CONSIDER A SMART HIGHWAY

We have lots and lots of sensors deployed

Cars are getting some form of “guidance” and if they accept it (and maybe pay a fee) get to drive faster.

Would we run into consistency issues of the sort seen in Lecture 6?
SMART HIGHWAY

**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
**TRACKING TRINITY…**

In this example, we are doing a few things in one picture:

- Data is being captured by IoT sensors.
- We are relaying it into a key-value storage layer, and saving it in some sort of sharded, replicated form.
- A “query” is pulling up images that show Trinity with the KeyMaker on her motorcycle.
REMINDER: FLOCK OF GEESE

In the last lecture we saw how the concept of a causal snapshot can help us create consistent views of a distributed system.

Can we use that same idea here?

Goals: We want temporally precise and causally consistent data, and then will search it for clear images of Trinity’s ride.
ANIMATION: A WAVE IN AN AQUARIUM

To illustrate this point visually, we made a simulation.

Rather than a flock of geese, it simulates a wave in an aquarium, as if 400 cameras were watching the water, each sending 20fps. We captured this “IoT sensor data” into files.

Then we took snapshots at a rate of 5fps and made a movie.
Existing file systems (like HDFS on the left) make mistakes when handling real-time data. But we can fix such problems (right).
WHY IS THE ONE ON THE RIGHT “BEST”? WELL... GARBAGE IN, GARBAGE OUT

Many machine learning systems are “tolerant” of noise, but HDFS was way worse than just noisy: it was inconsistent!

We might not trust the system when it tracks Trinity.

Inconsistent inputs can defeat any algorithm!
SMART SYSTEMS NEED CONSISTENCY!

As we saw, one dimension concerns *time*

- After an event occurs, it should be rapidly processed
- Any application using the platform should see it soon

Another centers on *coordination and causality*

- Replicate for fault-tolerance and scale
- Replicas should evolve through the same values, and data shouldn’t be lost
FREEZE FRAME FILE SYSTEM (FFFS\textsubscript{V1})

This was created by our TA, Theo, with Weijia Song!

The idea was to bring ideas from Lamport’s models into a file system so that the end-user (you) could benefit without needing to implement the mechanisms.

He took advantage of the fact that HDFS has a snapshot API, even though it didn’t work. FFFS “reimplements” this API!
HOW DOES IT WORK?

Normal file systems only store one copy of each file.

FFFS starts by keeping every update, as a distinct record. The file system state at a particular moment is accessed by indexing into the collection of records and showing the “last bytes” as of that instant in time.

So FFFS looks just like a normal file system to its users.
HOW DOES IT WORK?

Next, just like in our space-time figures, FFFS tags every record with a special kind of timestamp.

In our examples we used logical clocks and vector clocks.

FFFS actually uses a *hybrid clock*. This includes the IoT timestamp from the sensor, the platform timestamp from a clock, and a causal timestamp from a logical clock.
HOW DOES IT WORK?

Even though FFFS has multiple servers (in fact data spreads over them using the same key-value sharding discussed in Lecture 2), for an access at time T (you open “filename @ T”):

- It accesses data accurate for time T, despite clock skew
- It tracks causality, so that if it returns Y for some read, and update X → update Y, then it also returns X.
- In effect, FFFS does temporal reads along a consistent cut.
In effect, each time your application does a read from a set of files, that operation occurs along a consistent cut that:

- Is as accurate as $\text{FFFS}_{v1}$ can make it, given clock precision limits
- If $T' \geq T$, the cut for $T'$ includes everything the cut for $T$ included
- If you read multiple files, the results are causally consistent
- Reads are deterministic (other readers see the same data)
IN OUR HIGHWAY EXAMPLE?

When we query, we want the machine-learning tool to see data as a series of consistent snapshots across the full data set.

Then it can select data that includes video-snippets of Trinity with exactly one snippet per unit of time, no overlaps, no “lies”.

Thought question: How does the overlap issue relate to sensor overlap from the Meta system, discussed previously?
REVISIT THE SMART HIGHWAY

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
BEYOND FFFS$\textsuperscript{v1}$

A file system is not a natural API to use if the way you think of the application is through key-value data.

So for Azure IoT, as part of a system called Derecho, also invented at Cornell, we are building FFFS$\textsuperscript{v2}$. It will be inside Derecho and looks like a key-value storage layer for “objects”.

But in fact it can do anything FFFS$\textsuperscript{v1}$ could do.
DERECHO: WHAT IS IT?

Derecho is an open-source tool for developers creating new cloud µ-services. Download from GitHub.com/Derecho-Project

Derecho leverages RDMA to gain exceptional speed, but can map to TCP if RDMA isn’t available.

Currently targets C++ developers in Linux cloud environments like Azure IoT Edge, Azure Intelligent Edge, and Amazon AWS.
First tier absorbs much of the load

Back-end handles complex tasks

Notifications, cache-invalidations use multicast too.

Incoming traffic: RESTful RPC, WCF, other TCP-based protocols, etc.

This is just one example.

The developer defines the subgroups, controls the layout, tells us what pattern of sharding to use, etc.
ROLES DERECHO IS PLAYING

Derecho:
1. Automates the “mapping” from processes to these roles
2. Auto-instantiates C++ types for subgroups & shards
3. Fault-tolerant, repairs damage so data is always consistent.
4. Employs ultra-fast multicast (Paxos) for updates.
5. Offers a unique new read-only query mechanism that is consistent and can index in time, accurate to milliseconds.
In the market, you find key-value stores like Cassandra, but with weak consistency and lacking these time-indexing capabilities.

Others, like Microsoft FaRM, are proprietary tools for specific needs (FaRM for example supports Bing’s transactional queries and updates in a massive shared memory).

You also find file systems implemented over a layer like Derecho (think of Zookeeper), but they are slow and used mostly for configuration management.

Derecho’s object store lives in this space, but is both *simpler* and *more powerful*.
OBJECT STORE API (FFFS\textsubscript{V2})

Extremely simple: Stores any kind of “binary information”

- \texttt{Derecho::ObjectStore::put(key, value)}
- \texttt{Derecho::ObjectStore::get(key[, time])}
- \texttt{Derecho::ObjectStore::watch(key, Callback f)}

The key could be a string, an integer, even a complex object.
The value could be a byte array, a photo, a video, and can be huge.
We take a subgroup, shard it (for example, 2 replicas)

- **put** maps your key to some shard. It holds the key,value pairs

  - Replication uses an atomic multicast based on Paxos, so all copies are in a consistent state.

  - There is just one “most current” value, held by the store.

- **get** will fetch this most recently stored value.

- **watch** uses multicast to inform any watchers each time value changes.
STORING THE DATA

For the simple case, a normal C++ "Map" is used.

This is a hashed lookup structure employing a very efficient in-memory data layout.

A single key can only map to a single value. If you replace it, the old value is garbage collected.
SPEED? BREAKS RECORDS!

Every data path uses RDMA transfers (100Gbps bidirectional)

Derecho multicast is exceptionally fast and even with multiple copies, the entire update runs at nearly 125Gbps

Note: In today’s Derecho, RDMA is offered inside a µ-service, but in 2019 we will also have RDMA for clients outside the service.
WHAT ABOUT BIG OBJECTS?

If objects are large, and watchers just want “some” objects, we recommend a simple two step approach:

- Create a uid, and **put** the (uid, obj) pair, first.
- **put** a “meta-data” record that lists the uid.
- Now, via **get** or **watch**, clients learn about the update from the meta-data, which can list various attributes.
- They call **get** a second time to fetch object, if desired.
VERSIONED OBJECTS

We configure the object store to track versions. `put` creates a new version:

- `key`: The object store always tracks information on a per-object basis
- `version-number`: Just an integer
- `time`: If the object itself lacks a timestamp, we just use “platform” time.

Now `get` can lookup most current version, or a specific one, even by time. The object store is optimized to leverage non-volatile memory hardware.
SEQUENCES OF VERSIONS

With concurrent applications, you could worry that someone will do a get, then compute a new version, then put. But if some other process simultaneously does the same thing, one could overwrite the other.

In fact there is a feature in the object store to ensure that if this kind of race condition were to occur, put instead “fails” so that you can repeat the get and try again. You get consistency without any need for locking.
STORING DELTAS

Existing DHTs lack support for versioned data.

We implemented a highly optimized versioned data structure.

We implement a temporal index, and cache frequently accessed data.

- A server still manages a map (since many keys map to it), but you can think of the values for a specific key as being versioned.

- Sometimes deltas are more efficient. If you have a function to compute the delta, we won’t even create a new version unless you tell us to.

- Values (or deltas) are saved on NVMe & replicated for fault-tolerance.
OBJECT STORE AS A FILE SYSTEM

A file system is really an abstraction over a block store.

We plan to offer the Ceph object-oriented enhanced Posix API.

- Here we configure the object store to be versioned and persistent.
- Paxos for fault-tolerant updates, guaranteeing consistency.

This will offer back-compatibility, but for peak speed users should still use put/get/watch
We can implement “publish” using `put`.

- Acts as a message bus in the non-versioned case
- Acts as a message queue in the versioned mode.

... and we can support “subscribe” using `watch`

Thus the object store can support pub-sub APIs such as the OMG DDS specification, Kafka, OpenSplice, etc. We can also offer message queuing APIs such as the Azure or AWS queuing services.
STATEFUL OR STATELESS?

Configured to not store anything, we get a pure notification, sometimes called “topic-based publish-subscribe.”

Configured to store the most recent value, a new subscriber can see the most recent posting, then gets notifications for updates.

Configured to track versions, we have a true “queuing” service, acting like a mailbox. A subscriber can replay old data, or we could use replay as a debugging/auditing tool. But the object store has a truncate API that can be used to force it to discard old data.
SOME PERFORMANCE GRAPHS

From our TOCS submission
Mellanox 100Gbps RDMA on ROCE (fast Ethernet)

100Gb/s = 12.5GB/s
LARGE MESSAGES

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.

Raw RDMC is faster, but performance loss is small.

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

RDMC: our cheapest 1:K reliable RDMA send

Derecho Atomic Multicast (Vertical Paxos)
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 (56GB/S RDMA)

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

With lots of small groups we see linear capacity growth.

LARGE GROUP OF SIZE N (2…128) LIMIT WAS MEMORY FOR BUFFERING

BROKEN INTO SHARDS OF SIZE 2 OR 3 LINEAR AGGREGATE THROUGHOUT
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.
... 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)
CONCLUSIONS

Derecho is up and running solidly now, and can be used on pure TCP systems as well as on RDMA-enhanced ones. Our project site and collaboration hub is GitHub.com/Derecho-Project/.

The new object store offers a very simple and flexible C++ API.

It is a good example of how Lamport’s ideas can look in practice.