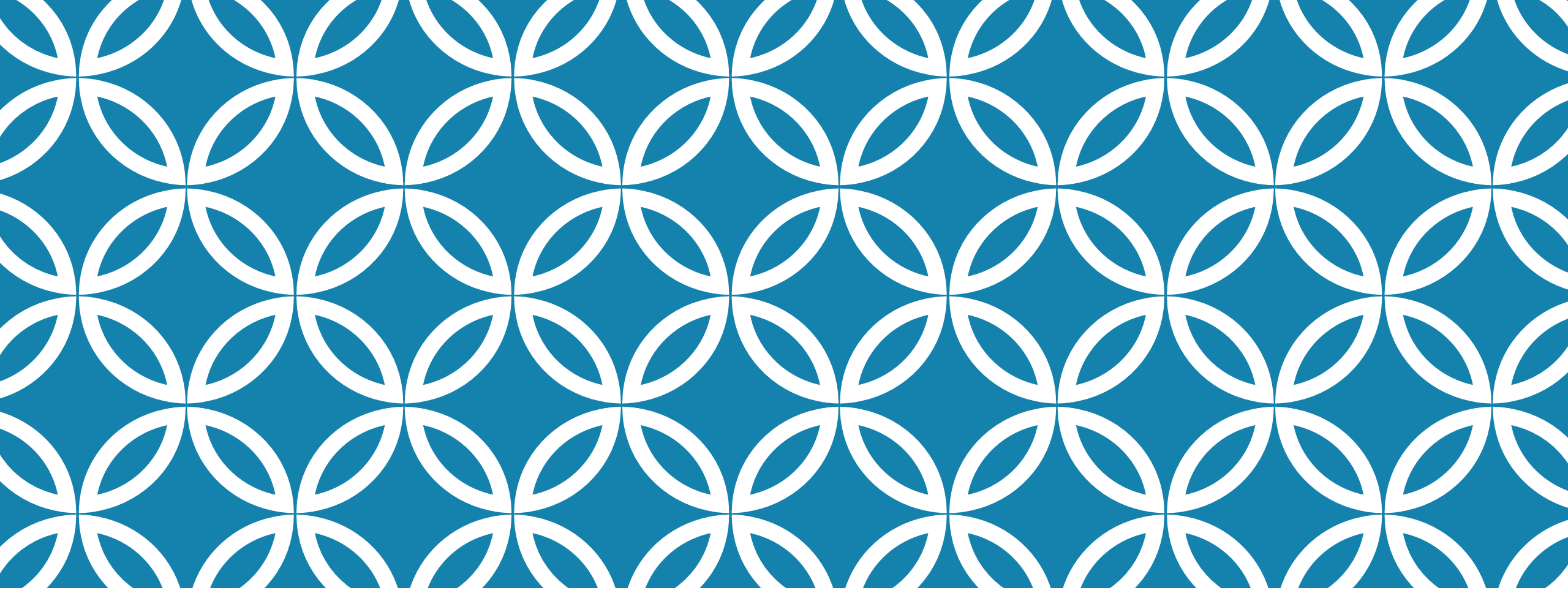


CS5412 / LECTURE 6 REPLICATION AND CONSISTENCY (PART I: THEORY AND PROTOCOLS)

Ken Birman

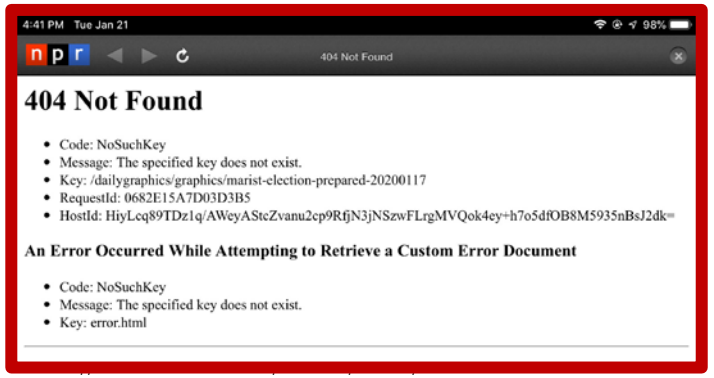
Fall, 2022

IN LECTURE 5, WE SAW CAP IN ACTION

In that example, some sort of DHT update was delayed or disrupted and my "click" on an article (apparently about an election poll in 2020) couldn't find some of the content (apparently some sort of graphic)

The DHT **put** must have been done asynchronously and not yet finished, or maybe the DHT was recovering from a crash or in the middle of an elastic resize event.

This resulted in a visible form of inconsistency



CAN WE BUILD CONTINUOUSLY CONSISTENT, HIGHLY AVAILABLE μ-SERVICES?

Just like in any μ-service, the data and computation will be sharded.

Suppose the data needed is a mix of the data in the request, and data hosted right in the shards that do the computation. This enables **CA**P: P won't matter because our service is self-sufficient!

Each shard will need multiple members for availability (2 or 3 replicas, on machines selected to not fail simultaneously), and should "self-repair".

CAN
HIGH

Just lik

Reminder: P was about "partition tolerance", meaning "the μ-service never has to pause and fetch data from any other μ-service, like a database or file system. It can reply purely using data it "holds".

Eric Brewer pushed for soft-state only, but if we can solve this problem, we can actually have a hard-state μ-service.

Suppose the data needed is a mix of the data in the request, and data hosted right in the shards that do the computation. This enables **CA**P: P won't matter because our service is self-sufficient!

Each shard will need multiple members for availability (2 or 3 replicas, on machines selected to not fail simultaneously), and should "self-repair".

SOME TASKS THAT REQUIRE CONSISTENT REPLICATION

IoT applications where we need to make sure all our actions are correct and reflect the most recent information

Replication of configuration parameters, input settings, 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

TODAY'S OUTLINE

Understanding our broad goal: how to build a μ-service (or set of interconnected μ-services) to be "self-sufficient" and offer **CA** and not need Brewer's **P** (they will need a different kind of **P**artition tolerance, to avoid split-brain crashes, but this is different from what Brewer meant)

Overview of the important models: crash-failure, Byzantine failure, virtual synchrony for self-managed membership

Deep dive on one simple example: Chain replication, using Zookeeper for configuration (membership) management

LESLIE LAMPORT STARTED THE AREA!

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.

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?

~~Superman~~ FRED SCHNEIDER TO THE RESCUE!

When people first became confused about how to put all of this theory into practice, Leslie didn't really respond.

Leslie explained that at the end of the day, he prefers to be seen as a theoretical computer scientist… not an engineer.

Leslie felt that although his work does define "building blocks" for state machine replication, software engineering tradeoffs 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

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 saw his role as being the definition of "building blocks" for state machine replication. Engineers should deal with practical issues.

FRED SCHNEIDER'S TUTORIAL?

In fact it didn't dive into implementation issues either.

Fred's work was more focused on showing how state machine replication could be deployed to solve higher level objectives of various kinds.

So his tutorial, although hugely impactful, still didn't somehow make state machine replication "practical" in cloud settings.

KEN TO THE RESCUE!

Early in my research career I became interested in this

My idea was simple: **Build** non-deterministic programs that can run as groups, with subsystems where the state-machine replicated data lives

- The "replicated objects" holding the data do need to be deterministic
- But the application as a whole doesn't need to be deterministic
- This became a widely accepted way around the limitation. It was first implemented in a 1985 system we called "Isis" (more on it later)

ISIS: MAIN BUILDING BLOCKS

One is called *atomic multicast*. The other is a form of atomic multicast that has a built in form of *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".

IN-MEMORY REPLICATION LIMITATION

The one issue with using atomic multicast is that our solution will be doing in-memory replication.

But if all the members fail, the state of that shard is wiped out. In many systems this is acceptable, and we just try to be sure it would be rare.

But there are many services that need persistent memory.

PERSISTENT LOGGING

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

In the early days, data was smaller and the actual "state" could still fit in memory. The logs were used mainly for recovery.

Modern cloud systems have such big data objects that today, the data might not all fit in memory, and reading the logs has become frequent

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. But can we detect failures accurately?

TIMEOUT CAN DECEIVE YOU!

Confusion about who is dead (and who looks dead, but isn't) causes Romeo and Juliette to end tragically…

In a distributed setting, we can never be sure if a crashed program is **really** dead, or only **seems** dead.

If we have enough healthy members, the system can tolerate a few crashes. If one or two processes become unresponsive we can exclude them and "move on"

Trying to be certain is impractical, unless you just press "reset" for any machine that seems at all faulty. Some datacenters actually do that!

JAMES HAMILTON

"Reboot. Reimage. Replace."

Cloud leader at Microsoft, then Amazon. James was focused on datacenters with huge numbers of computers

Don't sit around waiting to figure out if one of them is faulty.

Just reboot it. If that doesn't work, reimage it (reinstall the software). If it fails repeatedly, replace it. We call these the "three Rs"

MORE EXTREME MODELS

Crash faults aren't the only model Leslie considered. In fact, there are many models that capture extreme behavior, such as 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

Leslie devised a story to motivate this model. He set the story during the Byzantine empire, a war-torn period in southern Europe

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…

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

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!

Retreat!

Retreat!

Attack!

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

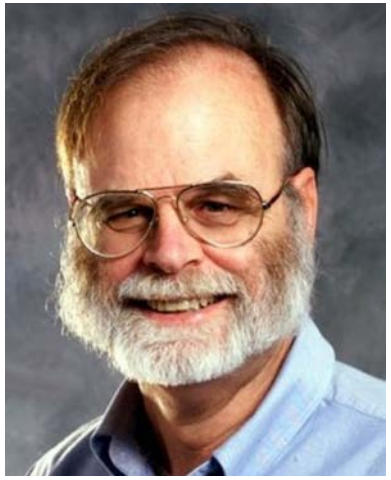
BASIC IDEA FOR ONE SIMPLE PROTOCOL

Byzantine agreement requires 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.

JIM GRAY: BUT FAILURES AREN'T BYZANTINE!



Around when people were discussing this, Jim Gray did a famous study: "How do programs fail, and what can be done about it?"

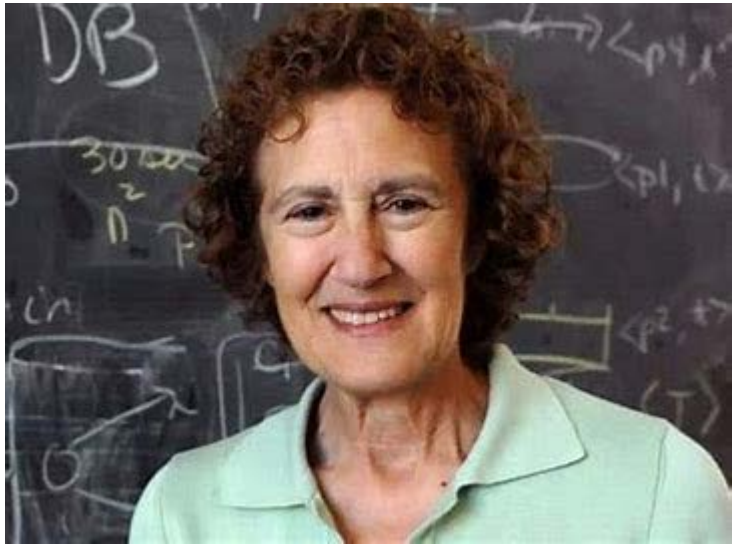
Mostly, the issue was software bugs, causing crashes. He argued for much better testing and quality assurance.

With this, failures still occur, but they usually cause programs to halt (crash).

IS BYZANTINE AGREEMENT EVER NEEDED?

Barbara Liskov and her student Miguel Castro created tools for Byzantine state-machine replication for objects and services.

The protocols are costly, but evolved into the ones used in blockchain systems, where Byzantine attacks are a common and real issue



Barbara Liskov won a Turing Award in 2008 for her many contributions to programming languages and systems

… BACK TO ISIS!

When I first came to Cornell in 1982, my initial research project focused on implementations of state machine replication in object oriented computing frameworks – like Java. We used the technique in individual objects.

The Isis Toolkit (1995) was the first practical solution in this area.

WHY "ISIS"?

Reference to Egyptian mythology.

In the Book of the Dead, Osiris is 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 picks up the pieces of your system and brings it back to life.

VIRTUAL SYNCHRONY

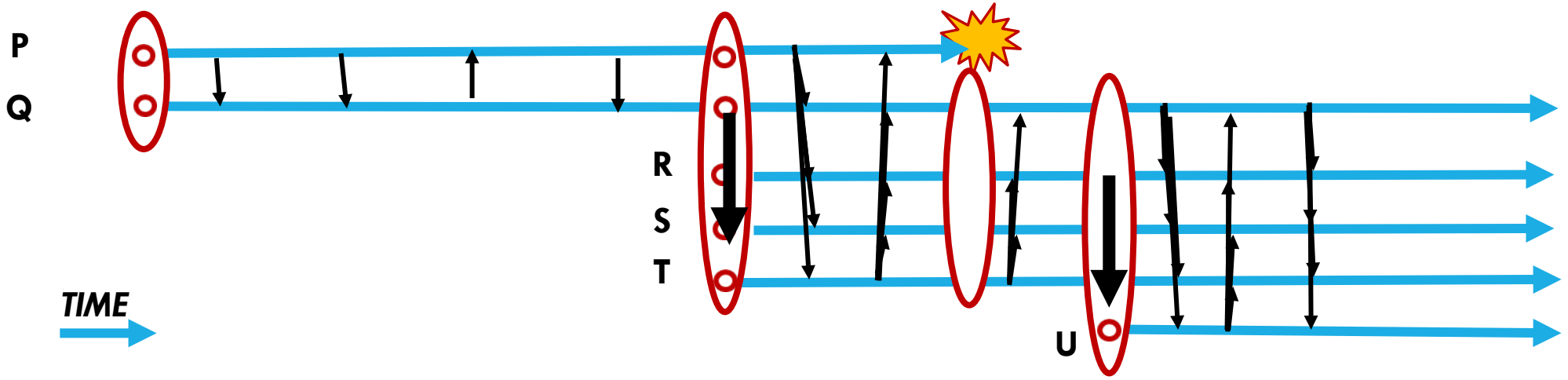
Isis does use atomic multicast and persistent replication, to support state machine replication, but it innovated by first addressing self-management.

This feature tracked who is in each group and shard. Then data was replication by the members in the list.

We called this model virtual synchrony.

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…

SPLIT-BRAIN CONCERN

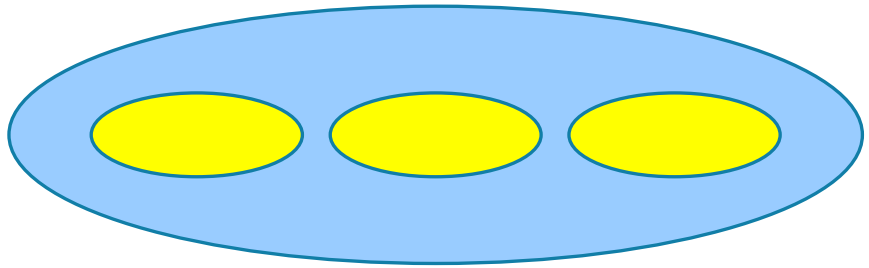
Tes.com

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 worry: What if P thinks Q has failed, and Q thinks P has failed?

SPLIT-BRAIN CONCERN

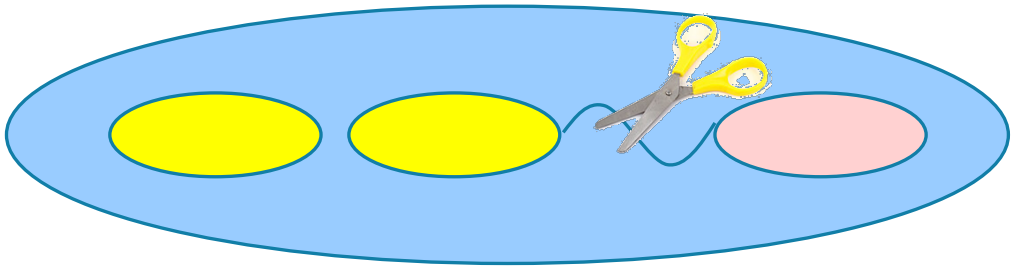


In fact, a split-brain worry is a kind of **partitioning tolerance** concern!

In CAP, Eric Brewer used the term*partition tolerance,* but he meant "μ-service A must reply to requests even if μ-service B is unreachable".

Here, we are worrying about a partition within the members of the μ-service itself, not between two different μ-services.

SPLIT-BRAIN CONCERN



In fact, a split-brain worry is a kind of **partitioning tolerance** concern!

In CAP, Eric Brewer used the term*partition tolerance,* but he meant "μ-service A must reply to requests even if μ-service B is unreachable".

Here, we are worrying about a partition within the members of the μ-service itself, not between two different μ-services.

WHAT CAUSED THE PROBLEM?

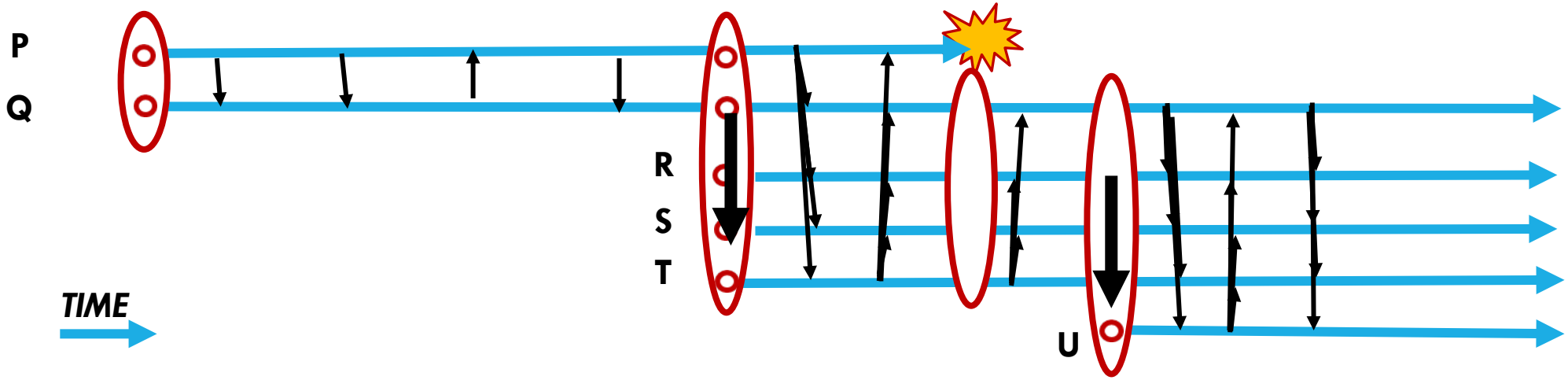
A split service could leave us with two distinct "views" of the service, each having different members.

What we really want is to be sure that at all times, there is just one official membership.

Isis solved this using a majority rule: a majority of the members from view k must approve membership for view k+1. Members can only vote once.

NOW WE CAN SEE HOW VIRTUAL SYNCHRONY WORKS, AND WHY IT SOLVES THE ISSUE!

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

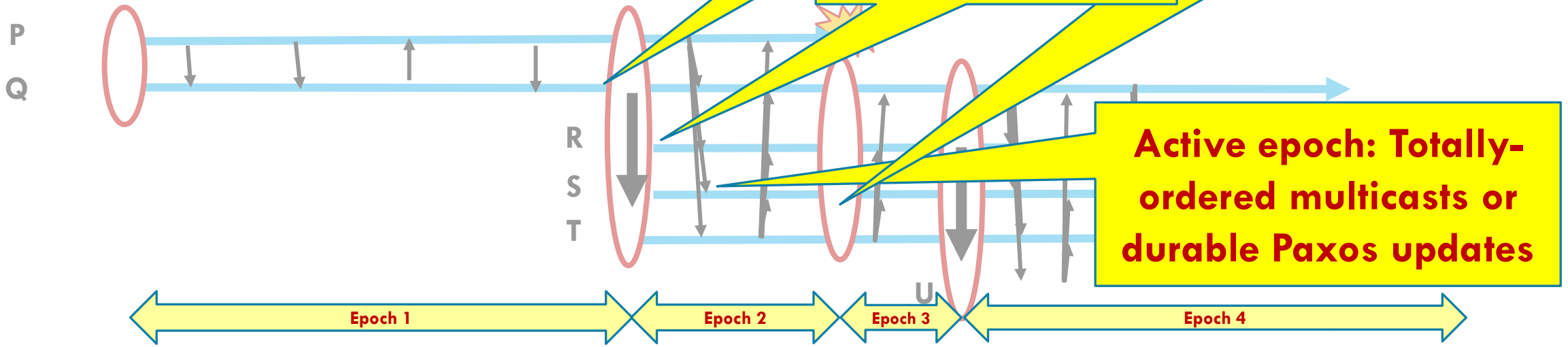


Joins, failures are "clean", state is transferred to joining members

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

NOW WE CAN SEE HOW V… ONY WORKS, AND WHY IT SOLVE… THE ISSUE

Epoch: A period from one membersh… e.



Joins, failures are "clean", state is transferred to joining members

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

BACK TO TODAY'S GOALS FROM SLIDE 6

✓ **Understanding our broad goal:** how to build a μ-service (or set of interconnected μ-services) to be "self-sufficient" and offer **CA**

✓ **Overview of the important models:** crash-failure, Byzantine failure, virtual synchrony for self-managed membership

➡ **Deep dive on one simple example:** Chain replication, using Zookeeper for configuration (membership) management

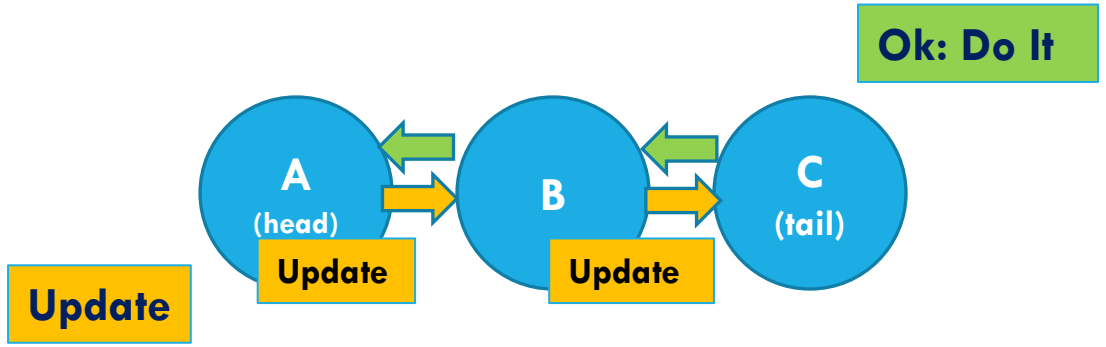
CHAIN REPLICATION

Assumes you have a way to manage the membership of your shard or replication group, but need a simple fault-tolerant way to do updates and queries.

For example, maybe you use the popular Zookeeper management service

Professors Van Renesse and Schneider proposed the following solution

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?

It needs Zookeeper: Unless you model the membership management layer, the protocol alone 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.

But with a correct membership management service, used correctly by the protocol, chain replication solves state machine replication

OTHER FAMOUS TOOLS THAT EXIST TODAY

We will learn about Zookeeper later in the course. This is a modern solution based on an approach very similar to the Isis one.

Many people have worked with a library called RaFT. RaFT was proposed as a practical software solution for modern programs that need reliability

In our next lecture we will hear about Paxos and Derecho.

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.