DERECHO: TOOL FOR BUILDING CLOUD/IoT SERVICES WITH GREAT PERFORMANCE

Motivated by AL/ML on or near the cloud edge
... requires low latency, high speed, accelerated computing.

But we also need high availability, fault-tolerance, consistency.

Derecho is easy to use.
It breaks performance records in this space!
A DERECHO: UNSTOPPABLE HIGH-SPEED WIND
TODAY’S PLAN

Derecho overview talk: Ken, 90m (depending on questions)

Derecho demo: Edward, 20m

Downloading and installing: Matthew, 20m

Discussion of APIs, demo: 20m

Challenge: Extend our demo in a small way.

Our TA: Lorenzo Rosa
## Papers for Full Details

<table>
<thead>
<tr>
<th>Title</th>
<th>Authors</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>RDMC: A Reliable Multicast for Large Objects.</strong></td>
<td>J Behrens, S Jha, K Birman, E Tremel.</td>
<td>IEEE DSN '18, Luxembourg, June 2018.</td>
</tr>
</tbody>
</table>
DERECHO IS A PLATFORM TO ENABLE MACHINE INTELLIGENCE FOR THE “INTERNET OF THINGS”

IoT devices simply don’t have enough power and lack the big picture.

Use the cloud-edge could host machine intelligence, enabling real-time reactivity using consistent, recently-acquired context.
RDMA (REMOTE DMA ON INFINIBAND/ETHERNET)

When available, RDMA is stunningly fast. Derecho leverages it.

But the system is also fast on datacenter TCP. RDMA not required.

... and we can run on other transports too, like OMNIPath
EXAMPLE WE WILL BUILD LATER, THEN EXTEND

First tier: inexpensive computation on meta-data

Key-value object store holds specialized knowledge models for categories (flowers, birds, dogs, trees…)

Flower $p=0.85$
Vegetable $p=0.6$

Most likely a sunflower!

Sunflower

Flowers

Vegetables

IoT Cloud Infrastructure

Sunflower $p=0.97$
Zucchini blossom $p=0.04$
In many emerging IoT uses, we would dynamically update ML models as the application learns from experience.

Many systems need to use the freshest knowledge, and may need a barrier at which “nobody is using the old data” any more.

As nodes join or leave our service, membership evolves.
THE SIZE OF IoT OBJECTS IS A SIGNIFICANT CHALLENGE

Videos: 100’s of MB, even GB in size (keep in mind that video compression is a slow computational task, so at the time we first acquire data it often will be “inflated” in size)

Photos: MB or 100’s of KB

Each extra copying step or storage step is costly!
YET, WE RELY HEAVILY ON REPLICATION

- High availability and fault-tolerance for valuable data
- The knowledge models are replicated in the shards. Updated versions need to be “re-replicated”, rapidly.
- The input photo is being replicated to the classifiers.
- The map of nodes (IP addr+resources to roles) is shared and must be updated each time nodes fail/join.
- The infrastructure itself replicates containers, files, etc.
New software framework for distributed programming (a C++ library)

With RDMA, 100x to 10,000x faster than other options. Fast on TCP too: 25x to 250x speedups

It can be used to build new / improve old cloud μ-services

Information-efficient Paxos/Atomic Multicast

Plus, an exceptionally efficient RDMA mapping

Zookeeper, HDFS, blob store (FFFS\textsubscript{v2}), BlockChains, DDS…
HARDWARE ACCELERATORS

ML systems work with big data and must leverage hardware.

Derecho was motivated by RDMA and NVM (3D-Xpoint/Optane)

Most recently, exploring efficient integration with GPU.
HARDWARE ACCELERATORS

We need a new service architecture that has minimal locking, zero-copy, and understands special hardware memory models.

This benefits your code, not just our code!
REVISITING OUR EDGE IOT EXAMPLE

IoT Devices

Google GRPC
Slow but universal

Vendor-supplied services (fairly rigid)

No Hardware Accelerators

Azure Functions

Image upload path

Azure IoT Hub

Blob Store

NoSQL Database

Image processing engine

...
REVISITING OUR EDGE IOT EXAMPLE

IoT Devices

Google GRPC
Slow but universal

Image upload path

Azure IoT Hub

Cow hoof-health evaluation microservice

Derecho is a tool for creating μ-services
REVISITING OUR EDGE IOT EXAMPLE

Image upload path

Hardware Accelerators and Machine-Learned Models Available. Managed by the “App Service”
CRITICAL PATH? MANY ELEMENTS!

Soon: Derecho coverage will encompass this entire space (but not the first hops to the client)

Today: Derecho Helps here
EVEN A SINGLE $\mu$-SERVICE COULD HAVE STRUCTURE

Outward-facing layer accepts Google’s GRPC (or whatever)

Knowledge repository

Perhaps this back-end works to refine and update the knowledge models
NOTICE THE SIMILARITY BETWEEN WHAT WE ARE CALLING SUBGROUPS WITH OBJECTS

In fact we built Derecho around “replicated objects”

The processes in a microservice run identical code... but define classes, and instantiate objects for each subgroup or shard that they belong to. They hold any replicated state.

Class methods are handlers for p2p and ordered_send requests
A LIBRARY? OR A SERVICE?

Derecho is provided as a **library**

But we are already using it to create stand-alone **services**:

- The **Derecho object store**: sharded service for key-value data
- The **Derecho DDS** is a pub-sub DDS layered on the object store
A LIBRARY? OR A SERVICE?

Derecho is provided as a **library**

But we are already using it to create stand-alone **services**:

- The **Derecho object store**: sharded service for key-value data
- The **Derecho DDS** is a pub-sub DDS layered on the object store

**HELP WANTED**

Other services that could be layered on the object store include Zookeeper, HDFS, Rsync, Ceph...
THE DERECHO EXECUTION MODEL

Virtual synchrony + atomic multicast or Paxos (you pick)
In the μ-service layer, an App Service manages services as pools of active instances. This (moderately dynamic) layer is our focus.

In many commercial products the set of servers is simply locked down and rarely changes. We allow this but don’t require it.
OUR MODEL: VIRTUAL SYNCHRONY “EPOCHS”

Membership is managed for you, and changes when processes fail, or a batch of them joins or leaves. Upcall notifies you.

Derecho automates layout into subgroups and shards, initializes joining processes, and repairs persisted state for restarts.

Model: state machine replication with multicast events, p2p queries, and self-healing behavior. Like “one-shot transactions.”
EPOCH MECHANISM IN ACTION

Derecho senses failure, swaps a warm-started process in, repairs any persisted data and initializes in-memory data.

Multicasts/Paxos updates always (appear to) happen in a single epoch…
WHAT MAKES IT AN EPOCH?

We cycle through modes:

- Reporting a membership view
- Running atomic multicasts or Paxos in an “unchanging view”
- End of the epoch (triggered membership changes)
- Efficient self-repair
- Switching to the next view, starting a new epoch

Switch to new epoch costs about 150ms
BENEFITS?

Clean, simple concept of current membership, roles.

Built in coordination with data updates, queries.

Automated recovery from any possible failure.
 USING DERECHO TO MANAGE DATA

Atomic multicast and Paxos in the library
THE BASIC IDEA IS SIMPLE

Decide if your data will live in an entire subgroup or a shard.

Decide if it should be persistent or volatile.

➤ **Persistent** data lives in files and will survive crashes.

➤ **Volatile** data lives in memory and can be lost if everything fails.
MODIFY DATA WITH STATE MACHINE UPDATES

To reduce locking, we separated our update path from queries

Updates (ordered_send) are state machine actions. A set of replicas pass through the same sequence of values.
class Foo {

    // Variables that define state of the Replicated<Foo> object

    // … Methods:
    void put(string s, double v){code...}
    const double get(string s){code...}
    REGISTER_RPC_FUNCTIONS(Foo,put,get)

};

auto outcome = fptr.p2p_query<Foo::rpc::get>(who, “John Doe”, 22.7);
auto outcome = fptr.ordered_send<Foo::rpc::put>(“John Doe”, 22.7);
class Foo_With_Data {
    Volatile<T> myObj_V; // Declares an in-memory variable
    Persistent<T> myObj_File; // Declares a persistent variable (in filesystem)
    Persistent<T, ST_SPDK> myObj_NVMe; // Declares a persistent variable (in NVMe). Coming soon.
}

// Program-level access to versioned storage:
auto x = *myObj; // Default: most current (myObj[NOW])
auto x = myObj[ver]; // Returns a specific version
auto x = myObj[time]; // Version that applied at specific time

myObj.Truncate(ver); // Discards versions newer than ver
myObj.Trim(ver); // Discards versions (0, ..., ver-1)
The Derecho-based object storage service leverages Derecho’s lock-free temporal accuracy and strong read consistency to offer these guarantees.
An ordered_send is used for updates and reaches all members of the shard. The handler sees the pending tail of the log (otherwise, “x := x+1;” wouldn’t work…).

A p2p query is sent to one replica, and sees **stable** data. In this example, data becomes stable when there are 2 replicas.
INSIDE A SINGLE SHARD OR A SINGLE SUBGROUP

If you prefer, you can think of every subgroup as sharded, but some as having just one big shard.
Inside Your Process

Each process has a single Derecho thread doing most of the work.

This allows Derecho to avoid locking...

Derecho’s core thread issues event-triggered upcalls to handlers you write. These must not block!

- Run an extra thread and hand off work to it if needed.
When processes join or leave, the layout function runs to assign roles and ensure that the new configuration will be “adequate”.

If successful, we end up with a new membership map to subgroup and shard assignments (replicated objects).

State transfer you saw initializes the replicated objects.
At first, P is just a normal program, with purely local private variables. P still has its own private variables, but now it is able to keep them aligned with the versions at Q, R and S.

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

```
Foo(1, 2.5, "Josh Smith")
Bar(12345);
Foo(1, 2.5, "Josh Smith")
Bar(12345);
Foo(1, 2.5, "Josh Smith")
Bar(12345);
Foo(1, 2.5, "Josh Smith")
Bar(12345);
```
LIMITATION

To use ordered_send, the sender must belong to the target subgroup or multicast

- We construct efficient RDMA data paths when the epoch starts
- This takes time and consumes resources

Thus you sometimes need to relay a request via P2P RPC
This client wants to query data at some point in time, and passes an index that the method will use to index the version vector.

c.RequestBar(who, 12345, Time = 10:31.05.476);
Client constructs a target list. The for loop issues the requests without waiting. Then the client iterates over results.

```cpp
for(auto t: {P,Q,R}) results.pushback(c.RequestBar(t, 12345, Time = 10:31.05.476));
```

```cpp
for(auto r: results) { process result r }
```
HERE A CLIENT USES A P2P RPC TO ASK P TO RELAY AN UPDATE REQUEST
A PROCESS FAILS

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

Derecho “trims” disrupted updates, like $X_{k+2}$

- Committed
- Now

Disrupted update automatically reissued
FAILURE DURING RELAYING IS TRICKY. DID THE ORDERED_SEND OCCUR BEFORE P CRASHED?

C knows that P failed, but can’t tell whether Foo was performed. This retry approach requires that the request be idempotent.
DERECHO CAN SUPPORT FANCY PATTERNS. HERE IS MAP-REDUCE IN A SHARDED GROUP

We obtain a completely atomic MapReduce primitive within Derecho!
FUTURE FEATURES (~2020)

We are considering extending the ordered_send API to be available in clients, relayed fault-tolerantly.

We are also planning to allow external (non-top-level group members) to use Derecho’s client APIs.

RDMA would be supported even for those external clients.
WHY NOT OFFER FAULT-TOLERANT “SHARD TO SHARD” ORDERED_SEND?

In fact this can easily be supported too.

But… APIs should be minimal, covering genuinely important stuff.

For example, MapReduce doesn’t need such an API: the MapReduce model has “end to end” fault-tolerance!
Memory Management

Underlying rules: RDMA requires that memory be pinned and registered. No automated marshalling occurs in RDMA hardware.
HOW DERECHO SENDS MARSHALLED DATA

An RPC-style call is marshalled: copying occurs

```cpp
auto outcome = fptr.ordered_send<Foo::rpc::put>("John Doe", 22.7);
```

Tag: `Foo::put` identifies the handler we are calling

John Doe: Data copied into message
22.7: Data copied into message

Issue: To use RDMA, the data must be in pinned and registered memory!
RECEIVING MARSHALLED DATA

void put(string& name, double& score) {
    do something with name and score
}

We don't copy on receive!

“John Doe”
22.7

Derecho-allocated memory region

Derecho’s event thread

RDMA

KEN BIRMAN (KEN@CS.CORNELL.EDU) 53
SENDERSIDE VS. RECEIVERSIDE DELAY

Any delay sending the next message leaves RDMA idle.

Once time is lost on the sender, we can’t recover it.

Receivers, on contrast, often have enough slack (“breathing room”) to absorb small receive delays. No RDMA channel backlog arises
Zero Copy Send Option

But you can also pre-allocate a message, avoiding copying:

```cpp
auto msg_builder = fptr.prepare_ordered_send<Foo::rpc::put>();
auto obj = msg_builder.build_arg<myClass>("Sally Smith", 31.2);
auto outcome = msg_builder.send(obj);
auto result = outcome.get();
```

Currently works only for `ordered_send`, but will also be supported for `p2p send` sometime soon.

This logic is off the critical path.

Derecho will free the object...
CONSEQUENCES OF SLOW HANDLER LOGIC?

Abstractly, the whole Derecho library on this node will be waiting for that handler to complete. But our logic is highly tolerant of short handler delays, and you won’t see any issue at all for this case.

Deadlock is a risk if a handler requires locks or does blocking I/O. Here delays can be quite long or unbounded. Avoid this.

Really slow behavior or unbounded delays cascade across nodes. Some other node may “kill” the slow node after a while.
EXAMPLES OF REASONABLE HANDLERS

If you wanted to keep a list of (name, score) tuples, you’ll need to copy the data into a new memory area. *Does a malloc + memcpy, but this is absolutely fine.*

In our GPU-accelerated photo analysis, we would need to DMA the photo into GPU memory from the handler. *Nonblocking, pretty fast.*

In our Paxos mode, Derecho writes updated data to SSD storage. *The synchronous disk writes were too slow, so we copy the data, then do an asynchronous write.*
EXAMPLES OF UNREASONABLE HANDLERS

Many Linux system calls can block. **Don’t do system calls in handlers.**

**Printing output** (even for debugging) is quite slow. Save debug info to a buffer but print it later, not in the handler.

Even with GPU, some ML tasks can take a few seconds. You might think of having the handler wait, then report “done”. **Don’t do this. A few seconds is WAY too long to sit in a handler.**
MOTIVATION: CONSIDER A SIMPLE RELIABLE BARRIER PROTOCOL ON RDMA

"Here is 100B message $m$"
"Deliver $m$"
"Garbage collect $m$"
A SIMPLE RELIABLE PROTOCOL ON 100G RDMA

“Ack”
“Ack”

TIMELINE, PROCESS P

0.75us + 100B/12.5GB/s = 0.750000008us

4.5us + 12 RDMA messages (limit: 75M/s)
A SIMPLE RELIABLE PROTOCOL ON 100G RDMA

Peak possible performance?

♦ Time to perform one 100B reliable multicast? 4.5us + “noise”
  ... based on time expended, limited to 222,222/s

♦ 12 RDMA verb operations out of 75M: limited to 6.25M/s

♦ RDMA could have transferred 36KB of data in 4.5u
  ... we left 99.8% capacity “unused”!

100B/12.5GB/s = 0.750000008us
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

Send continuously, as soon as new updates show up

 Receivers continuously report their acks, in an all-to-all pattern. This way every process can deduce delivery/garbage collection.
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 (Derecho’s RDMC or SMC multicast)

Control information exchanged continuously (one-sided RDMA writes via SST)
Derecho group with members \{A, B, C\} in which C is receive-only

**Derecho’s Data Plane = RDMC/SMC.**

**Derecho’s Control Plane = SST**

Data moved on RDMA multicast

Control is done using knowledge programming on the SST

<table>
<thead>
<tr>
<th>Suspected</th>
<th>Proposal</th>
<th>nCommit</th>
<th>Acked</th>
<th>nReceived</th>
<th>Wedged</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>4: -B</td>
<td>3</td>
</tr>
<tr>
<td>B</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>C</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>
SST CONCEPT

A simple table used to exchange control information or data
Derecho group with members \{A, B, C\} in which C is receive-only

Derecho’s Data Plane = RDMC/SMC. Derecho’s Control Plane = SST

A, B and C each have a replica of the SST

<table>
<thead>
<tr>
<th>Suspected</th>
<th>Proposal</th>
<th>nCommit</th>
<th>Aced</th>
<th>nReceived</th>
<th>Wedged</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>4: -B</td>
<td>3</td>
</tr>
<tr>
<td>B</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>C</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>

Data moved on RDMA multicast

Control is done using knowledge programming on the SST
**SHARED STATE TABLE (SST): DIRECT RDMA WRITES WITH NO LOCKING (SEQUENTIAL CACHE-LINE CONSISTENCY)**

Replicated at members

Update own row

Read-only copy of other rows

<table>
<thead>
<tr>
<th></th>
<th>Suspected</th>
<th>Proposal</th>
<th>nCommit</th>
<th>Acked</th>
<th>nReceived</th>
<th>Wedged</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>F</td>
<td>F</td>
<td>4: B</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>B</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>3</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>C</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>3</td>
<td>3</td>
<td>5</td>
</tr>
</tbody>
</table>

RDMA enables A to write directly to the replicas on B and C
DERECHO’S TWO DATA PLANES

SMC: A simple ring-buffer multicast implemented over the SST

RDMC: A Reliable Multicast for very large messages
SMC PROTOCOL

For small messages (< 10KB), or vectors of small messages, to shards or small subgroups.
In each process’ row, we set aside space for a ring buffer and a count of “available messages”

A uses one-sided RDMA to replicate its row to B, C. B sees the counter update, reads $m_{27}$, acks to A. A computes the minimum ack value and won’t reuse the slot until all receivers acknowledge that they are done. Here, C is lagging, so slot 27 won’t be reused yet.
All Derecho send primitives offer “urgent” semantics: the system sends as soon as possible.

For SMC, this minimizes latency but may make poor use of bandwidth.

To improve bandwidth utilization, send a small vector of updates in each SMC message, much like buffered file I/O.
SMC WITH 1-BYTE MESSAGES

Mellanox 100Gbps RDMA on ROCE (fast Ethernet)
100Gb/s = 12.5GB/s... 75M RDMA verbs/s

SMC can transfer up to around 10KB per message, but this 1B case lets us understand worst-case behavior.
RDMC PROTOCOL

For larger objects (even 100MB)

Works both in shards and in larger subgroups, tested with as many as 100’s of receivers.
RDMC: RDMA MULTICAST FOR BIG DATA

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”.
DERECHO’S CONTROL PLANE

Message ordering, stability.
CONTROL PLANE: PROTOCOLS “OVER” THE SST

Derecho group with members \{A, B, C\} in which C is receive-only

A, B and C each have a replica of the SST

Data moved on RDMA multicast

Control is done using knowledge programming on the SST

**Table:**

<table>
<thead>
<tr>
<th>Suspected</th>
<th>Proposal</th>
<th>nCommit</th>
<th>Acked</th>
<th>nReceived</th>
<th>Wedged</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>4: -B</td>
<td>3</td>
</tr>
<tr>
<td>B</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>C</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>

**Diagram:**

- Data moved on RDMA multicast
- Control is done using knowledge programming on the SST
DIRECT “ALL TO ALL” KNOWLEDGE SHARING

With one-way (“monotonic”) counters, suppose that P can ack to Q by just writing directly into Q’s memory (one-sided RDMA)

Now Q doesn’t wait for a message from P— it just looks at the memory cell to see how many messages P has received from it

Because of symmetry, we end up with a table!
We use a novel “monotonic logic” programming approach

- Coded using predicates that, once true, remain true
- Enables a highly asynchronous style of coding

Implemented in the SST
**CONSISTENCY: ATOMIC MULTICAST AND PAXOS**

**As an atomic multicast,** moves data from DRAM to DRAM
- Reliable, totally ordered, can use the “view” to coordinate
- Data purely in Volatile<T>

**As a storage solution,** also persists the data to SSD or NVM
- In this mode has the same properties as Lamport’s Paxos protocol
- Data in Persistent<T>
GUARANTEES?

We start with the same goals as with standard Paxos

- Total order: every process sees same events in same order
- Durability: No failure can cause the loss of a committed event

But then add one more guarantee: *Derecho will never enter a state in which data is “less replicated” than whatever minimum you specify.*

Let’s pause and look at performance, revisit the protocols later.
PERFORMANCE

Let’s see how well all this does, before we say more about the control plane and how it implements Paxos properties.
Derecho can make 16 consistent replicas at 2.5x the bandwidth of making one in-core copy.

Raw RDMC is faster, but performance loss is small.

100Gb/s = 12.5GB/s

memcpy (large, non-cached objects): 3.75GB/s
RDMA VERSUS TCP: RDMA IS 4X FASTER

Derecho Atomic Multicast: 100G RDMA

Derecho on TCP, 100G Ethernet
LATENCY: TCP IS ABOUT 125US SLOWER

Derecho Atomic Multicast: 100G RDMA

Derecho on TCP, 100G Ethernet
Derecho: Scaling (56G RDMA)

Large groups scale while maintaining a substantial percentage of their peak bandwidth.

With lots of small groups we see linear capacity growth.

Large groups scale while maintaining a substantial percentage of their peak bandwidth.

Large group of size N (2...128)
Limit was memory for buffering.

Broken into shards of size 2 or 3 linear aggregate throughout.
**DATA STORAGE: VERSIONED DATA**

Stores (k,value) pairs: put, get, and also watch

Data is versioned (and we can hold deltas rather than values)

Temporal access is precise but also causally consistent: like a snapshot isolation guarantee. You see only “committed” data

Derecho Object Store and DDS: Thin APIs on the versioned storage

Could support the Zookeeper API, or HDFS or even a log-structured file system API like Ceph
Performance is limited by the peak bandwidth possible with the SSD devices on our cluster… Our (fairly old, slow) SSDs “maxed out”. Derecho’s protocols are not a limiting factor.

Ramfs turns out to do several memcpy operations, and this limits performance.
VERSIONED STORAGE TAKE-AWAYS

... in both cases, the storage medium was the limiting factor

Derecho can deliver bytes far faster than RamDisk or SSD can soak them up, so performance looks flat.

Note: In this mode, Derecho is the world’s fastest durable Paxos!
DERECHO VS APUS (RDMA PAXOS), N=3
DERECHO VS LIBPAXOS, ZOOKEEPER, N=3 (ALL THREE CONFIGURED FOR TCP ONLY)
Derecho can hide behind familiar APIs.

- Services based on Derecho with their own APIs
  - Pub/Sub DDS
  - Key/Value Object store
  - File system: Ceph, HDFS or Zookeeper
  - Linux rsync

- Replication
  - Derecho Core (C++ [and Java])
  - RDMC + SMC + SST

- O/S Platform
  - For example, Ubuntu

- Hardware
  - RDMA + SSD or next generation NVM (TCP emulation of RDMA for backwards compatibility)
SOME REMARKS ABOUT SST PROGRAMMING AND THEORY

Is Derecho the “Ω” Paxos protocol?
REMINDER: DERECHO HAS A SINGLE PREDICATE-EVALUATION THREAD PER PROCESS

\[
\text{forever} \{ \\
\text{foreach} (P \text{ in predicate list}) \\
\text{If } P \text{ is true} \{ \\
\text{invoke trigger associated with } P \\
\}\}
\]
DERECHO’S PAXOS IS EXPRESSED ON THE SST PROGRAMMING MODEL

Lock-free, but we store monotonic values in the cells. If you miss some updates you can still deduce that they occurred.

Enables monotonic aggregation and even a monotonic form of knowledge-based reasoning ($K_A(P), K^1(P), \ldots$).

Result? Highly efficient batched receiver-side decision-making.
Predicates: like conditions in an if statement, but they include aggregates over the SST or even higher level “functions”.

For best performance:

- Predicates should trigger actions as soon as it is safe to do so
- Receiver-side batching: one decision covers multiple “events”
CENTRAL IDEA

In the past, we thought of protocols in terms of messages exchanged and leaders with special roles.

With modern hardware, it is better to think about knowledge exchanged and logical deductions.
If process A deduces $P$, we say A knows $P$ and write $K_A(P)$
$K_A(\mathcal{P})$ vs $K^1(\mathcal{P})$

Now suppose that every process in a shard containing \{A,B,C\} reports $K(\mathcal{P})$ via the SST (by setting an entry in its row to true):

$$(K_A(\mathcal{P}) \& K_B(\mathcal{P}) \& K_C(\mathcal{P}))$$

We write $K^1(\mathcal{P})$: Everyone knows $\mathcal{P}$

A process deduces $K^1(\mathcal{P})$ by aggregating $K_X(\mathcal{P})$ for all shard members
Derecho encodes Paxos as logic that reasons about knowledge and evolves through these deductive knowledge transitions.

Some decisions can be made symmetrically... we only need a leader when agreeing on the next view and cleaning up disrupted multicasts.

The huge wins are symmetric/asynchronous stability and safety deductions.
Example: the stability property used to commit Paxos updates:

- All messages from P:1 to R:17 have been acknowledged by all receivers and can be delivered (in order). Local deduction from $K_X(P)$
- State updates associated with P:1 to R:17 are stable. $K^1(P)$
WHY IS THIS “BETTER”?  

Participants decide independently, and many decisions cover batches of messages (receiver-side uncoordinated batching).

We also eliminate one round-trip delay (participants can make a decision without waiting for the leader to decide and share it).

Enables a streaming style of data flow.
WHERE ARE THE PAXOS QUORUMS?

Derecho uses a “virtually synchronous Paxos” model.

This has quorums, and uses a standard quorum Synod structure.
WHERE ARE THE PAXOS QUORUMS?

If any process ever suspects a majority, it shuts itself down
WHY DON’T **SHARDS** NEED QUORUMS, TOO?

The shard is only “active” if we have a top-level quorum

Additionally:

- Each new membership must support the desired layout (in Derecho, we use the term “adequate view”).
- To tolerate $f$ faults, a shard must have size $\geq f+1$, so that $f+1$ replicas exist (stability) before any delivery upcall occurs.
HOW EFFICIENT ARE THE PROTOCOLS?

Our protocols make decisions as soon as it is safe to do so. They are optimally efficient in a knowledge sense: no safe deductive path can be shorter (every message is needed)

In this sense our Paxos solution is theoretically optimal!
In theory, there is no difference between theory and practice. In practice, there is.
HOW EFFICIENT IS THE SYSTEM ITSELF?

Probably the biggest issue right now is the “first hop” delay.

Part of a broader issue: the O/S and runtime is full of locking and copying and uses TCP pervasively.

We dream about end-to-end lock-free zero-copy RDMA paths!
FINE... BUT HOW EFFICIENT IS DERECHO ITSELF?

RDMC and SMC map to RDMA with “ideal efficiency”. But...

- SMC has an urgent semantic... aggressive use of RDMA writes.
- In a shard of size 3, one SMC multicast uses 10 verbs. **Hence limited to roughly 7M/s on Connect X4 (switch limit 75M/s)**
- With a lazy (batching) semantic we could do better... but we might unpredictably increase SMC delay. And you can send vectors of requests (hand-implemented batching)
... HOW EFFICIENT IS DERECHO?

The Derecho critical paths are all as lock-free as possible, and as zero-copy as we can manage.

But even so, the SST predicate evaluation thread is a good example of a part of the system that could be improved.
MINIMAL DELAY: THE BIG GOAL

Only lock-free zero-copy code can run at full speed of hardware.

But we are still pretty far from true end-to-end lock-free paths even with Derecho in our service.

O/S for lock-free zero copy: the holy grail! Much work needed...

Awesome SOSP 2021 research topic...
SUMMARY

We’ve created the world’s fastest and most flexible C++ library for replication. The ultimate data replication story… for now…

Derecho is a high quality artifact that should be useful in the cloud O/S, in the IoT Edge and 5G Edge, or even in the network plane.

We welcome users, and are acquiring more and more of them. And we would also welcome more developers!