CS5412 / LECTURE 10
REPLICATION AND CONSISTENCY

Ken Birman

Spring, 2019

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.

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

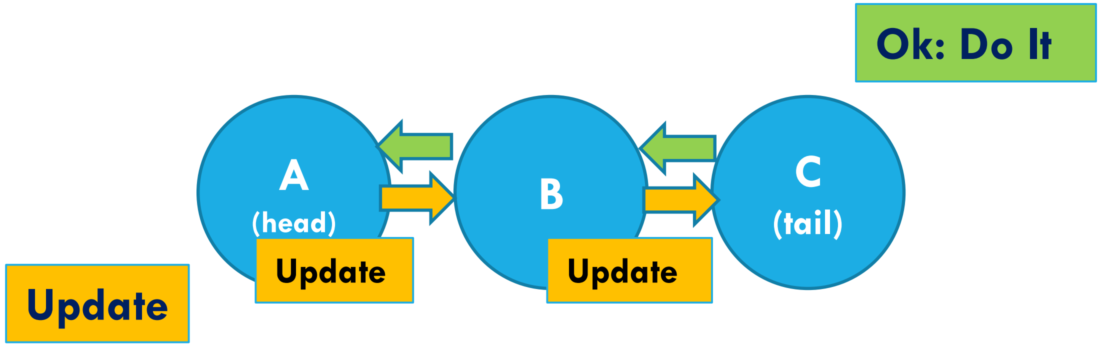
MEMBERSHIP AS A DIMENSION OF CONSISTENCY

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.

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.

COMMON CONCERNS

Where did the group come from? How will chain membership be managed? The model doesn't really 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!

RESTARTING A COMPLEX SERVICE

Imagine that you are writing code that will participate in some form of service that replicates data within a subset (or all) of its members.

How might you go about doing this?

- Perhaps, you could create a file listing members, and each process would add itself to the list? [Note: Zookeeper is often used this way]
- But now the file system is playing the membership tracking role and if the file system fails, or drops an update, or gives out stale data, the solution breaks.

MEMBERSHIP MANAGEMENT "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!

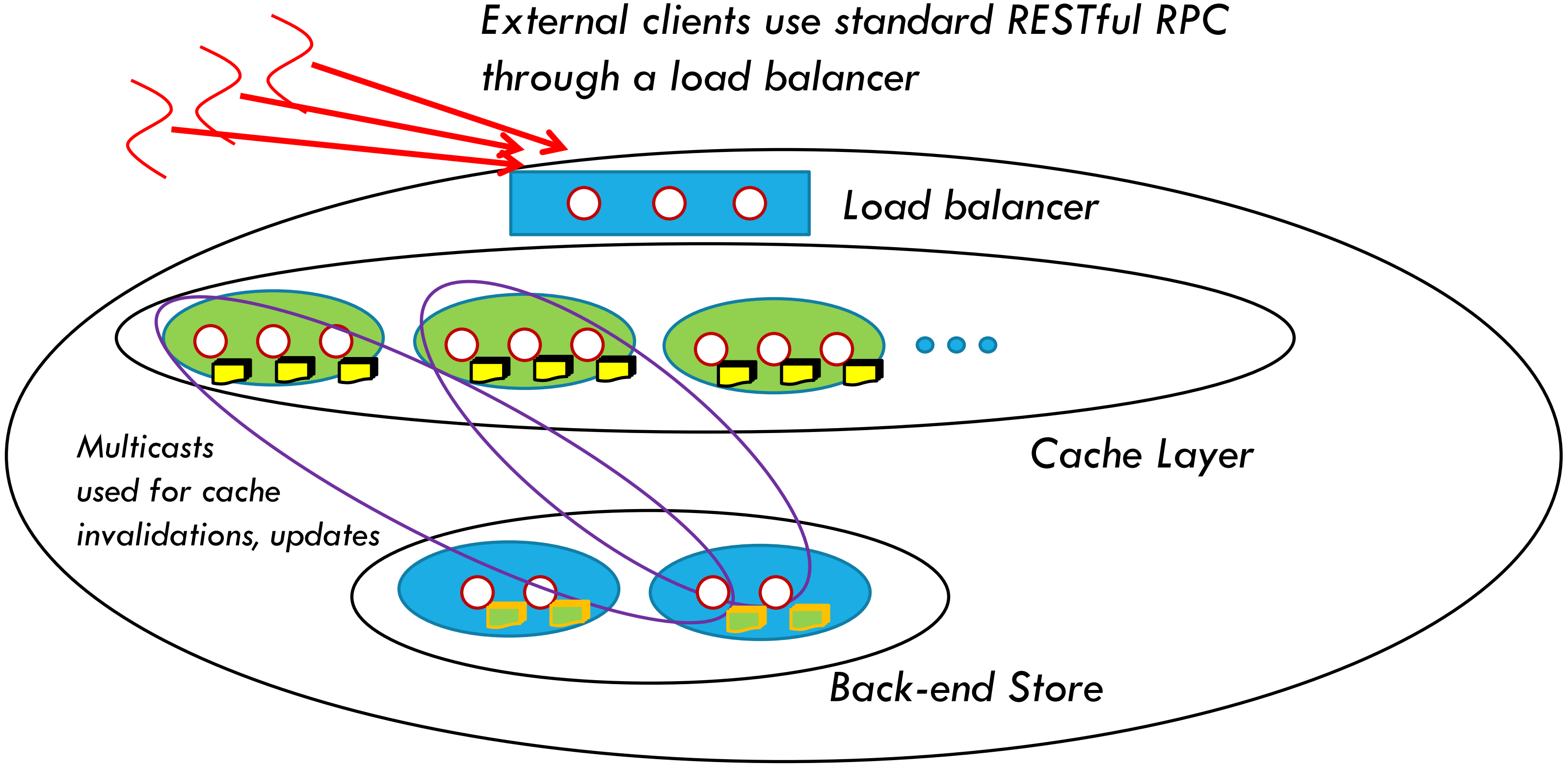
SOLVING THE SPLIT BRAIN PROBLEM

We use a “quorum” approach.

Our system has N processes and only allows progress if more than half are healthy and agree on the next membership view.

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

… YIELDING STRUCTURES LIKE THIS!



External clients use standard RESTful RPC through a load balancer

Load balancer

Cache Layer

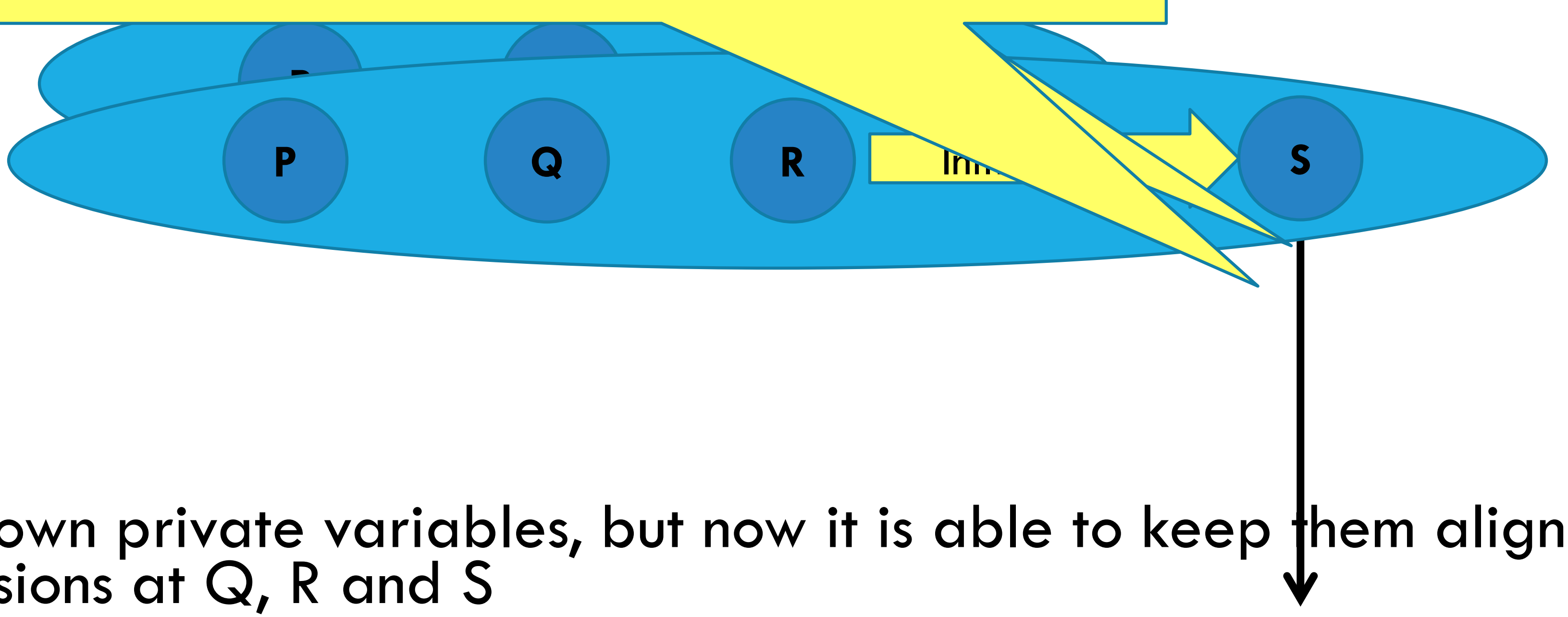
Multicasts used for cache invalidations, updates

Back-end Store

A PROCESS JOINS A GROUP

At first, P ate variables

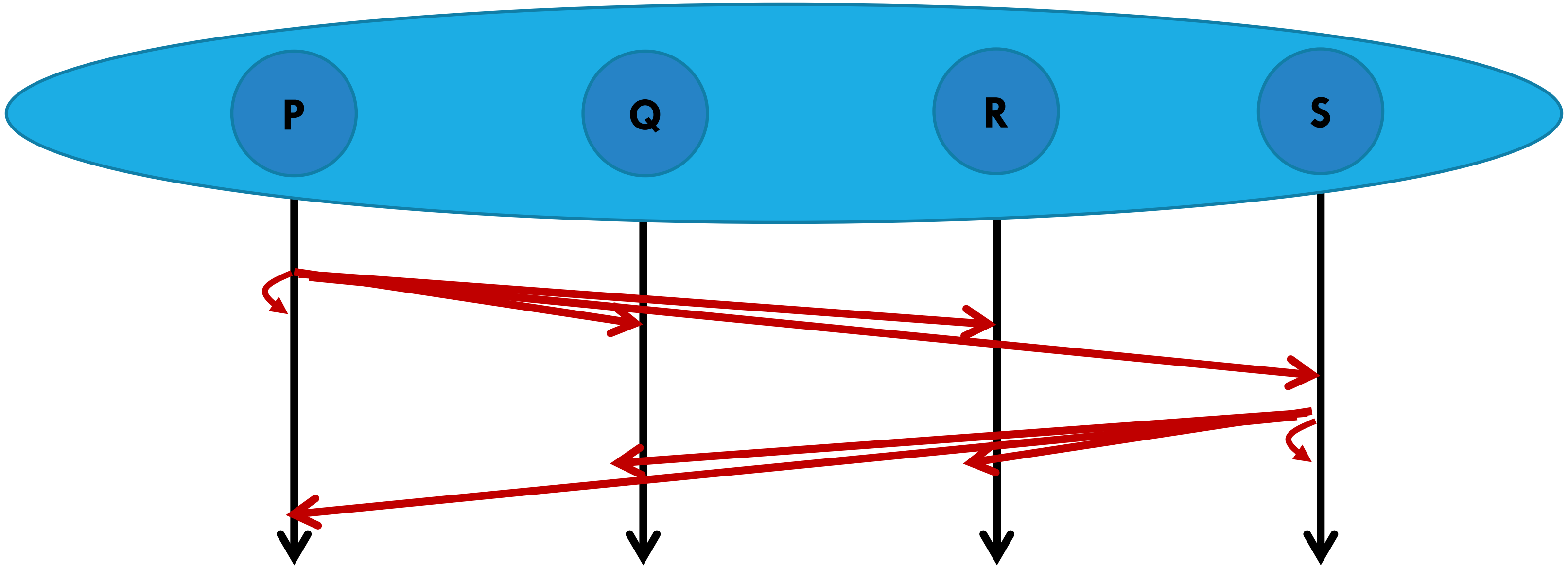
… Automatically transfers state (“sync” of S to P,Q,R)
Now S will receive new updates



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

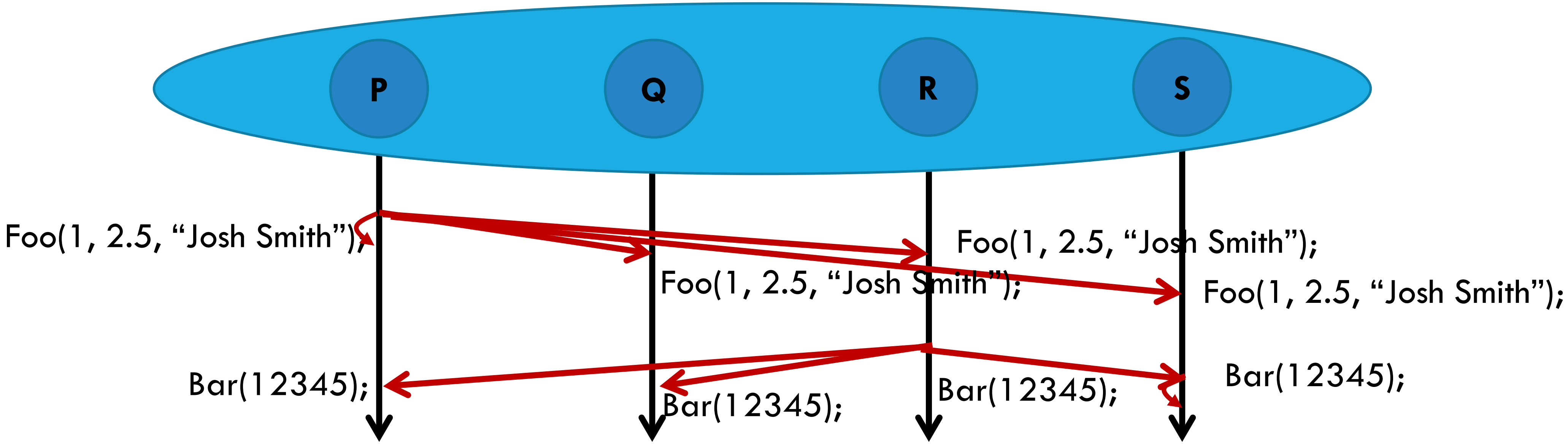
A PROCESS RECEIVING A MULTICAST

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.

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".

Derecho offers "state machine replication" for this purpose. Leslie Lamport was first to propose the model.

Leslie Lamport

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.

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.

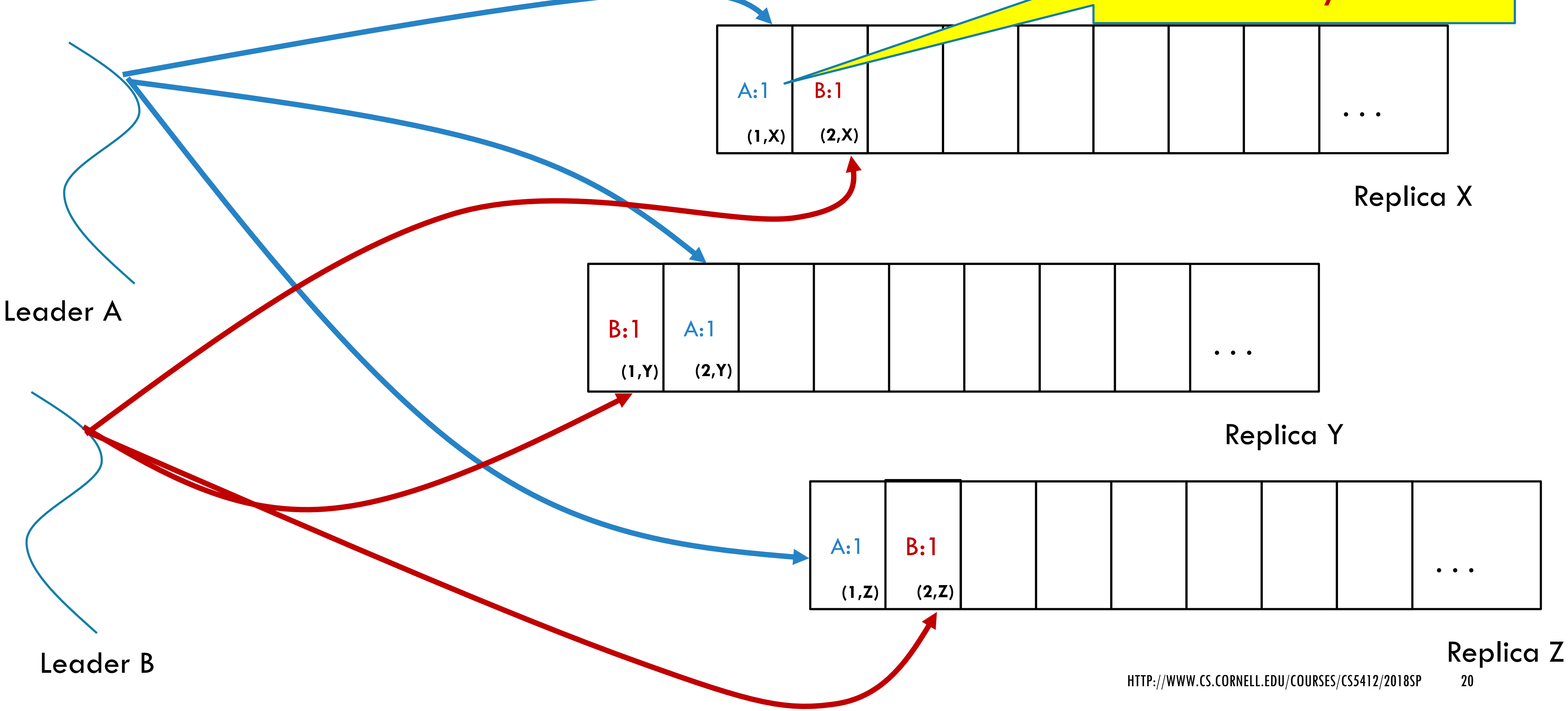
BUILDING AN ORDERED MULTICAST

Leslie proposed several solutions over many years. We'll look at one to get the idea (Derecho uses a much fancier solution).

This is Leslie's very first protocol, and it uses logical clocks.

Assume that membership is fixed and no failures occur

LESLIE'S ORIGINAL PROPOSAL: PRIORITY QUEUES AND LOGICAL CLOCKS



Pending updates occupy a slot but are not yet executed.

LAMPORT'S RULE:

Leader sends proposed message.

Receivers timestamp the message with a logical clock, insert to a priority queue and reply with (timestamp, receiver-id).

For example: A:1 was put into slots {(1,X), (2,Y), (1,Z)}
B:1 was put into slots {(2,X), (1,Y), (2,Z)}

Leaders now compute the maximum by timestamp, breaking ties with ID.

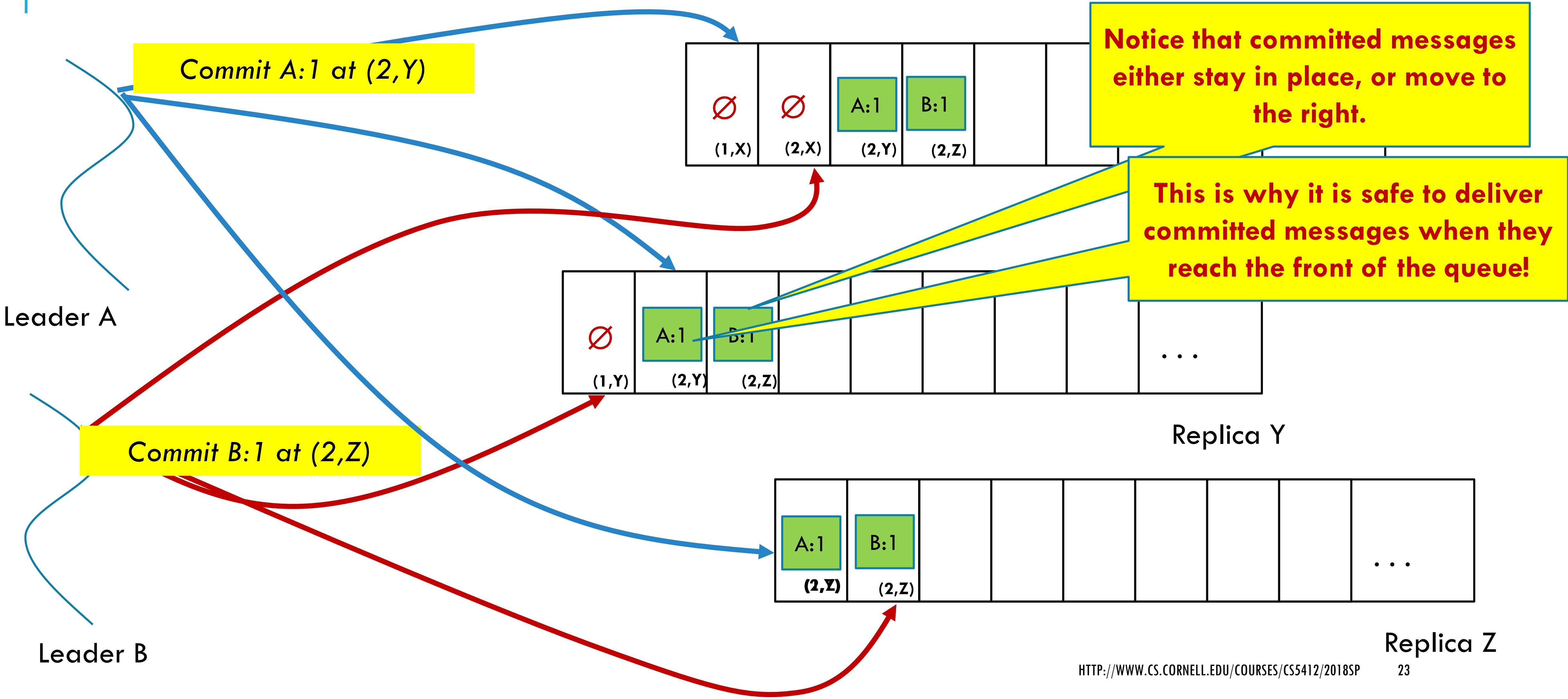
LAMPORT'S PROTOCOL, SECOND PHASE

Now Leaders send the "commit times" they computed

Receivers reorder their priority queues

Receivers deliver *committed messages, from the front of the queue*

LESLIE'S ORIGINAL PROPOSAL: PRIORITY QUEUES AND LOGICAL CLOCKS



Notice that committed messages either stay in place, or move to the right.

This is why it is safe to deliver committed messages when they reach the front of the queue!

IS THIS A "GOOD" SMR PROTOCOL?

It isn't dreadful. In fact the messages are delivered along consistent cuts!

But it can't handle crashes if the state will be durable (on disk). And it doesn't use modern networking hardware very well.

Also, adding logic to handle membership changes is tricky. Derecho uses an approach called "virtual synchrony" for membership changes.

DURABLE STATE: PAXOS CONCEPT

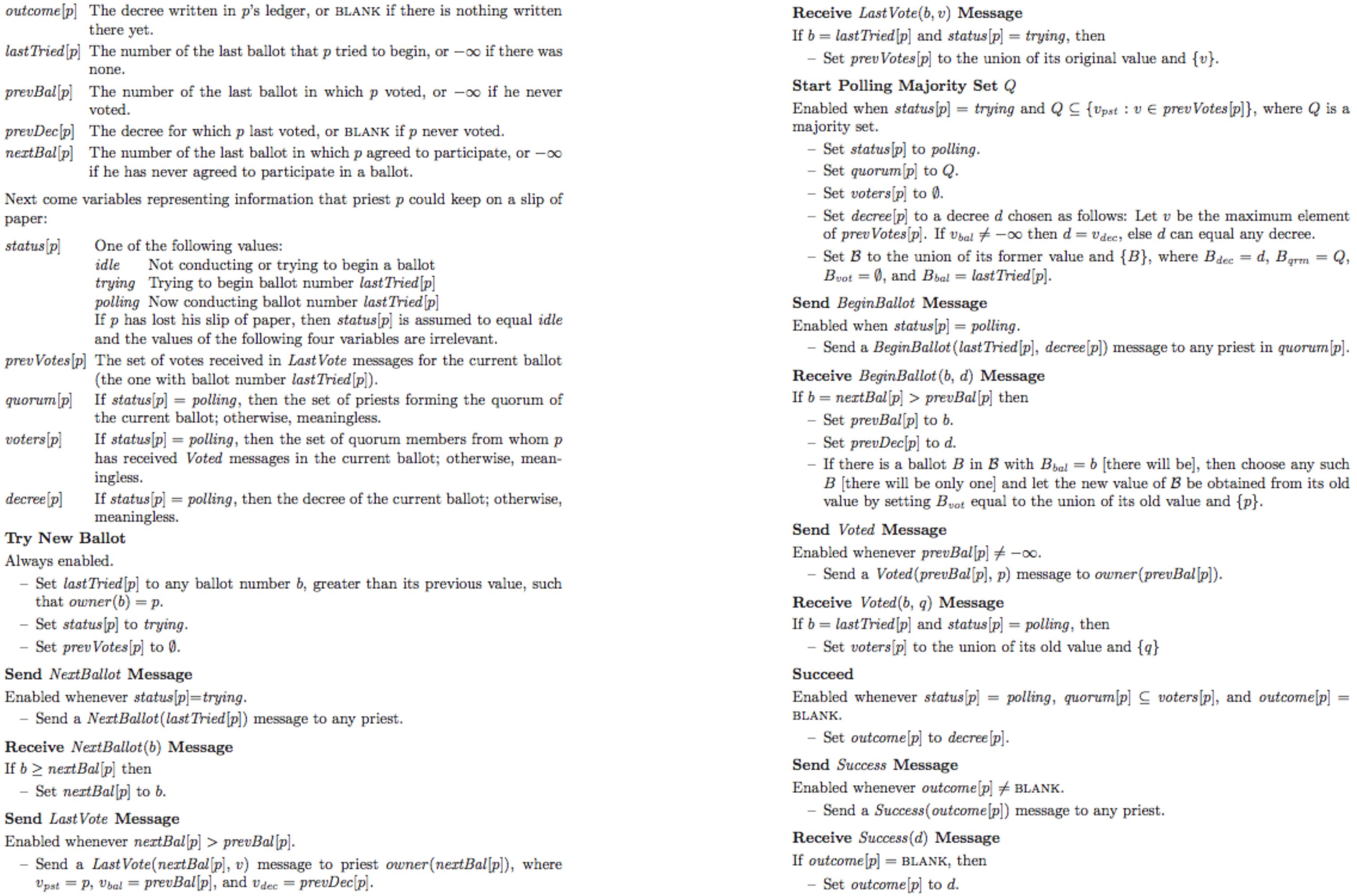
Paxos (Greek Island)

Our SMR protocol puts messages into identical order ("total order") but doesn't address logging them to disk or cleanup during recovery.

Paxos is the name of a collection of protocols that Lamport created to solve ordering, durability and "non-triviality" all at once. Our SMR protocol actually can be turned into a version of Paxos.

We say "version" because there are many ways to implementPaxos.

ACTUAL PAXOS PROTOCOL: VERY COMPLEX



outcome[p] The decree written in p's ledger, or BLANK if there is nothing written there yet.

lastTried[p] The number of the last ballot that p tried to begin, or $-\infty$ if there was none.

prevBal[p] The number of the last ballot in which p voted, or $-\infty$ if he never voted.

prevDec[p] The decree for which p last voted, or BLANK if p never voted.

nextBal[p] The number of the last ballot in which p agreed to participate, or $-\infty$ if he has never agreed to participate in a ballot.

Next come variables representing information that priest p could keep on a slip of paper:

status[p] One of the following values:
idle Not conducting or trying to begin a ballot
trying Trying to begin ballot number *lastTried*[p]
polling Now conducting ballot number *lastTried*[p]
If p has lost his slip of paper, then *status*[p] is assumed to equal *idle* and the values of the following four variables are irrelevant.

prevVotes[p] The set of votes received in *LastVote* messages for the current ballot (the one with ballot number *lastTried*[p]).

quorum[p] If $\mathit{status}[p] = \mathit{polling}$, then the set of priests forming the quorum of the current ballot; otherwise, meaningless.

voters[p] If $\mathit{status}[p] = \mathit{polling}$, then the set of quorum members from whom p has received *Voted* messages in the current ballot; otherwise, meaningless.

decree[p] If $\mathit{status}[p] = \mathit{polling}$, then the decree of the current ballot; otherwise, meaningless.

Try New Ballot

Always enabled.

- Set *lastTried*[p] to any ballot number b, greater than its previous value, such that $\mathit{owner}(b) = p$.
- Set *status*[p] to *trying*.
- Set *prevVotes*[p] to $\emptyset$.

Send *NextBallot* Message

Enabled whenever $\mathit{status}[p] = \mathit{trying}$.

- Send a *NextBallot*(*lastTried*[p]) message to any priest.

Receive *NextBallot*(b) Message

If $b \geq \mathit{nextBal}[p]$ then

- Set *nextBal*[p] to b.

Send *LastVote* Message

Enabled whenever $\mathit{nextBal}[p] > \mathit{prevBal}[p]$.

- Send a $\mathit{LastVote}(\mathit{nextBal}[p], v)$ message to priest $\mathit{owner}(\mathit{nextBal}[p])$, where $v_{pst} = p$, $v_{bal} = \mathit{prevBal}[p]$, and $v_{dec} = \mathit{prevDec}[p]$.

Receive *LastVote*(b, v) Message

If $b = \mathit{lastTried}[p]$ and $\mathit{status}[p] = \mathit{trying}$, then

- Set *prevVotes*[p] to the union of its original value and $\{v\}$.

Start Polling Majority Set Q

Enabled when $\mathit{status}[p] = \mathit{trying}$ and $Q \subseteq \{v_{pst} : v \in \mathit{prevVotes}[p]\}$, where Q is a majority set.

- Set *status*[p] to *polling*.
- Set *quorum*[p] to Q.
- Set *voters*[p] to $\emptyset$.
- Set *decree*[p] to a decree d chosen as follows: Let v be the maximum element of *prevVotes*[p]. If $v_{bal} \neq -\infty$ then $d = v_{dec}$, else d can equal any decree.
- Set $\mathcal{B}$ to the union of its former value and $\{B\}$, where $B_{dec} = d$, $B_{qrm} = Q$, $B_{vot} = \emptyset$, and $B_{bal} = \mathit{lastTried}[p]$.

Send *BeginBallot* Message

Enabled when $\mathit{status}[p] = \mathit{polling}$.

- Send a $\mathit{BeginBallot}(\mathit{lastTried}[p], \mathit{decree}[p])$ message to any priest in *quorum*[p].

Receive *BeginBallot*(b, d) Message

If $b = \mathit{nextBal}[p] > \mathit{prevBal}[p]$ then

- Set *prevBal*[p] to b.
- Set *prevDec*[p] to d.
- If there is a ballot B in $\mathcal{B}$ with $B_{bal} = b$ [there will be], then choose any such B [there will be only one] and let the new value of $\mathcal{B}$ be obtained from its old value by setting B_{vot} equal to the union of its old value and $\{p\}$.

Send *Voted* Message

Enabled whenever $\mathit{prevBal}[p] \neq -\infty$.

- Send a $\mathit{Voted}(\mathit{prevBal}[p], p)$ message to $\mathit{owner}(\mathit{prevBal}[p])$.

Receive *Voted*(b, q) Message

If $b = \mathit{lastTried}[p]$ and $\mathit{status}[p] = \mathit{polling}$, then

- Set *voters*[p] to the union of its old value and $\{q\}$

Succeed

Enabled whenever $\mathit{status}[p] = \mathit{polling}$, $\mathit{quorum}[p] \subseteq \mathit{voters}[p]$, and $\mathit{outcome}[p] =$ BLANK.

- Set *outcome*[p] to *decree*[p].

Send *Success* Message

Enabled whenever $\mathit{outcome}[p] \neq$ BLANK.

- Send a *Success*(*outcome*[p]) message to any priest.

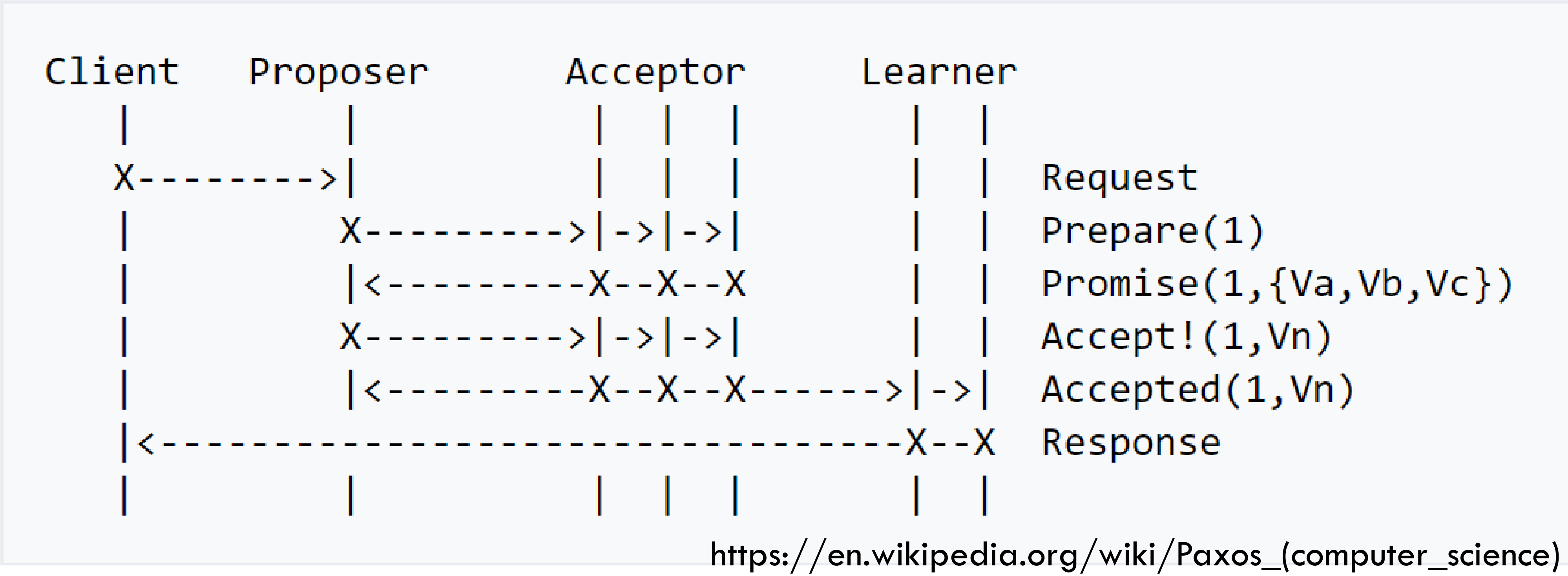
Receive *Success*(d) Message

If $\mathit{outcome}[p] =$ BLANK, then

- Set *outcome*[p] to d.

Paxos drilldown: If time permits (but we probably won't cover this slide)

PAXOS MESSAGE FLOW

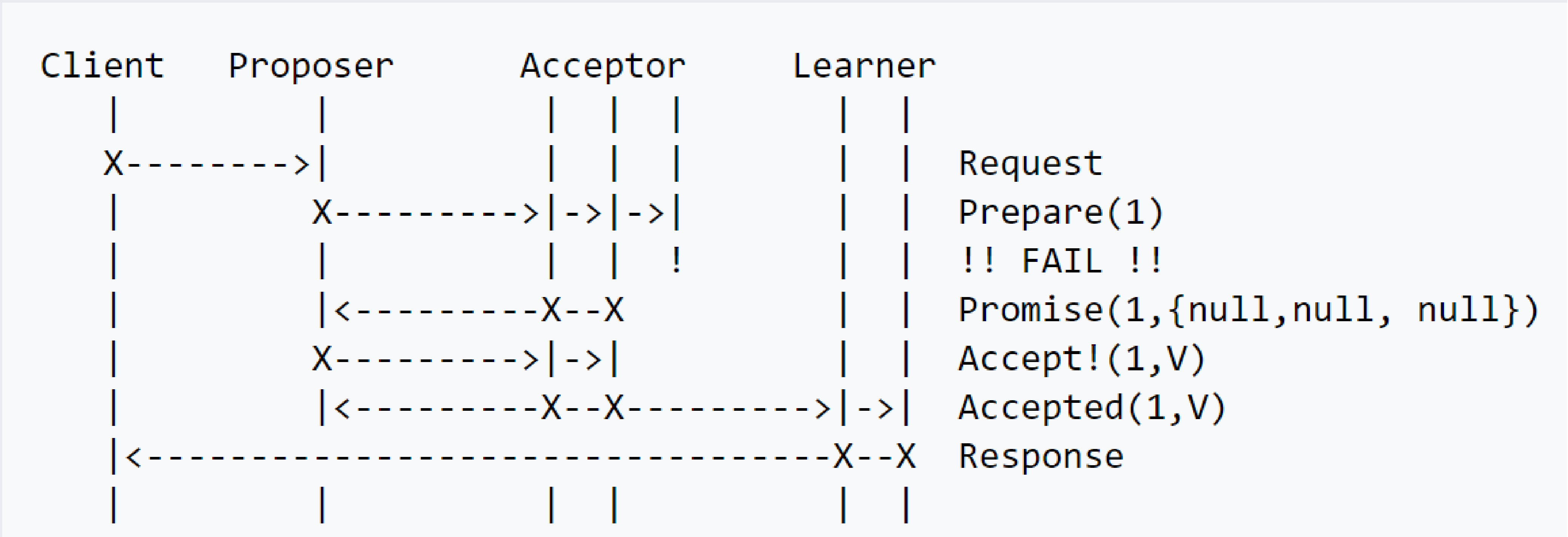


```
Client   Proposer      Acceptor     Learner
   |         |          |  |  |       |  |
   X-------->|          |  |  |       |  |  Request
   |         X--------->|->|->|       |  |  Prepare(1)
   |         |<---------X--X--X       |  |  Promise(1,{Va,Vb,Vc})
   |         X--------->|->|->|       |  |  Accept!(1,Vn)
   |         |<---------X--X--X------>|->|  Accepted(1,Vn)
   |<---------------------------------X--X  Response
   |         |          |  |  |       |  |
```

https://en.wikipedia.org/wiki/Paxos_(computer_science)

Paxos drilldown: If time permits (but we probably won't cover this slide)

MESSAGE FLOW: FAILURE OF ACCEPTOR

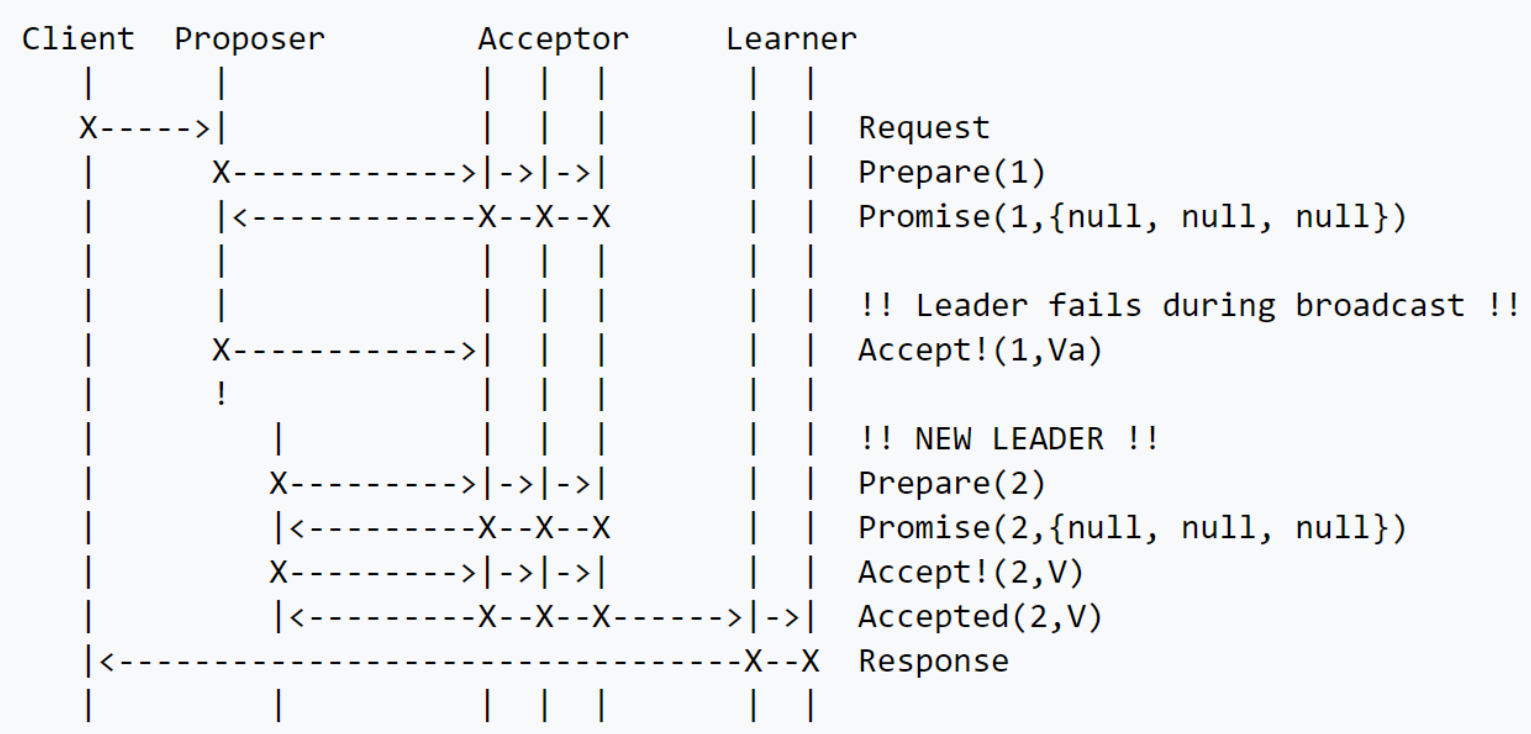


```
Client   Proposer      Acceptor     Learner
   |         |          |  |  |       |  |
   X-------->|          |  |  |       |  |  Request
   |         X--------->|->|->|       |  |  Prepare(1)
   |         |          |  |  !       |  |  !! FAIL !!
   |         |<---------X--X          |  |  Promise(1,{null,null, null})
   |         X--------->|->|          |  |  Accept!(1,V)
   |         |<---------X--X--------->|->|  Accepted(1,V)
   |<---------------------------------X--X  Response
   |         |          |  |          |  |
```

https://en.wikipedia.org/wiki/Paxos_(computer_science)

Paxos drilldown: If time permits (but we probably won't cover this slide)

MESSAGE FLOW: FAILURE OF PROPOSER

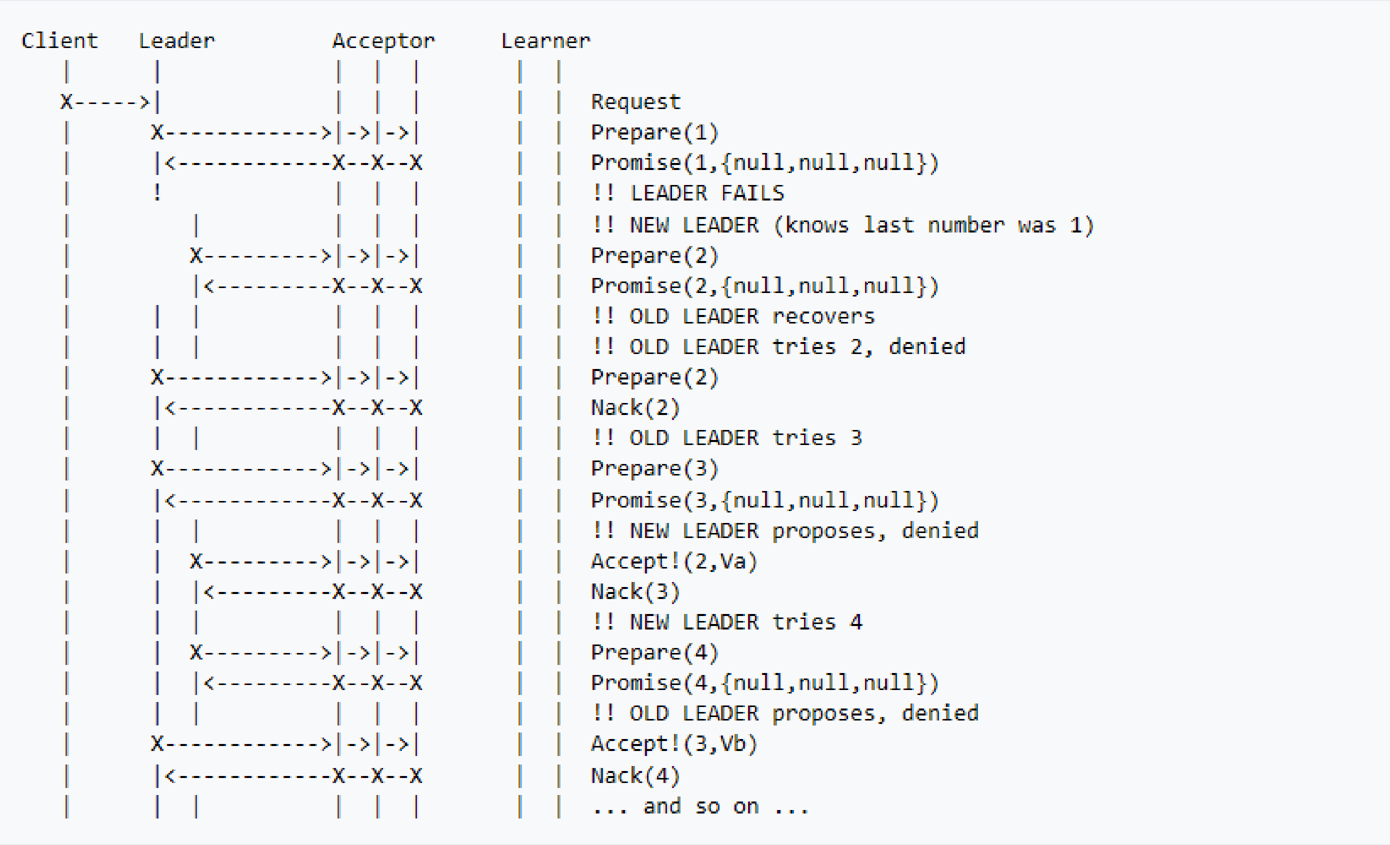


```
Client  Proposer        Acceptor     Learner
   |      |             |  |  |       |  |
   X----->|             |  |  |       |  |  Request
   |      X------------>|->|->|       |  |  Prepare(1)
   |      |<------------X--X--X       |  |  Promise(1,{null, null, null})
   |      |             |  |  |       |  |
   |      |             |  |  |       |  |  !! Leader fails during broadcast !!
   |      X------------>|  |  |       |  |  Accept!(1,Va)
   |      !             |  |  |       |  |
   |         |          |  |  |       |  |  !! NEW LEADER !!
   |         X--------->|->|->|       |  |  Prepare(2)
   |         |<---------X--X--X       |  |  Promise(2,{null, null, null})
   |         X--------->|->|->|       |  |  Accept!(2,V)
   |         |<---------X--X--X------>|->|  Accepted(2,V)
   |<---------------------------------X--X  Response
   |         |          |  |  |       |  |
```

https://en.wikipedia.org/wiki/Paxos_(computer_science)

Paxos drilldown: If time permits (but we probably won't cover this slide)

MESSAGE FLOW: 2 COMPETING PROPOSALS



```
Client   Leader         Acceptor     Learner
   |      |             |  |  |       |  |
   X----->|             |  |  |       |  |  Request
   |      X------------>|->|->|       |  |  Prepare(1)
   |      |<------------X--X--X       |  |  Promise(1,{null,null,null})
   |      !             |  |  |       |  |  !! LEADER FAILS
   |         |          |  |  |       |  |  !! NEW LEADER (knows last number was 1)
   |         X--------->|->|->|       |  |  Prepare(2)
   |         |<---------X--X--X       |  |  Promise(2,{null,null,null})
   |      |  |          |  |  |       |  |  !! OLD LEADER recovers
   |      |  |          |  |  |       |  |  !! OLD LEADER tries 2, denied
   |      X------------>|->|->|       |  |  Prepare(2)
   |      |<------------X--X--X       |  |  Nack(2)
   |      |  |          |  |  |       |  |  !! OLD LEADER tries 3
   |      X------------>|->|->|       |  |  Prepare(3)
   |      |<------------X--X--X       |  |  Promise(3,{null,null,null})
   |      |  |          |  |  |       |  |  !! NEW LEADER proposes, denied
   |      |  X--------->|->|->|       |  |  Accept!(2,Va)
   |      |  |<---------X--X--X       |  |  Nack(3)
   |      |  |          |  |  |       |  |  !! NEW LEADER tries 4
   |      |  X--------->|->|->|       |  |  Prepare(4)
   |      |  |<---------X--X--X       |  |  Promise(4,{null,null,null})
   |      |  |          |  |  |       |  |  !! OLD LEADER proposes, denied
   |      X------------>|->|->|       |  |  Accept!(3,Vb)
   |      |<------------X--X--X       |  |  Nack(4)
   |      |  |          |  |  |       |  |  ... and so on ...
```

Paxos drilldown: If time permits (but we probably won't cover this slide)

WHAT MAKES PAXOS COMPLICATED?

In some sense, the protocol is dealing with many issues all at once.

It has no agreed-upon "current membership" (although it does have a static list of members, some of which might currently be unavailable).

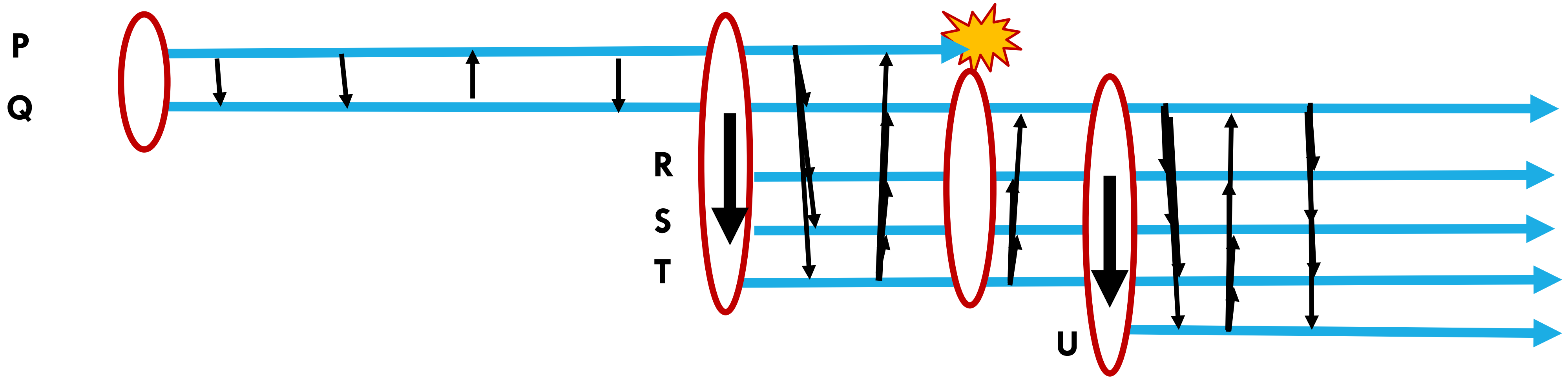
To compensate for not knowing which are up, it uses a competition to get a quorum of "acceptors" to agree on each update, and this is messy.

Tracking *membership* at a more basic level simplifies the logic dramatically!

Paxos drilldown: If time permits (but we probably won't cover this slide)

VIRTUAL SYNCHRONY: MANAGED GROUPS

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



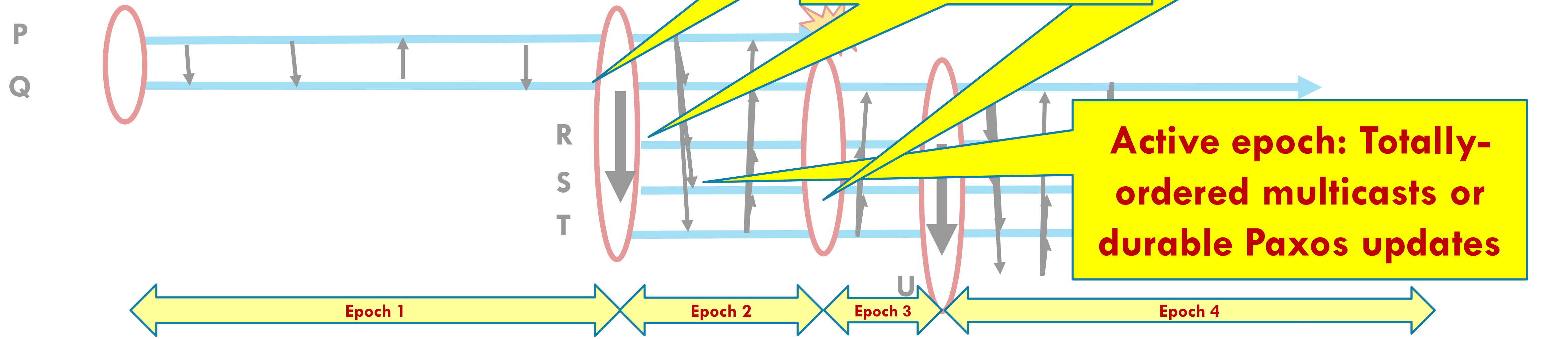
Joins, failures are “clean”, state is transferred to joining members

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

VIRTUAL SYNCHRONY: MA S

Epoch Termination

Epoch: A period from one membersh p e.



Epoch Termination

State Transfer

Active epoch: Totally-ordered multicasts or durable Paxos updates

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

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

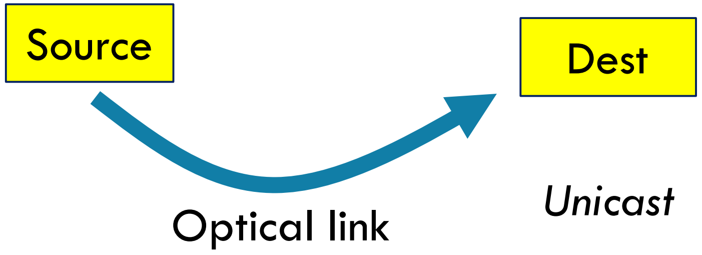
DERECHO'S VERSION OF PAXOS

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.

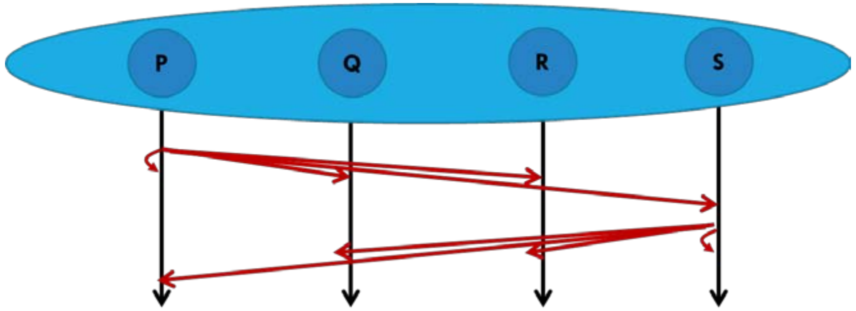
RDMA can actually transfer data to a remote machine faster than a local machine can do local copying.

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

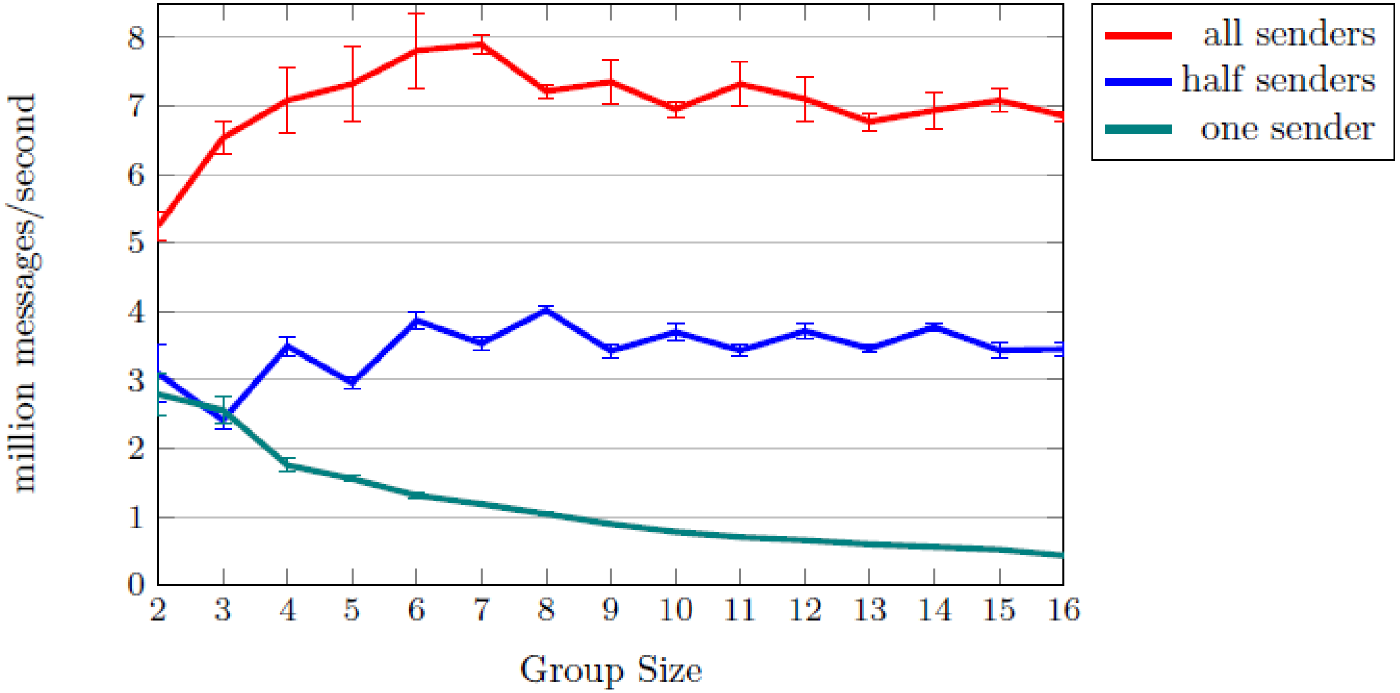
SMALL MESSAGES USE A DIRECT RDMA COPYING PROTOCOL WE CALL SMC.

Mellanox 100Gbps RDMA on ROCE (fast Ethernet)

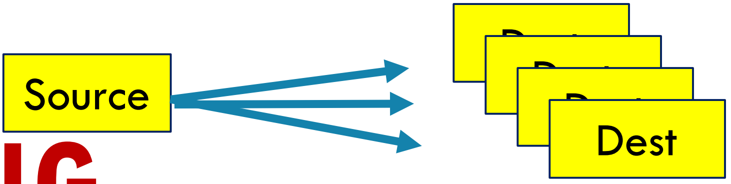
100Gb/s = 12.5GB/s



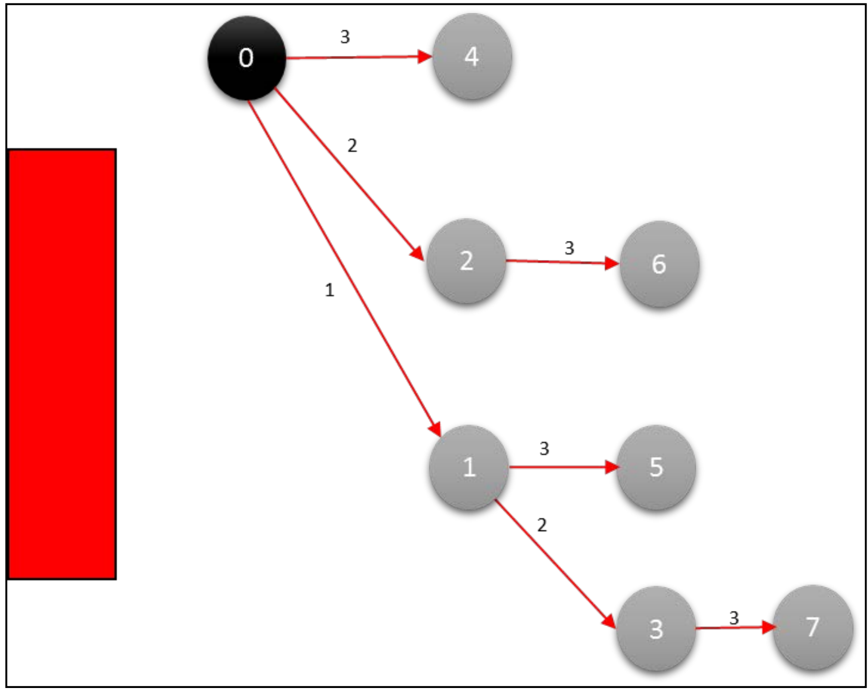
SMC Protocol, 1 byte messages



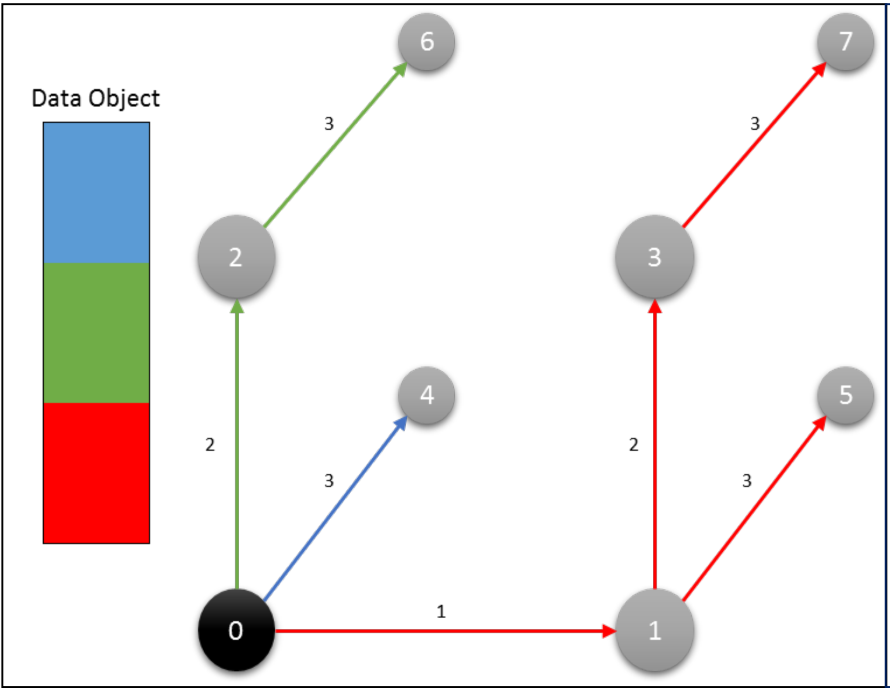
LARGE MESSAGES USE A RELAYING METHOD WE CALL RDMC



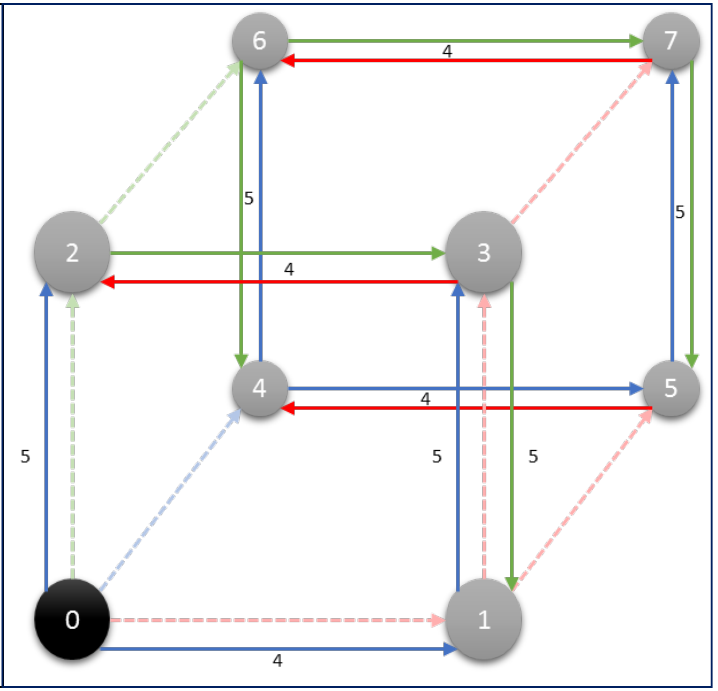
Multicast



Binomial Tree

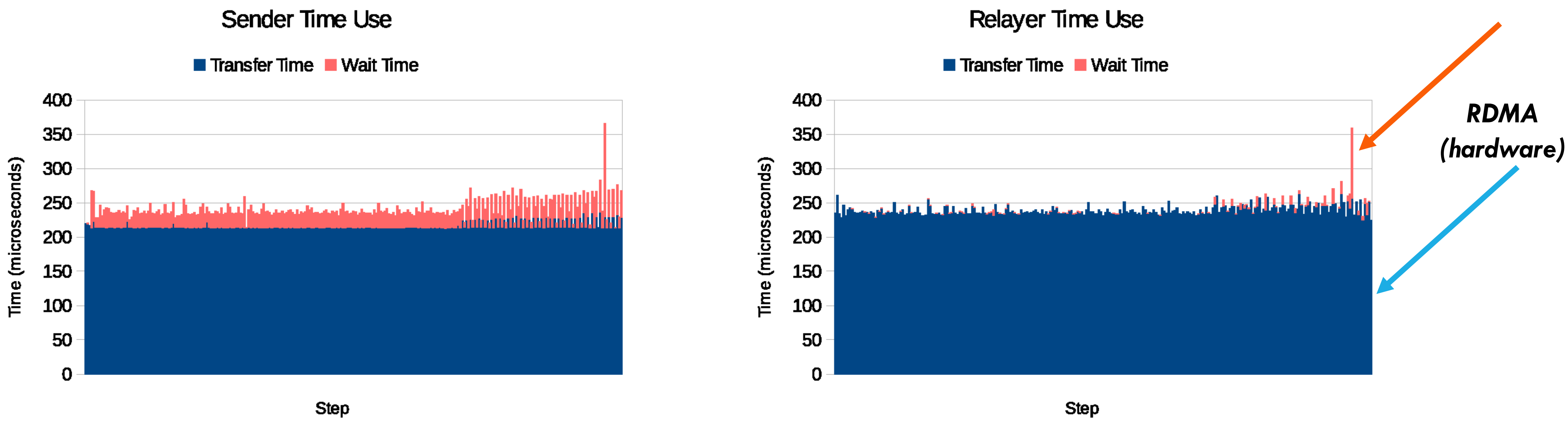


Binomial Pipeline



Final Step

RDMC SUCCEEDS IN OFFLOADING WORK TO HARDWARE



RDMC (software)

RDMA (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?

Recall that in virtual synchrony we know the membership for each epoch.

Derecho also knows which members are “senders”. The application tells it.

Within the senders, Derecho just uses round-robin order: message 1 from P. Message 1 from Q. Message 1 from R. Now message 2 from P…

If some process has nothing to send it can “pass” (it sends a *null* message).

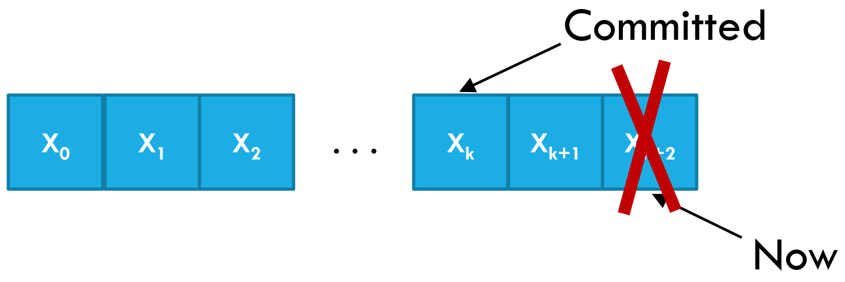
ARE WE FINISHED?

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

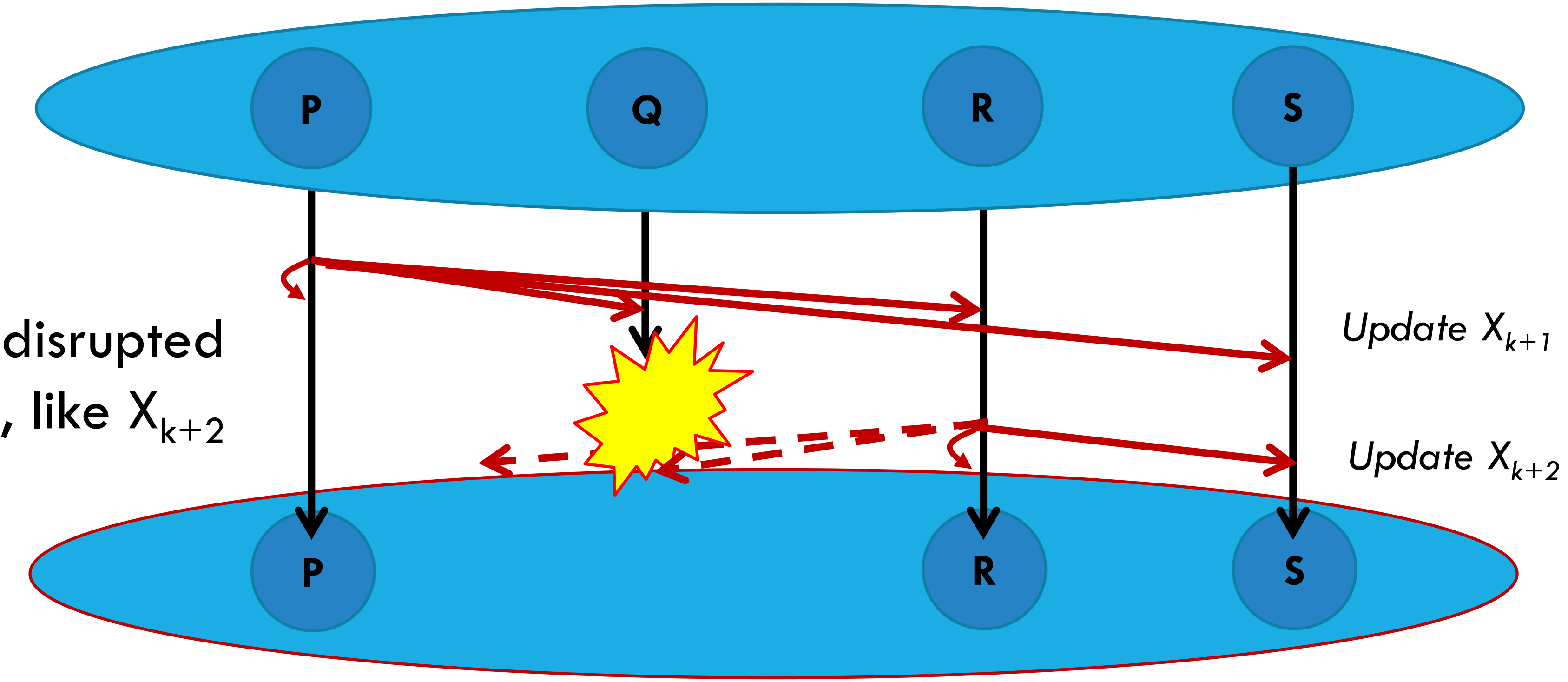
Derecho's method for this is a bit too complex for this lecture, but in a nutshell it cleans up from failures, then runs a protocol (based on quorums) to agree on the next view (the next epoch membership), then restarts.

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)

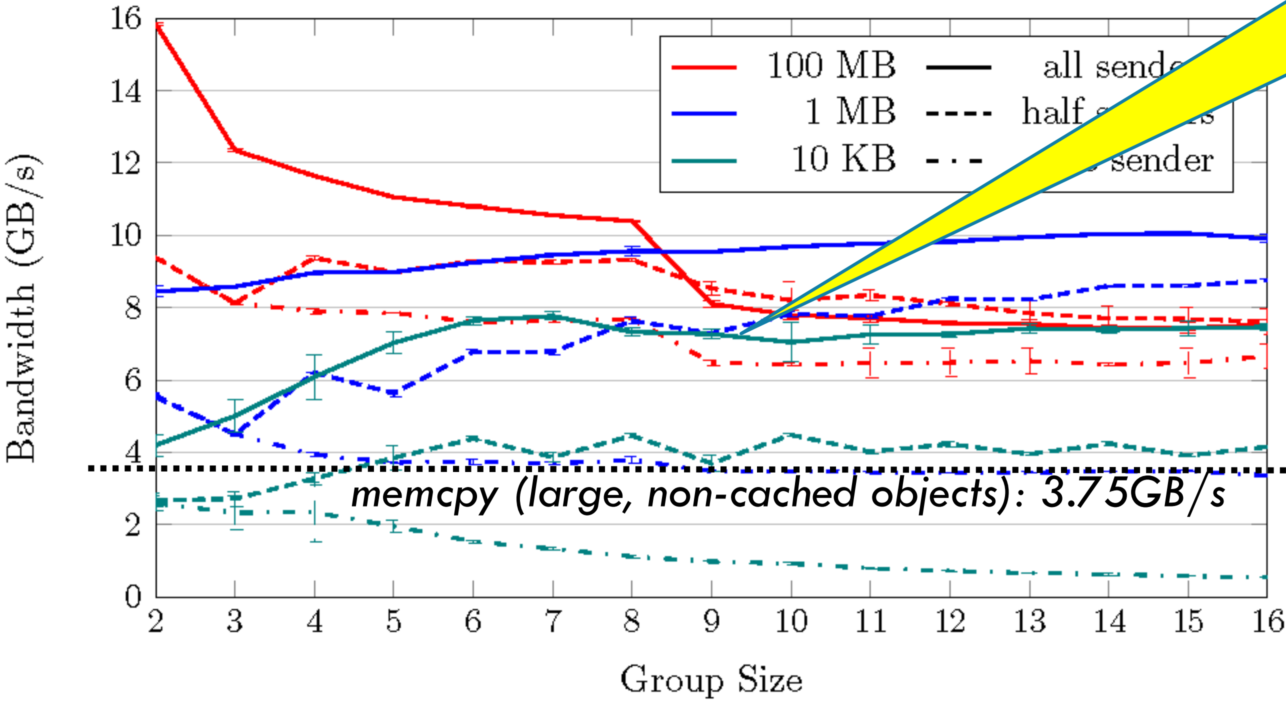
Derecho can make 16 consistent replicas at 2.5x the bandwidth of making one in-core copy

LARGE MESS

Mellanox 100Gbps RDMA on ROCE (fast Ethernet)

100Gb/s = 12.5GB/s

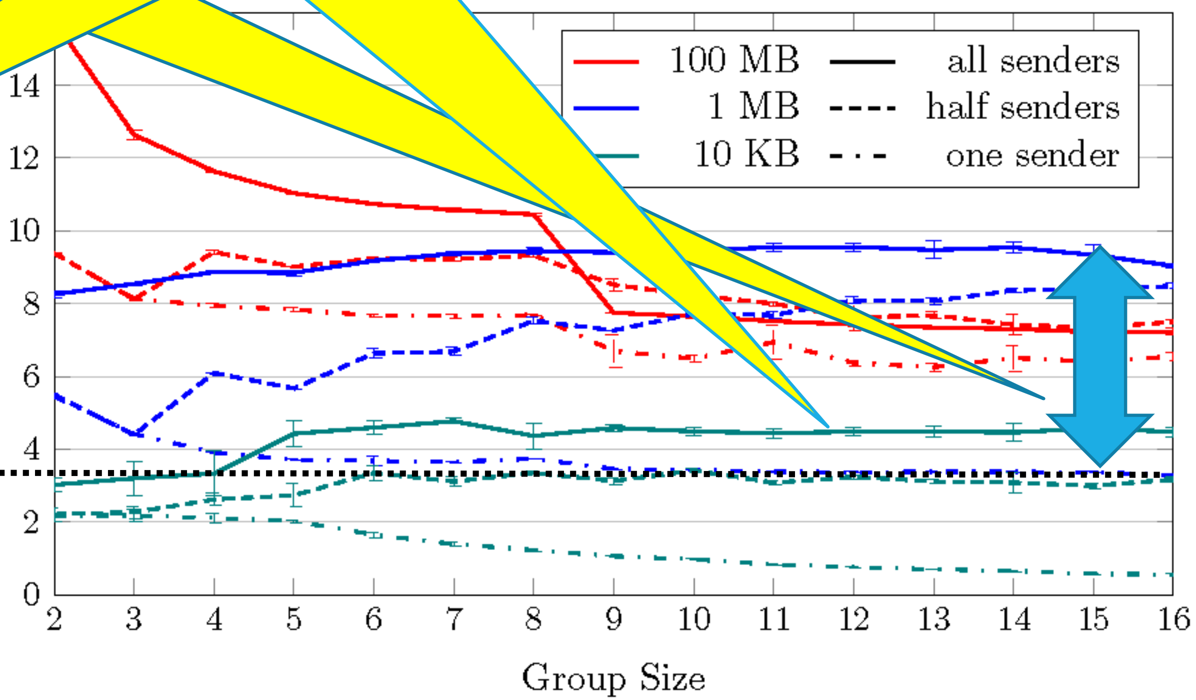
Raw RDMC multicast via Derecho API



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

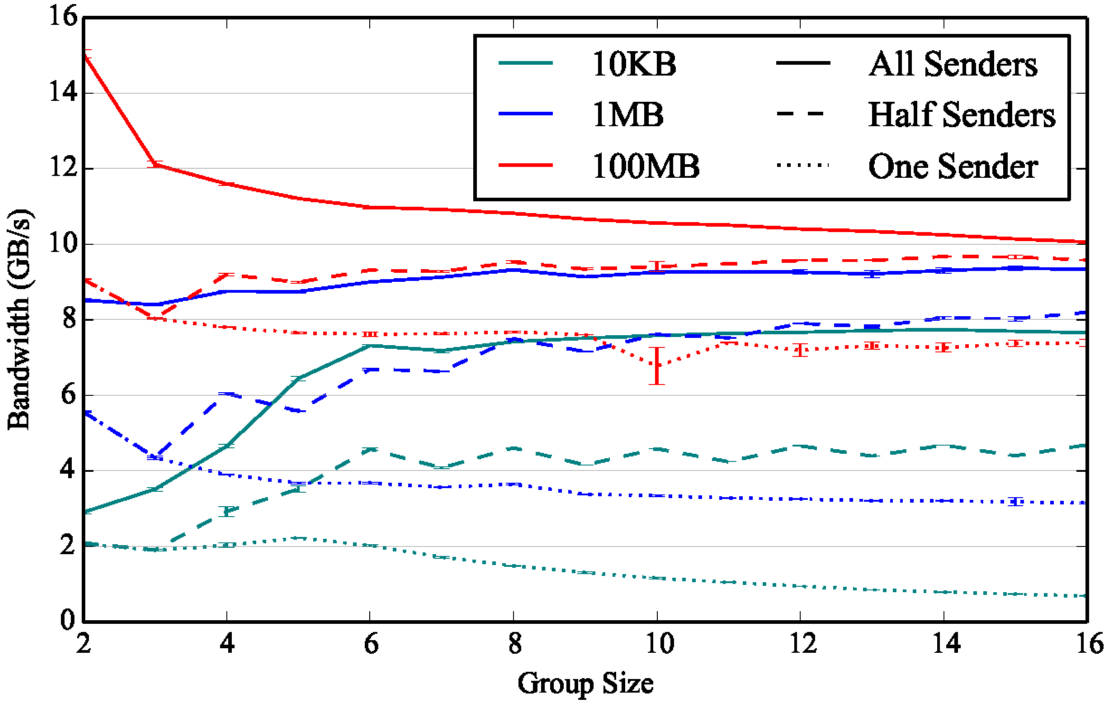
Raw RDMC is faster, but performance loss is small

ertical Paxos)

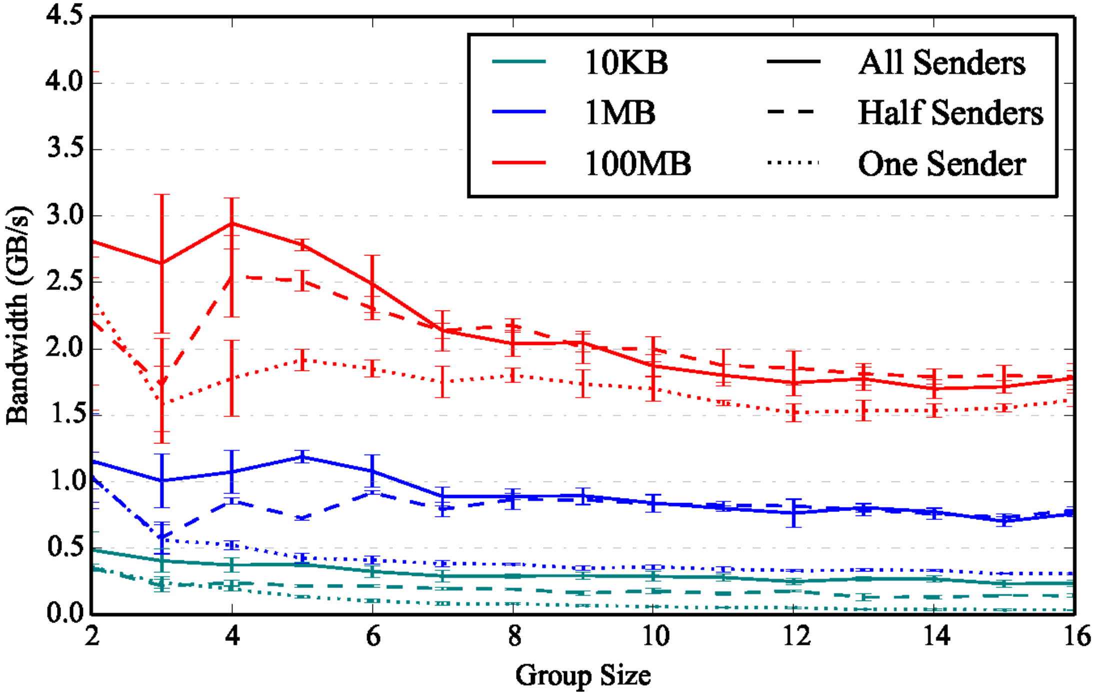


DERECHO CAN ALSO RUN ON TCP. WE FIND THAT RDMA IS 4X FASTER

Derecho Atomic Multicast: 100G RDMA

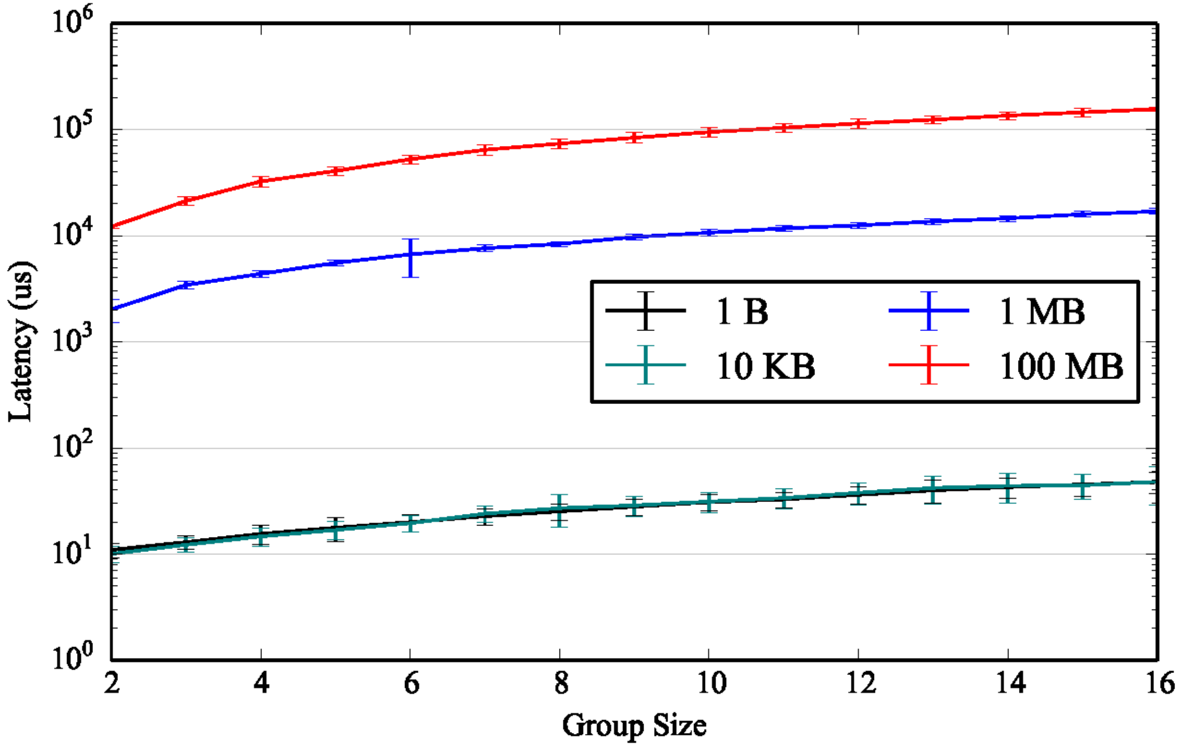


Derecho on TCP, 100G Ethernet

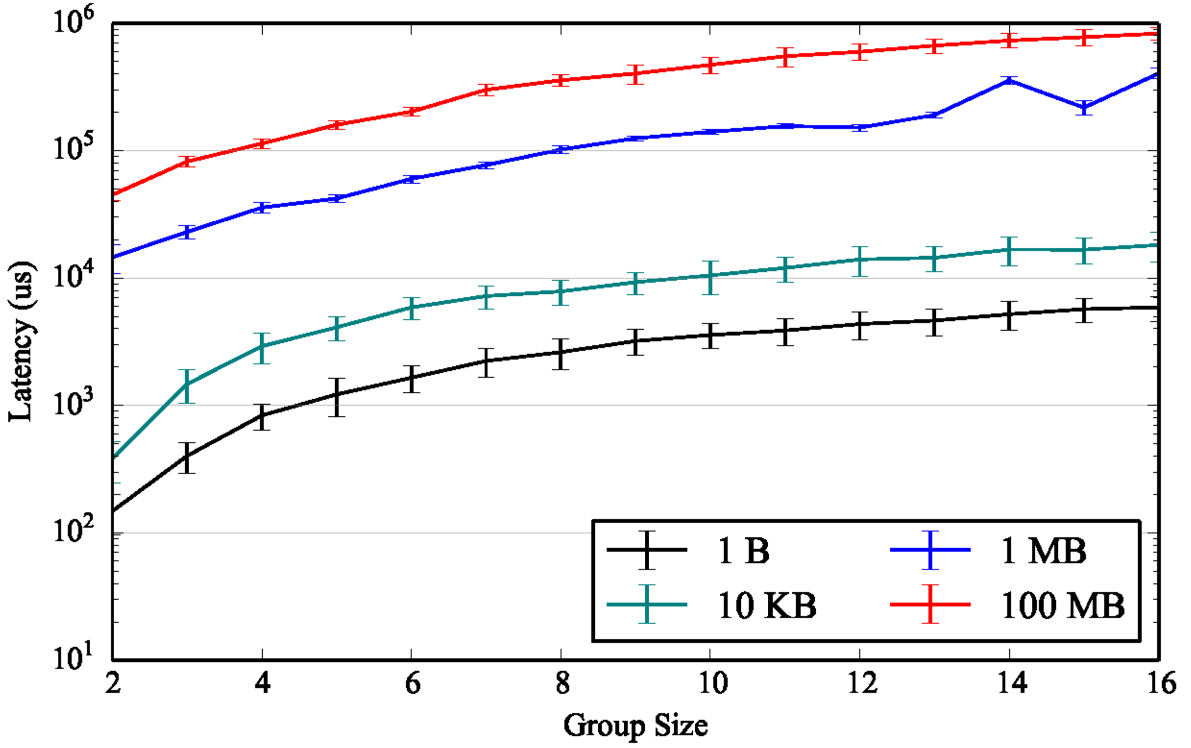


TCP INCREASES DELAYS BY ABOUT 125us

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

FOUR WAYS OF GETTING TO THE SAME PLACE!

Several of these solutions use Zookeeper to manage a file with membership data. Chain replication would do that. Then, it gets ordering and consistency by passing data in FIFO order down the chain.

Lamport's original atomic multicast protocol would also use some other method to manage membership. It gets ordering via a 2-phase protocol with logical clocks.

Paxos also uses a 2-phase (at minimum) commit. The slots in the log determine the delivery ordering. Proposers compete to fill them in.

Derecho has a virtual-synchrony membership service, then uses a fixed order. Senders send messages (or a null) in round-robin order, based on the view.

Paxos drilldown: If time permits (but we probably won't cover this slide)

WHAT ABOUT THE DERECHO OBJECT STORE?

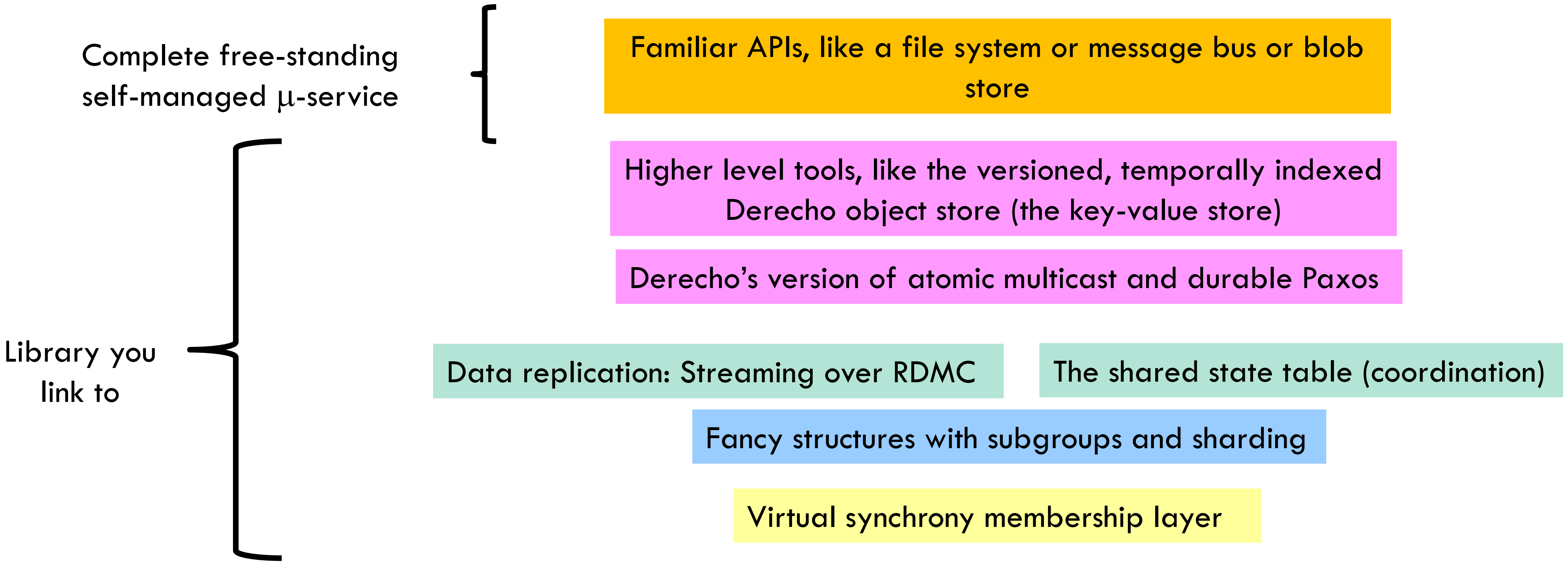
We heard about this in the lecture about the “Freeze Frame File System”.

- 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.