Tasks that require consistent replication

- Copying programs to machines that will run them, or entire virtual machines.
- Replication of configuration parameters and input settings.
- Copying patches or other updates.
- Replication for fault-tolerance, within the datacenter or at geographic scale.
- Replication so that a large set of first-tier systems have local copies of data needed to rapidly respond to requests.
- Replication for parallel processing in the back-end layer.
- Data exchanged in the “shuffle/merge” phase of MapReduce.
- Interaction between members of a group of tasks that need to coordinate:
  - Locking
  - Leader selection and disseminating decisions back to the other members
  - Barrier coordination
A common approach is “chain replication”, used to make copies of application data in a small group.

We form our group into a chain and send updates to the head.

The updates transit node by node to the tail, and only then are they applied: first at the tail, then node by node back to the head.

Queries are always sent to the tail of the chain: it is the most up to date.
FRAMEWORKS LIKE DERECHO SUPPORT MUCH FANCIER USES

We listed a set of roles, like managing the “layout” of a complex μ-service, orchestrating self-repair after a crash, mapping updates to RDMA.

But how can we understand the associated computing model?

In today’s lecture we explore this and some related models.
MEMBERSHIP MANAGEMENT: ONE NEED

Automates tasks such as tracking which computers are in the service, what roles have been assigned to them.

May also be integrated with fault monitoring, management of configuration data (and ways to update the configuration).

Often has a notification mechanism to report on changes
STATE MACHINE REPLICATION MODEL

We’ve seen it a few times since Lecture 2, but let’s be more detailed.

Take some set of machines that are initially in identical states.

Run a deterministic program on them.

If we present updates in identical order, the replicas remain in the same state.

This is a very general way to describe replicating a log or a file or a database, coordinating reactions when some event happens in a fog computing setting, locking and barrier synchronization, etc.
Leslie proposed several solutions over many years.

His very first protocol assumed that the shard membership is fixed and no failures occur, and used a two-phase commit with a cute trick to put requests into order even if updates were issued concurrently.
Pending updates occupy a slot but are not yet executed.
LAMPORT’S RULE:

Leader sends proposed message.

Receivers timestamp the message with a logical clock, insert to a priority queue and reply with (timestamp, receiver-id).

For example: A:1 was put into slots \{\{(1,X), (2,Y), (1,Z)\}\}

B:1 was put into slots \{\{(2,X), (1,Y), (2,Z)\}\}

Leaders now compute the maximum by timestamp, breaking ties with ID.
LAMPORT’S PROTOCOL, SECOND PHASE

Now Leaders send the “commit times” they computed

 Receivers reorder their priority queues

 Receivers deliver committed messages, from the front of the queue
LESLIE’S ORIGINAL PROPOSAL: PRIORITY QUEUES AND LOGICAL CLOCKS

Notice that committed messages either stay in place, or move to the right.

This is why it is safe to deliver committed messages when they reach the front of the queue!
IS THIS A “GOOD” SMR PROTOCOL?

The 2-phase approach works well for a totally ordered atomic multicast.

But it isn’t correct if the state will be durable (on disk).

Also, handling membership changes is a bit tricky. So Lamport pushed further and proposed his Paxos protocol.
Leslie Lamport proposed the Paxos model as a way to

- Express the state machine replication problem in a more formal way
- Solve it, using the “Paxos protocol”. Now we call it the “classic” one:
  - Over the years, many Paxos protocols have been developed
  - All of them solve the Paxos specification, but the details vary widely

One source of confusion with Paxos concerns storage of data

- Is Paxos “delivering messages” to an application that has the state/storage role?
- Or is Paxos itself a kind of “database of messages”? **Leslie likes this model.**
HOW PAXOS WORKS.

Similar to the original protocol, but in Paxos, Lamport only requires a reply from a majority (quorum) of the participants.

He has two “phases”. In the first, a leader keeps proposing to place messages into slots and the receivers either accept or reject the proposal. They write the pending messages to logged storage, but then wait.

In the second phase, a leader with a quorum of accept votes can use a two-phase commit to finalize the update.
HOW DO THESE BEHAVIORS DIFFER?

In the first protocol, we do an atomic multicast to update in-memory state.

- This is sometimes called a vertical Paxos protocol.
- The key insight is that it isn’t safe if all the processes might crash.

In the second protocol, Paxos “implements” a durable storage system

- Like an append-only log, and updates are appended
- The application would read the log, but the log is durable.
NOT EVERYTHING NEEDS SUCH AN ELABORATE MODEL AND SOLUTION

In many key-value stores, membership management is just done by a service that periodically updates a file listing the members and mapping.

Then the members are expected to “shuffle” data until they have the proper key-value tuples.

There can be a brief period of inconsistency but it wouldn’t last for long.
VIRTUALLY SYNCHRONOUS MEMBERSHIP

Manages a group of replicas that are using Paxos.

Members can join the group, leave it, crash (failed members are ejected).

“State transfer” to initialize the new members when they join

Updates occur entirely in a single epoch: during a period when membership is stable. Every replica receives the identical updates.
VIRTUAL SYNCHRONY: MANAGED GROUPS

Epoch: A period from one membership view until the next one.

Joins, failures are “clean”, state is transferred to joining members.

Multicasts reach all members, delay is minimal, and order is identical...
In Derecho, a “true” Paxos protocol is used to compute each new membership epoch (“view”).

We pick a leader, and it attempts to gain agreement on the next view and other configuration parameters for the epoch (like the mapping of view to subgroup layout and sharding).

If the leader fails, a different leader takes over. This terminates when a quorum is able to accept and switch to the new epoch.
LIFE IN AN EPOCH

Then, during the epoch, multicasts or durable Paxos updates can be done under an assumption that no failures occur.

This simplifies protocols: Derecho’s protocols are *optimally efficient*.

If some process does fail, the event triggers a new epoch-agreement protocol, which also finalizes the prior epoch.
MULTICASTS VERSUS PAXOS

For a virtual synchrony multicast, a totally-ordered in-memory message triggers a state machine update. The state itself lives in memory.

With durable Paxos, state lives in a file or append-only log, on disk. Here because the state is non-volatile, even a failure can’t cause it to be lost.

In both cases, if a process joins a group, state transfer is used to initialize it.
Epoch: A period from one membership view until the next one.

Joins, failures are “clean”, state is transferred to joining members.

Multicasts reach all members, delay is minimal, and order is identical…
Imagine an air traffic control computer, and a backup network fails, but just for a moment… both machines remain active.

- Machine X never noticed the outage and thinks A is the active controller.
- But machine Y “rolled over” to the backup, and is talking to B.
- Now we have two controller computers for the same landing strip!

Partitioning (split brain) and similar inconsistencies can never happen with virtually synchronous Paxos solutions. Use virtual synchronous Paxos whenever you are at risk of unsafe behavior in the event of inconsistency.
LET’S TAKE A CLOSER LOOK

What might a service using these concepts be doing?

How would a person code such a service?
EXAMPLE: RECEIVE DATA, REPLICATE, STORE

This is an example of an atomic multicast pattern.

The membership of the group of replicas is \{P,Q,R\}. P played a server role by relaying the image.
EXAMPLE: RECEIVE, **DISCARD UNINTERESTING CONTENT.  IF INTERESTING, REPPLICATE AND SAVE.**

Same pattern but now the server, P, needs to run additional machine-learning logic. It also draws on a previously replicated machine learned model.
EXAMPLE: RECEIVE, DISCARD UNINTERESTING CONTENT. IF INTERESTING, RUN A PARALLEL ALGORITHM TO DO EDGE EXTRACTION.

Now we also want Q and R to run some form of user-provided logic, to do the edge extracting (P probably runs it too.)
WHAT DID OUR EXAMPLES SHOW?

We need a way to write a piece of code that can

- Open a TCP connection endpoint. The sensor will connect to it.
- Read in an image or video, over the connection.
- Run some sort of “artificially intelligent” test procedure that makes use of a machine-learned model that was uploaded previously (tier 2)
- Then, if data is interesting, “ask” for it to be replicated
- When replicas arrive at Q, R, write them to storage.

For this, Azure IoT might use a REST library. It handles these standard steps and is easy to use.
Here, all three are playing server roles, for different sensor devices. We will want to make sure they store the updates in identical order, keeping them consistent.
WHAT ABOUT FAULT-TOLERANCE?

Our replica group, \{P,Q,R\} could have crashes

We want the service to remain active while one or two nodes are down

Then we need to self repair
SELF-REPAIR EXAMPLE: PETSTORE.COM

Suppose our service keeps a replicated inventory and machine Q crashes. On restart, we must repair the log on Q before we let it resume activity.
DISCUSSION: GUARANTEES

Once we talk about guarantees, there are many we might want!

- Consistency (every replica sees the same sequence of updates)
- Durability (once an update has been persisted, it won’t be forgotten)
- Fault-tolerance (keeps running even if a replica has crashed)
- Recovery (repairs itself and restores full functionality)
- Congestion/flow control (won’t overwhelm slow components)
- Attack tolerance (costly, but hardens service against malicious attack)
THREE PERSPECTIVES ON REPLICATION

Membership of the replica set, fault sensing

How do we send multicasts?

How can we customize the solution?
HOW DO PEOPLE WRITE CODE WITH DERECHO?

You write a program in C++, Java, or a similar language (it has to be one that can load a C++ library).

You initialize Derecho when the program starts. This launches a thread used by Derecho itself.

Derecho joins your program to other group members, then decides which members play which roles.
In Derecho, roles are really associated with replicated objects.

You design a C++ class, perhaps “T”, and then tell Derecho to create a subgroup of type Replicated<T>.

Derecho uses your guidance to decide which members will belong to this subgroup, and then runs the constructor to initialize it.
Events such as membership changes or delivery of multicasts are via events.

You register a method to handle these events.

The method would update the state associated with the group. There is also a very fast way to do read-only queries.
class Foo: Replicated<Foo> {

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

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

};

auto outcome = fptr.p2p_send<Foo::put>(who, “John Doe”, 22.7);
```cpp
class Foo: Replicated<Foo> {
    Volatile<T> myObj_V("FileName"); // Declares an in-memory variable
    Persistent<T> myObj_P("FileName"); // Declares a persistent variable
}

// Program-level access to versioned storage:
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)
```
SAME SERVICE VIEWED AS A FILE SYSTEM

Programs accessing data captured using Derecho via append-only files

Version vectors opened as files

Posix “snapshot” feature used for indexing

External clients use standard RESTful RPC through a load balancer

Multicasts used for cache invalidations, updates

Load balancer

Cache Layer

Back-end Store
... SMART MEMORIES CAN BE COMPLICATED!
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.
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. 1/4\textsuperscript{th} 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);
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::put>(who, "John Doe", 22.7);
auto result = cache.p2p_query<MemCacheD::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::put>("John Doe", 22.7);

auto results = cache.ordered_query<MemCacheD::get>("John Doe");
for(auto future : results) { code... }
API SUMMARY: DEFINE SUBGROUP/SHARD

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.
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.
SOME PRACTICAL COMMENTS

Derecho is very flexible and strongly typed when used from C++.

But people working in Java and Python can only use the system with byte array objects (size_t, char*).

You can't directly call a “templated” API from Java or Python, so:
- First you create a DLL with non-templated methods, compile it.
- Then you can load that DLL and call those methods.
- You still need to know some C++, but much less.
CONCLUSIONS?

A software library like Derecho automates many aspects of creating a new \( \mu \)-service.

The Paxos model is used to ensure consistency, fault-tolerance. There are two cases: ordered multicast (non-durable) and persistent (on disk).

You code in an event-driven style.