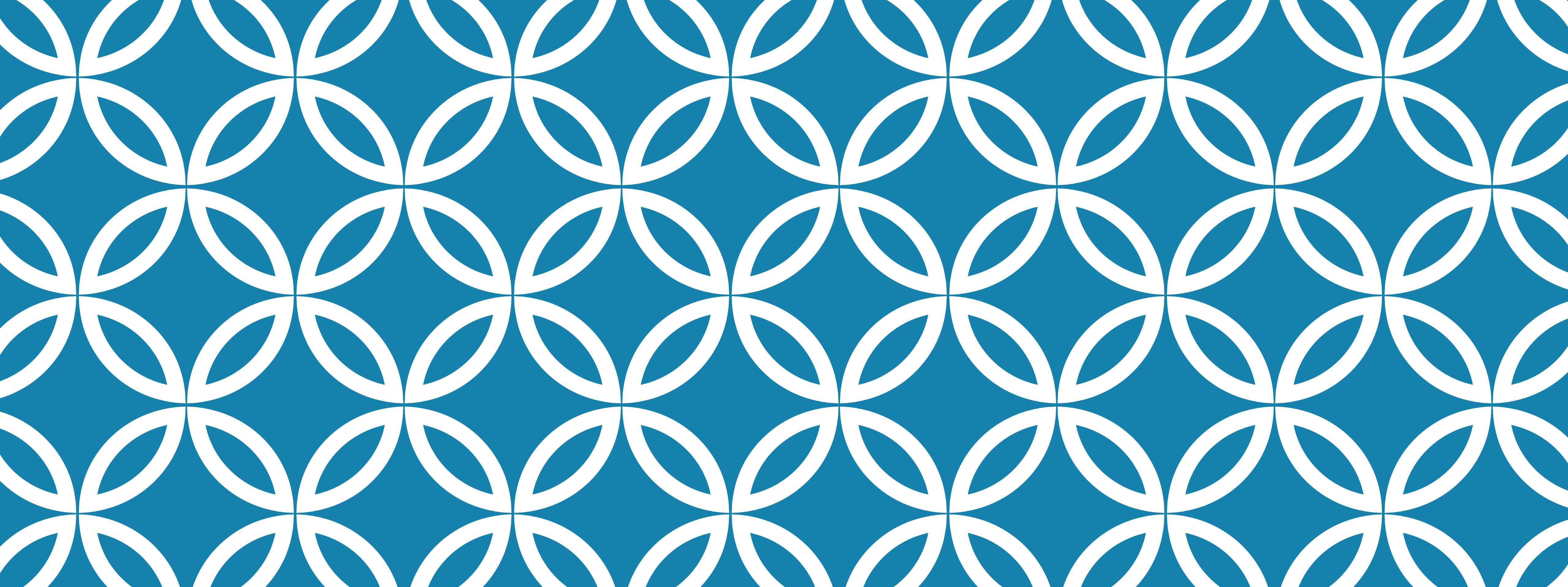


CS5412 / LECTURE 7
REPLICATION AND CONSISTENCY
(PART I: THEORY AND PROTOCOLS)

Ken Birman

Spring, 2022

CAN WE BUILD CONSISTENT, HIGHLY AVAILABLE μ-SERVICES?

When building μ-services everything needs to be sharded for scaling.

But so far, our shards have just a single member each.

If we also want fault-tolerance we would have two or more members per shard, not just one.

CAN WE BUILD CONSISTENT, HIGHLY AVAILABLE μ-SERVICES?

We also learned about the CAP folk-theorem.

It says that consistency is just not needed in the cloud and that we should build systems with weak consistency if they would be more available (responsive) even when partitioned.

Web sites tolerate staleness. We can hide many forms of staleness, or explain it away. Usually, our customer won't even notice.

CAN WE BUILD CONSISTENT, HIGHLY AVAILABLE μ-SERVICES?

But in IoT settings, inconsistency is a big problem.

- IoT systems monitor physical things, like motors and power grids.
- Inconsistencies can be confusing – they might tell us the motor isn't on, yet in fact it *is* on. We don't want stale IoT data!
- Many people think that IoT requires strong consistency.

Could we build μ-services that just replicate the entire state: **CA** but not **P**?

- If they don't depend on a second layer of services, **P** won't arise.

OTHER 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

~~Superman~~ LESLIE LAMPORT TO THE RESCUE!

Leslie Lamport

This is Leslie Lamport, who was a pioneer in bringing rigorous models and reasoning to distributed computing systems.

For Leslie, our question relates to replicating state. If we have replicas of the state of a μ-service, it can tolerate failures, and if we can make it consistent we get C+A. Now we might not actually *need* P.

State machine replication is the name Lamport proposed for this new model

HOW DOES STATE MACHINE REPLICATION WORK?

We start with some set of processes (like servers in a DHT shard)

Initialize them into the identical starting state.

Then make sure each process sees the identical events in the identical order. If the code is deterministic, they will remain consistent.

From this idea we can build up abstractions like fault-tolerant computing.

THERE ARE MANY NEW IDEAS HERE!

Can we really build "deterministic" computer programs?

What does it really mean to build a program that can be replicated this way? How does this even fit with an IoT setting using μ-services?

Anyhow, how did we know which servers should do this replication? And what if one of them fails, but the others keep running?

DETERMINISM

The idea here is to think of each program as code that reads inputs, then computes on the input, then produces outputs.

A non-deterministic program might do various different things even with the identical inputs.

A deterministic program will always behave identically.

WHY MIGHT A PROGRAM *NOT* BE DETERMINISTIC?

Think about a program that reads the clock.

If I make two copies, they won't see the same value because computer clocks advance at such high rates (nanosecond increments) that you basically can't read two clocks in parallel and see the same time!

So even if you just print the time, your program is non-deterministic.

MORE ISSUES…

Most modern programs are multithreaded.

But this implies that the thread scheduling order is random and unpredictable. The only way to be fully sure of the order is to use locking in some very rigid way.

Since most programs don't use locking in such a rigid way, the scheduling order can't be controlled, and so each copy behaves differently.

MORE ISSUES…

Some programs read input from more than one potential source, like more than one client on a network.

Those programs could get two inputs more or less at the same time– in which case one replica might see A, then B. But the other might see B first.

In fact this could even happen if we have one network connection to a multithreaded client.

LESLIE'S ANSWER? "MEH."

Leslie Lamport

When people first raised these concerns, Leslie didn't really respond.

He said that at the end of the day, he is a theoretical computer scientist and not an engineer. He said his role is to "inspire" not "implement".

Leslie did define "building blocks" for state machine replication, but he didn't worry much about those issues.

THE MAIN BUILDING BLOCKS

One is called *atomic multicast*. The other is called *durable logging*.

Both implement state machine replication, but in different settings.

Atomic multicast is a pure networking concept. It doesn't save data into storage of any form.

ATOMIC MULTICAST

We have a sender, and a group of receivers.

- In some situations, this group of receivers is just a list of processes.
- In others, the group is some form of "name" for the group, and a group membership service is used to track the mapping from the name to the current list of members.

Now we can offer the sender an atomic multicast API

```
outcome = atomic_multicast(destinations, message);
```

ATOMIC MULTICAST

With an atomic multicast, we usually just say that the message is a vector of bytes. From last week (and in the homework) you've learned that actually we can represent all sorts of objects as byte vectors, using *serialization.*

Atomic multicast normally requires some form of protocol that implements the sending (just like TCP, which implements reliable one-to-one data streams over the more basic network hardware).

ATOMIC MULTICAST

The requirements are:

- The atomic multicast is all or nothing. If any receiver delivers a message, then every receiver must do so (unless it crashes, obviously).
- If a crash does occur, this can't break the delivery guarantees. Worst case? A sender crash after just one copy was sent. This must be self-repaired inside the protocol that implements the primitive.
- Additionally, messages must be delivered in the identical order at all the receivers. If A and B are simultaneously sent, the order can be A B or B A, but everyone must "agree".

PERSISTENT LOGGING

This idea starts with atomic multicast, but assumes that each receiver will be saving messages in an append-only file: a *log*.

We want all the receivers to either have identical logs, or we could allow the logs to have gaps but use a merge-and-repair protocol when reading data, to fill in any gaps.

With persistent logging, data becomes durable: After some point the protocol can guarantee "this information will not be lost."

HOW MANY FAILURES?

Leslie proposed many protocols to implement the state machine replication model using atomic multicast or durable logging.

He also pioneered the art of proving that his protocols would be correct if the number of non-faulty receivers is sufficiently large compared to the number of faulty ones.

Depending on the fault model we use, this can be easy… or quite hard…

FAILURE MODELS

Most cloud computing systems worry about crash failures and network link failures (partitioning).

These are relatively easy to detect and protect against.

With accurate detection we just need F+1 processes to tolerate F failures.

MORE EXTREME MODELS

In fact there are many models that try to capture extreme behavior, such as being under a hacking attack.

These are generally called Byzantine fault models.

With Byzantine faults, we usually need 3F+1 processes to overcome F failures – a single Byzantine process can cause big trouble!

BYZANTINE AGREEMENT

Based on a story Leslie Lamport cooked up.

A group of knights lead small armies towards a castle. Inside, the besieged King has an army too, but he would be defeated if the armies all attack. However, if he can **split** them, he will win.

So, he bribes a knight. Obviously, the knight's own army won't attack. But the knight also needs to block at least one additional attacker…

SOME PROBLEMS TO THINK ABOUT

The basic issue is when the knights don't agree. If some attack and some retreat, the force will be defeated and the king in the castle wins!

But the traitorous knight is motivated to lie. It could say retreat to some knights and attack to others – to split their forces.

We need a form of **majority voting** that works even when some knight tries to confuse and disrupt! This is solvable but explains the 3F+1 rule.

THE BYZANTINE MODEL IN ACTION

Retreat!

Attack!

Attack!

Retreat!

Arthur sees Retreat, Attack, Retreat	Retreats
Lancelot sees Attack, Attack, Retreat	Attacks
Phillip is a traitor, unimportant what he does	---

No solution works
for 3 knights, 1 round

THE BYZANTINE MODEL IN ACTION

Retreat!

Retreat!

Retreat!

Attack!

With four knights and 2 rounds, we can untangle the puzzle (they all retreat)

BASIC IDEA FOR ONE SIMPLE PROTOCOL

The main concept is to run multiple rounds of "voting"

- " I am Lancelot. In round 3 I heard **attack, retreat, retreat, retreat**".
- " Therefore, in round 4, Lancelot votes **retreat**".

After F+1 rounds, the non-faulty knights will converge. The single faulty knight cannot stop them from overwhelming his confusing inputs.

TOO MANY PROTOCOLS!

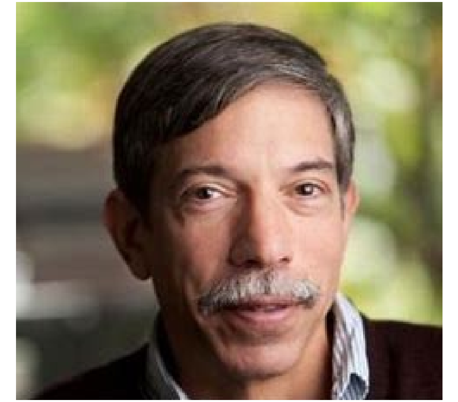
Cloud engineers became overwhelmed

For practical problems like a DHT, should we use a "fail stop" solution? Or a "halting failures" model, without notification? Or even a Byzantine one?

Cloud developers needed *practical* solutions. But Leslie kept writing paper after paper on variations of these scenarios, without clarity of which to actually implement.

~~Superman~~

FRED SCHNEIDER TO THE RESCUE!

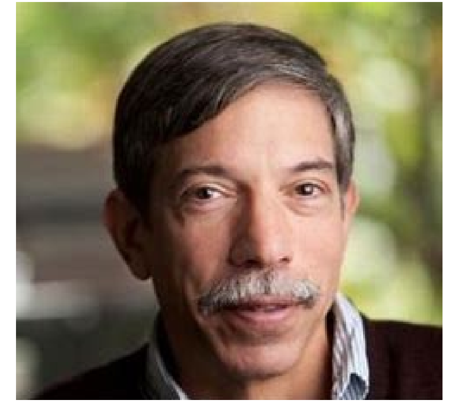


When people first raised these concerns, Leslie didn't really respond.

Leslie explained that at the end of the day, he is a theoretical computer scientist… not an engineer.

Leslie felt that although his work does define "building blocks" for state machine replication, practical issues were beyond his scope.

~~Superman~~
FRED SCHNEIDER TO THE RESCUE!



But Fred Schneider saw this as an opportunity

Working with Leslie, he gradually identified a wide range of ways to actually implement state machine replication solutions. He didn't implement them, but he created a *tutorial* on how to approach the issue.

This work showed that state machine replication could be useful

A FEW BIG TAKEWAYS

One is that state machine replication is way more complex to actually deploy in a real system than anyone initially realized.

We can simplify some aspects by breaking out modules and applying the model only within those modules. A non-deterministic program could still have a state-machine replicated "component" inside it.

But the needed code can be quite substantial!

… WHICH IS WHERE KEN CAME IN!

When I first came to Cornell my initial research project focused on implementations of state machine replication in object oriented computing frameworks – like Java.

The Isis Toolkit became widely used and pretty famous. This is how I earned tenure.

WHY ISIS?

Reference to Egyptian mythology.

When Osiris was torn to pieces by Set, Isis gathered all the pieces and wrapped them in linen. Osiris came back to life. Their son, Horus, went on to defeat Set, who was then banished from Egypt and the underworld.

The Isis Toolkit was created to pick up the pieces of your messed up system and bring it back to life.

VIRTUAL SYNCHRONY

One feature Isis introduced was a practical form of self-managed group and shard membership tracking.

We called this model virtual synchrony. It combines with atomic multicast and durable state machine replication.

SPLIT-BRAIN CONCERN

Tes.com

Suppose your μ-service plays a key role, like air traffic control. There should only be one "owner" for a given runway or airplane.

But when a failure occurs, we want to be sure that control isn't lost. So in this case, the "primary controller" role would shift from process P to some backup process, Q.

The issue: With networks, we lack an accurate way to sense failures, because network links can break and this looks like a crash. Such a situation risks P and Q both trying to control the runway at the same time!

WHY IS THIS IMPORTANT?

Think about air traffic control (a setting that actually does use Isis!)

A plane needs permission to land. Suppose that the computer system tells two different air traffic controllers to take charge of the single runway.

- One says “Flight US 270 you are cleared for landing.”
- The other says “Flight US 270, wait. Flight Delta 110 clear for takeoff”.

This would be a very dangerous inconsistency!

WHAT CAUSED THE PROBLEM?

This is an inconsistency related to who is the "leader"

It can be reduced to the same inconsistencies that CAP is introducing. So if we are going to do things "like" air traffic control, we need consistency.

Key idea: Maybe we can have C+A and not even need P. P matters if a service is really caching data from some other service. ***If each service is self-contained and fault-tolerant – and consistent – we won't need P.***

IOT SYSTEMS WILL HAVE <u>MANY</u> ISSUES LIKE THIS EXAMPLE

With IoT we will need consistency from the start!

… drones, and cars, and smart power grids all need consistency!

Virtual synchrony membership eliminates the split-brain problem and makes it easier to implement state machine replication. In effect, by having "C" include agreement on membership, our job gets easier!

SOLVING THE SPLIT BRAIN PROBLEM

We use a "quorum" approach.

Our system has N processes and only allows progress if more than half agree on the next membership view. Example: if N=5, we say that after a failure, we need 3 or more of the original N to resume.

Since there can't be two subsets that both have more than half, it is impossible to see a split into two subservices.

GROUP MEMBERSHIP SERVICE INSIDE ISIS

In Ken's Isis Toolkit (1985) there was a special subsystem to keep track of group membership, using a quorum method to prevent split brain issues.

Members could join… leave… fail (crash), and Isis would track the state of the system automatically, reporting changes to all the remaining members.

This was enough to enable them to automatically reconfigure and even to self-repair. If too many members failed, the system gracefully shuts down.

VIRTUAL SYNCHRONY MODEL

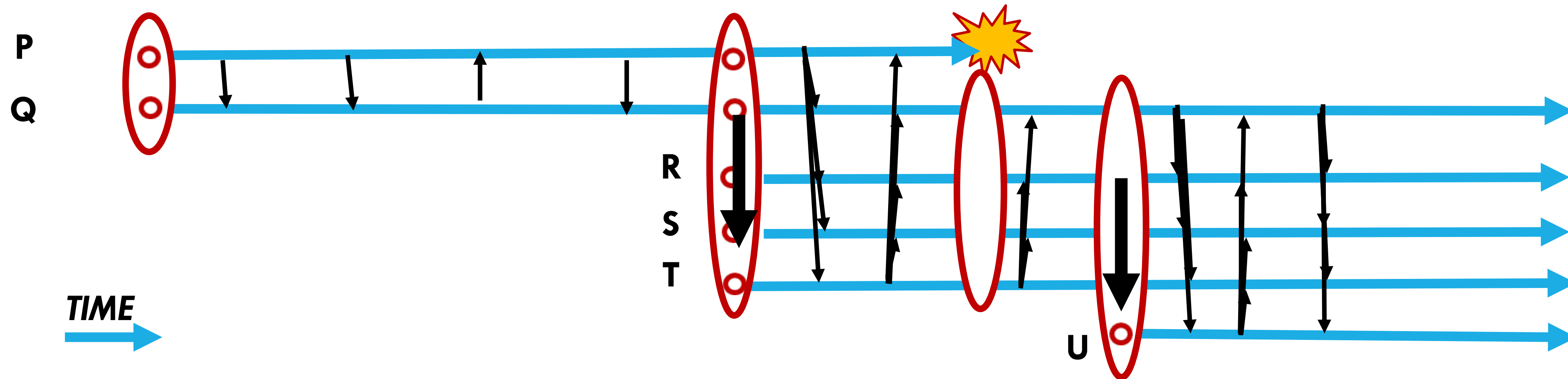
The term relates to how this was integrated with state machine replication.

In virtual synchrony, we use state machine replication when the membership is stable – not changing.

But we pause the state machine replication layer and reconfigure membership if a join, leave or failure occurs. This is done to make it look as if membership changes were atomic and instantaneous.

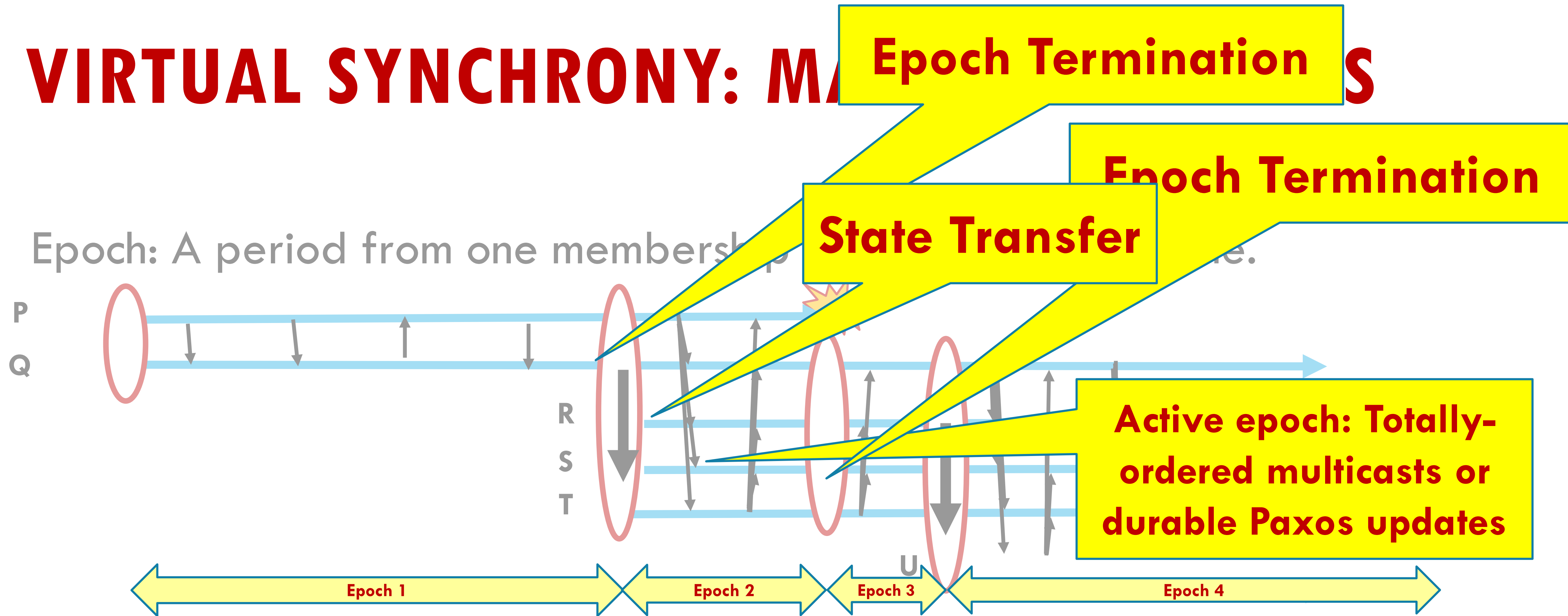
VIRTUAL SYNCHRONY: MANAGED GROUPS

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…



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

Multicasts reach all members, delay is minimal, and order is identical…

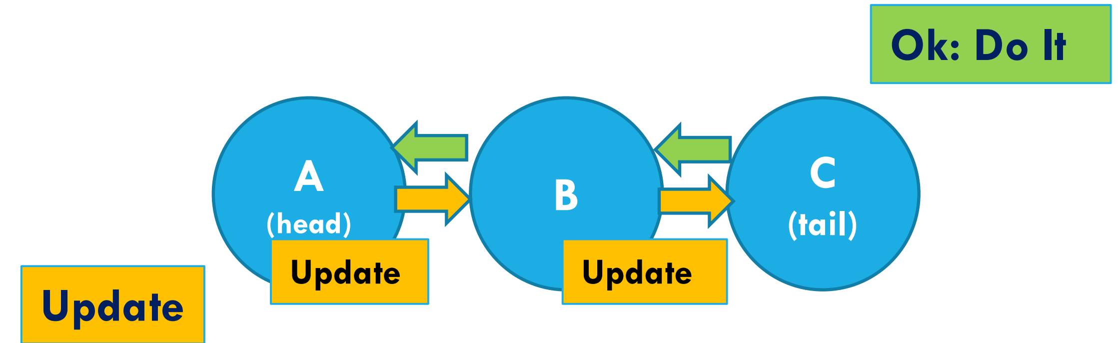
A FEW FAMOUS WAYS OF "PUTTING IT ALL TOGETHER" INTO A PLATFORM

I'll show you one: Chain Replication (Fred Schneider and Robbert van Renesse created this… it is very simple)

In the next lecture we will see others. They include

- Paxos, a family of protocols Leslie designed for atomic multicast (he calls this "vertical Paxos") and for persistent logging ("classic Paxos")
- Derecho, Cornell's newest solutions. These are the world's fastest!

EXAMPLE: CHAIN REPLICATION



A common approach is “chain replication”, used to make copies of application data in a small group. *It assumes that we know which processes participate.*

Once we have the group, we just form 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.

DOES CHAIN REPLICATION SATISFY STATE MACHINE REPLICATION?

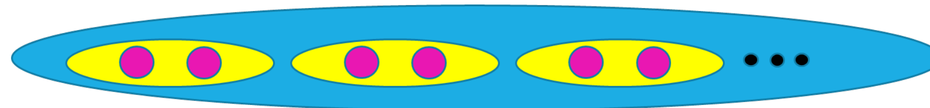
In some ways, but it is an incomplete story.

This is actually why Lamport felt that a formal model (a mathematical one) and a methodology for proving things about protocols was needed.

Chain replication is provably correct, but it assumes a membership mechanism, which it does not include. Without it, the chain replication scheme is not quite as strong as Paxos.

WHAT ABOUT A SHARDED DHT?

This is an image from one of our recent slide sets



One μ-service spanning many machines, split into shards with two machines per shard.

Could we just use chain replication on a shard-by-shard basis? Some systems do this, but as noted, integration with membership tracking matters

SUMMARY:

CAP says "relax consistency and don't stress about it" but in IoT settings, we *do* worry about consistency. The air traffic control example illustrates the concern. Many systems need consistency.

In a key-value store we might see this when replicating updates in a shard.

With Leslie Lamport's state machine replication model as a building block we can work from. Virtual synchrony is a model for managing dynamically evolving system membership. Given these conceptual tools there are simple solutions, like chain replication, that implement consistency.