CS5412 / LECTURE 8
REPLICATION AND CONSISTENCY

Ken Birman
Spring, 2020

RECAP

We discussed several building blocks for creating new μ-services, and along the way noticed that "consistency first" is probably wise.

But what additional fundamental building blocks we should be thinking about? Does moving machine learning to the edge create new puzzles?

We'll look at replicating data, with managed membership and consistency. Rather than guaranteed realtime, we'll focus on raw speed.

REMINDER: MANY MICROSERVICES ARE SHARDED AND USE REPLICATION

In our IoT cloud, "functions" run to react to IoT events are stateless and rather lightweight. Heavy lifting is done in microservices.

Many of those run on a pool of machines and are sharded – we treat the microservice as if it had many little "sub"-microservices inside.

A replicated shard (usually 2 or 3 members) uses the
the state machine replication model Lamport proposed

Leslie Lamport

OTHER TASKS THAT REQUIRE CONSISTENT REPLICATION

Copying programs to machines that will run them, or entire virtual machines.

Replication of configuration parameters and input settings.

Copying patches or other updates.

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

Replication so that a large set of first-tier systems have local copies of data needed to rapidly respond to requests

Replication for parallel processing in the back-end layer.

Data exchanged in the "shuffle/merge" phase of 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

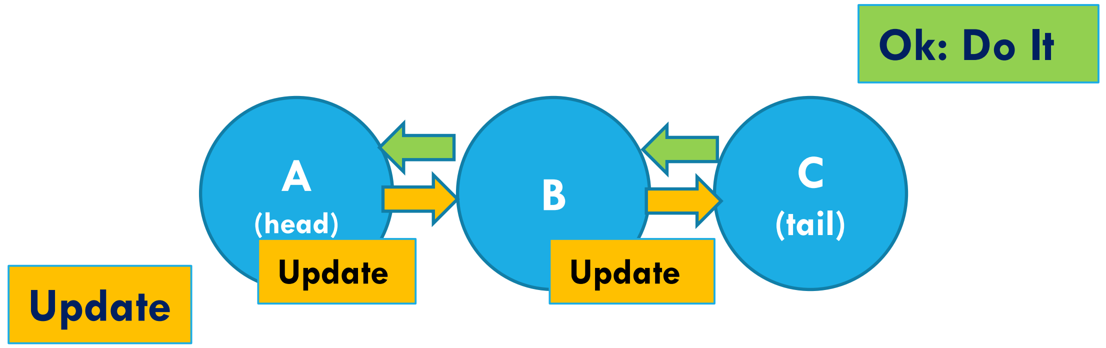
PAXOS: OVERARCHING “APPROACH”

Paxos (Greek Island)

Paxos is the name of a collection 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.

There are many ways to implement Paxos.

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 just 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 SATISFY PAXOS?

In some ways, but it is an incomplete story.

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

Chain replication is provably correct, but it assumes a membership mechanism, which it does not include. Without it, the chain replication scheme is not quite as strong as Paxos.

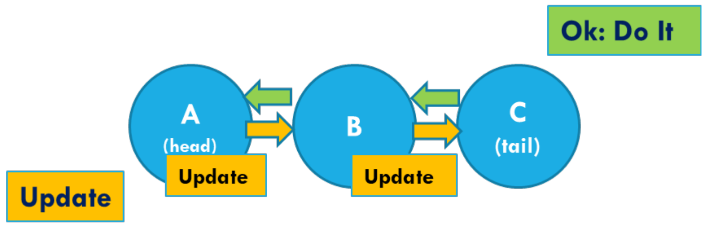
MEMBERSHIP AS A DIMENSION OF CONSISTENCY

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!

You build one program, linked to the Derecho C++ library.

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

Suppose your μ-service plays a key role, like air traffic control. There should only be one "owner" for a given runway or airplane.

But when a failure occurs, we want to be sure that control isn't lost. So in this case, the "primary controller" role would shift from process P to some backup process, Q.

The issue: With networks, we lack an accurate way to sense failures, because network links can break and this looks like a crash. Such a situation risks P and Q both trying to control the runway at the same time!

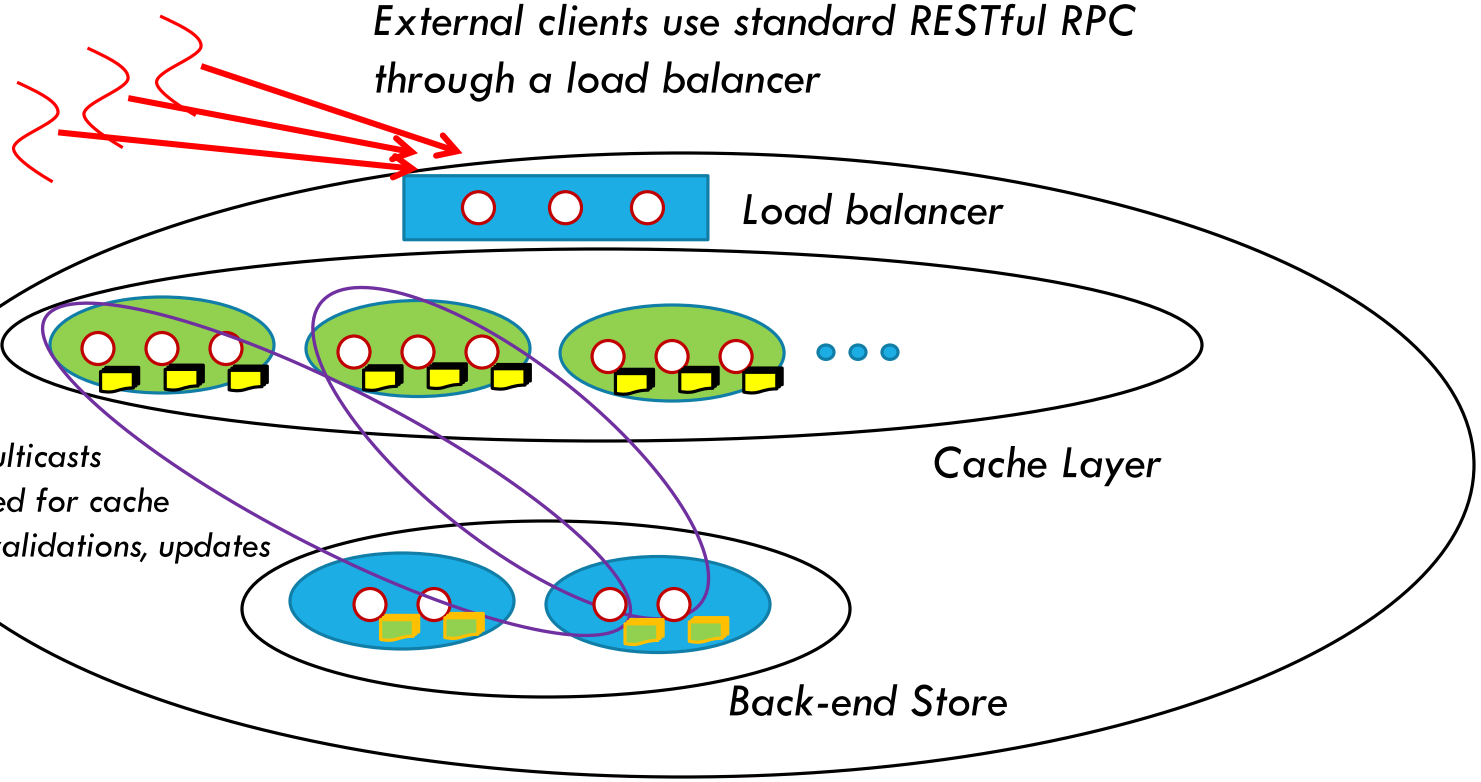
SOLVING THE SPLIT BRAIN PROBLEM

We use a “quorum” approach.

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.

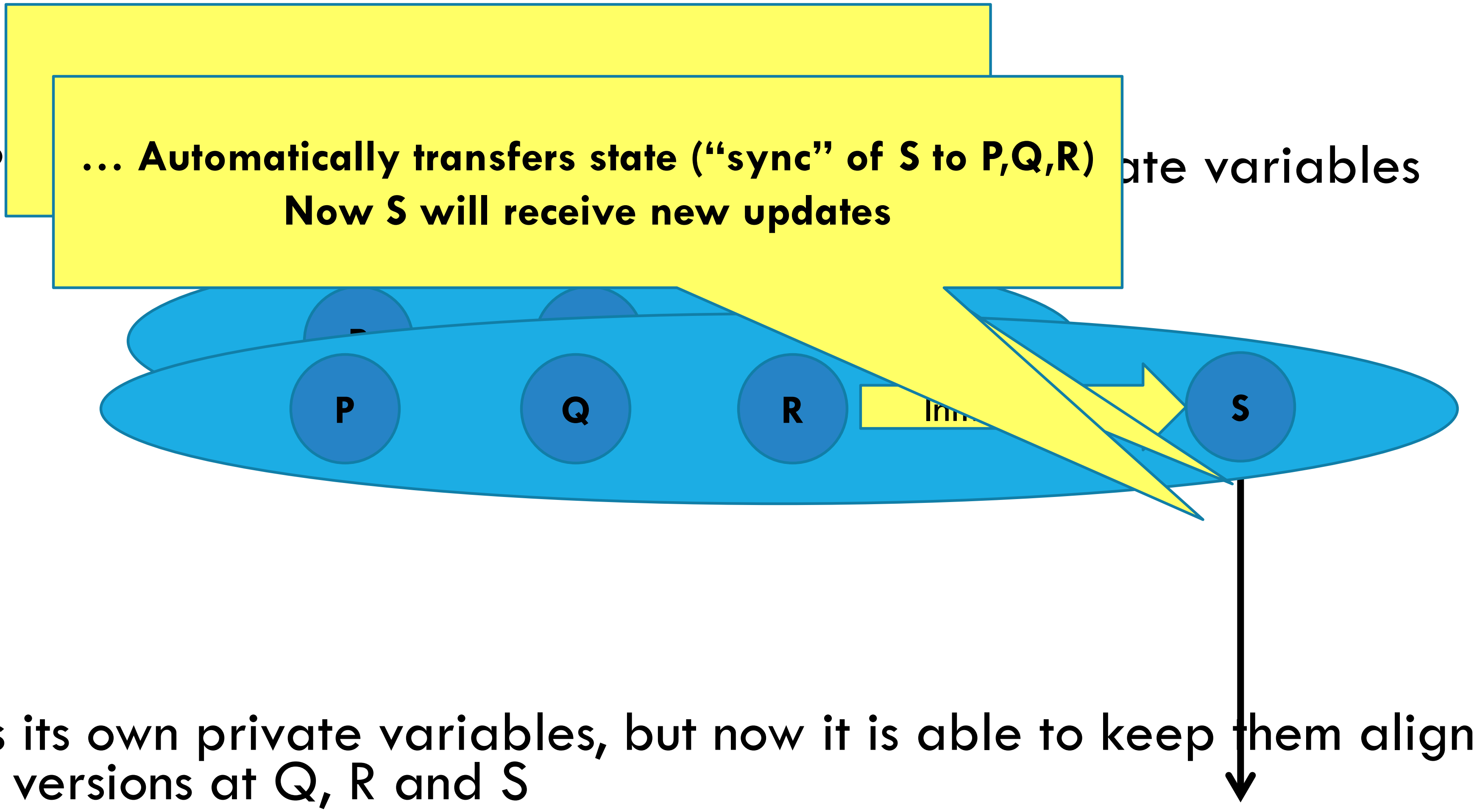
Since there can’t be two subsets that both have more than half, it is impossible to see a split into two subservices.

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


External clients use standard RESTful RPC through a load balancer
Load balancer
Multicasts used for cache invalidations, updates
Cache Layer
Back-end Store

A PROCESS JOINS A GROUP

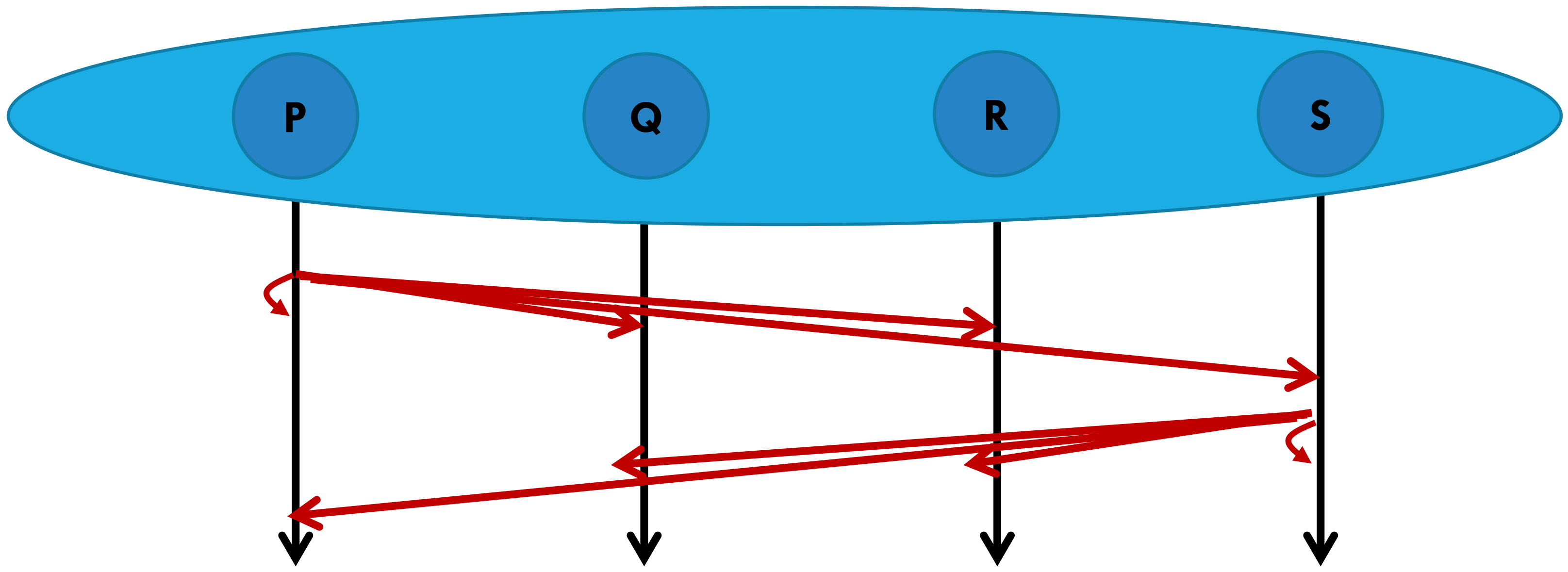
At first, P … ate variables


… Automatically transfers state ("sync" of S to P,Q,R)
Now S will receive new updates
P
Q
R
S
Ini

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

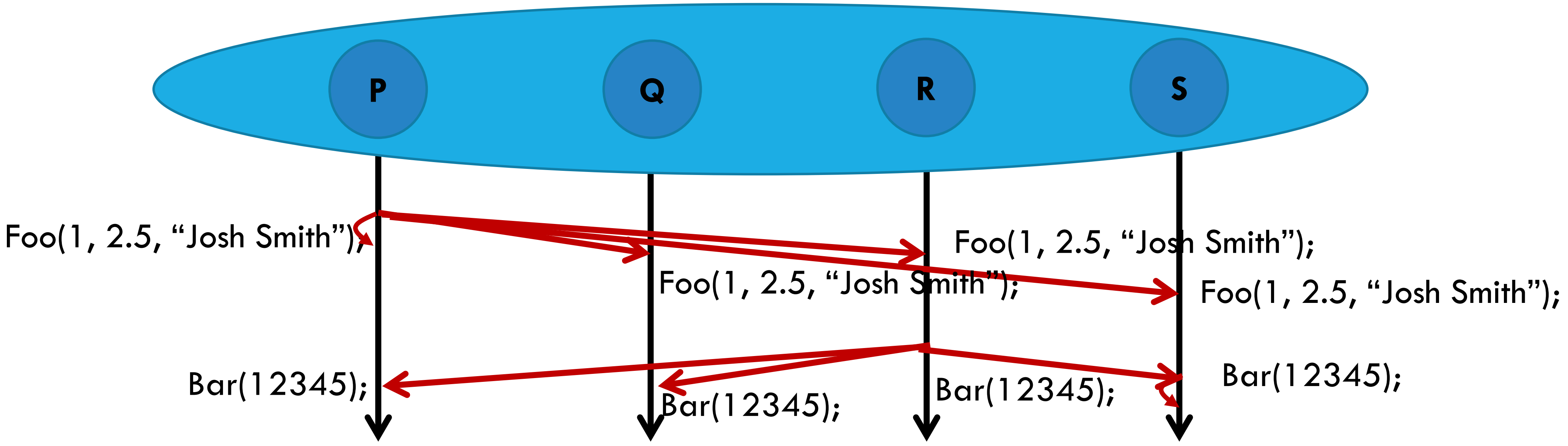
A PROCESS RECEIVING A MULTICAST

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.


P
Q
R
S
Foo(1, 2.5, “Josh Smith”);
Foo(1, 2.5, “Josh Smith”);
Foo(1, 2.5, “Josh Smith”);
Foo(1, 2.5, “Josh Smith”);
Bar(12345);
Bar(12345);
Bar(12345);
Bar(12345);

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

Next, automated 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++.

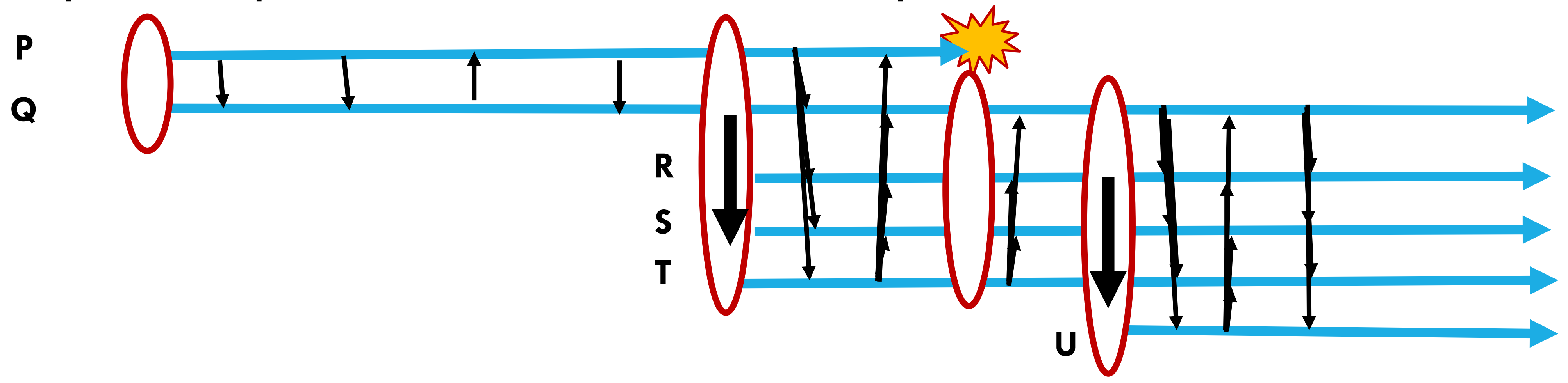
In these examples, we send an update by “calling” a method, Foo or Bar. The atomic multicast or Paxos is used to do the call, invisible to you.

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.

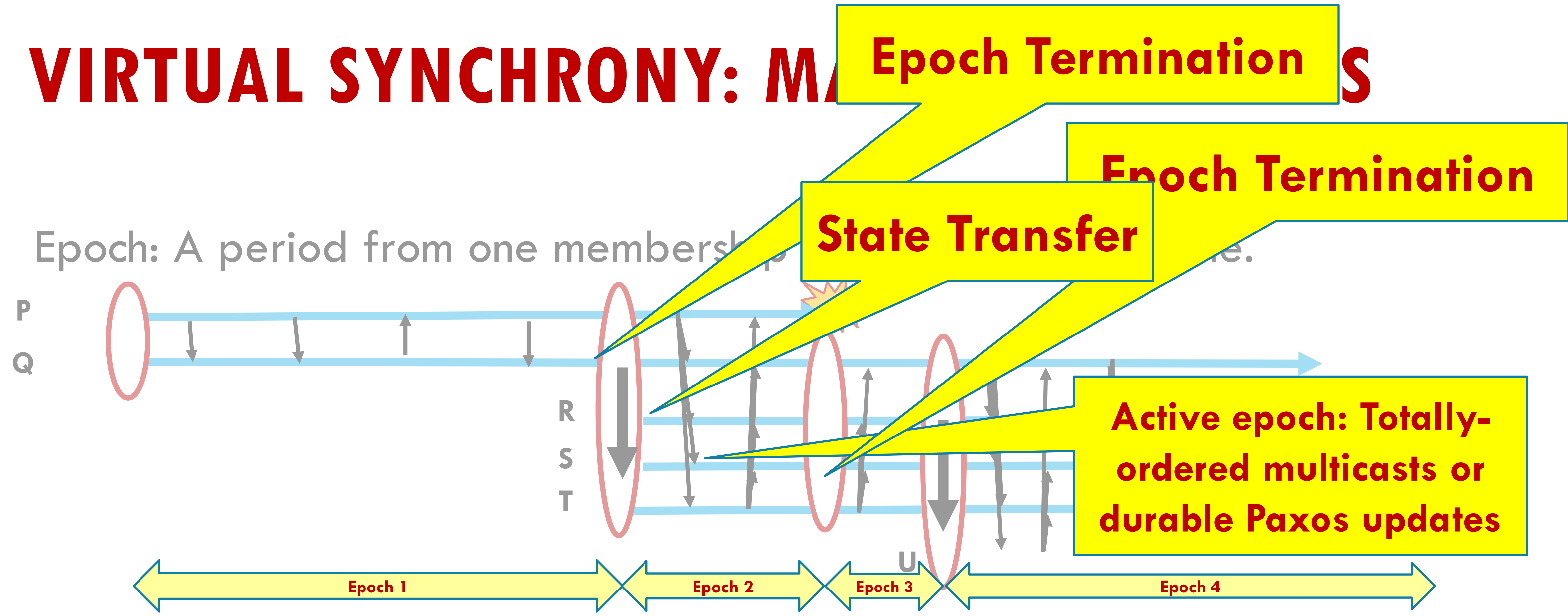


Joins, failures are "clean", state is transferred to joining members

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

VIRTUAL SYNCHRONY: MA S

Epoch: A period from one membersh p e.



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

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

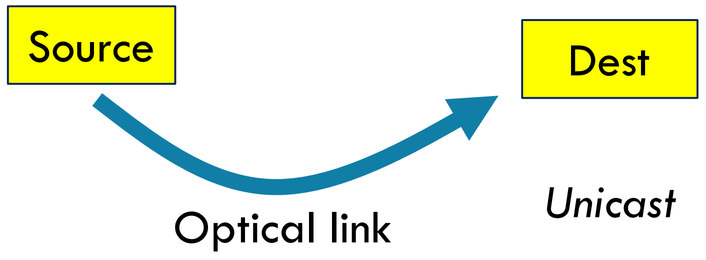
DERECHO'S VERSION OF PAXOS

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.

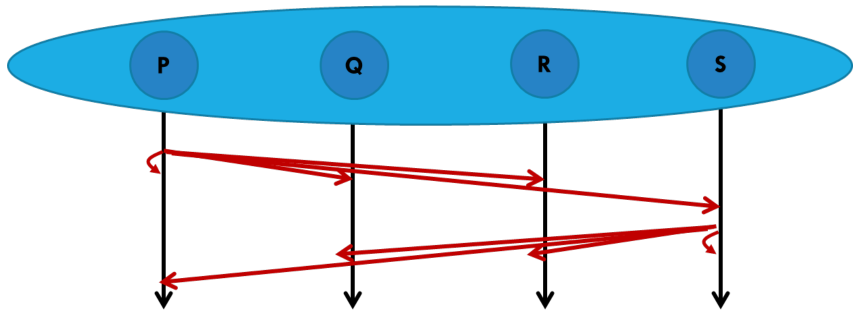
RDMA can actually transfer data to a remote machine faster than a local machine can do local copying.

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

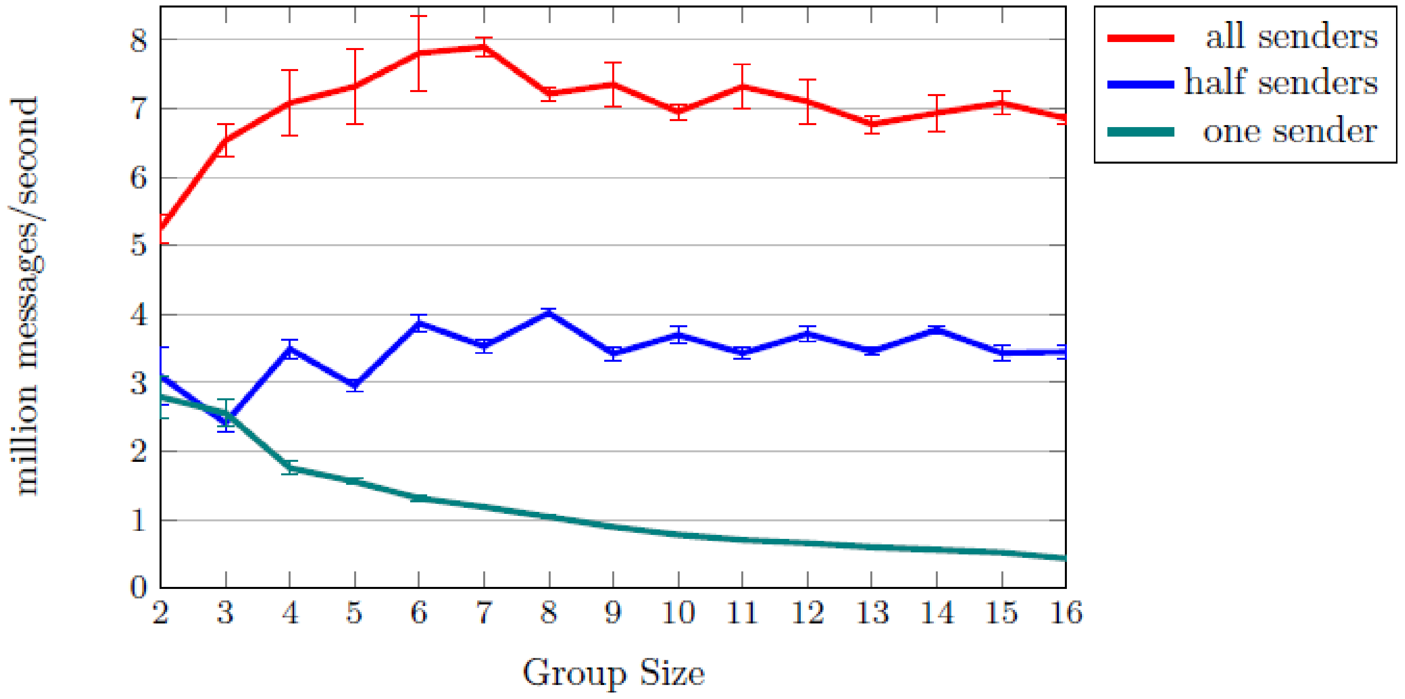
SMALL MESSAGES USE A DIRECT RDMA COPYING PROTOCOL WE CALL SMC.

Mellanox 100Gbps RDMA on ROCE (fast Ethernet)

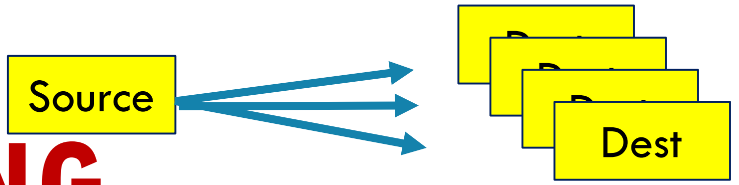
100Gb/s = 12.5GB/s



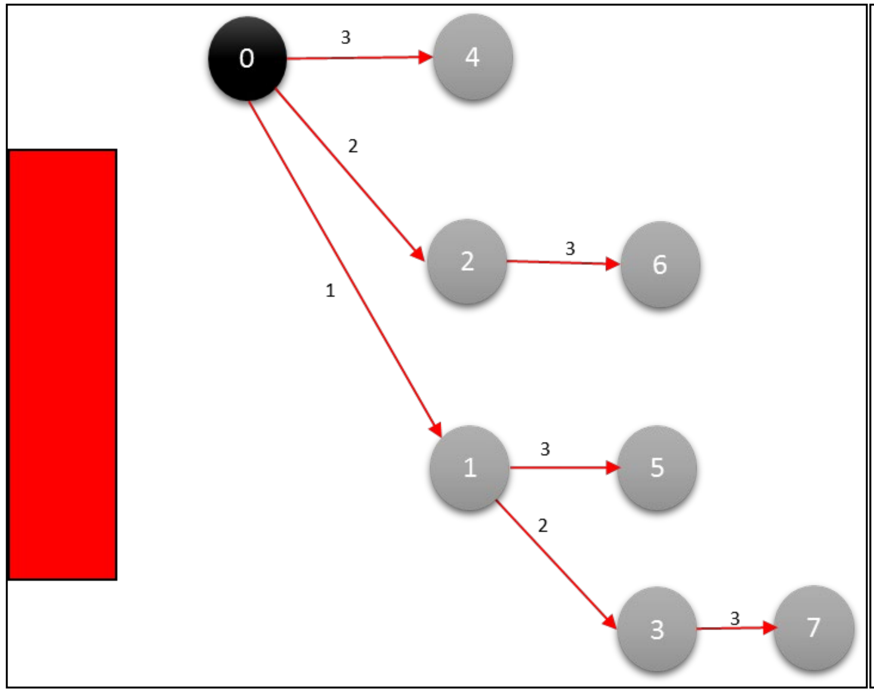
SMC Protocol, 1 byte messages



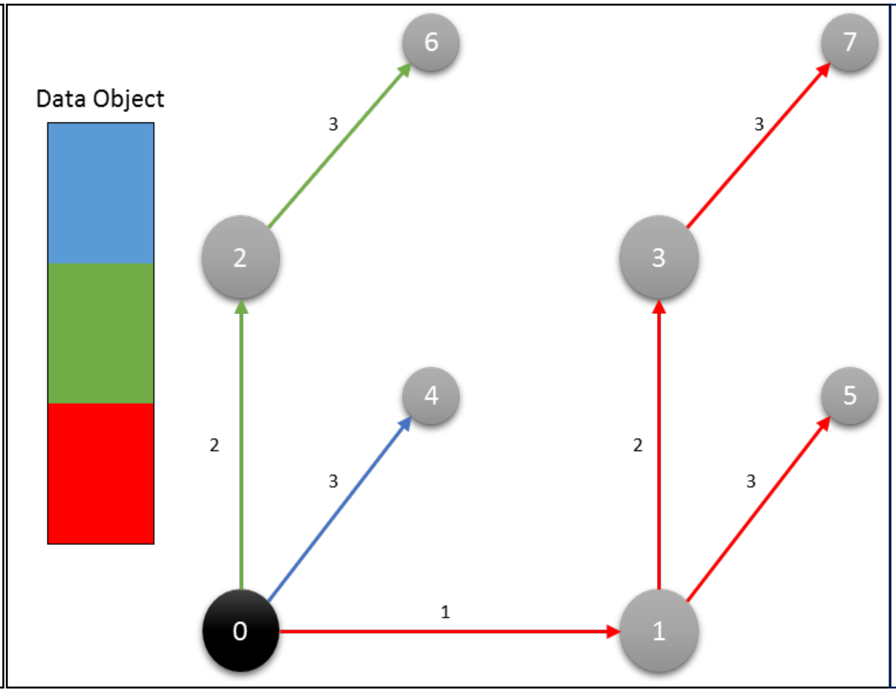
LARGE MESSAGES USE A RELAYING METHOD WE CALL RDMC



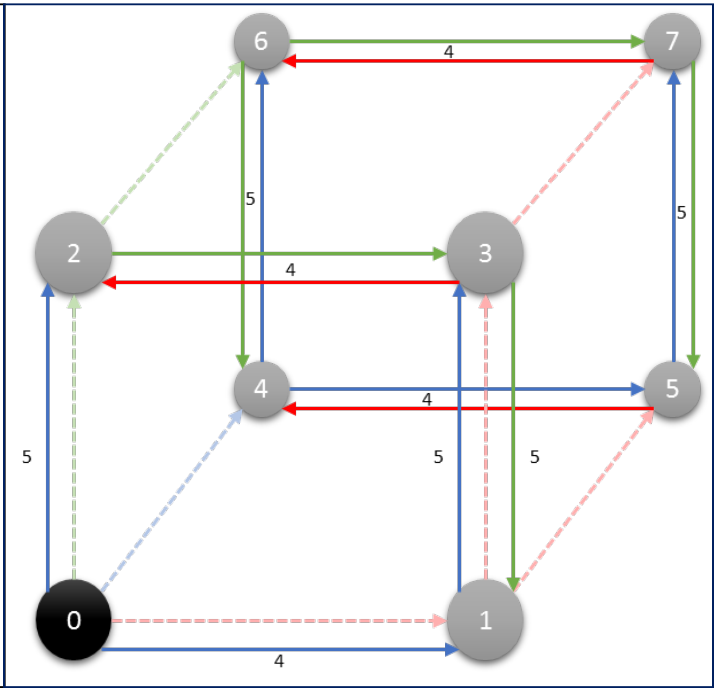
Multicast



Binomial Tree

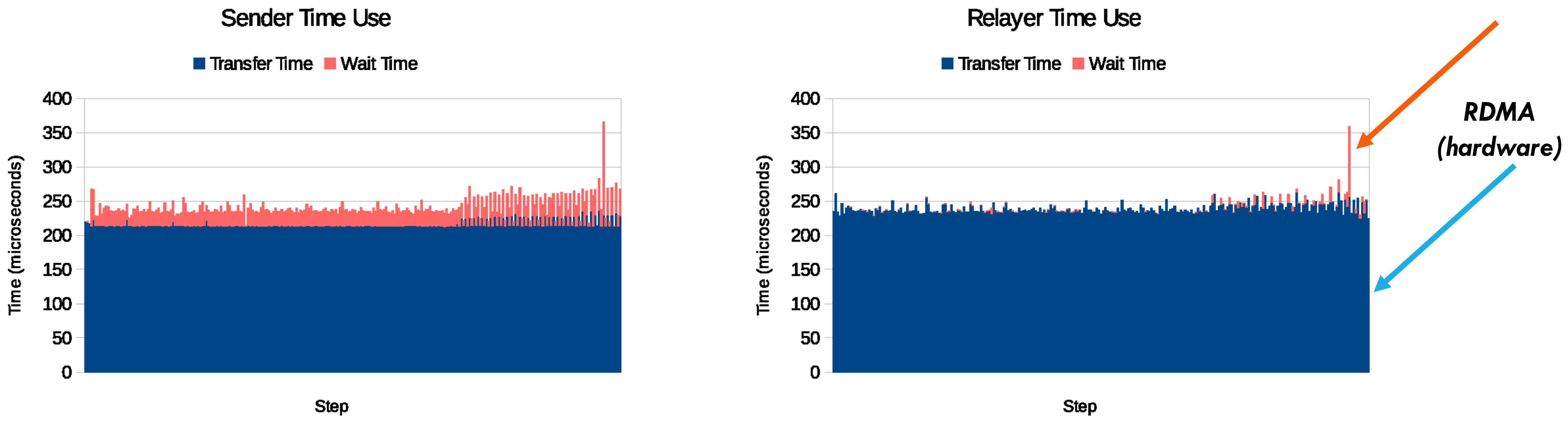


Binomial Pipeline



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…

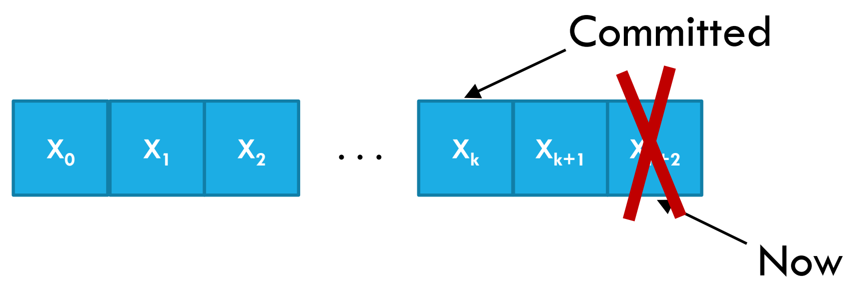
ARE WE FINISHED?

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.

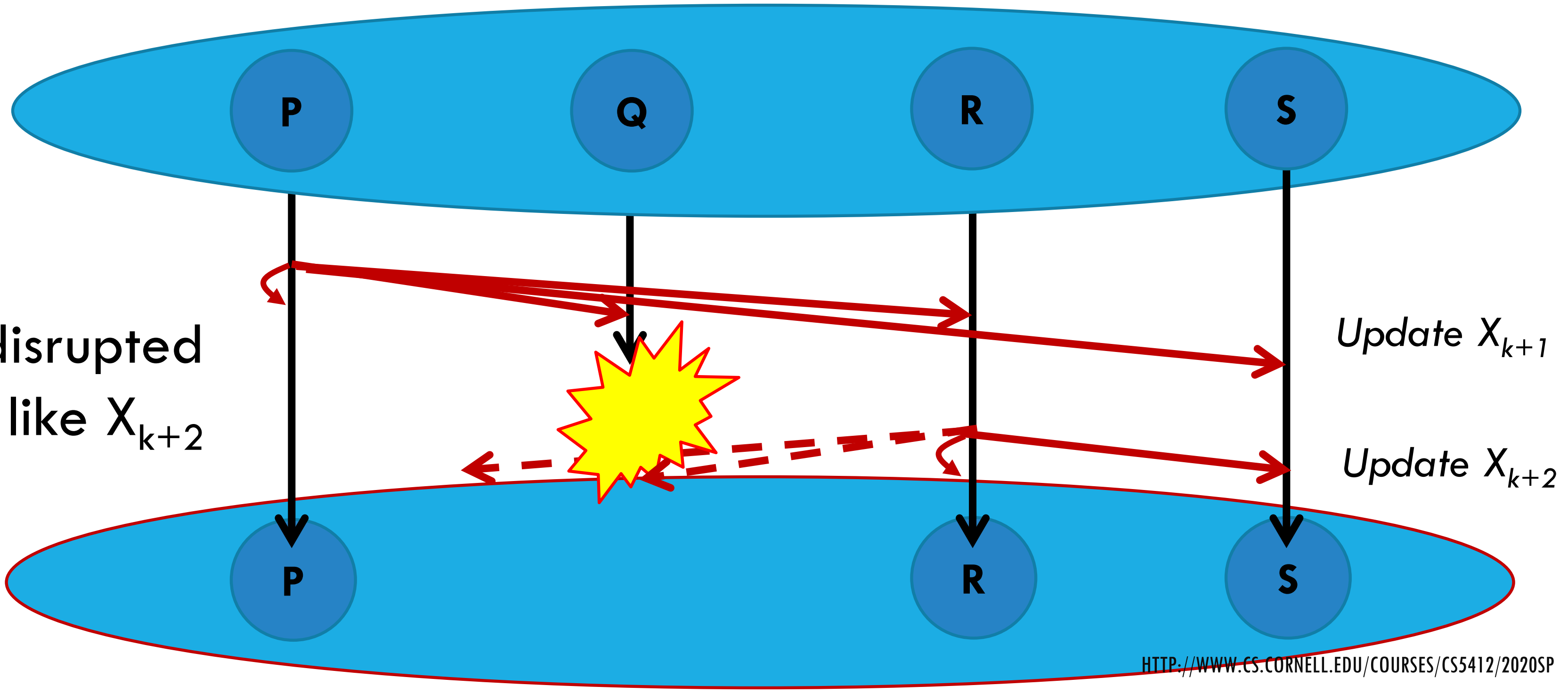
If a multicast was disrupted by failure, it then will be reissued.

A PROCESS FAILS



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

Derecho "trims" disrupted updates, like X_{k+2}



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

We can compare the “basic” RDMC multicast (the one seen earlier) 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)

LARGE MESS

Derecho can make 16 consistent replicas at 2.5x the bandwidth of making one in-core copy

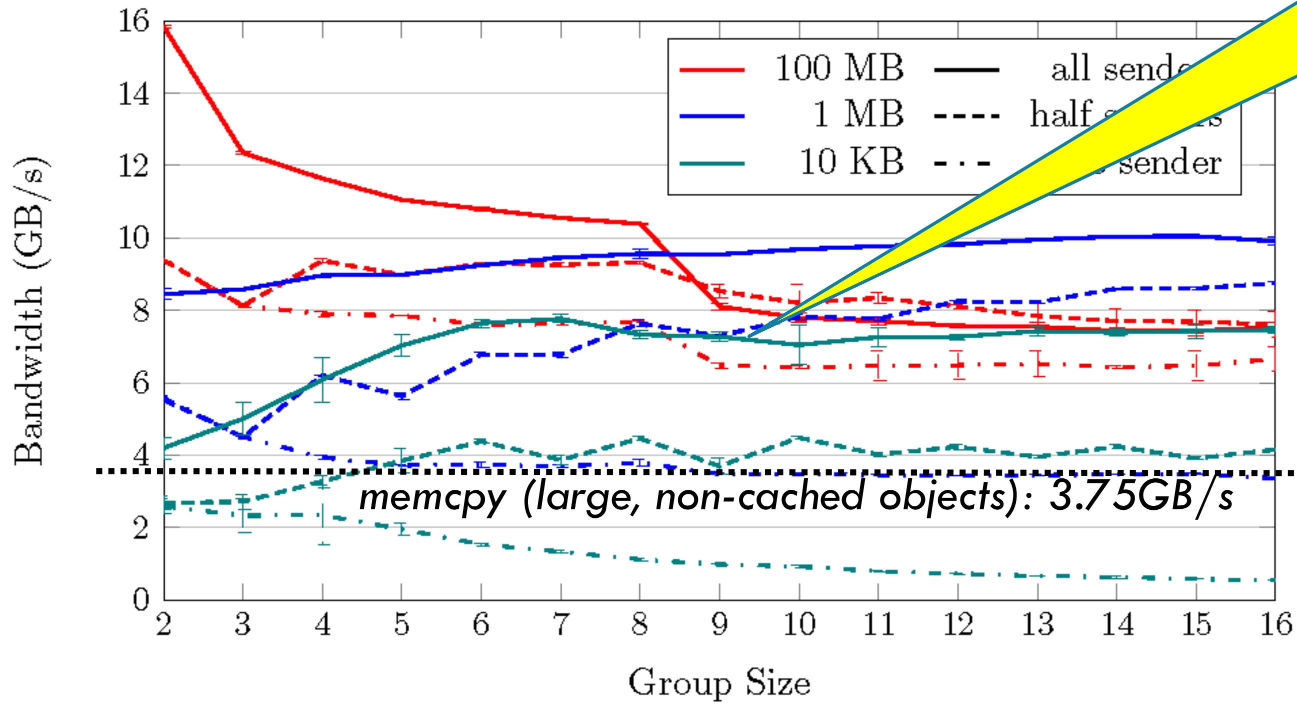
Mellanox 100Gbps RDMA on ROCE (fast Ethernet)

100Gb/s = 12.5GB/s

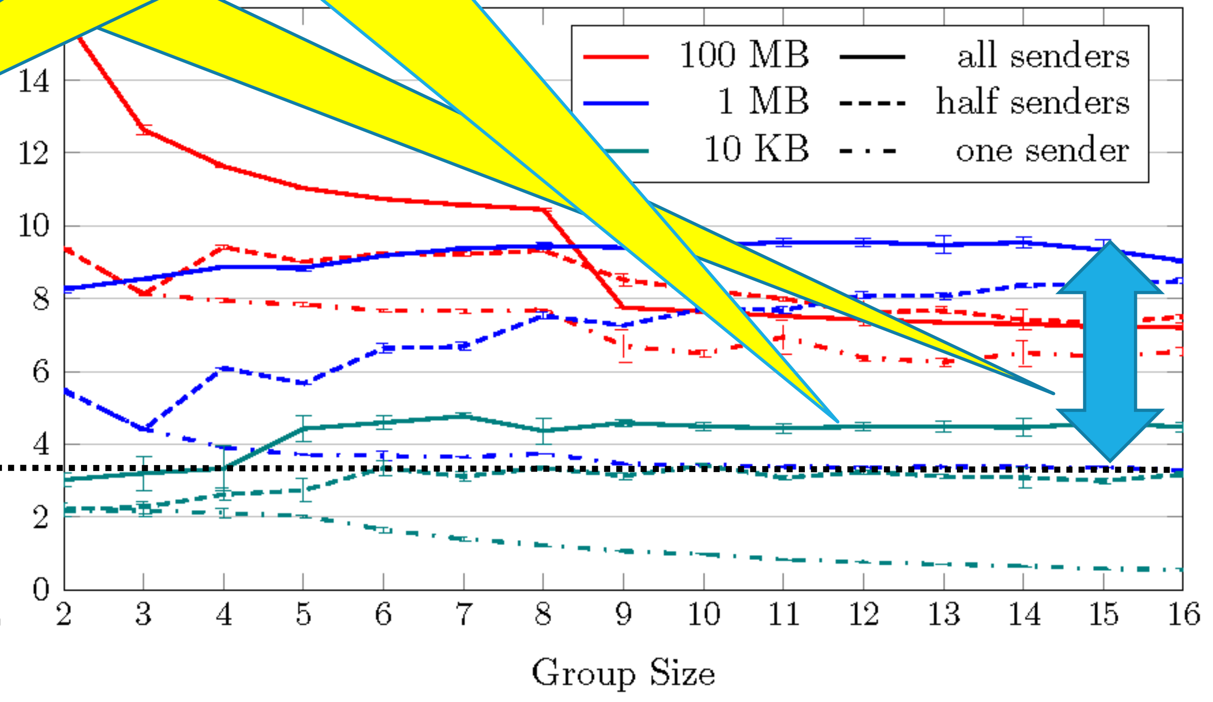
Raw RDMC is faster, but performance loss is small

Raw RDMC multicast via Derecho API

ertical Paxos)

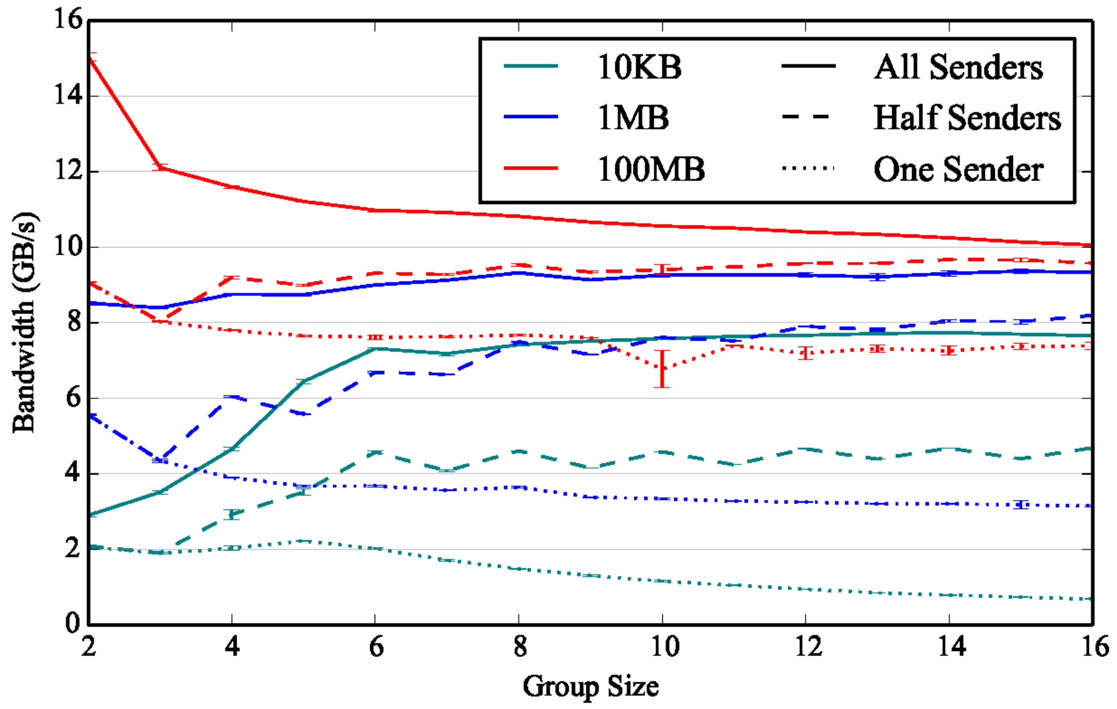


memcpy (large, non-cached objects): 3.75GB/s

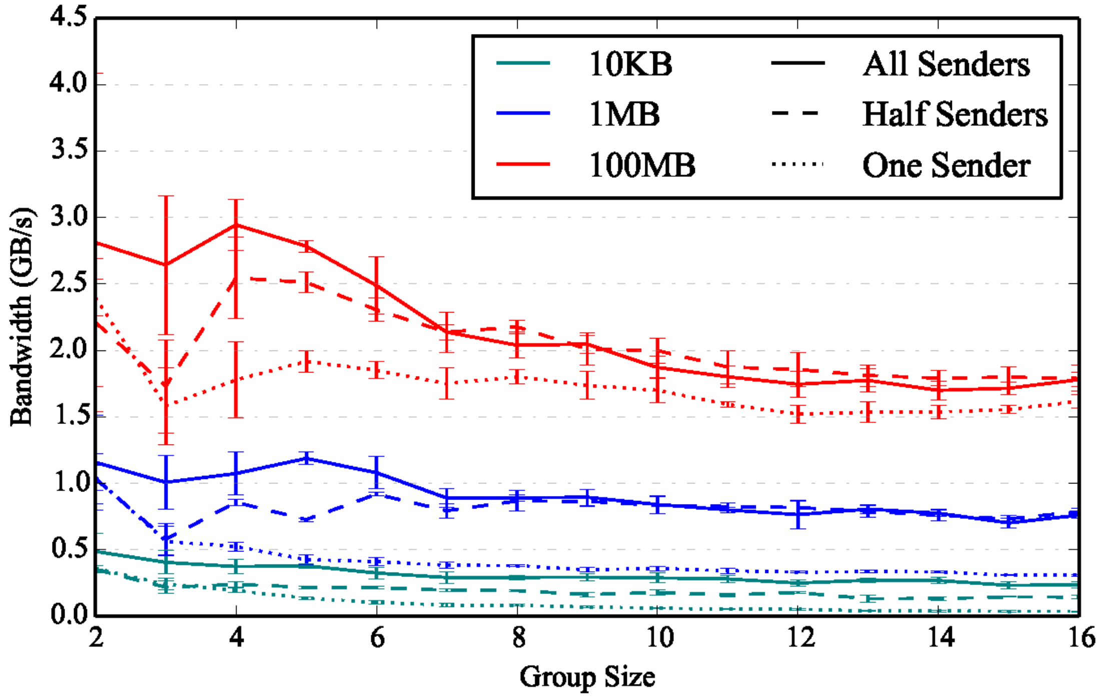


DERECHO CAN ALSO RUN ON TCP. WE FIND THAT RDMA IS 4X FASTER

Derecho Atomic Multicast: 100G RDMA

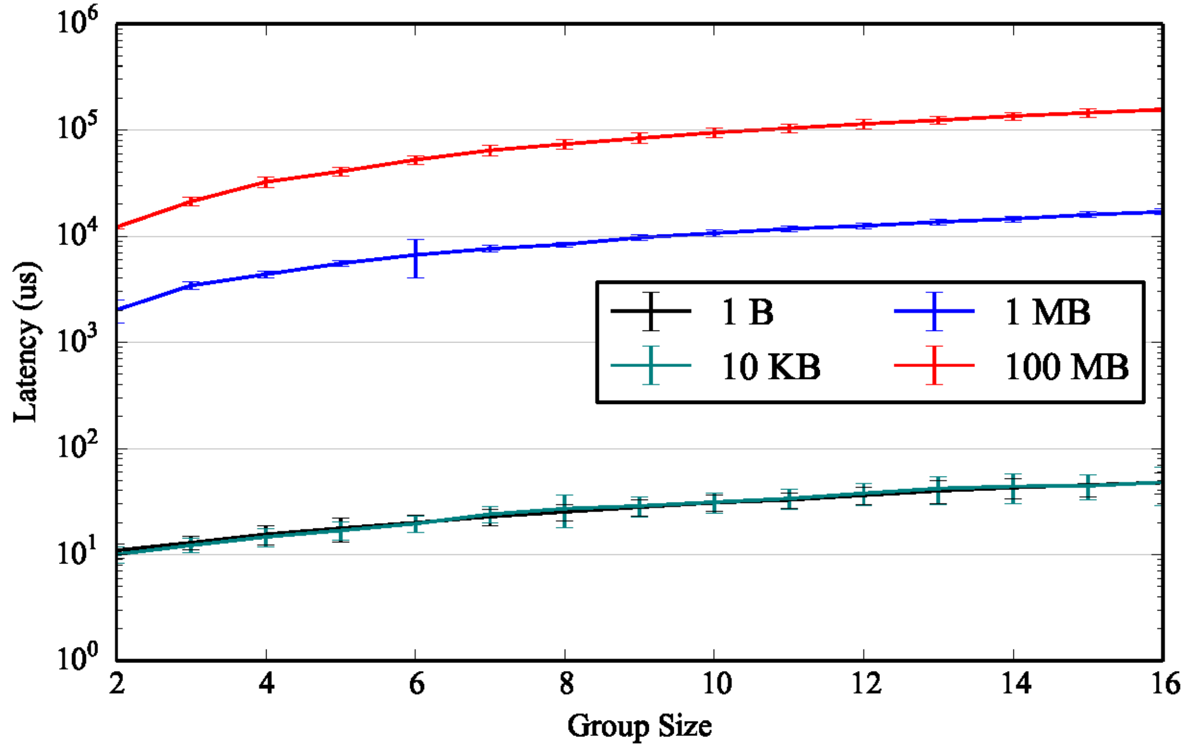


Derecho on TCP, 100G Ethernet

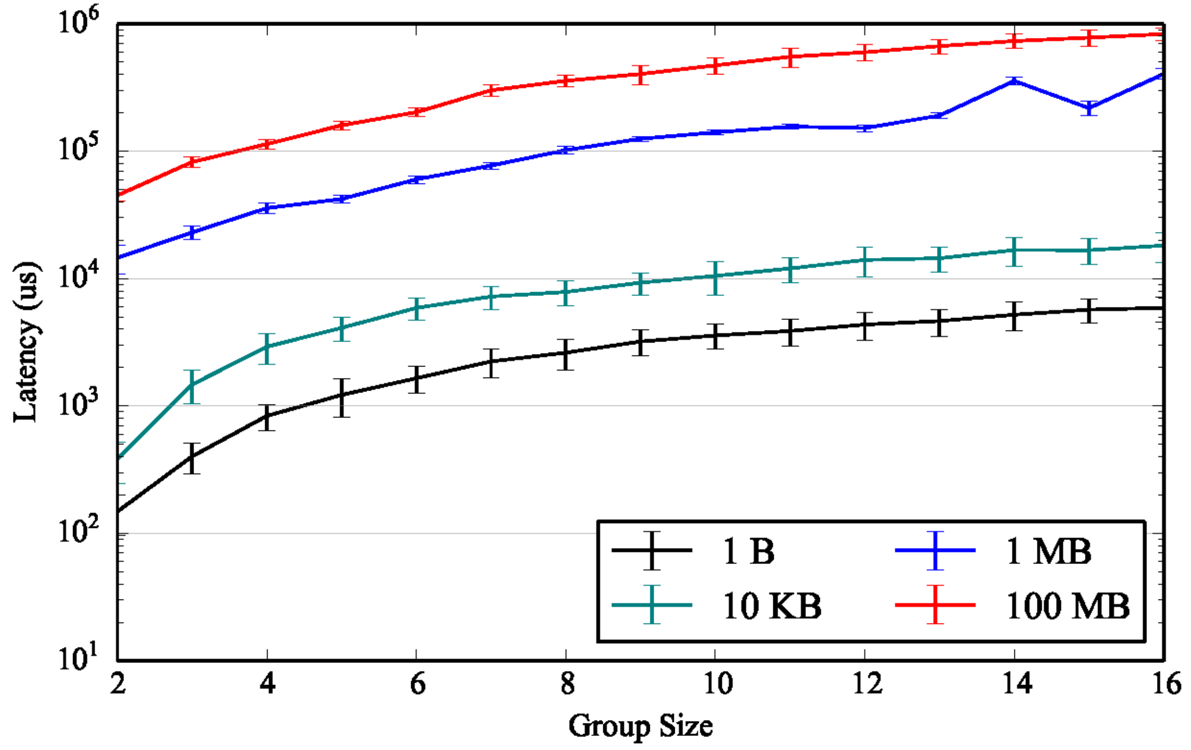


TCP INCREASES DELAYS BY ABOUT 125US

Derecho Atomic Multicast: 100G RDMA



Derecho on TCP, 100G Ethernet



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

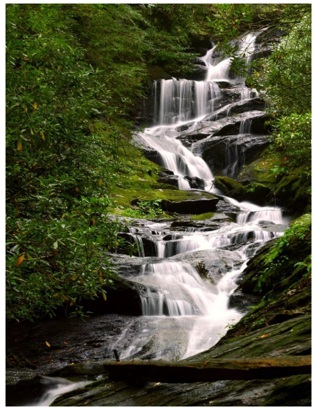
Yet the members of a group don't need to act identically. Tasks can be "subdivided" using ranking or other factors

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!

CASCADE: DERECHO OBJECT STORE



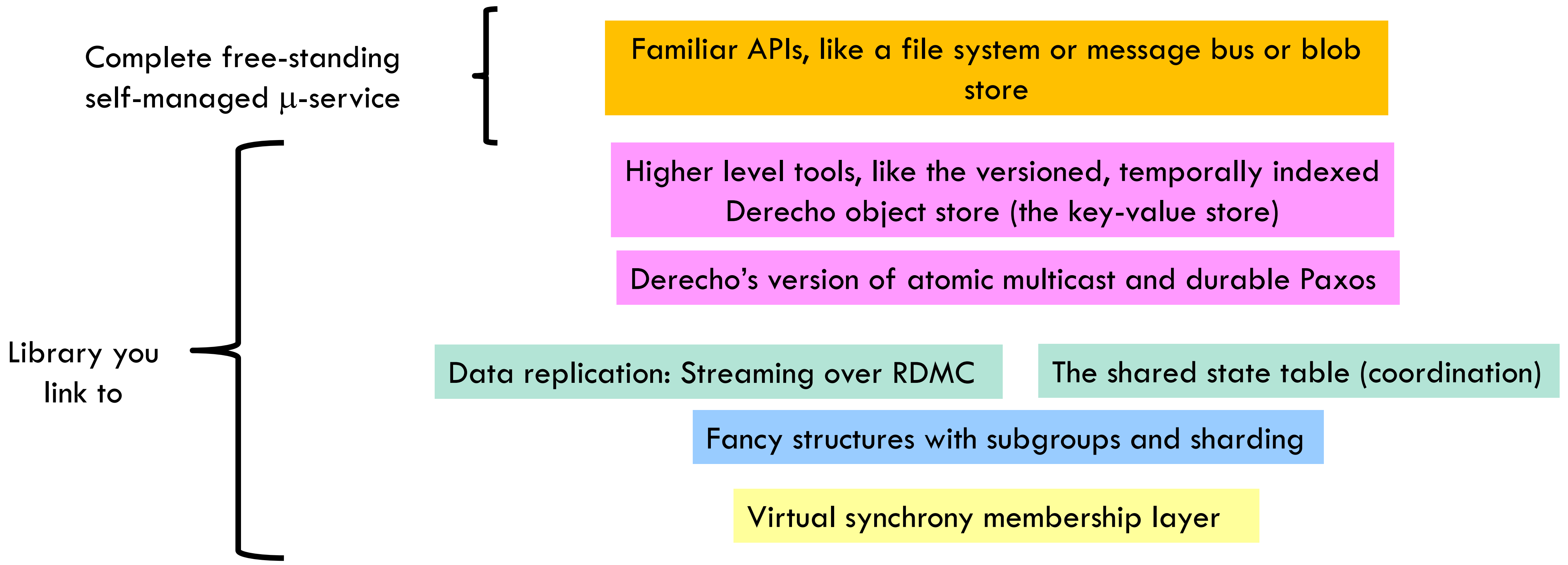
Not everyone finds it easy to use a Paxos or atomic multicast API. DHTs are easier to use: put, get, maybe “scan”. Derecho’s object store is a DHT:

- It offers a (key,value) API with operations like put(k,v), get(k), watch(k).
- Like FFFS, it understands time, and supports put(k,v,**t**) and get(k,**t**).

The object store is a library within a library: it was built on top of Derecho.

- It can be used as a library “within Derecho”,
- Or, you can set it up to run as a μ-service and talk to it from a function in the Azure function server.

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.

CONCLUSIONS?

A software library like Derecho automates many aspects of creating a new μ-service.

The Paxos model is used to ensure consistency, fault-tolerance. There are two cases: ordered multicast (non-durable) and persistent (on disk).

You code in an event-driven style.