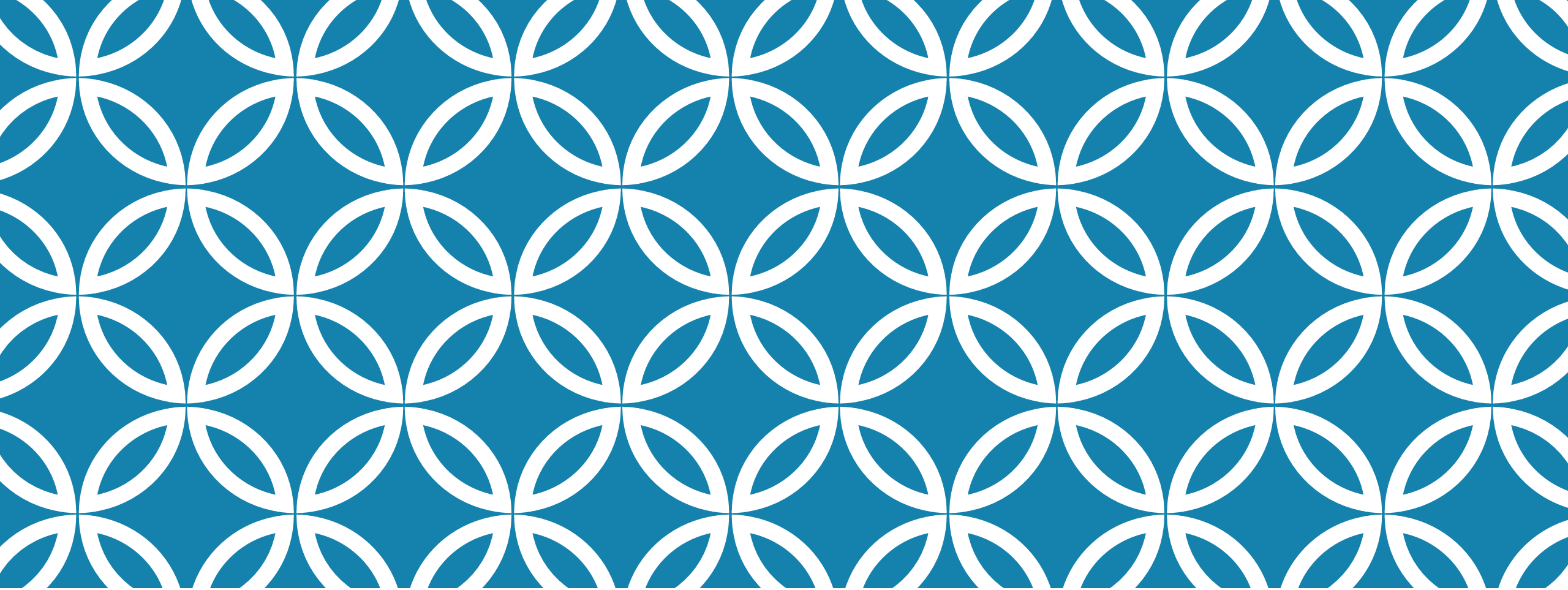


CS5412/LECTURE 10 CONSISTENT STORAGE FOR IoT

Ken Birman

CS5412 Spring 2020

CONSIDER A SMART HIGHWAY

We have lots and lots of sensors deployed

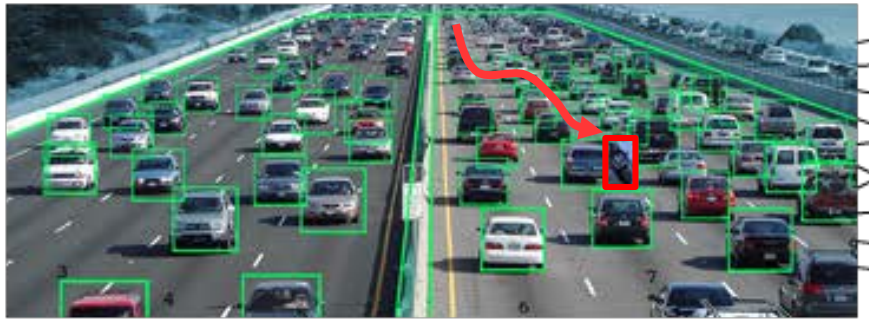
Cars are getting some form of "guidance" and if they accept it (and maybe pay a fee) get to drive faster.

Would we run into consistency issues of the sort seen in Lecture 9?

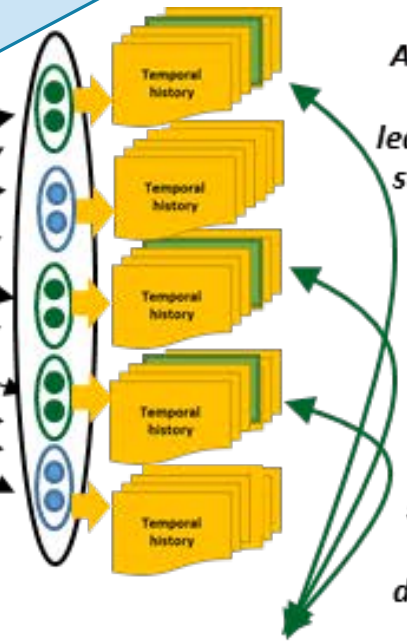
SMART HIGHWAY

Function tier implements this "routing" but doesn't do more, so we won't discuss it – the real work in this example is in the blob store.

Massive inflow of real-time data



Hashing the vid… maps streams to …ards.



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.

Movie is generated by combining sub-results

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)

MACHINE LEARNING TASKS IN THIS SYSTEM

Deciding which images are worth overloading: done on camera

Deciding if an image should be retained: Occurs in the smart service that will then hold the data, if it is retained…

… later this same smart service responds when asked "did camera HW101-018 capture any images of a women with a passanger on a motorcycle at 10:02.035am on January 21 2017?"

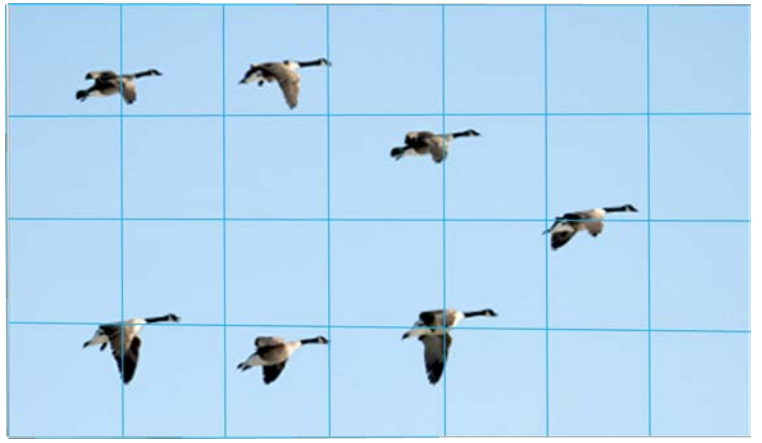
Deciding which images to use in the "movie" we are creating to document Trinity's crazy driving

TRACKING TRINITY…

In this example, we are doing a few things all at once

- Data is being captured by IoT sensors and uploaded via a function tier (it doesn't do much).
- 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

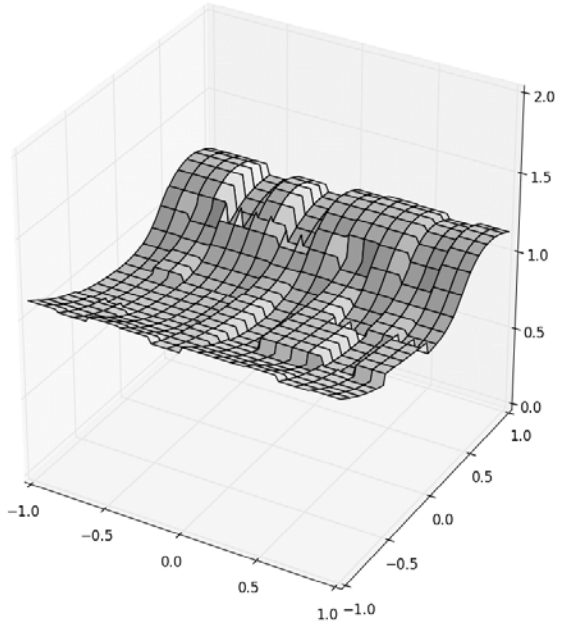
To illustrate this point visually, we made a simulation.

Rather than a flock of geese, it simulates a wave in an aquarium, or a phase-shift in a power grid, as sampled by a grid of sensors. We captured this "IoT data" into files.

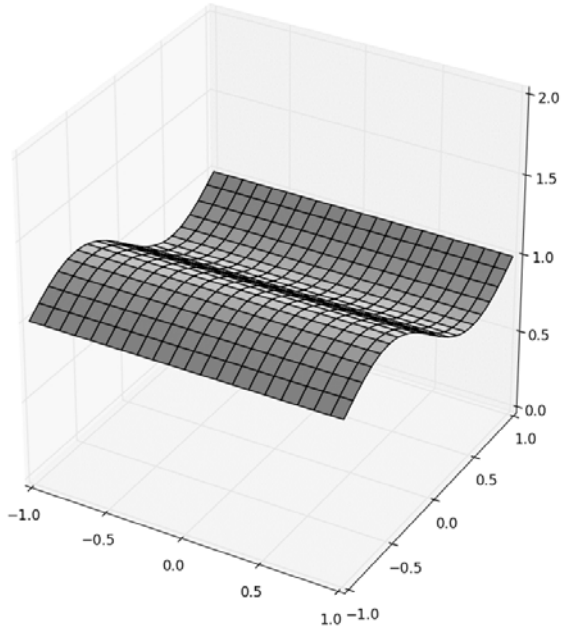
Then we took snapshots and made a movie.

CONSISTENCY PROBLEM: HDFS DOES BADLY!

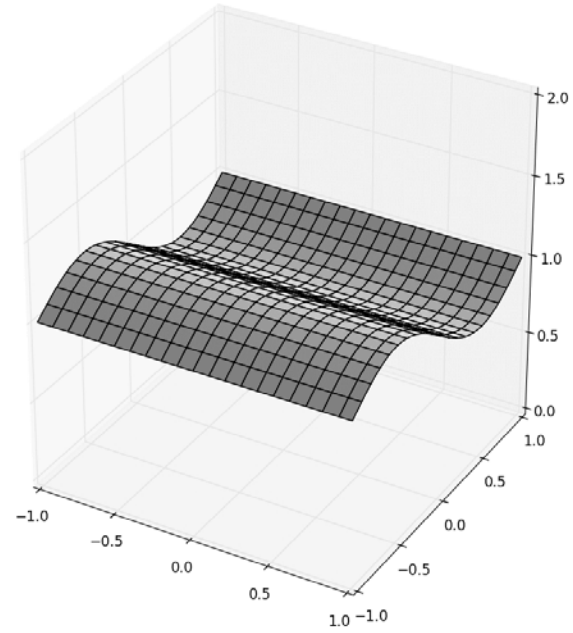
HDFS



FFFS+Server Time

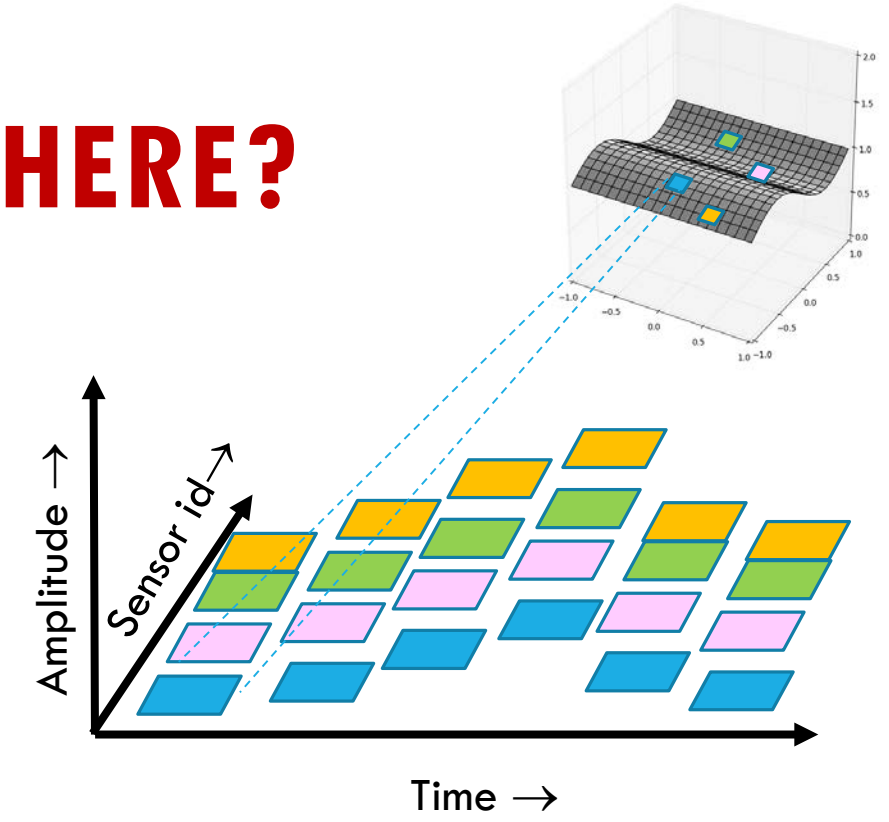


FFFS+Sensor TIME



Existing file systems (like HDFS on the left) make mistakes when handling real-time data. But we can fix such problems (right).

WHAT ACTUALLY HAPPENS HERE?



Each “cell” is like a small photo.

A new sensor reading is like a new photo.

Here we see a blue sensor, a yellow, a green and a pink one that each send 6 photos over a period of time.

WHAT MADE HDFS SO NOISY?

It was confused about time. Sometimes a snapshot included data from the frame prior to the one we wanted, or after it.

Sometimes data was completely missed. This is because HDFS is slow to "settle" down after an update.

Sometimes it violated the gap-freedom property: *inconsistent cuts!*

WHY WOULD THERE BE A CAUSAL CONNECTION BETWEEN SENSOR VALUES?

In fact there isn't: Those are completely independent and parallel

But we often construct secondary indices (like our B-Tree in hw2)

Those depend on the data in them, and can evolve through versions too, which creates a more complex happens-before relationship that fits Lamport's model well.

WHY IS CONSISTENCY SUCH A BIG DEAL?

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 ($FFFS_{V1}$)

This was created by the 2019 TA, Theo Gkountouvas, with Weijia Song!

The idea was to bring Lamport's model into the file system.

Fhey 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.

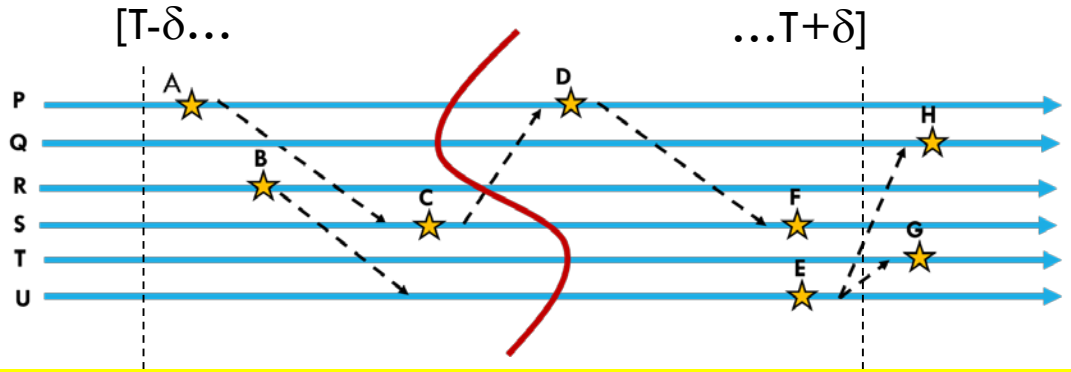
REMINDER: CONSISTENT CUTS

We talked about this on Tuesday.

Lamport suggested "visiting" machines in a distributed system at a set of instants that represent a gap-free snapshot of the execution.

- Math term: "closed under the $\rightarrow$ relationship"
- If B is included in the set, than any A $\rightarrow$ B is included too.
- He calls this a consistent snapshot. A consistent cut is the same but doesn't include the full history (it looks just at the state when you visited the machines, at that moment).

HOW DOES IT WORK?

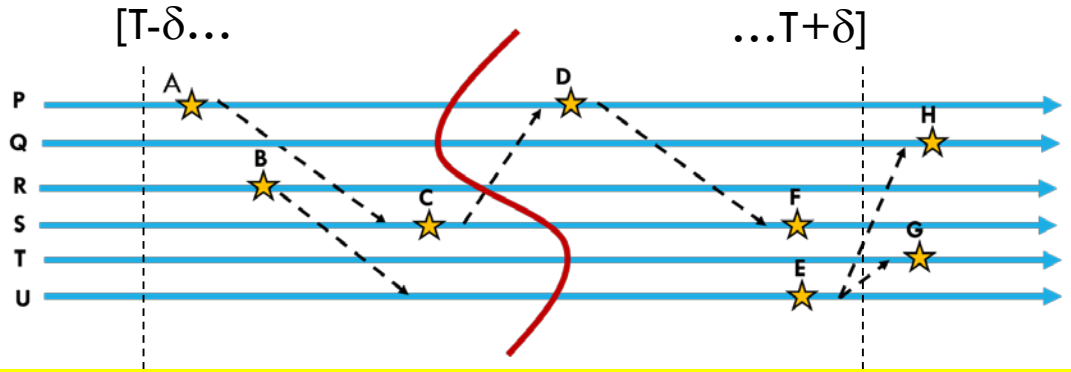


***Due to clock skew, T could fall anywhere in* [T-δ, T+δ]**

FFFS keeps a history of versions of the data it receives

- When you overwrite file records, it keeps the old version too!
- Data is indexed by time, but also by logical time.
- When you ask for a snapshot at time T, FFFS needs to ask the servers what data each has for each sensor.

HOW DOES IT WORK?

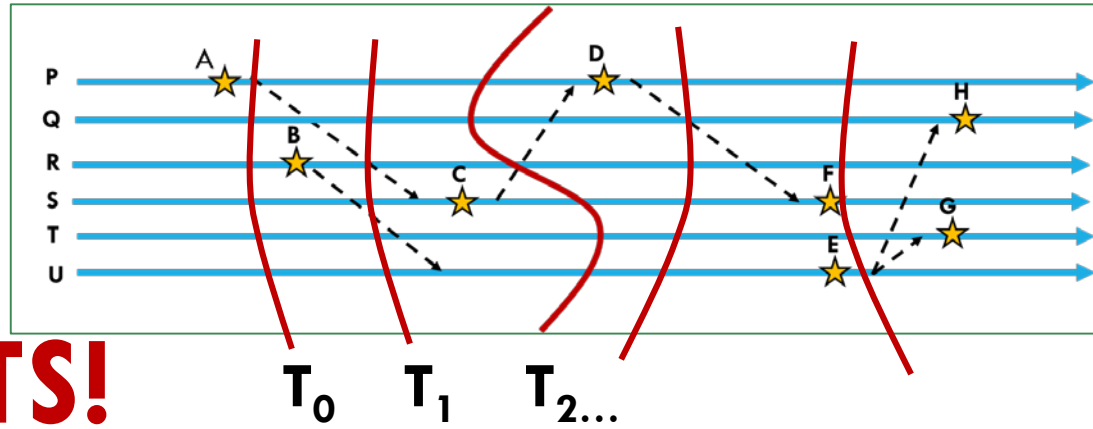


***Due to clock skew, T could fall anywhere in* [T-δ, T+δ]**

FFFS is smart about Lamport's → relationship

- It tracks down the candidate values to return using the time, T
- Above, notice that A happens at P, and C at Q, and A → C
- In fact, A could have "caused" C. Information about A reached C
- FFFS tracks causality, so that if some read returns C, FFFS would return A or some subsequent state for P.
- In effect, FFFS does temporal reads *along a consistent cut.*

WHAT IF YOU DO MANY READS? CONSISTENT CUTS!



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 $FFFS_{v1}$ 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.

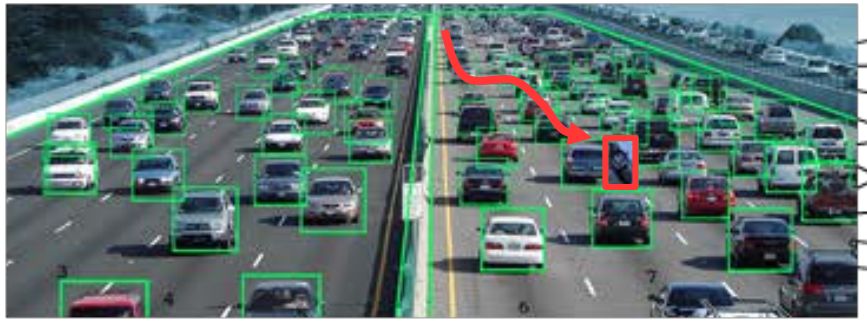
Then it can select data that includes video-snippets of Trinity with exactly one snippet per unit of time, no overlaps, no "lies".

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

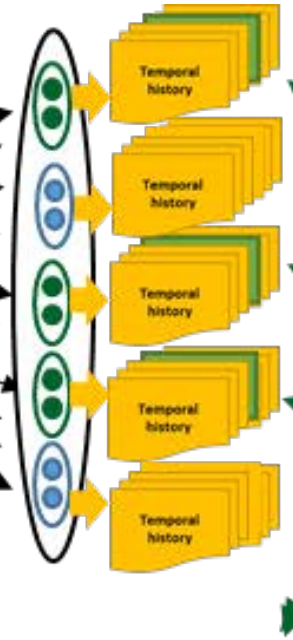
REVISIT THE SMART HIGHWAY

Use FFFS as the blob store!

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.

Movie is generated by combining sub-results

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)

FILE SYSTEM API GOT IN OUR WAY!

$FFFS_{v1}$ is actually a bit slow, partly *because* it uses a file system API. To talk to it, you open files, read/write/seek/close/delete.

This is not ideally matched to modern ML, where we prefer to use computational patterns like MapReduce.

What we really would want is to have a set of servers host a DHT and right next to the DHT, also host the computational task.

A DHT IS A MUCH MORE NATURAL CHOICE

If we track "versions" than we can use a DHT put operation.

The key is the sensor id. In our animation we had 400 of them.

Each value is a byte-array with a "new reading" from the sensor. In fact these are really structured records that include highly accurate time ("synchrophasor measurements").

CASCADE: A DHT "LIKE" $FFFS_{V1}$

So we decided to create a key-value store, versioned like FFFS, but with a sharded structure like key-value products.

We called it Cascade. It is starting to work now.

It was built using Derecho.

CONCEPT: SERVERS THAT *EACH* HOSTS A SHARD OF THE DHT PLUS CODE TO COMPUTE ON SHARDED DATA

A server

Some function we want to use in MapReduce

MapReduce can talk to the DHT shard without going over the network

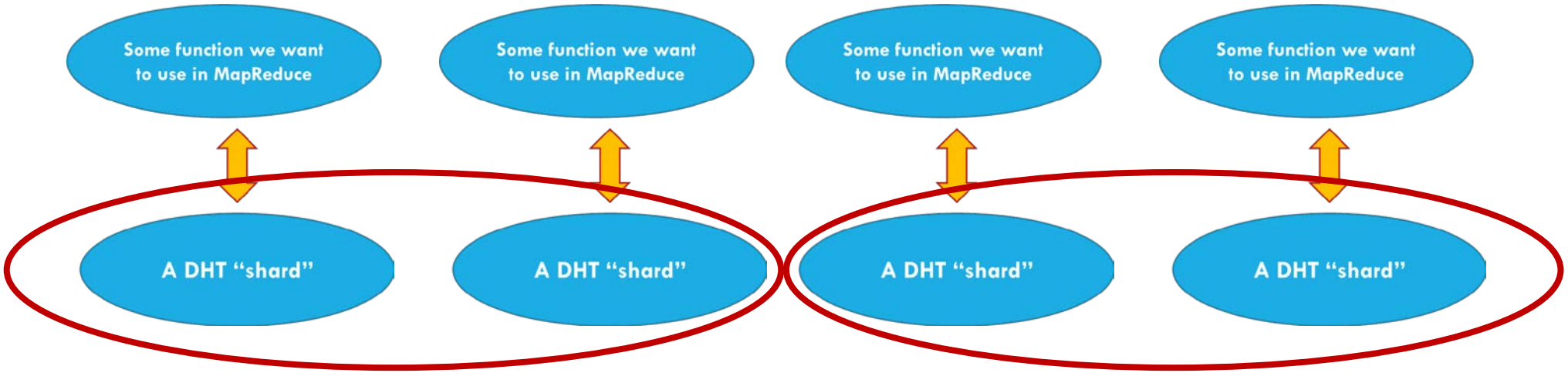
A DHT "shard"

On this one machine, we have *both* the code MapReduce will run *and* one of our DHT shards

CONCEPT: SERVERS THAT *EACH* HOSTS A SHARD OF THE DHT PLUS CODE TO COMPUTE ON SHARDED DATA

Our datacenter has many servers…

OUR CODE IS "SIDE BY SIDE" WITH THE SHARD AND CAN ACCESS LOCAL DATA CHEAPLY!



This looks like homework 2! A key-value DHT that lives on the same machines where some kind of logic is running. In homework 2, it was a B-Tree. In this example, it would be MapReduce or Hadoop…

CASCADE API ($FFFS_{V2}$)

Extremely simple:

- Cascade::**put**(key, value),
- Cascade::**get**(key[, time]). You can also query the version # for the object
- Cascade::**cput**(key, value, new-intended-version-#)
- Cascade::**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.

HOW YOUR CODE "REACHES" THE DHT

If your code uses a key that maps to the same node where it is running now, the request is handled locally, with no network I/O.

If your key maps to some other shard, Cascade will fetch the object over a high-speed RDMA link. If you did a put, it will use Derecho state machine replication (multicast) to do the update.

LOCALITY BENEFIT

If the code is talking to the local shard, no locking or copying is required, which makes access ultra-fast!

- We don't need locks because any updates create new versions
- Cascade actually does support version overwrite too, but in our example, it isn't using that feature.
- We won't need to copy the data (which could be big!) since we can access it directly right where Cascade holds it

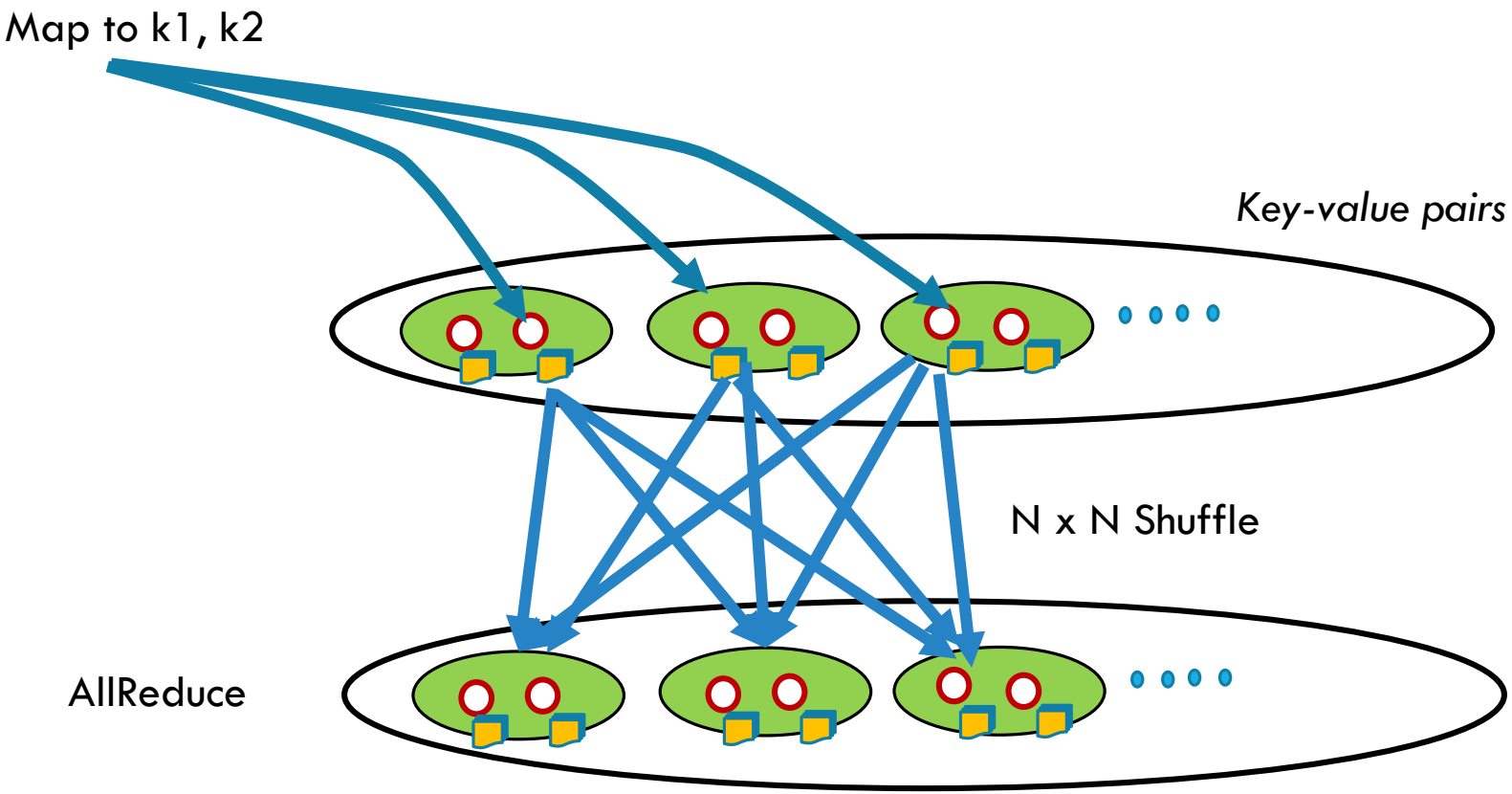
WHAT IF WE NEED A GPU FOR FAST COMPUTE?

Many ML algorithms are only feasible if we can do parallel computing on a GPU or TPU chip.

Cascade is smart enough to remember what it downloads into GPU memory. It won't DMA the same object again and again.

In effect, Cascade "manages" GPU memory like a cache!

MAP-REDUCE ON DATA IN CASCADE



We obtain a completely atomic MapReduce primitive within Derecho!

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.

STORING DELTAS

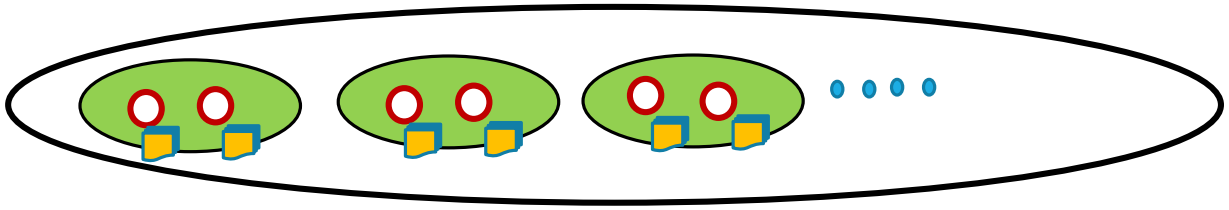
Existing DHTs lack support for versioned data.

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.

SIMPLEST CASE



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.
- **cput** is like put, but only replaces the prior value if the version # matches

WHAT IF WE DON'T WANT VERSIONS?

Version *numbering* is standard in Cascade. But it doesn't always track the version *history*.

For a given key, you can ask it to keep just the most current version, or you can ask it to keep all prior versions.

In our power grid example, we used the "all prior versions" approach, but for some kinds of data "just keep the current one" is best.

SEQUENCES OF VERSIONS

With concurrent applications, perhaps some task will do a **get**, then compute a new version, then **put**. But if some other task simultaneously does the same thing, they could try to create the same version.

In **cput** is a in the object store to address this kind of race condition. If an update races occurs, **cput** notices that a task is trying to overwrite a version that is already present. It "fails" so that the task can repeat the **get** and try again. This yields consistency without locking!

WHY IS AVOIDING LOCKING SO IMPORTANT?

In a database course you are taught to lock an object, read it, then write the new version and then commit.

But Jim Gray told us this won't scale, so we don't want to do that. It led to Eric Brewer's CAP conjecture, and weak consistency.

Cascade has strong consistency from state machine replication plus versioning. cput avoids locking, avoiding Gray's concern!

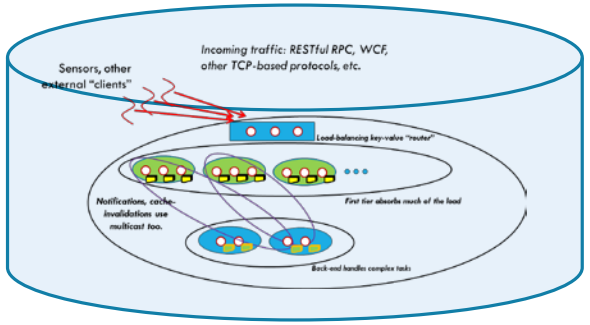
WHY IS AVOIDING LOCKING SO IMPORTANT?

Summary: conditional version put:

cput allows you to read a version. When you update it, you'll do a cput but you also specify the required version number.

- This will normally be successful.
- But if the version was changed under your feet, you get an error. Then throw away the update and try again.

OBJECT STORE AS A FILE SYSTEM



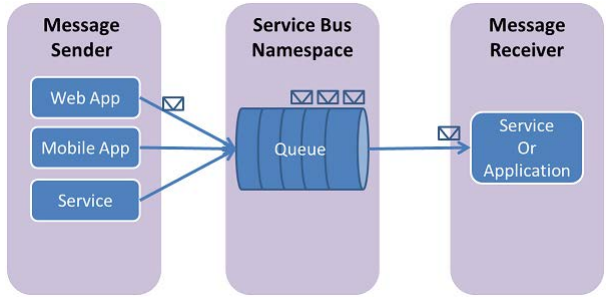
A file system is really an abstraction over a block store.

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**

OBJECT STORE AS A "BUS"



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.

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.

HOW IS CASCADE IMPLEMENTED?

The entire key-value architecture is being built over Derecho

In effect, you talk to Cascade (put/get/watch) and it "translates" your actions into Derecho's state machine replication operations, which run on RDMA (if available).

Effect is that your Cascade operation is extremely fast

A LIBRARY? OR A μ-SERVICE?

Cascade can be set up as a separate μ-service on its machines.

- In this configuraton it could replace a blob store, or a DHT like CosmosDB or Cassandra or Dynamo.
- But your code wouldn't run on the same machines.

Used as a library, Cascade can run on the same machine as your code. This eliminates extra copying and message passing.

SUMMARY

Lamport's ideas give us a way to fix the inconsistencies seen in existing cloud storage systems, like HDFS or the Azure blob store

In prior work, Cornell created a file system, $FFFS_{v1}$ to show this.

Right now, the newer Cascade project is recreating the same options but in a (key,value) DHT approach that fits nicely with today's most popular ML and AI platforms.