



# **CS5412 / LECTURE 8**

## **REPLICATION AND CONSISTENCY**

### **(PART II: PRACTICAL OPTIONS)**

**Ken Birman**  
**Spring, 2022**

# RECAP

Eric Brewer told us that stale data and inconsistency is fine. Why worry? And in fact this worked for cloud computing web sites, like Amazon.com

But his CAP model doesn't fit well with IoT settings that involve watching real-world devices and controlling real-world actions. Consistency matters for these cases: we need data that isn't stale, we need fault-tolerance, and we need scalability too!

So we discussed 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: VIRTUAL SYNCHRONY

The idea was to break down a distributed systems into a

- **Multicast protocol.** Sends messages only while membership is stable.
- **Membership service.** Tracks which processes are in the system, and what role each process is playing (like which shard it is in).
- **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.

# CLASSIC PAXOS



To understand the issue, it may help to start by understanding a little about classic Paxos.

We'll focus on the behavior of the system when membership is fixed. This is because modern systems use virtual synchrony or a similar mechanism so that Paxos is always paused when membership must be changed.

# LESLIE LAMPORT'S VISION



Leslie starts with  $2F + 1$  processes (for example, to tolerate 1 crashed processes, we need 3 in total).

Paxos tolerates processes that get overloaded and don't reply, but later recover and act normally. *Membership is not changed in such cases.*

The state is stored on logs – this is a persistent append-only update model.

# LESLIE DIDN'T USE VIRTUAL SYNCHRONY

At the time, he wasn't familiar with my model.

So he assumed that the membership was fixed, and that some processes simply couldn't be reached due to being crashed – a temporary problem. Crashed processes would later restart on their own.

His idea: each update just needs to reach a majority of the members.

# 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.



# (2-PHASE COMMIT “OVERSIMPLIFIES”)

Each of the two phases can require a few rounds of messages.

This relates to cases with concurrent updates occurring, or where some processes actually do fail and then recover with some data loss.

We won't get into those details in CS5412.

# HOW DOES PAXOS READ DATA?

The central issue is that due to being failed when an update occurred, some acceptors (some log copies) might lack certain writes.

To compensate, Paxos has a client (a “learner”) read multiple replicas. Then the learner merges the log contents, which fills in any gaps.

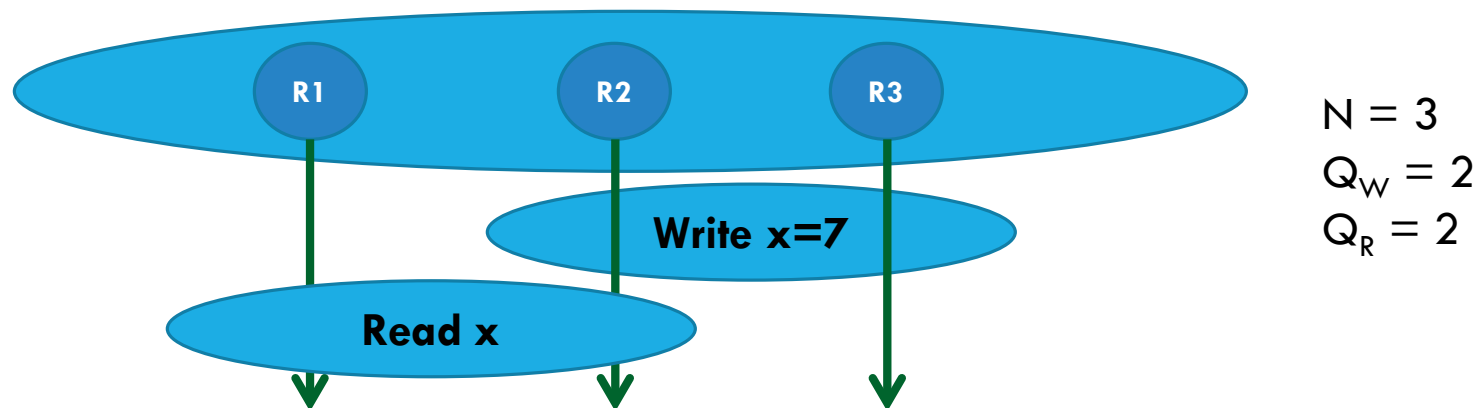
Accordingly, we define the read quorum,  $QR$  to be large enough to overlap with any prior update that was successful. E.g. might have  $QR = 2$

# VERIFY THAT THEY OVERLAP

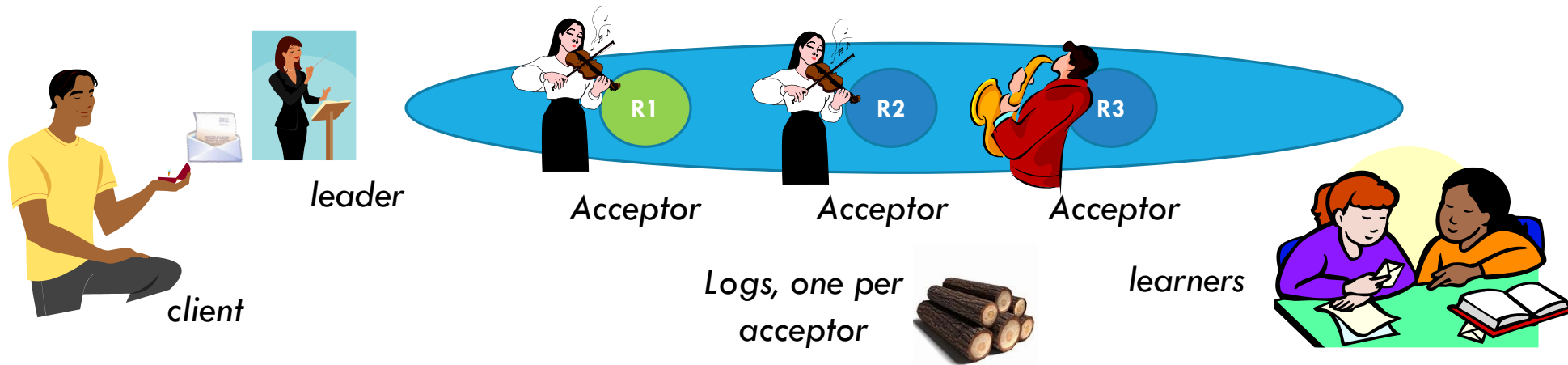
So: we want

- $Q_W + Q_R > N$ : Read overlaps with updates
- $Q_W + Q_W > N$ : Any two writes, or two updates, overlap

The second rule is needed to ensure that any pair of writes on the same item occur in an agreed order



# VISUALIZING THIS



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

# SO... PAXOS IS LIKE A SINGLE APPEND-ONLY LOG, BUT IMPLEMENTED WITH MANY LOGS!

Thinking of Paxos as a way to make a durable log of messages is the right way to view the classic protocol. Logs are on disk and are durable.

We use multiple logs, but the way we merge them when learning causes them to behave like one gap-free log.

Atomic multicast keeps everything in memory. It runs much faster, but the application would need to handle any logging.

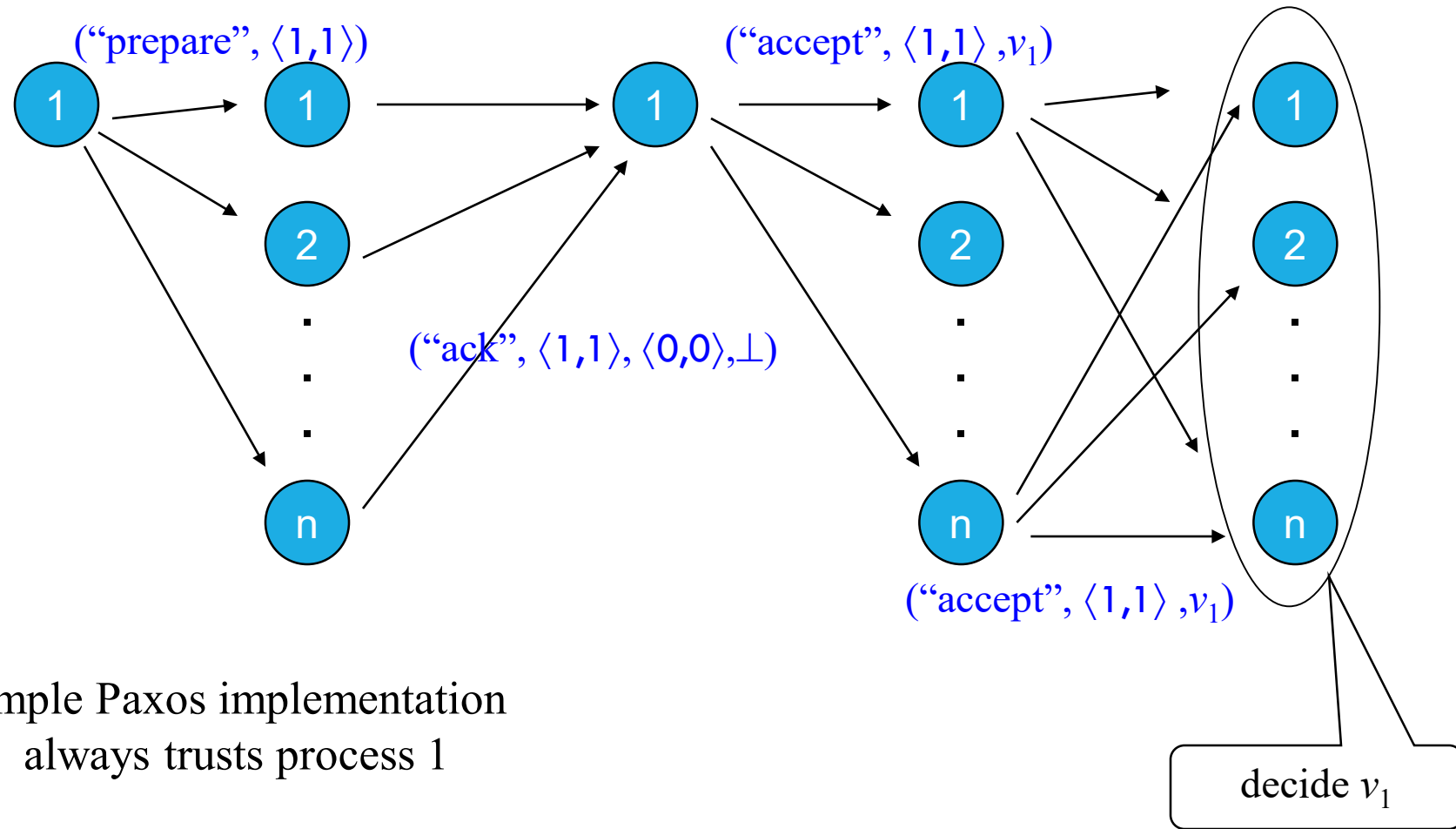
# PAXOS BALLOT NUMBERS

So... Paxos thinks of the log as a series of slots to be filled with updates.

But the first phase might not succeed on the first try. What if fewer than  $QW$  acceptors are able to accept some update?

This leads to the idea of “ballots” – the first phase loops, with the leader trying to get  $QW$  successes on a series of proposals, each with an increasing ballot number. (Of course, ideally, we succeed right away).

# IN FAILURE-FREE SYNCHRONOUS RUNS



Simple Paxos implementation  
always trusts process 1

# CRITICISMS OF PAXOS

The protocol is very slow and clumsy.

It has a proof of correctness, which is great, but is also very complex and many implementations have had bugs. Over the decades more and more Paxos implementations have been proposed and proved to implement state machine replication, but by now it is no longer clear what Paxos “is”!

In fact the classic Paxos protocol wasn't even invented by Lamport! There were at least three prior systems using it.



# 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 TO THE RESCUE!



A Derecho is a powerful wind

Derecho is a system created at Cornell that implements a new version of Paxos in which we also tried to leverage modern datacenter networks.

- It can support atomic multicast or durable (logged) Paxos.
- It runs on standard TCP but also supports a new and more modern hardware implementation of TCP in which the memory of one computer can be read or written directly from some other computer. This is called remote direct memory access: RDMA.
- When setting up Derecho, a configuration file tells it which to use.



# DERECHO IS A SOFTWARE LIBRARY

Derecho is a C++ library that handles membership, atomic multicast and persistent logging.

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

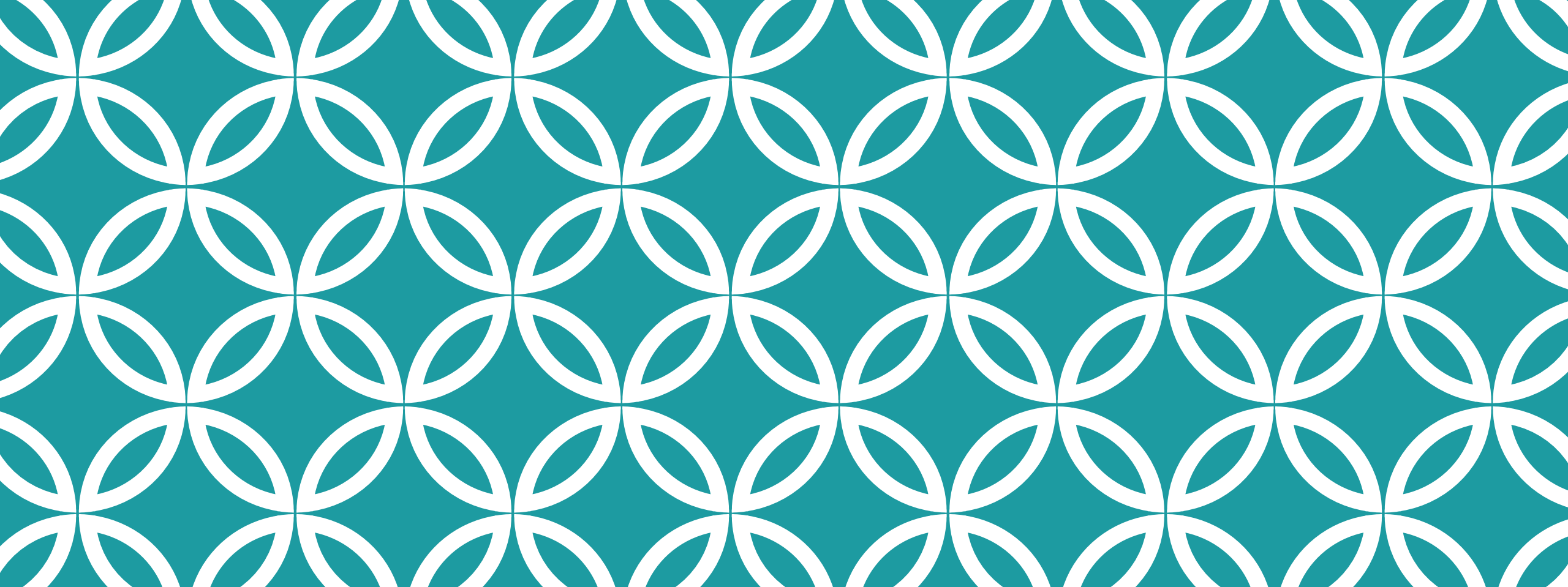
The developer builds a new microservice by linking against the library.

# DERECHO IS EXTREMELY FAST

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.

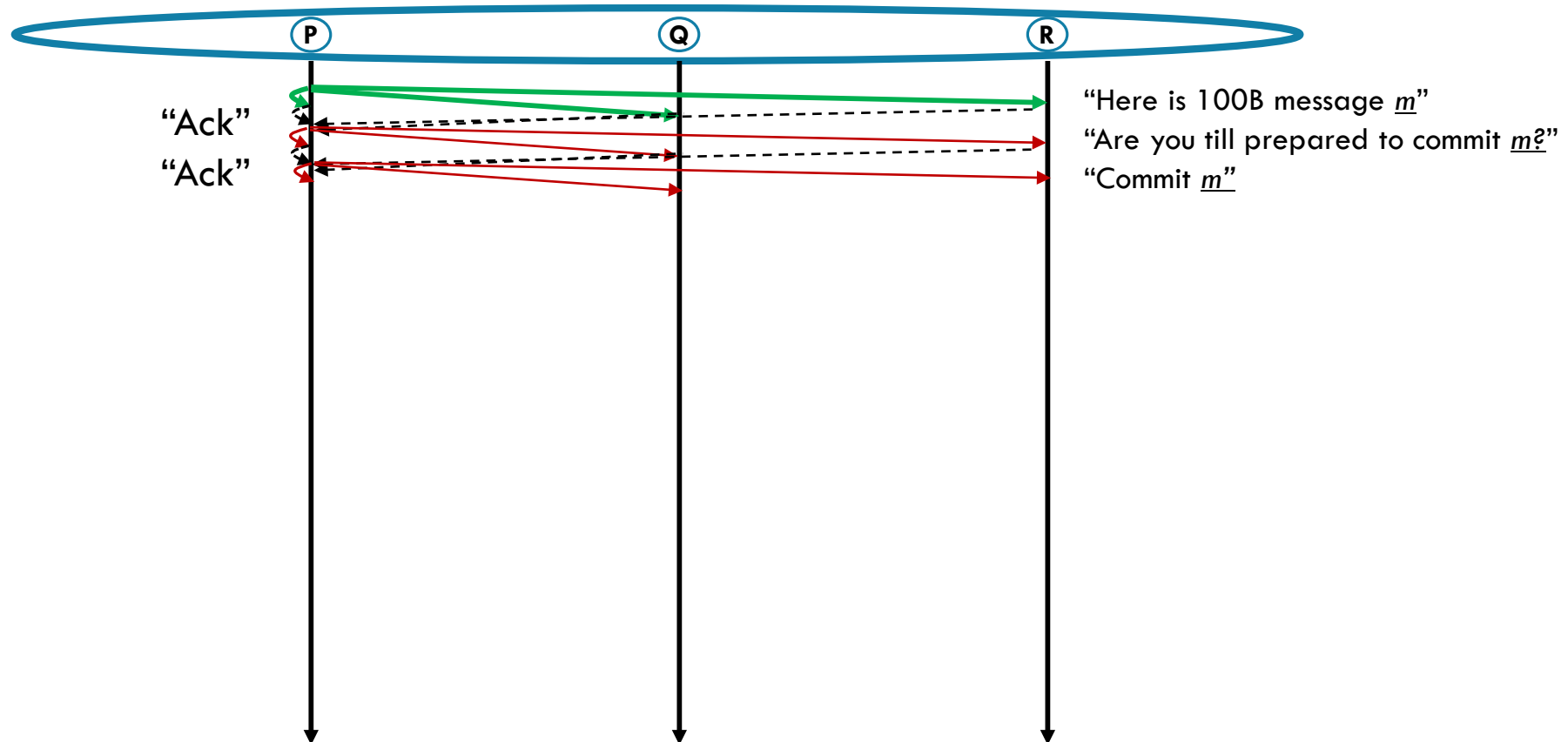
However, modern developers don't really want a C++ library. So we are creating Cascade, a new DHT for the IoT cloud, layered on Derecho.



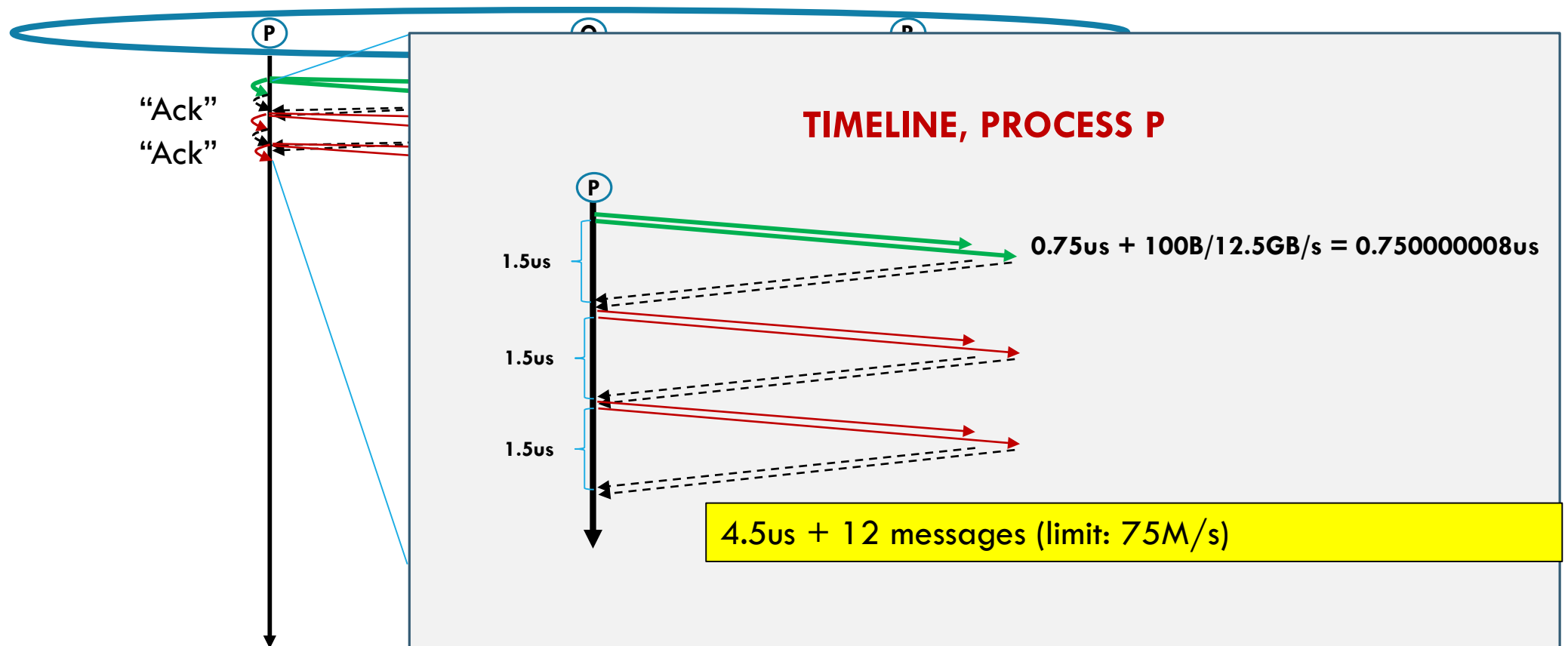
# DERECHO INTERNALS




# 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?  $4.5\mu\text{s} + \text{“noise”}$   
... 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”!

**P**

$100\text{B}/12.5\text{GB/s} = 0.750000008\mu\text{s}$

(limit: 75M/s)



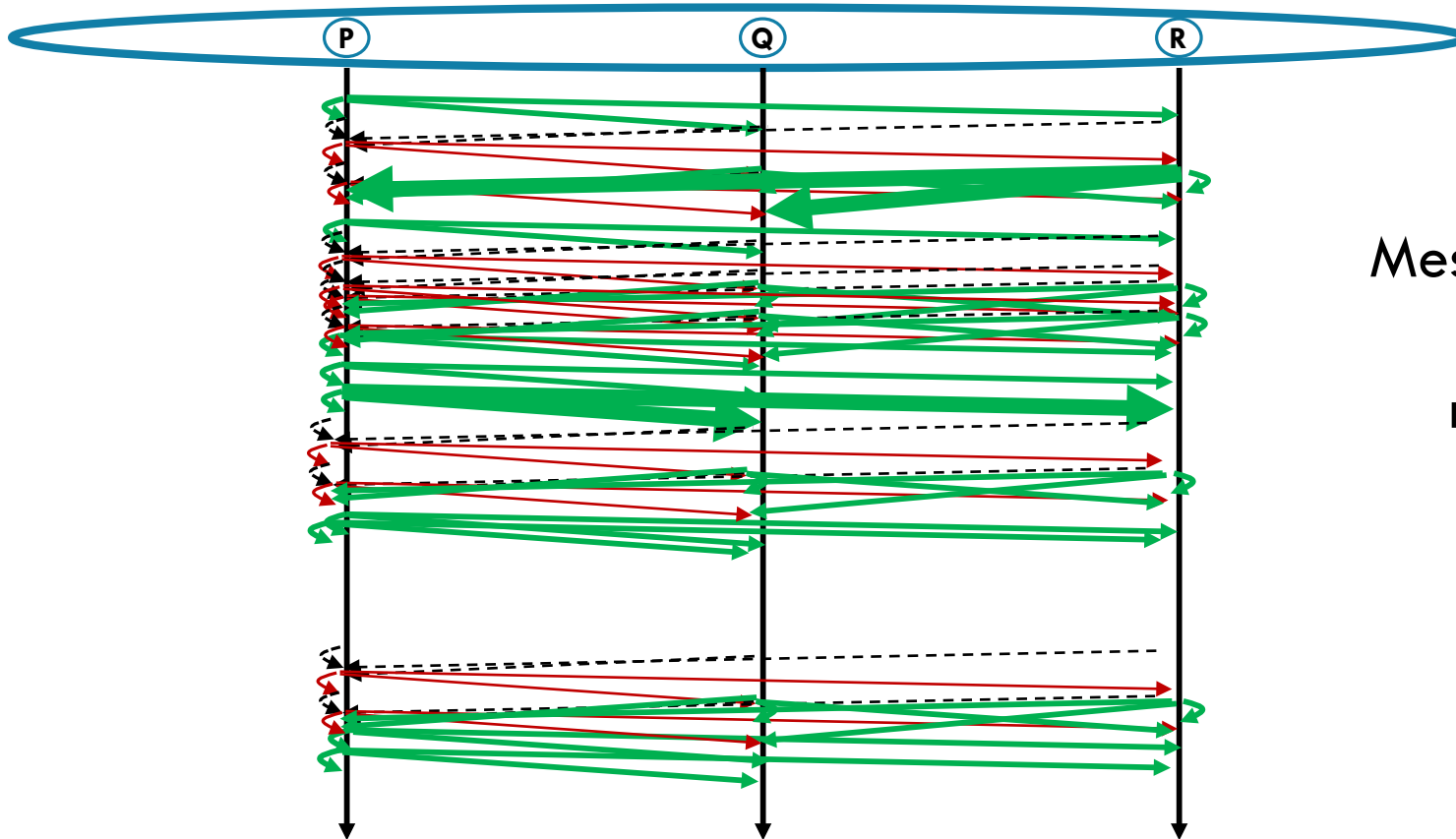
# A FEW IDEAS

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...



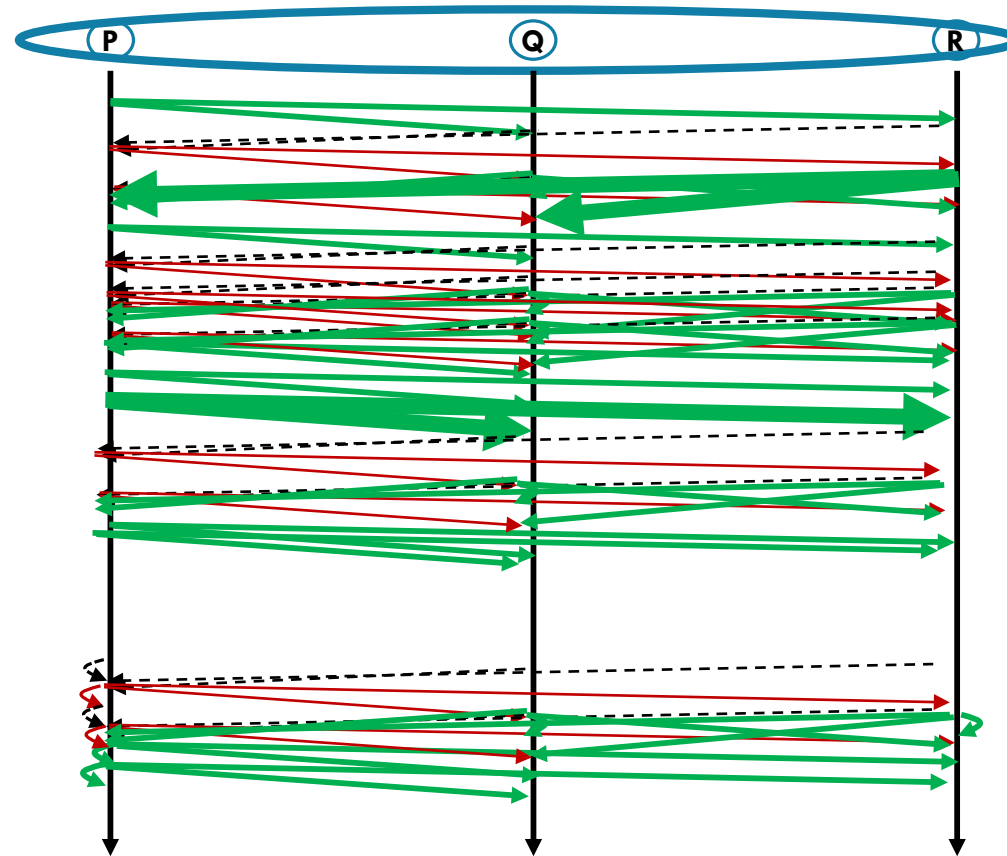
Messy and unpredictable with sudden bursts of data movement... Unlikely to perform well 😞

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

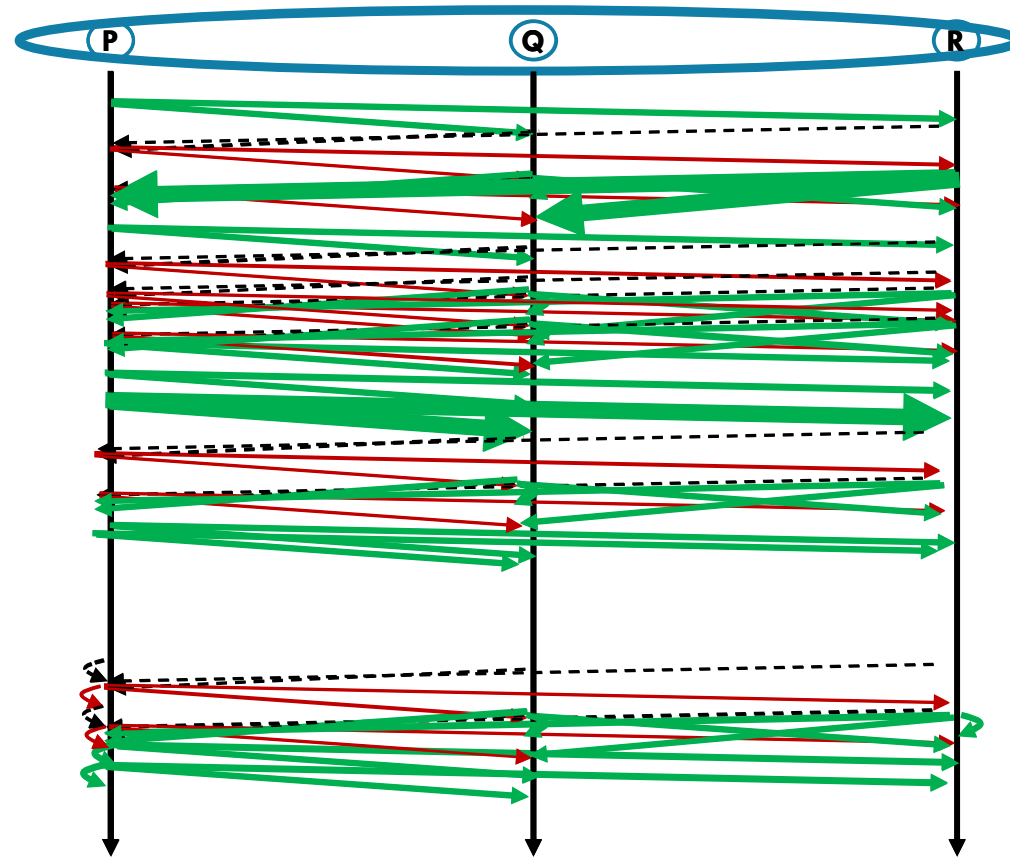
**Data plane:** The actual data messages. Send them continuously, as soon as new updates show up

**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.

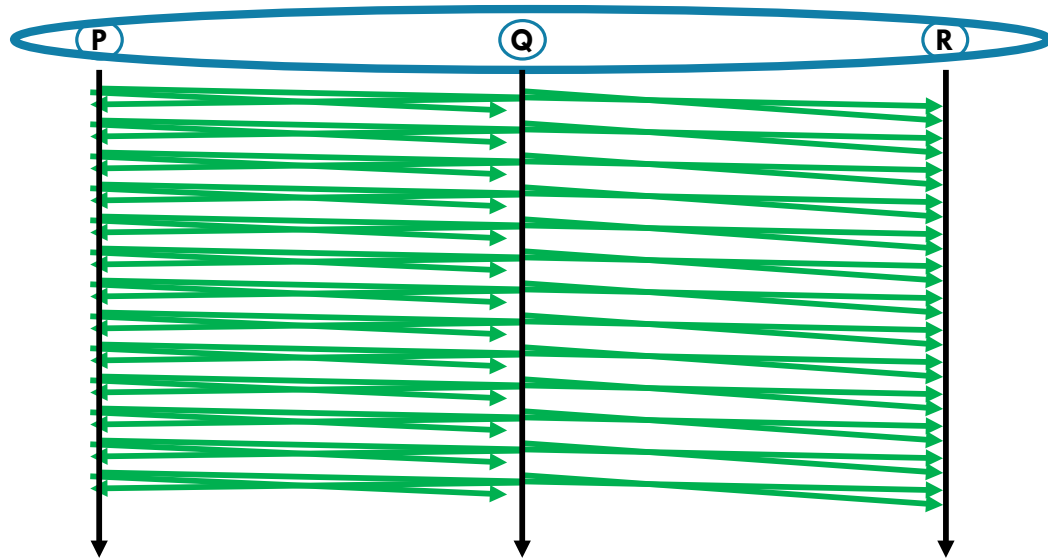
# BETTER: SEPARATE DATA PLANE AND CONTROL PLANE, MAKE THEM LOCK-FREE



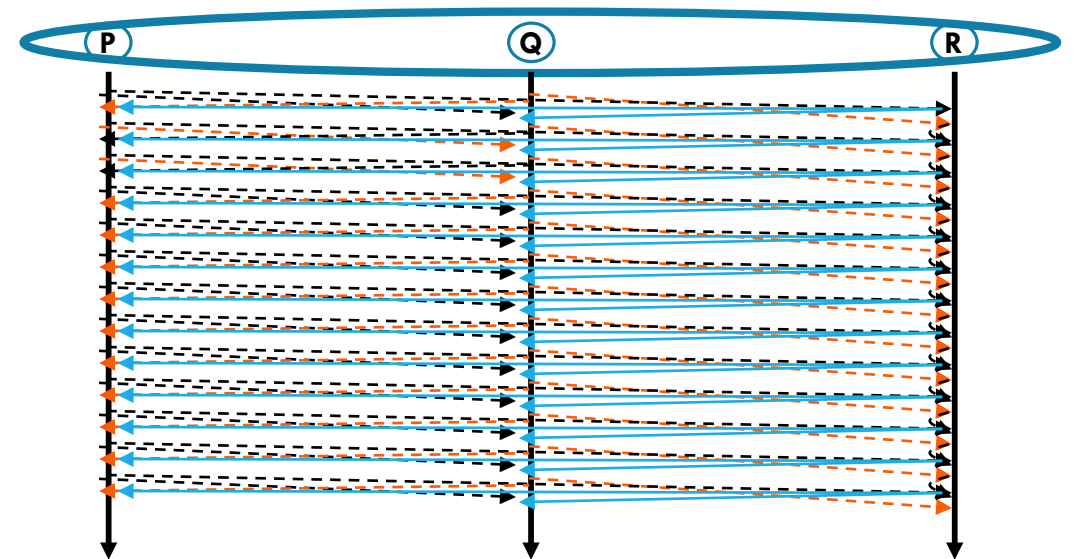
# BETTER: SEPARATE DATA PLANE AND CONTROL PLANE, MAKE THEM LOCK-FREE



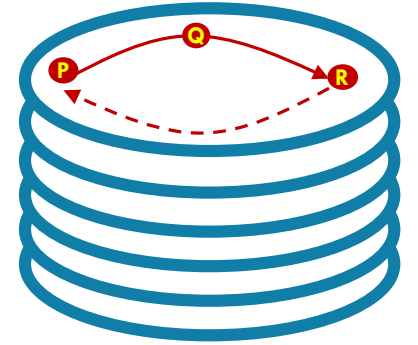
# BETTER: SEPARATE DATA PLANE AND CONTROL PLANE, MAKE THEM LOCK-FREE



Data plane runs steadily



Control information exchanged continuously



# HOW TO PUT THESE INTO ORDER?

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

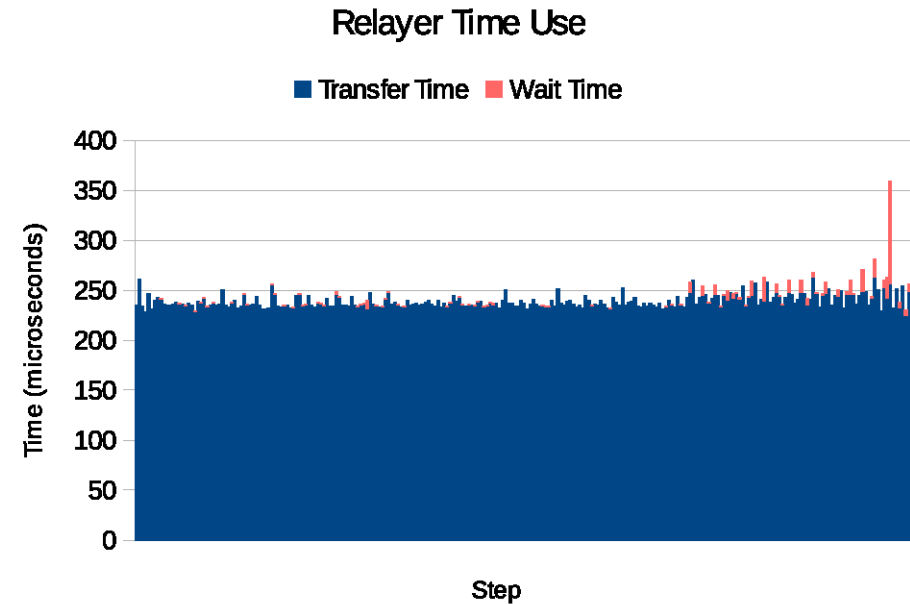
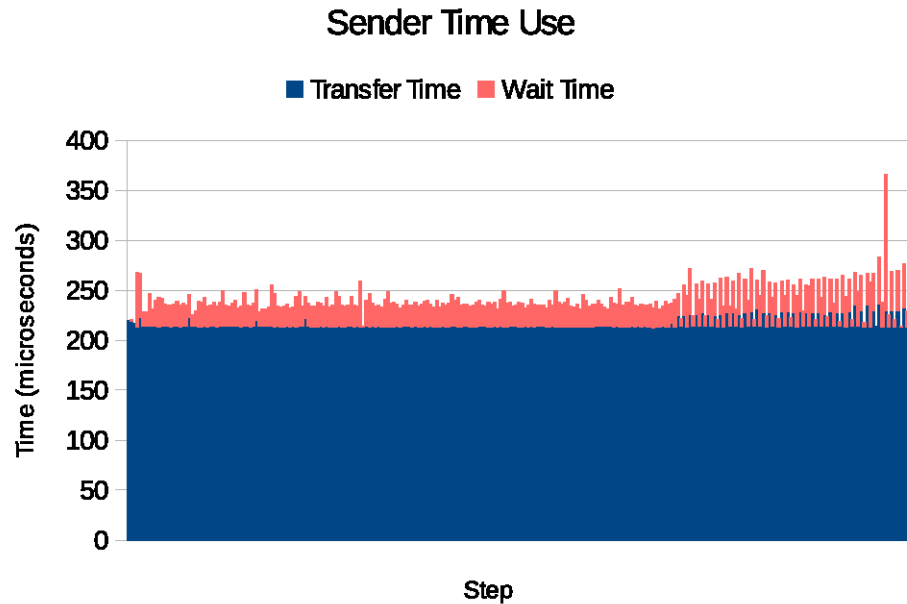
By “aligning” the flow of information with the network and not waiting for round-trip responses, it can run at the full network speed continuously.

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

Analogous real-world situation: filling a bucket from a steady stream of water (Derecho), versus filling it one cup at a time (Paxos).

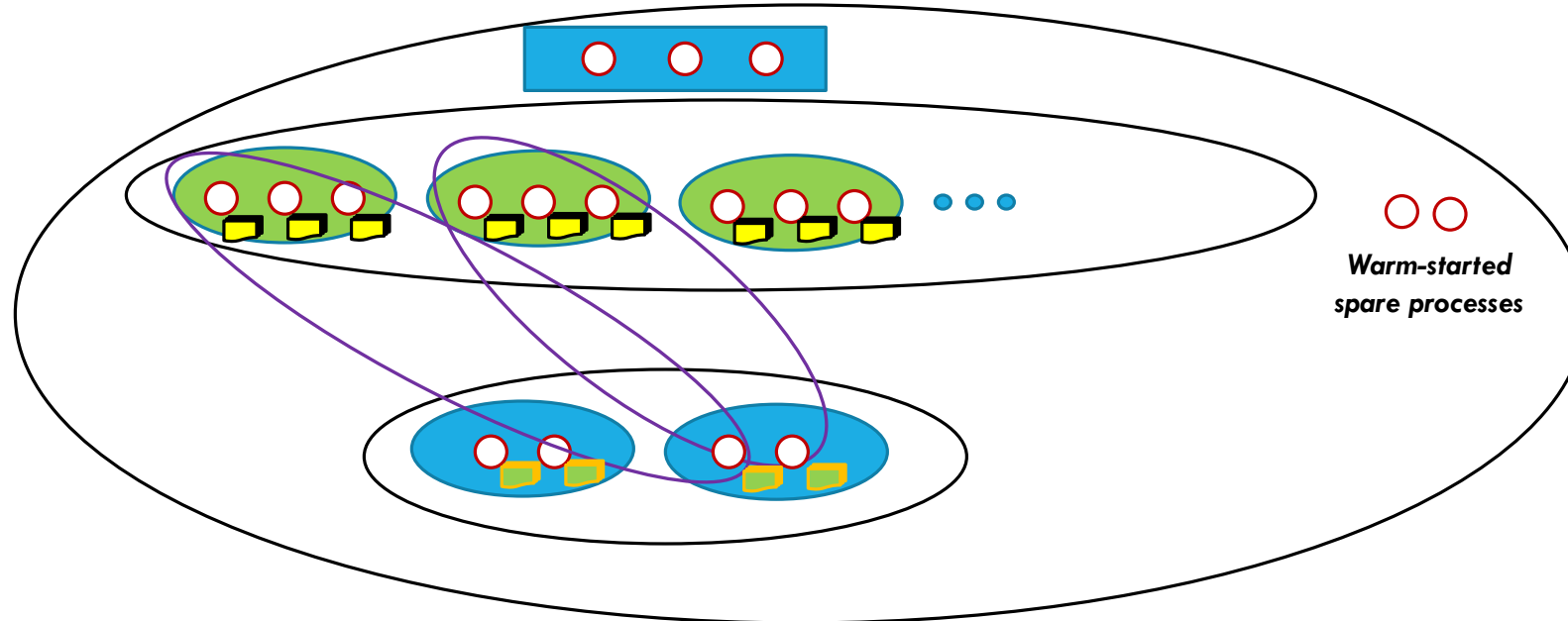


# DERECHO KEEPS THE NETWORK BUSY (BLUE) AND SPENDS VERY LITTLE TIME IN PROTOCOL SOFTWARE (PINK)



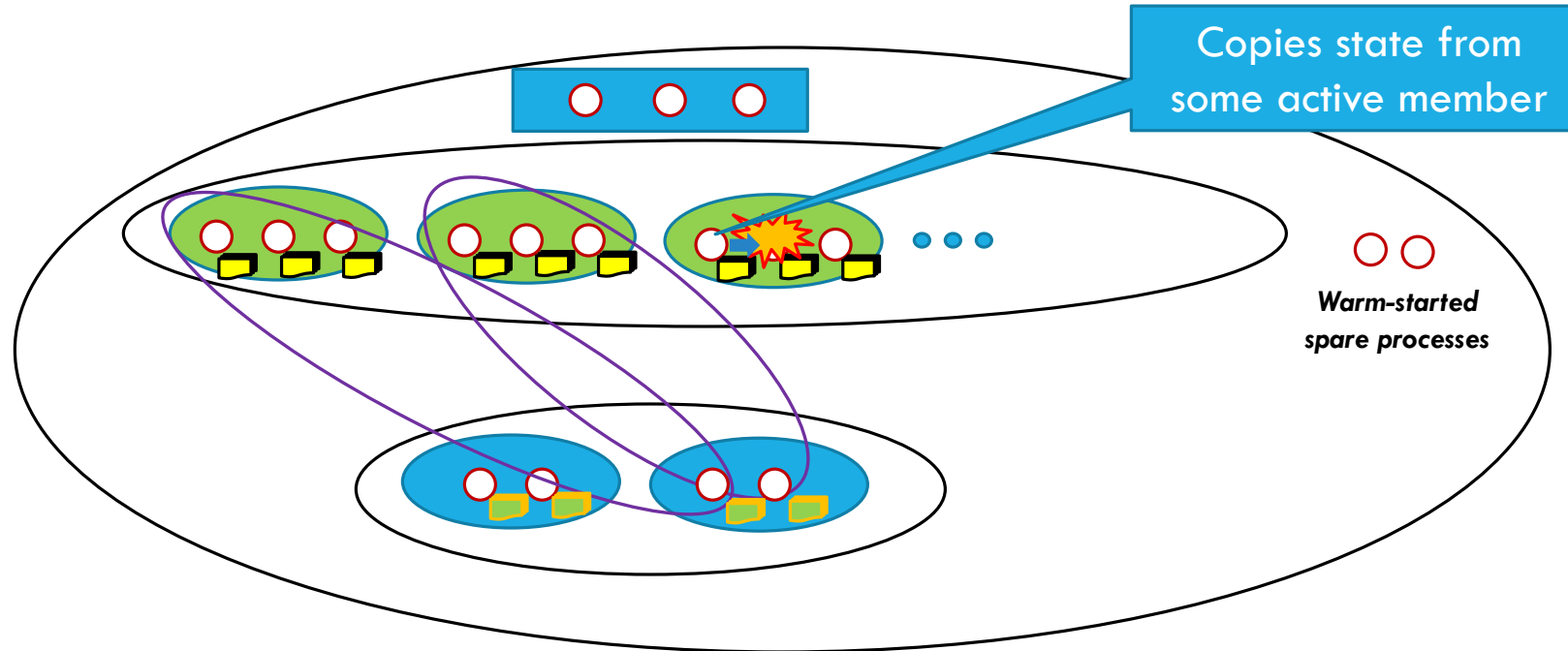
This snapshots Derecho under continuous heavy load

# DERECHO'S EPOCH MECHANISM IN ACTION



Derecho focuses on sharded services for modern cloud settings, and it packages this fast version of atomic multicast or Paxos with an automated way to organize the application into shards

# DERECHO'S EPOCH MECHANISM IN ACTION



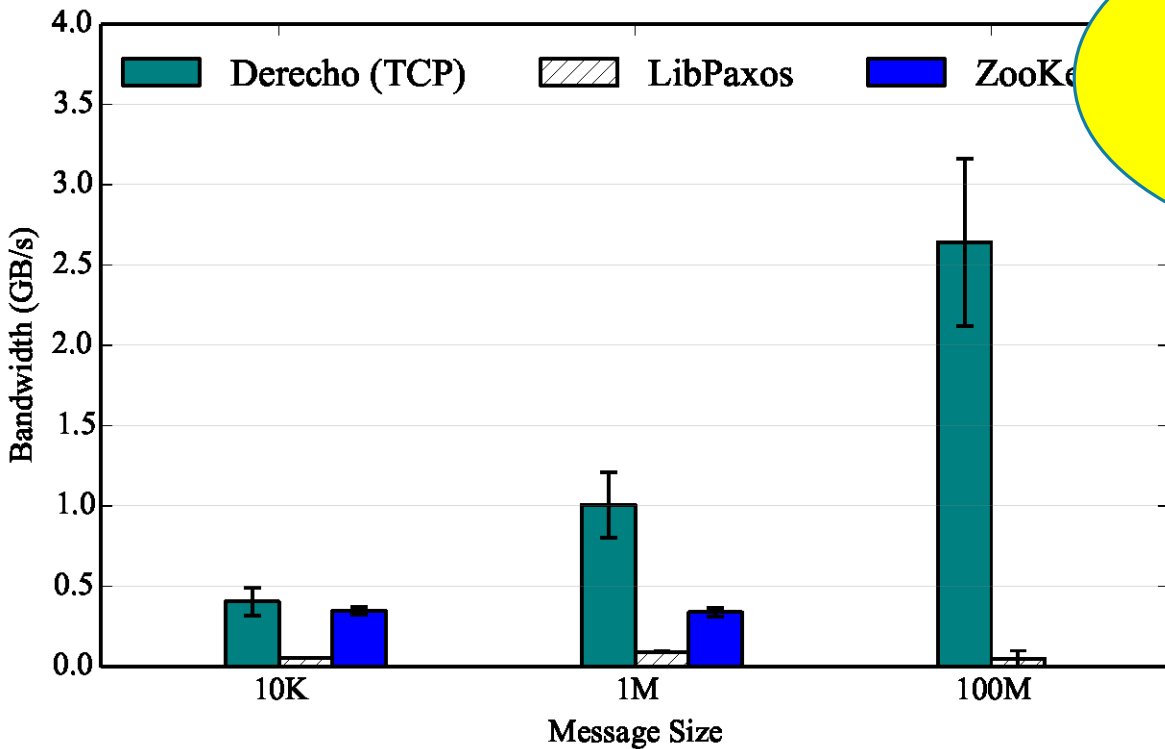
For example, if a failure occurs, Derecho automatically repairs it.

# ATOMIC MULTICAST PERFORMANCE

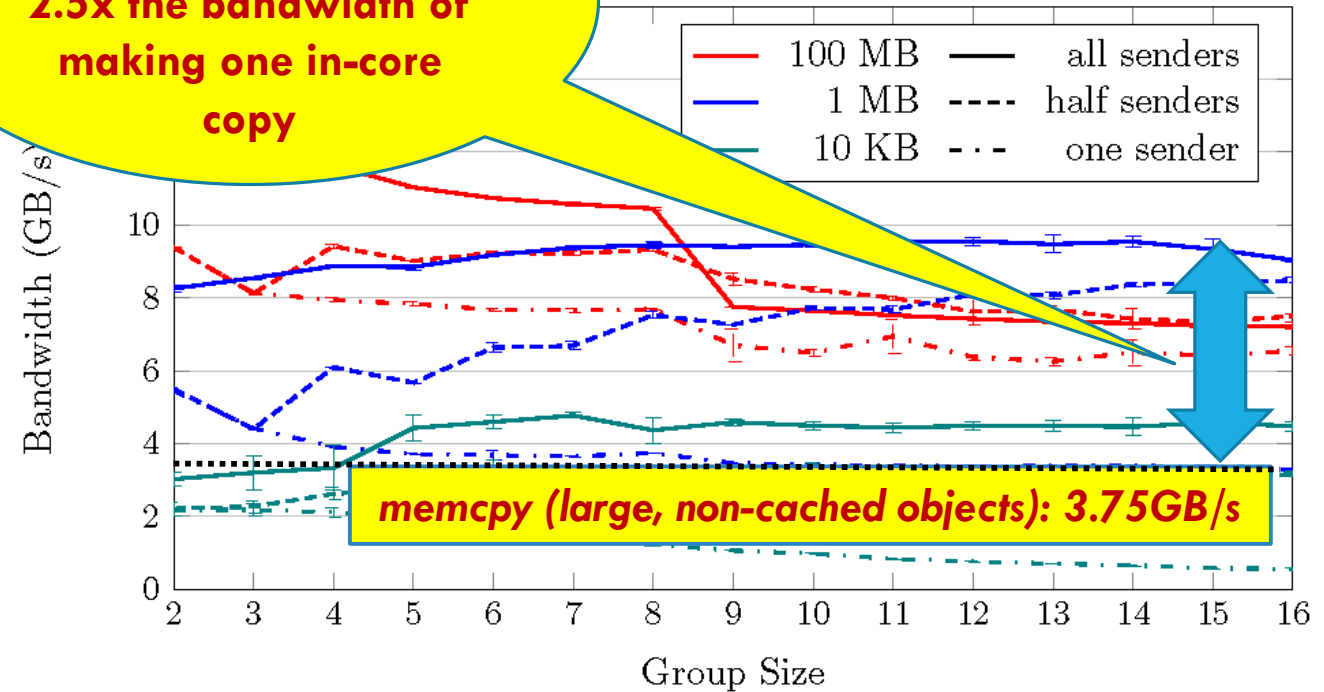
Mellanox 100Gbps RDMA on ROCE (fast Ethernet)

100Gb/s = 12.5GB/s

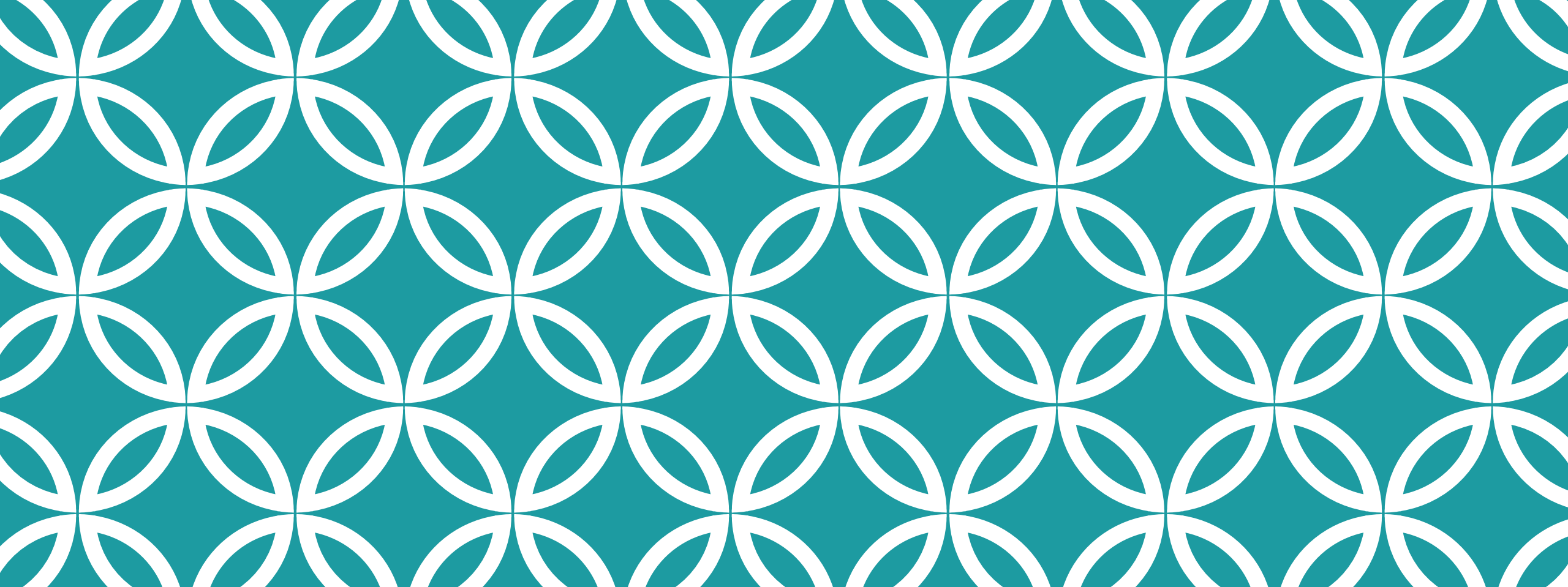
**Derecho can make 16 consistent replicas at 2.5x the bandwidth of making one in-core copy**



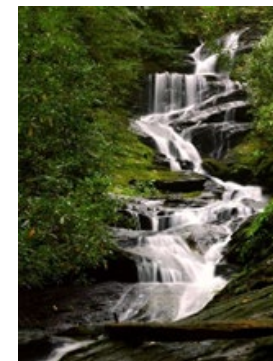
Derecho is faster than LibPaxos or Zookeeper even on TCP



Derecho Atomic Multicast on RDMA



# OUR NEW TOY: CASCADE





# SO WHY DO WE NEED MORE?

Derecho is a C++ library for sharded data + atomic multicast/Paxos.

Not everyone is a C++ coding wizard, so only really good developers can use Derecho directly.

Cascade project starts with Derecho and turns it into... a DHT for IoT!

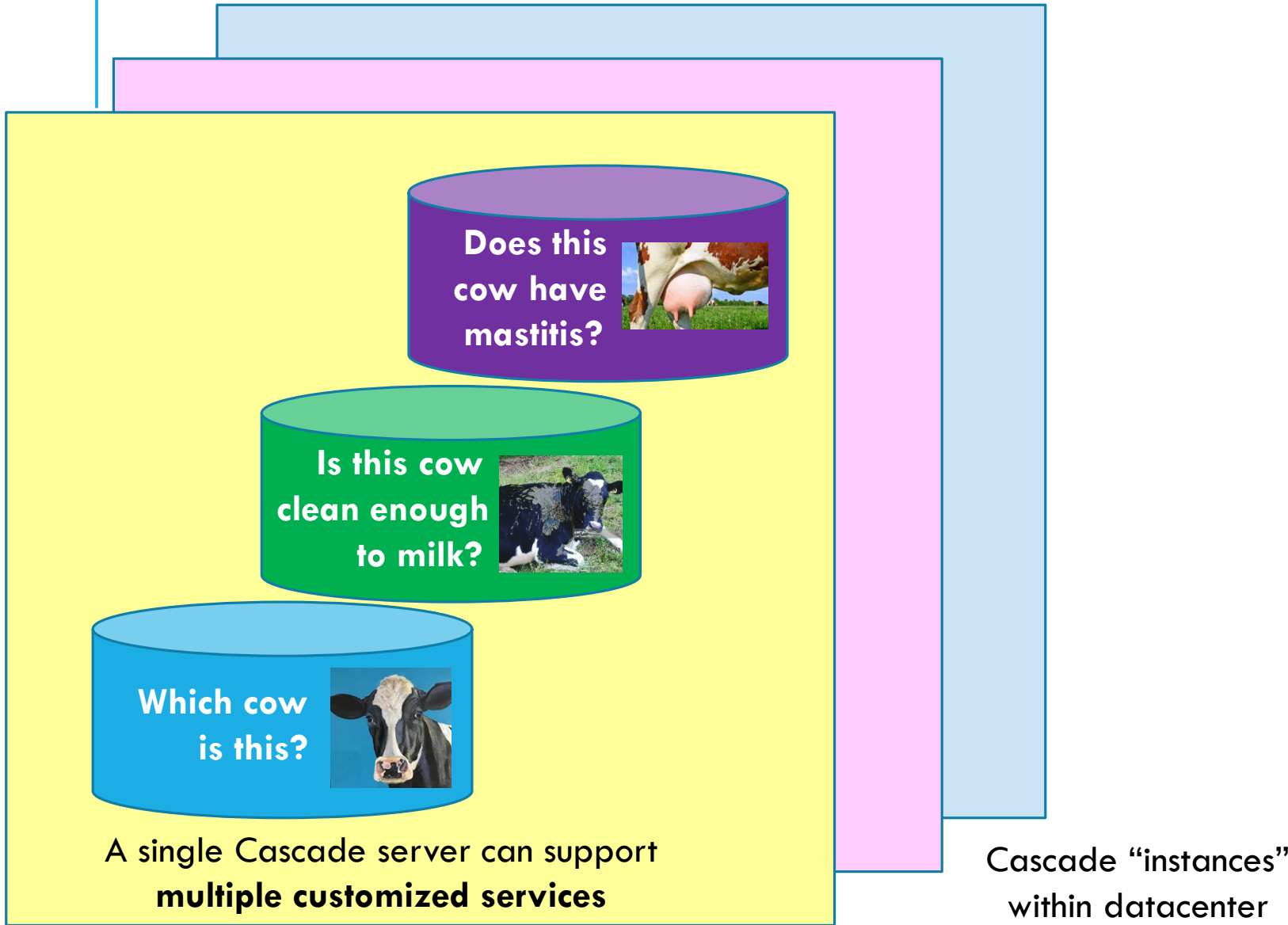
# CASCADE IS...

A DHT with a fancy, flexible API (for example, versioned put, notifications when (key-value) pairs of interest change)

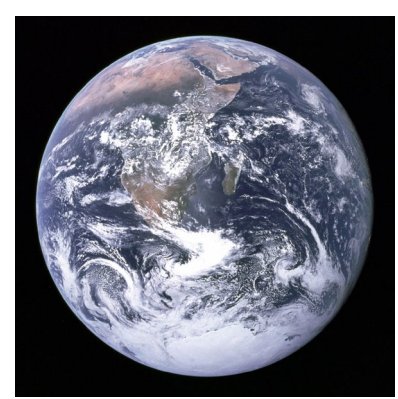
Layered on Derecho: Blazingly fast!

Customizable: You can integrate your own code into Cascade using the watch-a-key feature

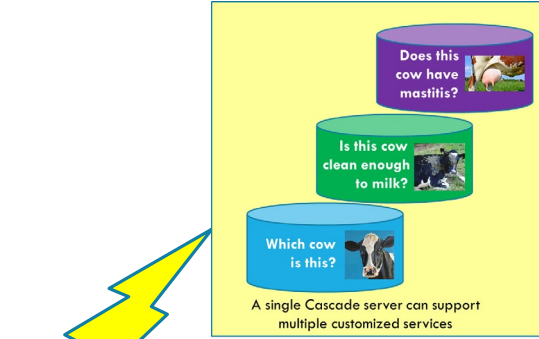
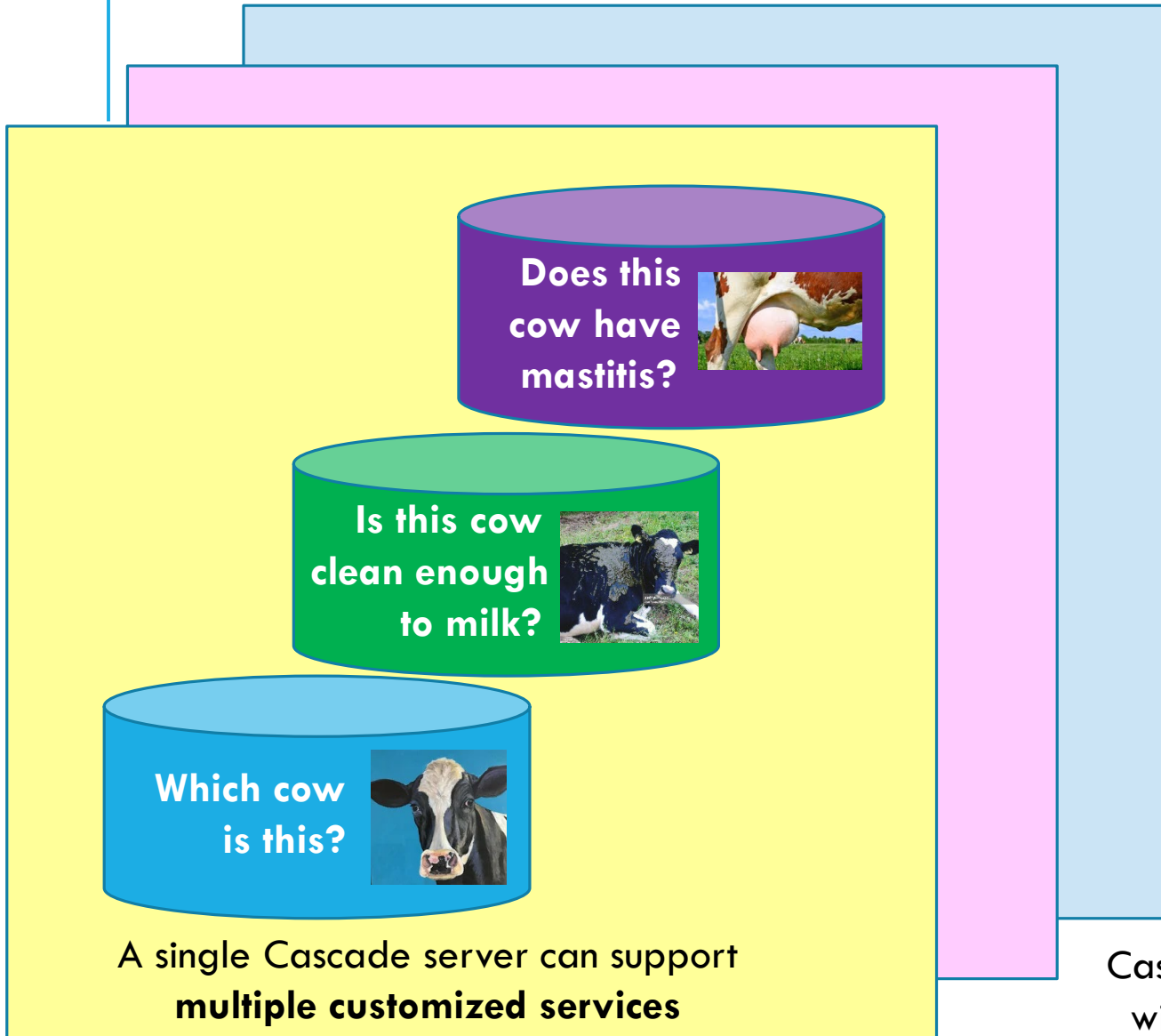
# CASCADE IS DESIGNED TO BE CUSTOMIZED



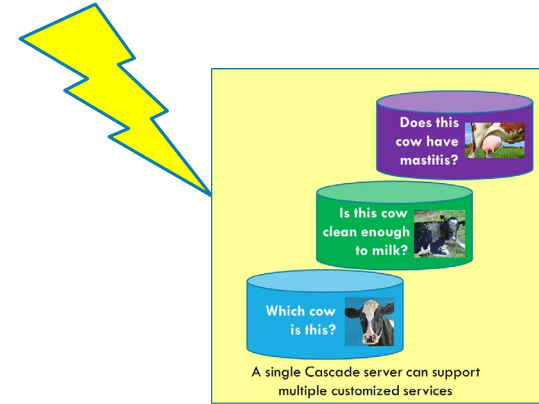




# CASCADE IS DESIGNED TO SCALE



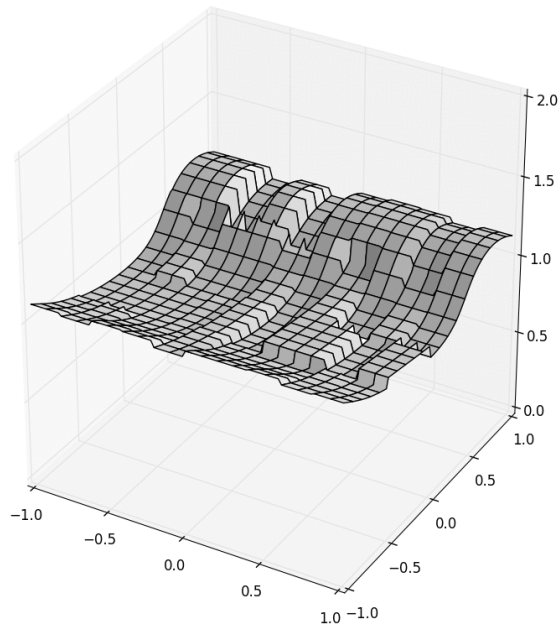
**WAN replication via read-only mirroring, using Zhen Xiao's WAN Agent.**



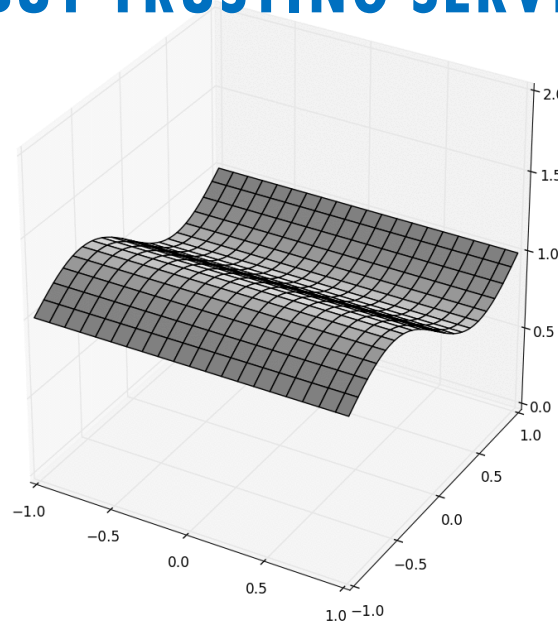
Cascade "instances" within datacenter

# CASCADE OFFERS STRONG CONSISTENCY

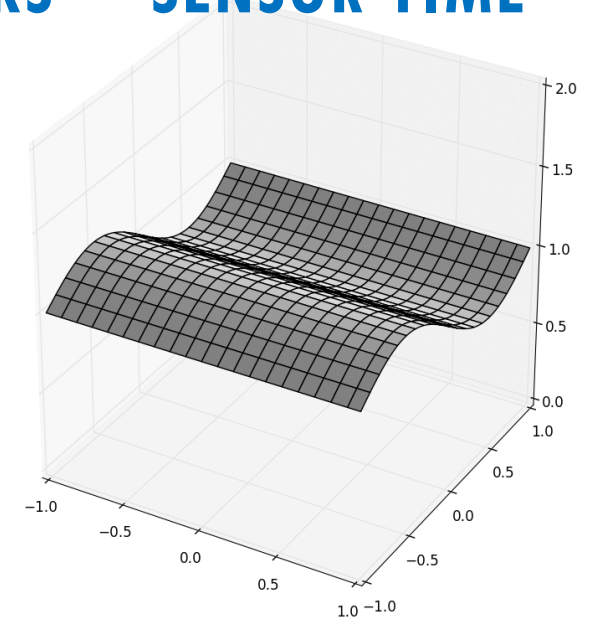
## HDFS



## CASCADE USING CONSISTENT CUTS BUT TRUSTING SERVER CLOCKS



## CASCADE WITH SENSOR TIME



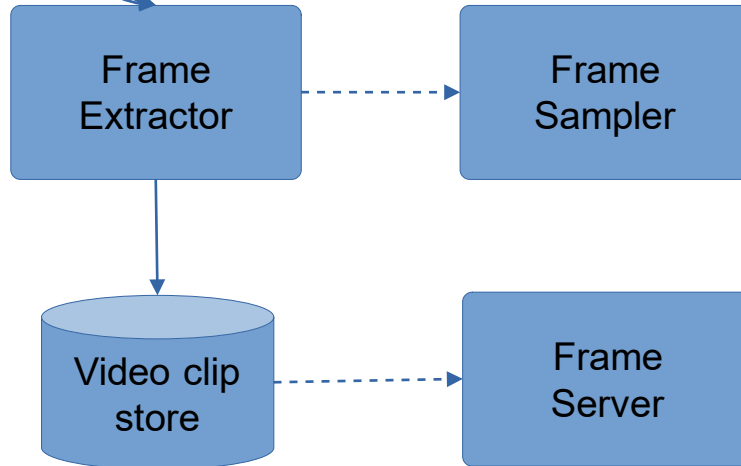
Cascade consistent cuts + GPS-timestamped sensor data result in clean input to the D-AI algorithm (in this case, a simple visualization)

# A REALISTIC DAIRY IMAGE PIPELINE

Dairy Farm

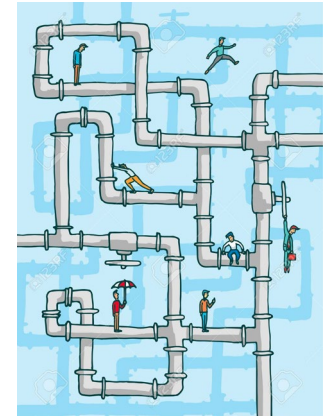


**The Farm Server (IoT Edge)**

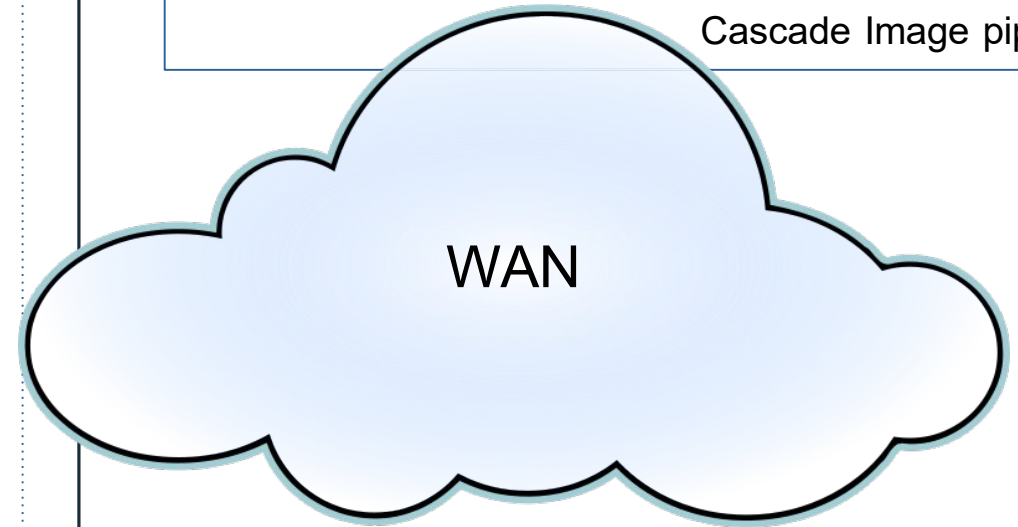


Data Center

Image Pipeline Front End  
(As an external client)



Cascade Image pipeline



# ... DETAILED VERSION (PyLINQ ON MSFT AZURE)



Integrate daily data



Date	cow_id	daily_yield	daily_fat	...	daily_protein
12/3/20	1	14	3.96	...	2.89
...	...	...	...	...	...
1/10/21	237	20	4.42	...	4.55

<field>/<cow\_id>{<ts(ver)>  
daily\_protein/cow\_id1{ver\_1} = 2.89

Download blobs from Azure & Store to Cascade VCSS subgroup

Upload daily date to Azure Blob Storage

Date	cow_id	...
12/3/20	1	...
...	...	...
12/3/20	237	...

```
client = cascade_py.ServiceClientAPI()
daily_yield_pool = client.get_object_pool("VolatileCascadeStore","daily_yield")
daily_yield_pool.put(cow_id,val)
```

daily\_fat/cow\_id237{ver\_38} = 4.42

```
white_cow_net = load_cow_model( \
    "cow_symbol.json", \
    'white_cow.params')
black_cow_net = load_cow_model( \
    "cow_symbol.json", \
    'black_cow.params')
...
action = cascade_context.get_next_action()
if action.type == "white_cow":
    pred = white_cow_net(action.frame)
else:
    pred = black_cow_net(action.frame)
```

Streaming image frames through TCP portal

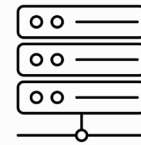


Farm server

Filtered image frames



External Client to Cascade



Store to subgroup VCSS

Trigger image analysis



Action = black\_cow\_infer



CV model

Image analysis

cow id: 127

LINQ query to retrieve data of most recent 10 days from Cascade about cow

```
1 client = cascade_py.ServiceClientAPI()
2 cow_id = "cow_id128"
3 recent_data = []
4 for field in ["daily_yield", "daily_fat", "daily_protein"]:
5     field_pool = client.get_object_pool("VolatileCascadeStore",field)
6     query_result = field_pool \
7         .where(lambda x:x.endswith("/"+cow_id)) \
8         .order_by(lambda x:x.version) \
9         .reverse() \
10        .take(10)
11    recent_data.append(query_result)
```

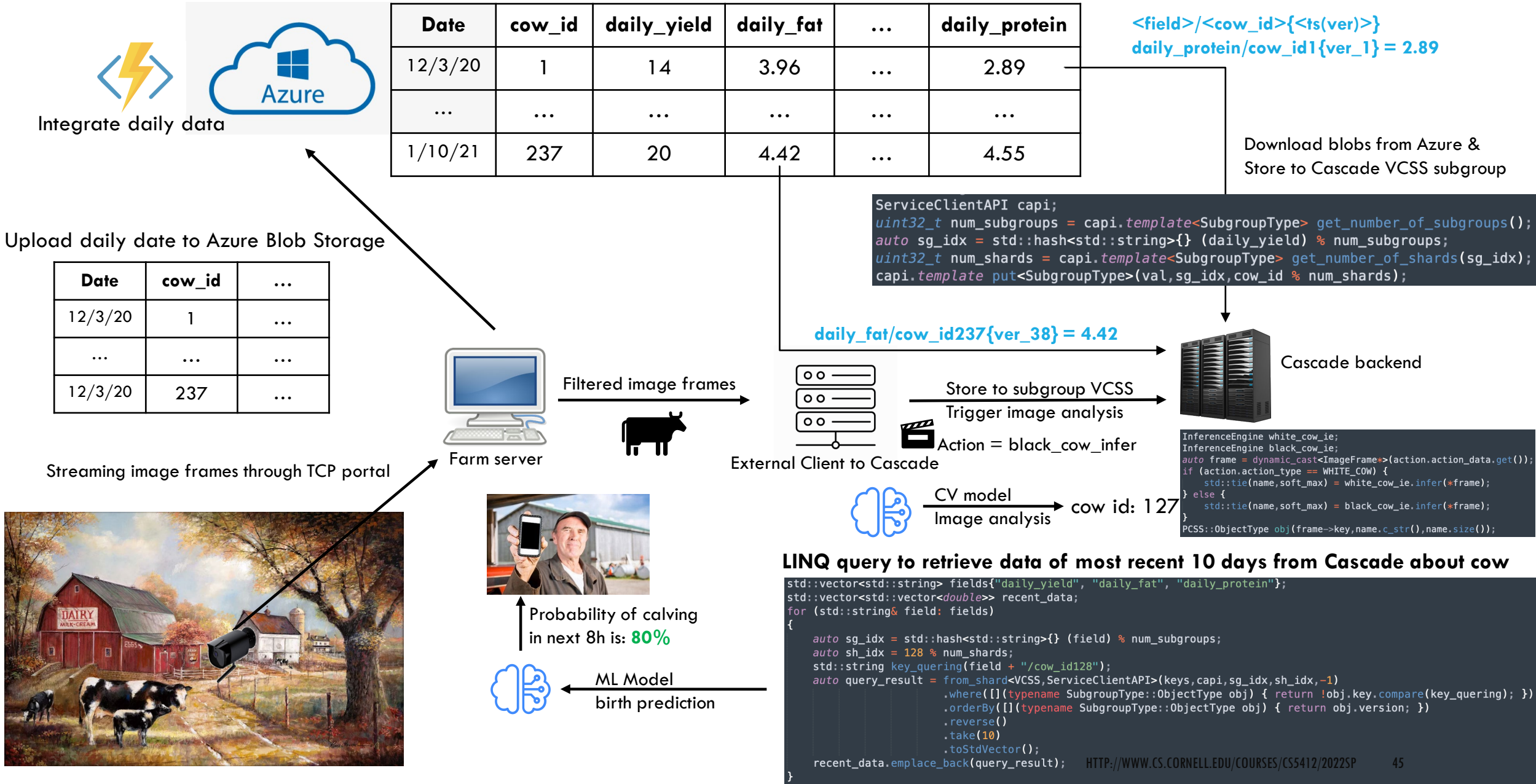


Probability of calving in next 8h is: 80%



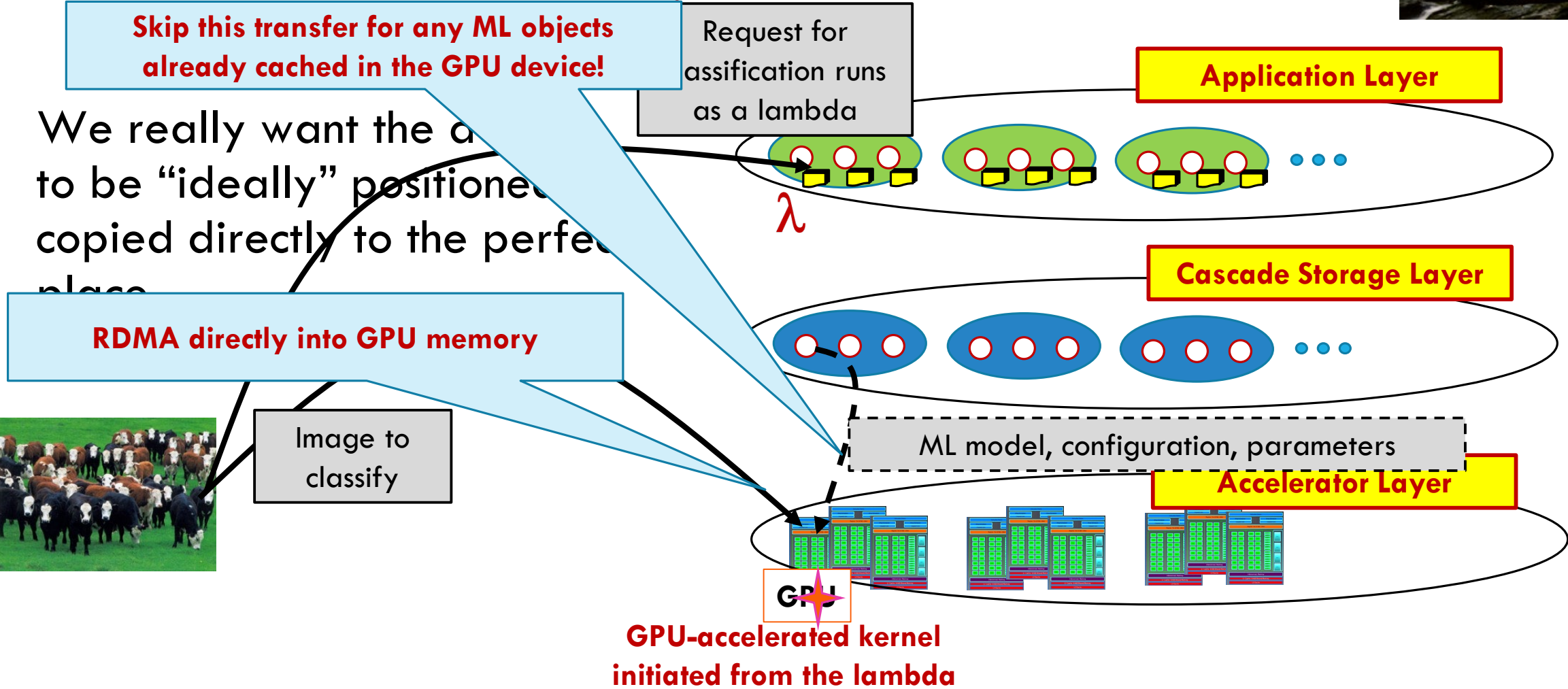
ML Model  
birth prediction

# C++ IS SIMILAR (BUT MORE EFFICIENT)





# INSIDE A CASCADE $\mu$ -SERVICE



# 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... but people don't want to use a C++ library these days. They work in higher level AI packages like Tensor Flow and Databricks, and those run on DHTs.

So, Cornell is now building Cascade: A DHT that leverages Derecho.