USING CORNELL’S DERECHO SYSTEM TO CREATE SMART DATA WAREHOUSES FOR THE INTERNET OF THINGS.

Ken Birman
Cornell University
THE FULL TEAM

Ken Birman: Project leader
Jonathan Behrens*: RDMA multicast protocol (RDMC)
Sagar Jha: Shared state table (SST), core protocols
Matt Milano, Edward Tremel: API and higher-level functionality
Weijia Song, Theo Gkountouvas: Volatile<T> and Persistent<T>
Robbert van Renesse: Consulting Research Professor

* Behrens is now at MIT pursuing his PhD
THE INTERNET OF THINGS: A BIG DEAL

Smart power grid…
Smart highways to guide smart cars…
Smart homes

Huge market

Scale demands a cloud with special properties
EXAMPLE: “SMART” POWER GRID

Today’s grid uses mostly 1980’s technology

Human-controlled... robust, but struggling with renewable energy.

Core idea: Machine learning could bring transformative flexibility
IOT DIMENSION: MICROGENERATION

Solar panel on the rooftop

Battery in the wall

Heater or A/C unit that we can activate early or delay

Think of “delayed power use” as a new form of power: “schedulable” demand!
SMART METERS

In-house control center for all these fairly dumb devices

Knows what’s going on, regulates power consumption for heavy power-use devices like A/C, heater, etc.

Ideally, the “endpoint partner” for the more centralized cloud-based optimization that works to balance supply and demand
The intelligence lives in the cloud: machine learning that watches the real-time data stream in and reacts.

The external infrastructure is mostly dumb sensors and actuators.
GRIDCLOUD... TOMORROW

Today, could monitor/control/archive the full USA in real-time (bulk grid), or a large community “distribution” network.

.... But our user community will eventually want to conduct studies on archived data, including “temporal forensic analysis”, and also watch for and react in real-time to various phenomena

- Such needs will arise in many IoT settings
- Far beyond what could be done today with GridCloud
CONSISTENCY CONCERNS ARISE, TOO

Real-time data captured three ways. Inconsistencies are like noise. Machine learning in the cloud needs strong consistency.
HOW THIS WORKED

We captured 400 feeds of data, appended to files @ 10 fps
Then queried the data at a slower rate: 2 fps
HDFS snapshots mix data from different times, violate causality
Our Freeze Frame FS is temporally precise and causally consistent, because it attaches a hybrid logical clock (HLC) to each update and uses a consistent query method.
(Keep this idea in mind for later… we’ll see it again)
MACHINE LEARNING NEEDS CONSISTENCY!

One issue is real-time consistency
- After an event occurs, it should be rapidly reported in the cloud
- Any application using the platform should soon see it

Another issue is replication consistency
- Replicate for fault-tolerance and scale
- Replicas should evolve through the same sequence of values, and data shouldn’t be lost in crashes
- System configuration should also be consistent, such as mapping of processes to their roles

11
SMART MEMORY: QUERYING A SCALABLE, ASYNCHRONOUS DERECHO SERVICE

Massive inflow of real-time data

Hashing the video-id maps streams to shards.

A smart memory interprets data using machine learning and analytics, then stores findings in temporal version-vectors.

Processes communicate to compare data from distinct video streams.

Within a shard, Paxos state-machine replication guarantees consistency, durability, fault-tolerance.

Query: motorcycle at 10:03.22-10:03.56am?

Temporal queries access versions within the histories, returning precise, causally-consistent results. This query accesses three shards (the green ones)

Movie is generated by combining sub-results
EACH SHARD HAS JUST PART OF THE DATA. THE SERVICE SCALES BECAUSE WORK IS SPREAD OUT.

Data is acquired by a non-blocking pipeline. Data is saved in time-stamped version-vectors, which are replicated for high availability. All the shard members are active.

Queries occur out of band, and won’t disrupt data acquisition.
COULD CLOUD “EVOLVE” TO DO THIS?

Existing platforms have great real-time responsiveness only when serving requests at the edge, from what could be stale data.

Inconsistency is pervasive, CAP is a kind of strange folk religion.

Fault handling and dynamic reconfiguration can provoke waves of inconsistency and performance variability that may last for minutes. Our goal: Fix these things, but build on standards.
Our premise: as cloud continues to expand and to engage with the Internet of Things, developers need new tools to make it easy to create “online” cloud services that will interact with things.

- Need high availability, real-time responsiveness
- Consistency, fault-tolerance, coordinate at multiple sites
- Clean integration with today’s machine-learning tools
SMART MEMORY: QUERYING A SCALABLE, ASYNCHRONOUS DERECHO SERVICE

External clients use standard RESTful RPC through a load balancer

Volatile copies in cache
Persisted versioned data
HARDWARE OPPORTUNITY

Our large-scale software library goal coincides with an exciting hardware development.

RDMA: Network can do DMA transfers memory-to-memory at line speeds with no software involvement during the I/O.

Utilization of RDMA promises incredible speedups.
RDMA RELIABLE UNICAST

RDMA: Direct **zero copy** from source memory to destination memory

On 40 Gb/s network RDMA reaches 36Gb/s. Unless used very carefully, TCP/IP is more like 250Mb/s... with care, maybe 10Gb/s

Like TCP, RDMA is reliable: if something goes wrong, the sender or receiver gets an exception. This only happens if one end crashes
Open source, for Linux platforms. **Shatters performance records.**

Simple API aimed at C++ programmers (but DLL can be imported and used from Java, C#, etc)

Covers everything to make your cloud-service development job easy and quick.
A PROCESS JOINS A GROUP

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 track the versions at Q, R and S.

... 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.
CLIENT MAKES AN UPDATE REQUEST

Client not in the group uses p2p send/query to issue requests.
**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 $X_{k+2}$.
A MULTI-QUERY

All members compute, then send back a share of the result. Members subdivide the computational task (e.g. 1/4th each)
A MULTI-QUERY

In fact we support two cases:

- Within a subgroup/shard (can leave side-effects, can see the state left by the most recent state-machine updates)

- Add hoc target list: can query *any list of processes at all*

- With a target list, query specifies the time at which the query is being done, hence might not see the most recent updates.
Client not in the group asks the group to do a query. A proxy forwards the query, collects the result, and forwards it back.

RequestBar(12345);
CLIENT QUERY: AD-HOC CASE

Client constructs a target list, but also specifies the time

RequestBar(12345, Time = 10:31.05.476);
WHERE DOES DATA ACTUALLY RESIDE?

In Derecho, the Group is a collection of replicated objects, of type Replicated<T> (you get to define class T).

In your class, you can have stateful objects, just like in normal C++, except that multicast will be used to update these data fields, keeping them consistent.

We support two cases: in-memory (volatile) and persisted.
In Derecho, replicated data fields are automatically versioned and the system can fetch versions on demand. Updates are pending until fully replicated, which we report via a “committed” event (an upcall). In persistent case, also saved on disk.

```cpp
class Foo: Replicated<Foo> {
    Volatile<T> myObj; // Declares an in-memory variable
    Persistent<T> myObj; // Declares a persistent variable

    auto x = myObj; // Default: most current (myObj[NOW])
    auto x = myObj[n]; // Fetches version n
    auto x = myObj[-1]; // Returns the previous version
    auto x = myObj[time]; // Version that applied at specific time

    myObj.Flush(); // Delays until prior updates commit (takes ~10us)
    myObj.Truncate(n); // Discards versions 0..(n-1)
}
```
NEW VERSIONS ARISE FROM TOTALLY-ORDERED STATE-MACHINE UPDATES

An update is a totally ordered multicast event. Update occurs first, then after every member has “stabilized” it, we commit it.

Every update creates a new version of every object... The handler is called in the same order at every group member

Version vector is compact (uses a deduplicated representation)
TEMPORALLY INDEXED ACCESS IS CONSISTENT, FAST, FLEXIBLE

No matter what processes are queried, even in totally unrelated subgroups or shards, temporal indexing always is:

- Temporally precise (as accurate as feasible given your clocks)
- Strongly consistent (closed under causal ordering $\rightarrow$)
- Deterministic (same request always returns same result)
- Extremely fast (first access may need to search, but second is satisfied from memory)
QUERIES VIA VERSION VECTORS

Much like queries with “snapshot isolation”
Model is read-only for queries, append-only for updates

Application can query any set of Derecho nodes at all, irrespective of the group structure and shard structure!

- Indexed reads eliminates need for totally ordered multicast
- Query will be done without blocking, with strong consistency
EXAMPLE: MAKING OUR MOVIE

Highway data would be sharded: shard X has videos for certain areas of the California highway system.

When motorcycle is sighted, we can guess at which shards might have cameras that could have “glimpsed” it passing.

Send query to one representative per shard, then merge results, using a filter that selects best snippet for each second.
OTHER ASPECTS OF PROGRAMMING API

Extremely simple API focuses on
- Point to Point Send, RPC-style Query
- Multicast Send and Query

Yet we can cover all of these cases, and moreover, by working in C++ 14, obtain super-efficient marshalling, polymorphism
Replicated<MemCacheD>& cache = g.get_subgroup<MemCacheD>(0);
auto outcome = cache.p2p_send<MemCacheD::RPCTag::put>(who, “John Doe”, 22.7);
auto result = cache.p2p_query<MemCacheD::RPCTag::put>(who, “Holly Hunter”);

class MemCacheD {
    Volatile<std::map<string,double>> quick_index;
    Persistent<image::jpg> photo_archive;
    void put(string s, double v){code...}
    const double get(string s){code...}
    auto register_functions(RPCManager& m, unique_ptr<MemCacheD>* this_ptr) {
        return m.setup_rpc_class(this_ptr, &MemCacheD::put, &MemCacheD::get); }
    enum class RPCTag { put, get }
};

Group<MemCacheD, CacheLayer ,…> g{sg_info, [MemCacheD factory], [CacheLayer factory] …};
Group<MemCacheD, CacheLayer…> g(group_leader_ip, subgroup_info);

Replicated<MemCacheD> cache = g.get_subgroup<MemCacheD>(0);

auto outcome = cache.ordered_send<MemCacheD::RPCTag::put>("John Doe", 22.7);

auto results = cache.ordered_query<MemCacheD::RPCTag::get>("John Doe");
for(auto future : results) { code… }
A “layout” method (not shown on these slides) maps membership of each view to generate subgroup and shard membership.

We provide some standard versions that can be customized, they are smart about issues like minimizing churn.

The complicated picture we’ve used is generated by one of them.
Cache Layer
Back-end Store
Multicasts used for cache invalidations, updates

Load balancer
External clients use standard RESTful RPC through a load balancer

... GENERATING THIS COMPLICATED SERVICE!
CONSISTENCY: A PERVERSIVE 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.
**DERECHO = RDMA MULTICAST + CONTROL**

Core idea:

1. Create a multicast using RDMA unicast to move the bytes.
2. Run the data stream asynchronously, out of band from the control layer. But superimpose a control plane that watches the data and notifies the application when it is safe to deliver messages.
3. The notifications will be in identical order at all replicas.
RDMC: AN RDMA MULTICAST
Derecho is a multicast and paxos

As a multicast, moves data from DRAM to DRAM at insane speeds
- Reliable, totally ordered, can use the “view” to coordinate
- Data purely in Volatile<T>

As a storage solution, moves data to persisted storage on receipt
- In this mode has the same properties as Lamport’s Paxos protocol
- Data in Persistent<T>
Latency for sending a message of size 256MB (left) and 8MB (right) transfers, on our Fractus cluster (100Gbs RDMA). Derecho is more efficient with larger messages.
How time breaks down: mostly, data transfer “work”, although there are some pauses. Notice how a scheduling delay at a sender impact downstream receivers.
**SMALL MESSAGES: SPECIALIZED PROTOCOL**

Binomial pipeline was too slow with small messages, so we created an optimized small-message protocol: the “SST multicast”

Speed would be unchanged up to about 1KB (due to underlying RDMA “block size”)
Next, we created groups of size $N$, varying $N$, and had 1, half or all members send at peak rate. Performance is very steady.
CONCURRENT TRANSFERS: OVERLOAD TOR NETWORK

Same experiment.

AptLab has an oversubscribed TOR: per-link bandwidth slows to 18.5Gbps under peak load. Now RDMC slows as it hits this limit.

With topology information, could run RDMC once at the TOR level and again in each rack. That would reduce TOR stress.
HOW FAST IS DERECHO?

Raw RDMC multicast via Derecho API

Atomic Multicast (Vertical Paxos)

100Gb/s = 12.5GB/s
LARGE GROUPS?

Mellanox 64Gbps RDMA on Infiniband

64Gb/s = 8GB/s

One large group

Shards of size k

![Graph showing bandwidth vs group size](image)

![Graph showing aggregated bandwidth vs top-level group size](image)
When configured to persist to SSD, the SSD itself is the bottleneck, or the “RamDisk” in volatile<T> mode.

We can deliver bytes far faster than SSD can soak them up, so performance looks flat, at the SSD DMA rate.

Note: In this mode, Derecho is the world’s fastest version of Paxos!
**VOLATILE**<T>** PERSISTENT**<T>

![Graph showing bandwidth (Gbps) vs. group size for volatile and persistent data types with different group sizes and data sets (1K, 1M, 100M)].

- **VOLATILE**<T>:
  - vola all 10K
  - vola hal 10K
  - vola one 10K
  - vola all 1M
  - vola hal 1M
  - vola one 1M
  - vola all 100M
  - vola hal 100M
  - vola one 100M

- **PERSISTENT**<T>:
  - pers all 10K
  - pers hal 10K
  - pers one 10K
  - pers all 1M
  - pers hal 1M
  - pers one 1M
  - pers all 100M
  - pers hal 100M
  - pers one 100M

(Insert graph showing the bandwidth (Gbps) vs. group size for volatile and persistent data types with different group sizes and data sets (1K, 1M, 100M)).
RDMC is like a torrent of data pouring down a flooded river. Reliable (unless something crashes; we’ll discuss that soon) but at a crazy, insane data rate.

Derecho needs stronger Paxos-style properties, but without losing all this awesome speed.

Key idea: Asynchronous “control” plane that uses monotonic properties to express goals, and runs out of band from the data.
SO HERE’S THE IDEA…

Build a little tool to let us attach strong properties to our raging torrent of data

We call this the SST (shared state table). It uses RDMA too.

Then re-express class protocols as stable predicates using the SST for the variables (the shared state) needed to sense strong properties like Paxos safety (we have other use cases in mind too)
SHARED STATE TABLE (SST)

A simple shared table implemented with RDMA

Key idea: each element of a rack-scale system owns one row in the table. The row format is defined by a C++ 11 struct that has well known sizes, “plain old data”

If you update “your” row, the data is pushed to other nodes

Then we support events: “on (predicate) do { function }”

- Current version: close to 2 million events per second with 6 nodes
- Runs on one-sided RDMA writes (or reads, but writes are faster)
- Same memory model as for “volatile” variables in C++
HOW THE SST WORKS

State in process P:

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>17</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

State in process Q

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

P updates its row (local to P). Then uses RDMA to send it to Q.

Q has a read-only copy of P's row and eventually sees the update.
HOW THE SST WORKS

Q saw P’s update and updated column Y in its own row
Then pushed the change to P, which will eventually see it.
SST IS LIKE A 3-D TABLE

We think of the SST as “one table”

Actually, each member has its own replica

Updates are concurrent so replicas aren’t in the identical states, or even pass through identical states. Also, they change faster than a single process can actually “track”
… just like the term sounds: “things that remain true, once they are established.”

- Example: if a counter is greater than 10, it remains greater than 10 even if incremented. Value advances monotonically.

- Another example: if a replicated object has been safely delivered to all its destinations and persisted by them, and the ordering is agreed upon, this is a “stable” property: Once safe, always safe.

Derecho employs only properties that are not just stable but also monotonic: if they hold for K, they hold for values < K
Derecho is a group communication infrastructure

Programs can attach to groups ("join") them

Data is replicated by doing a group send operation
**IMPLEMENTATION: DERECHO = RDMC + SST**

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

\[ V_3 = \{ A, B, C \} \]

DNS record and current view, showing senders

<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>

DNS: myGroup.somewhere.org | 123.45.67.123:6543 (Leader: A)
PAXOS AND ATOMIC MULTICAST PROTOCOL

For multicasts, we deliver in round-robin order, rotating through the senders.

- This delivery upcall creates new data versions
- A second upcall occurs soon after: “committed(n)” to tell the application when prior delivered events have stabilized.
- We use the SST to report how many stable versions each process has, and we commit \( n \) when the minimum reaches \( n \).
VIRTUAL SYNCHRONY EPOCHS

After a crash, we aggressively gossip “failure suspicions”. The SST row of the crashed processes will be “frozen”.

- Shuts down unless a majority of top-level members are healthy.
- Then we wedge the current Derecho configuration.

Pick the lowest ranked non-suspected member as the leader.

- For pending updates, it decides which should commit
- Then notifies all processes, which echo the notification
THINGS WE CAN PROVE

We have proofs that this protocol offers guarantees matching the Paxos guarantees (if durable), or vertical Paxos (if volatile).

There is a simple, correct, way to restart from total-failure.

Message deliveries occur after the minimum number of message passing events required for Uniform Agreement (Consensus).

Progress with the ♦P failure detection.

… establishing that Derecho is a constructive lower bound for UA
USE OF HARDWARE IS HIGHLY EFFICIENT TOO

RDMC tree is (basically) a binary fanout, which is optimal.

We have a way to hand off requests to the RDMA NIC, as if we compiled the data transfer into hardware.

Derecho polls for RDMA completions only while RDMC and SST activity is occurring. When idle switches to interrupt-driven mode, hence zero CPU load when no Derecho traffic is active.
 HOW FAST IS DERECHO? 

Using Mellanox 100Gbps RDMA on ROCE (fast Ethernet)

100Gb/s = 12.5GB/s

Raw RDMC multicast

For the 10KB case, Derecho automatically switches to use its optimized small messages protocol

Speed of a single-core memcpy: 3.75GB/s

Not shown: Derecho adds 10-50us of delay before delivery
MULTICAST VS PAXOS

In our Derecho formalism, the only difference is that with Paxos, we need to persist data (for example to SSD disk), whereas a multicast is identical but the data lives only in memory.

When configured to persist to SSD, the SSD itself is the bottleneck.

Our SSD has data-type dependent behavior.
**Derecho as Paxos: Limited by SSD Speed**

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Speed of SSD (MB/s)</th>
<th>Derecho (MB/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>YouTube video (mp4)</td>
<td>67.7645</td>
<td>63.2405</td>
</tr>
<tr>
<td>Random bits</td>
<td>72.1108</td>
<td>67.6758</td>
</tr>
<tr>
<td>Facebook web page</td>
<td>126.451</td>
<td>112.909</td>
</tr>
<tr>
<td>Repeating text data</td>
<td>457.55</td>
<td>300.398</td>
</tr>
<tr>
<td>All Zeros</td>
<td>471.104</td>
<td>292.986</td>
</tr>
</tbody>
</table>

**Guarantees**

Guarantees are exactly the same as for Paxos (Corfu).

**Basically, a replicated persistent log on SSD storage units**

4-member group, 4MB objects
Derecho would scale better than Corfu

Best known of prior log-based Paxos solutions was Corfu.

Derecho is 2x faster than Corfu in our lab, but also can create large numbers of replicas with little slowdown; Corfu lacks such a capability (in fairness, the need doesn’t arise).

Corfu will slow down with large number of replicas because its underlying replication protocol doesn’t scale as well as RDMD.
Ad-hoc queries are sent using parallel RPC, will use RDMA but not RDMC. This is quite fast if the argument lists are small.

Better to create a suitable subgroup if arguments include large objects, and pattern of queried nodes repeats frequently
The file system is a secondary way to access version vectors.

Enables applications to store data in a Derecho service, but then to access it from file system snapshots:

- Snapshot looks like a read-only directory named by the time.
- Standard POSIX feature, widely used with HDFS or Ceph.

Allows existing machine learning apps to be used unchanged.
VISION: FAMILIAR TOOLS OVER DERECHO

API layer
- File system: Ceph or Zookeeper
- Pub/Sub: Kafka
- Pub/Sub: Rabbit
- Management: CM

Replication
- Derecho
  - RDMC + SST

Containers
- Docker + Mesos or Linux or QNX

Hardware
- RDMA, SSD or next generation NV memory
  (SoftRoCE for backwards compatibility)
WIDE AREA GOAL

We want to experiment with Derecho over SoftRoCE and SoftiWarp, which offer the Verbs API but run over standard IP.

May also need to borrow an old proactive erasure coding idea from the Smoke and Mirrors File System, but should be able to offer a high speed solution over fast WAN links too.
KEY TAKEAWAY?

The Internet of Things revolution is extremely promising for all of us.

Huge amounts of data will need to be captured, persisted. Incredible demand for speed and real-time. **The cloud can do it!**

Derecho bring a new generation of blazingly fast programming tools for people building services for this new world.
WHY NOT JOIN US?

The Internet of Things is coming... and will bring extreme scale

Everyone needs speed and consistency.

Derecho is an open-source project and with help, we could make it a universally adopted solution for the full space of replication in scalable settings. *All of us win if a standard emerges!*

Download Derecho from GitHub.com