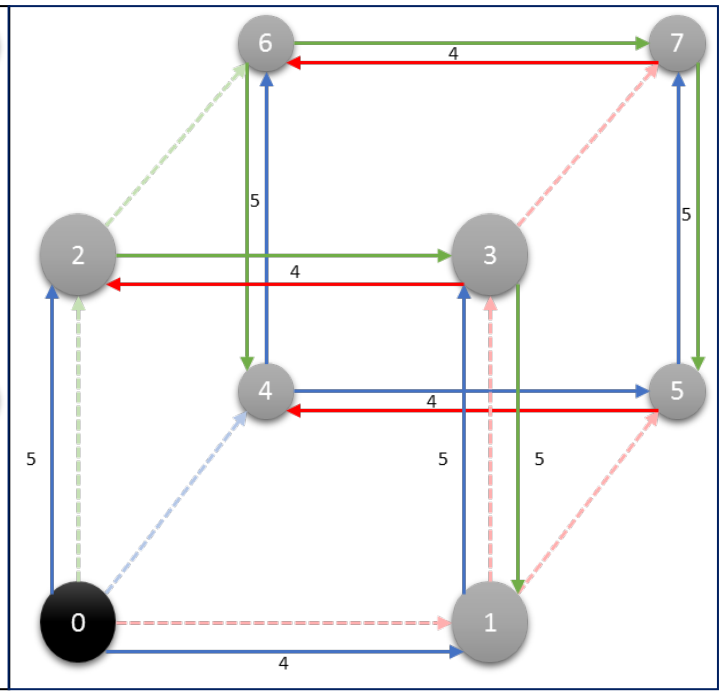
Multicast



Binomial Tree

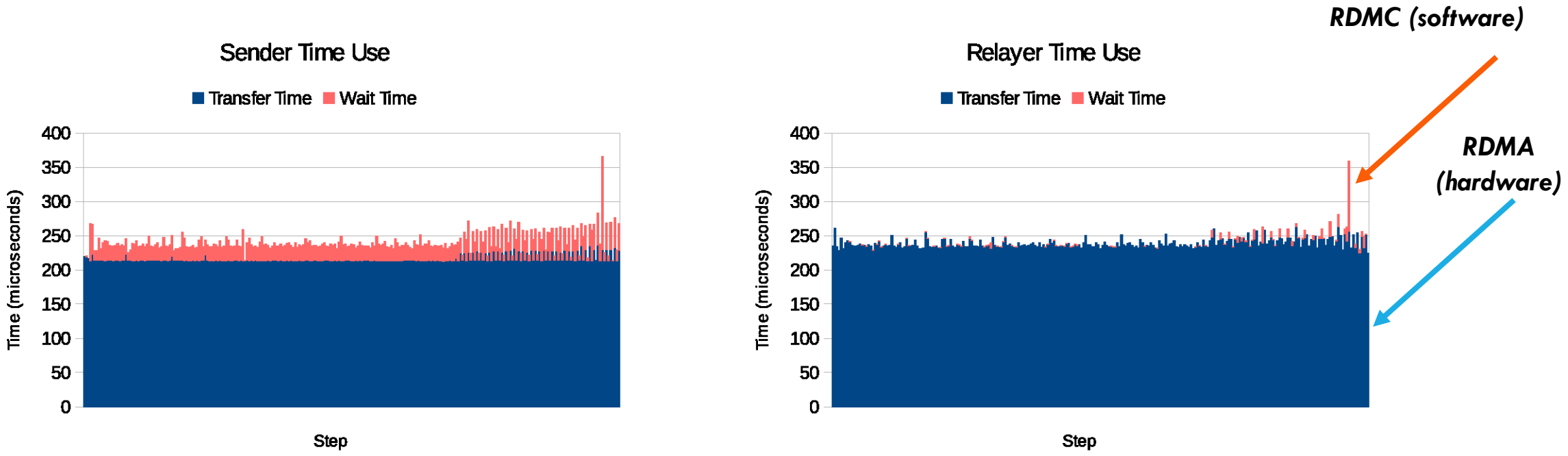


Binomial Pipeline



Final Step

RDMC SUCCEEDS IN OFFLOADING WORK TO HARDWARE



Trace a single multicast through our system… Orange is time "waiting for action by software". Blue is "RDMA data movement".

HOW DOES DERECHO PUT MESSAGES IN ORDER?

Derecho asks each subgroup or shard to designate which members are "active senders" in a given view.

- Within the senders, Derecho just uses round-robin order: message 1 from P: P:1 Q:1 R:1 P:2 Q:2 R:2…
- If some process has nothing to send Derecho automatically inserts a *null* message. P:1 Q:1 - - Q:2 R:2…

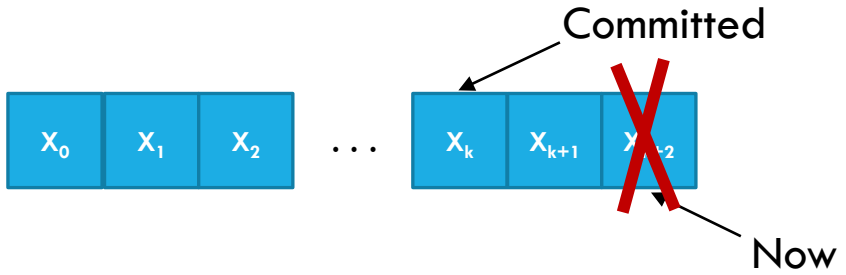
ARE WE FINISHED?

We still need to understand how to end one epoch, and start the next.

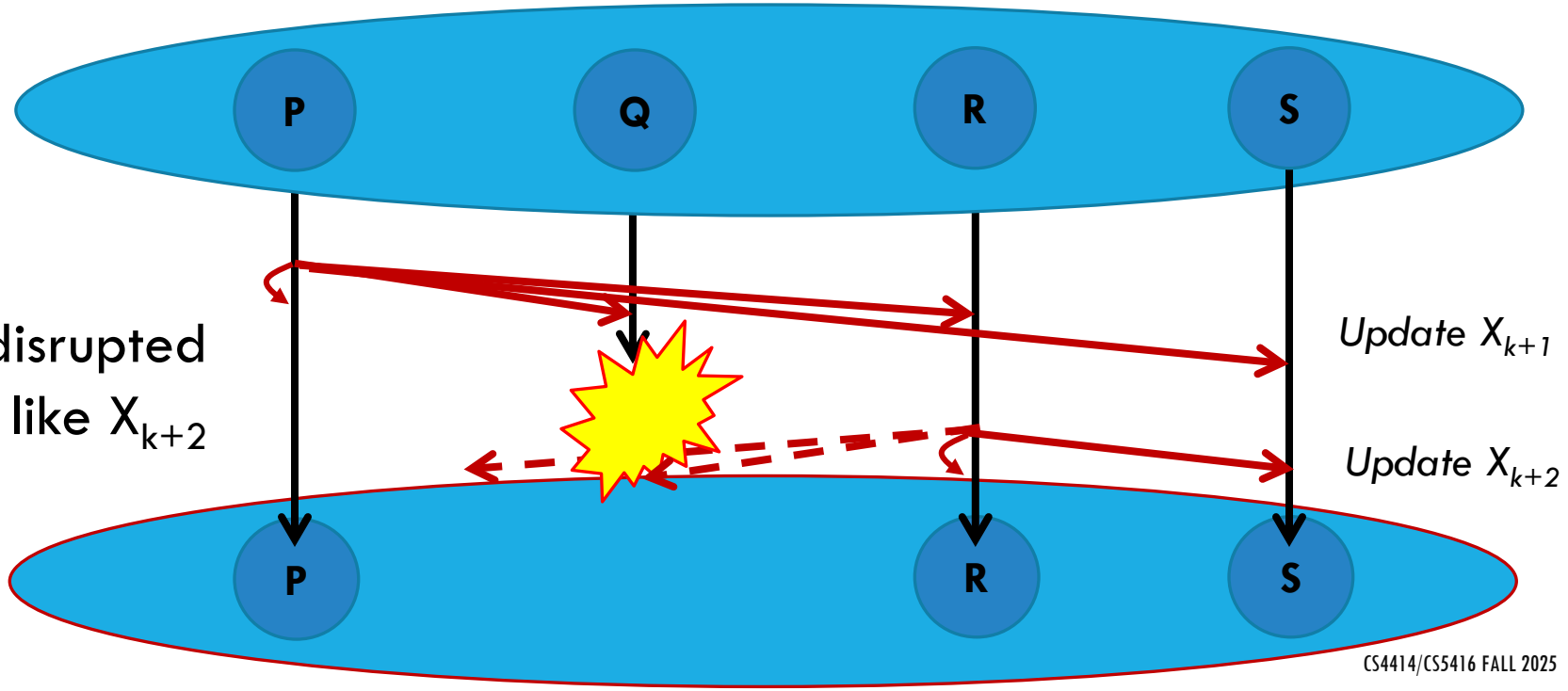
Derecho's method for this is a bit too complex for this lecture, but in a nutshell it cleans up from failures, then runs a protocol (based on quorums) to agree on the next view (the next epoch membership), then restarts.

If a multicast was disrupted by failure, it then will be reissued.

A PROCESS FAILS



Failure: If a message was committed by any process, it commits at*every* process. But some unstable recent updates might abort.



EXAMPLE OF HOW THIS WOULD BE USED?

In chain replication, suppose a failure disrupts our chain while an update is underway.

Cleanup could discard the update if it hasn't yet reached the tail, and finalize it everywhere if it definitely did reach the tail.

Derecho uses this same kind of reasoning to clean up its own multicasts in the event of a disruptive crash.

HOW MUCH COST DOES ORDERING AND PAXOS RELIABILITY OF THIS KIND ADD?

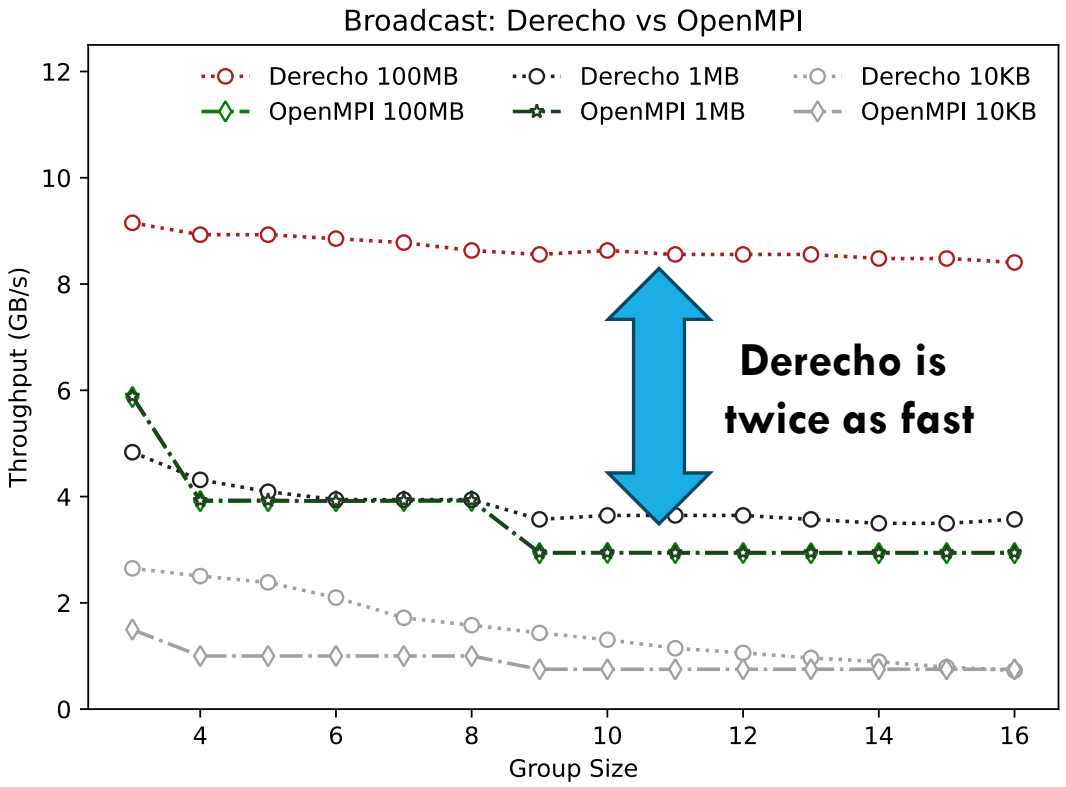
We can compare the Open MPI multicast, which has no guarantees, with an ordered Paxos protocol layered on RDMC in Derecho.

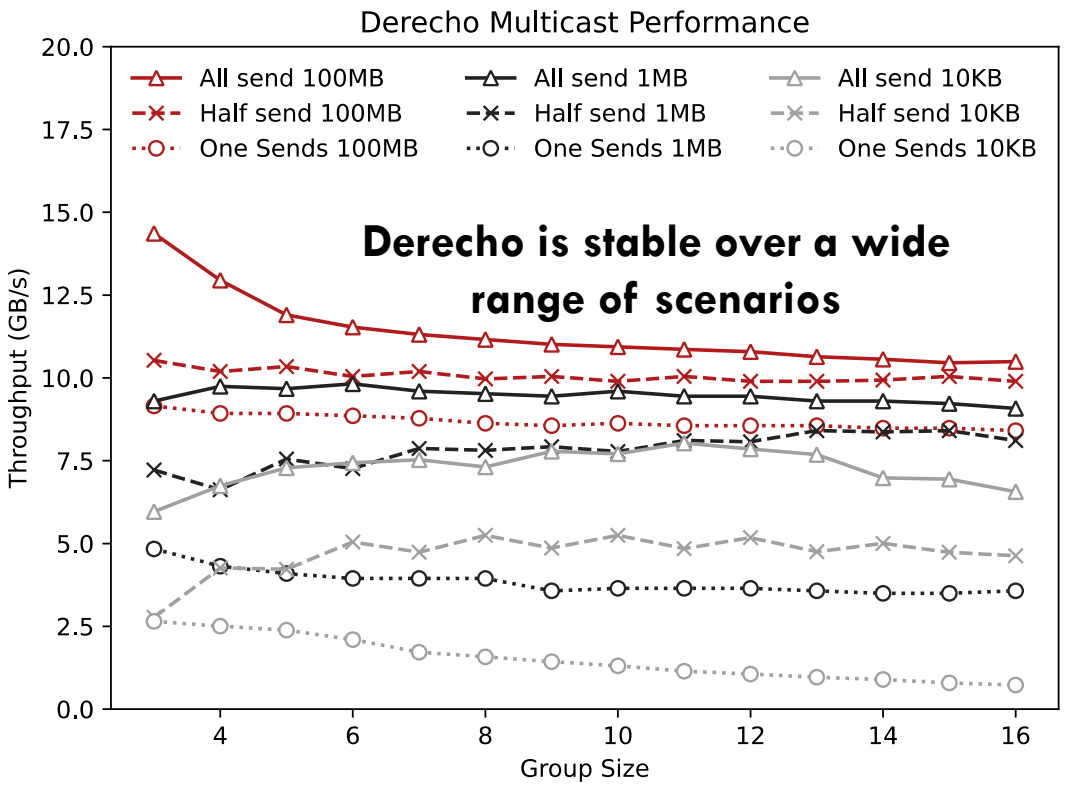
Our next slide shows what we get for various object sizes and group sizes.

Red: “a video” (100MB), Blue: “a photo” (1MB), Green: “an email” (10K).

Again, 3 cases: all send (solid), half send (dashed), one sends (dash dot)

DERECHO ATOMIC MULTICAST IS TWICE AS FAST AS THE NON-ATOMIC OPEN MPI RDMA MULTICAST





CONSISTENCY: A PERVASIVE GUARANTEE

Every application has a consistent view of membership, and ranking, and sees joins/leaves/failures in the same order.

Every member has identical data, either in memory or persisted

Members are automatically initialized when they first join.

Queries run on a form of temporally precise consistent snapshot

Yet the members of a group don't need to act identically. Tasks can be "subdivided" using ranking or other factors

FRAMEWORKS BUILT OVER DERECHO INHERIT THESE PROPERTIES

For example, we can build a key-value store, or a file system, or a publish-subscribe queuing system.

They run at extremely high speeds – higher than standard ways of building things. Consistency doesn't harm them at all.

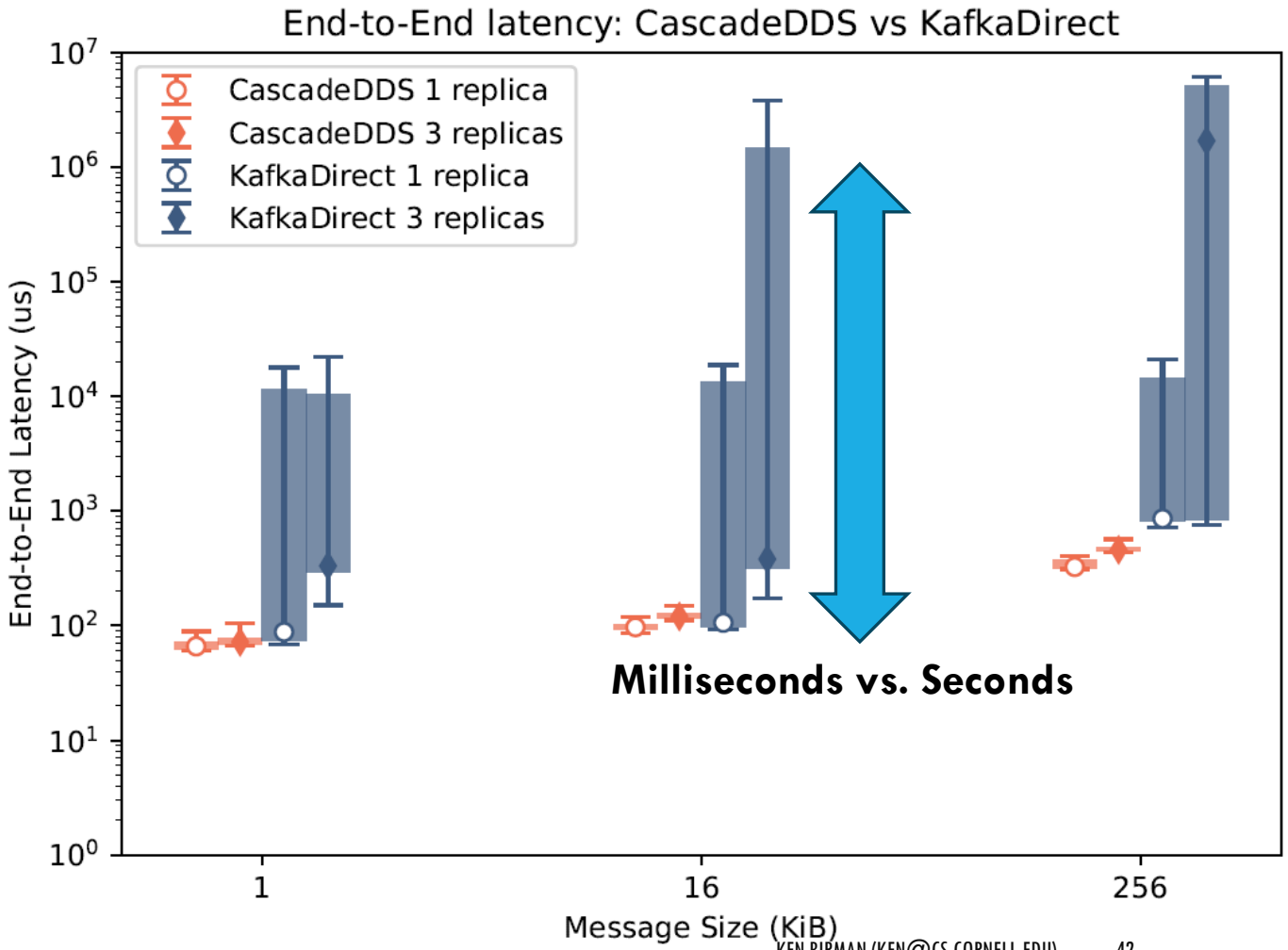
Applications, like MLs, using these services inherit consistency without needing to do anything special!

EXAMPLE: CASCADE DDS COMPARED TO KAFKA-DIRECT. BOTH USE THE SAME API.

Smaller is better

Tighter is better

Logarithmic Y axis



REPLICATION: MANY WAYS TO GET THERE!

By now we have heard of many ways to implement similar functionality.

- The actual Paxos protocols Leslie proposed. Those are slow.
- There are many famous "variations" on Paxos. RaFT is popular. Not faster, but easier to implement.
- Chain replication, but with a suitable membership service.
- A tool called Zookeeper that we didn't discuss. Used in Hadoop.
- Derecho, fastest of them all!

LAYERS ON LAYERS!

Complete free-standing self-managed μ-service

Familiar APIs, like a file system or message bus or blob store

Library you link to

Higher level tools, like the versioned, temporally indexed Derecho object store (the key-value store)

Derecho's version of atomic multicast and durable Paxos

Data replication: Streaming over RDMC

The shared state table (coordination)

Fancy structures with subgroups and sharding

Virtual synchrony membership layer

SOME PRACTICAL COMMENTS

Derecho is very flexible and strongly typed *when used from* C++.

But people working in Java and Python can only use the system with byte array objects (size_t, char*).

You can't directly call a "templated" API from Java or Python, so:

- First you create a DLL with non-templated methods, compile it.
- Then you can load that DLL and call those methods.
- You still need to know some C++, but much less.

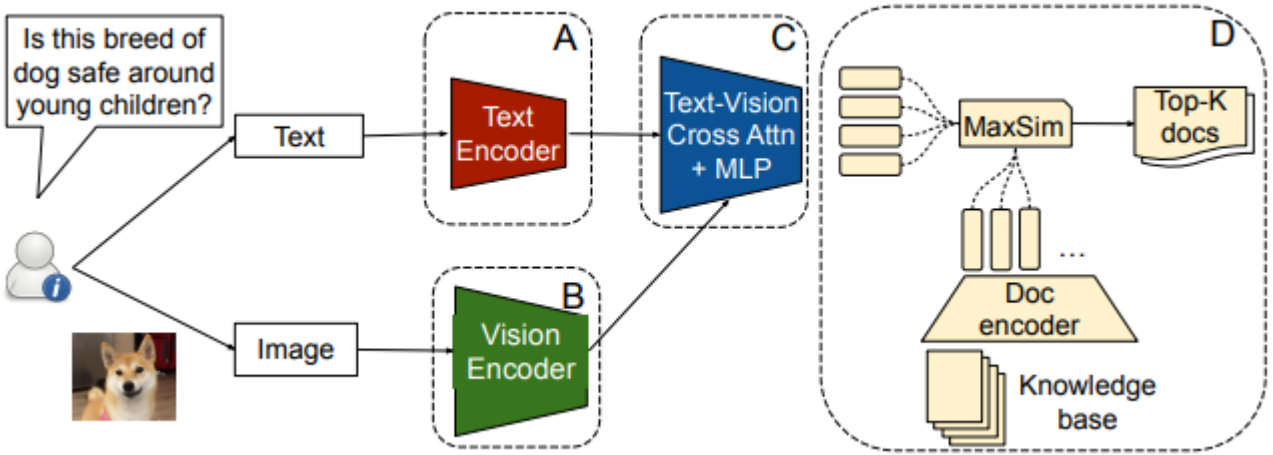
CORNELL VORTEX PROJECT

Underway right now. Alicia and Jamal and Shouxu are all involved.

The goal is to offer a super-efficient strongly efficient platform for hosting ML tasks, especially focused on inference and knowledge retrieval.

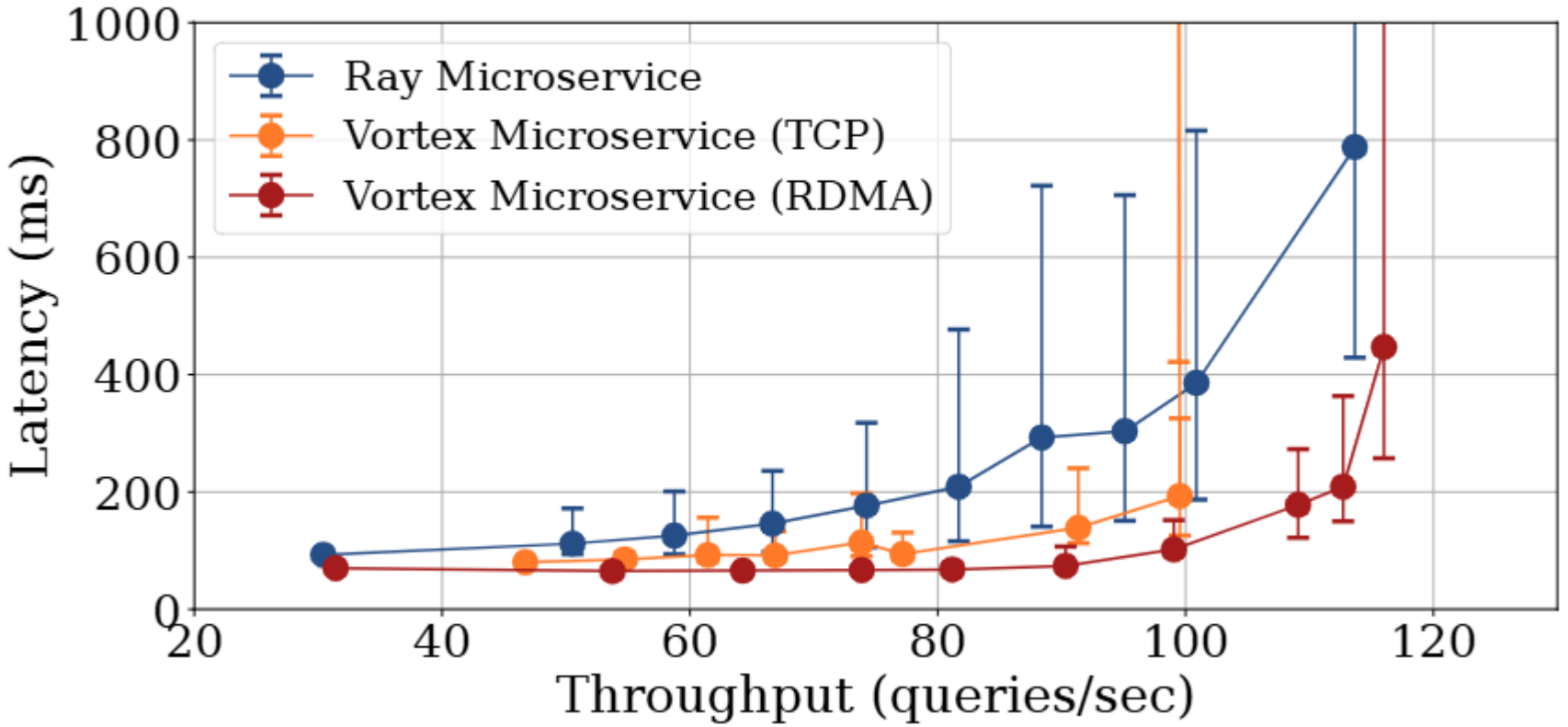
- Can support really low latency, which predictably fast responses
- And the MLs achieve very high throughput, making them inexpensive to operate compared to slower options

EXAMPLE VORTEX USE CASE: PREFLMR DEPLOYED AS A DOCUMENT RETRIEVAL SERVICE

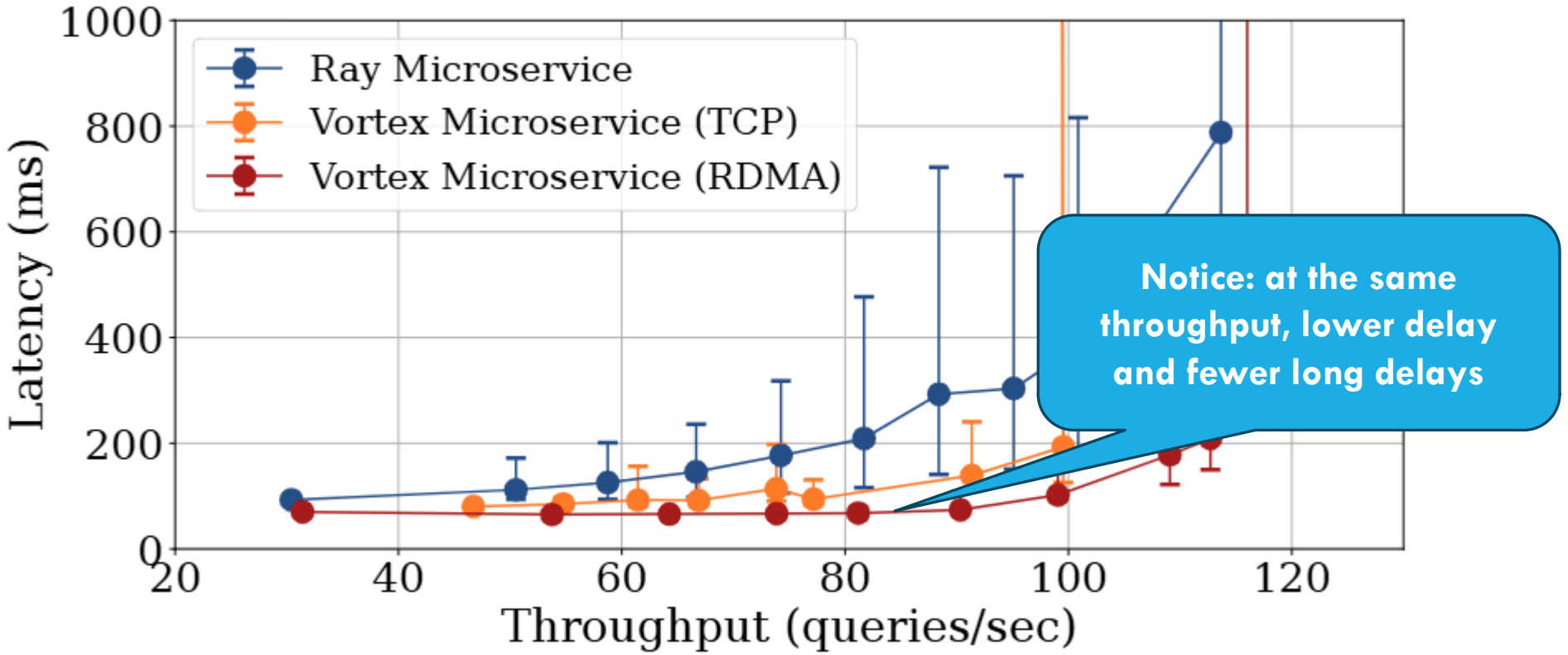


PreFLMR is a pipeline of ML components that can be used to answer questions about images

PREFLMR ON VORTEX WITH AND WITHOUT RDMA (COMPARED TO RAY SERVE)



PREFLMR ON VORTEX WITH AND WITHOUT RDMA (COMPARED TO RAY SERVE)



CONCLUSIONS?

Don't worry about consistency unless your code has a genuine need!

When needed, the state machine replication model is used to ensure consistency, fault-tolerance. Two cases: atomic multicast, persistent updates.

A software library like Derecho automates many aspects of creating a new consistent, fault-tolerant service. Virtualizing membership simplifies: the application is "notified" when it needs to do something.

SELF-STUDY QUESTIONS

Suppose your new job is for a manager who took CS4414/5416

The manager is worried about data consistency risks in the existing ML training framework the company uses. It collects all kinds of preexisting data (emails, memos, sales materials, other data) and fine tunes models.

Is data consistency a worry for ML training on unchanging data, or is the issue seen only with continuously updated data?

SELF-STUDY QUESTIONS

In a busy air traffic control center, John on ATC terminal A is responsible for takeoffs on runway 3. If the system crashes, Bill on ATC terminal B will take over from John.

Sarah, on ATC terminal B is responsible for controlling landings on runway 3. Her system needs to be certain nobody is taking off on runway 3 before authorizing a plan to land on runway 3.

Does "split brain" arise in this situation? Would virtual synchrony membership have the same risk?

SELF-STUDY QUESTIONS

A membership view is really a map of processes in the system to the roles each process plays, like “member 2 in shard 7 of the KV store”.

Visualizing an ML pipeline or training system, would you expect this kind of map to remain stable for long periods (in which case Derecho can be very efficient), or to dynamically change rapidly (in which case the overheads of virtual synchrony could be a significant cost)?

SELF-STUDY QUESTIONS

Although consistency throughout a platform pays off, even the people who build strongly consistent tools like Derecho oppose making them default for all of Linux and using them all the time.

Do you agree or disagree with this position? What challenges would arise if we tried to make distributed system membership services and replication groups universal features of distributed Linux platforms, used by default?

SELF-STUDY QUESTIONS

If we think of each component of PreFLMR as a small pool of servers that can be elastically resized to add workers or reduce workers, we would need to create a new Derecho membership view for each resizing event.

This could be costly.

Suppose we want to support dynamic resizing but without changing the membership view, and without paying a huge cost to migrate ML models and other dependent objects each time a new worker is launched. How could that be done?