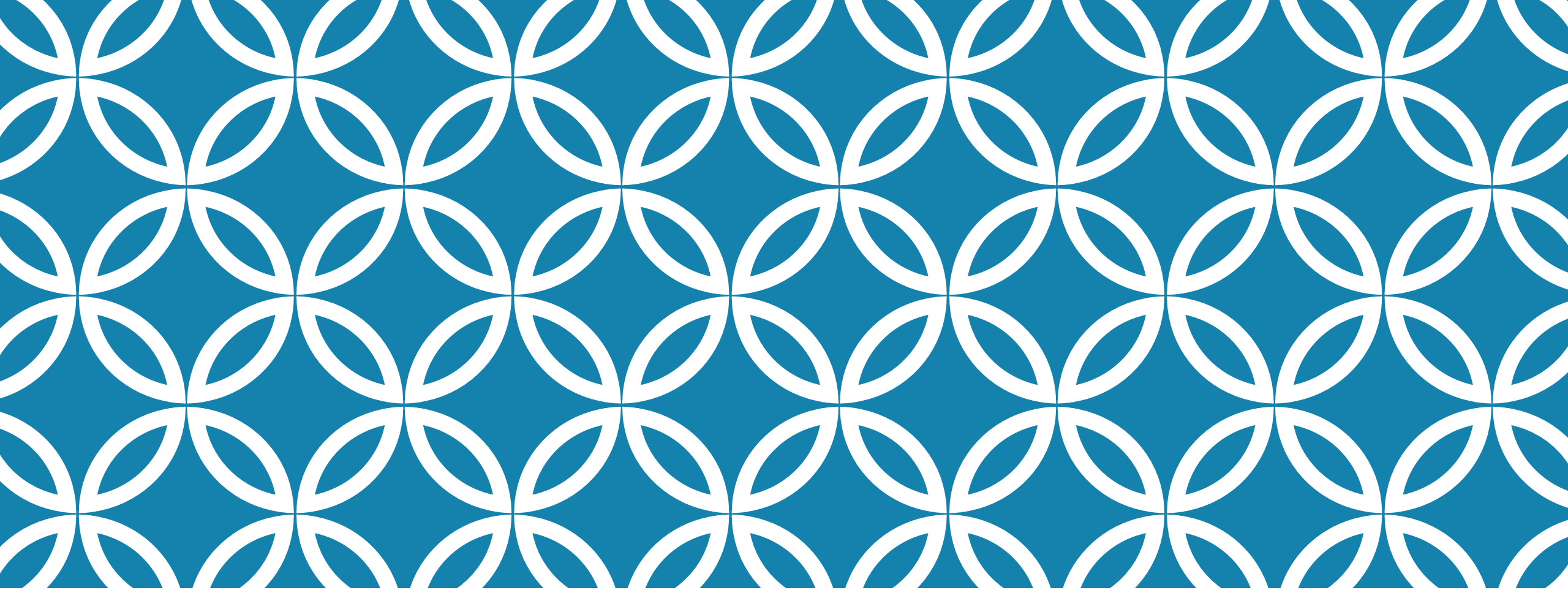


CS 5412/LECTURE 21 FAULT TOLERANCE IN APACHE

Ken Birman

Spring, 2020

HOW DO APACHE SERVICES HANDLE FAILURE?

We've heard about some of the main "tools"

- Zookeeper, to manage configuration
- HDFS file system, to hold files and unstructured data
- HBASE to manage "structured" data
- Hadoop to run massively parallel computing tasks
- Hive and Pig to do NoSQL database tasks over HBASE, and then to create a nicely formatted (set of) output files

BUT WHEN A FAILURE OCCURS…

Won't that cause "damage" all through the hierarchy?

- How do people working with Apache think about failure?
- What are the specific roles Zookeeper plays?
- What happens when a failed element later restarts?

In Derecho, we saw how all of this can be "combined" in one model (with new group views, and dynamic self-repair), but Apache applications might be spread over thousands of nodes in lots of distinct programs!

KEY ASPECTS

What does Apache do to “detect” failures?

What if a failure is just some form of transient overload and self-corrects?

- How would the component realize it was dropped by everyone else?

How can Apache self-repair the damaged components, and resume?

KEY ASPECTS

In fact Apache uses Zookeeper to sense failures.

Then it basically "cleans up", which means getting rid of partially written output from the failed components. YARN knows which files those are.

Then it restarts the things that failed. But it gives up if the same failure repeats again and again (why?)

CAN EVERY PROBLEM BE SOLVED THIS WAY?

We will be discussing this question later in the class!

We can think of Apache as a world of

- Hierarchical structure: layers and layers of very complex systems!
- Roll-forward reliability: if it fails, restart it.

But why is it even possible to "clean up"? This is the puzzle. What if an ATM machine already distributed the $500? Can we get it back?

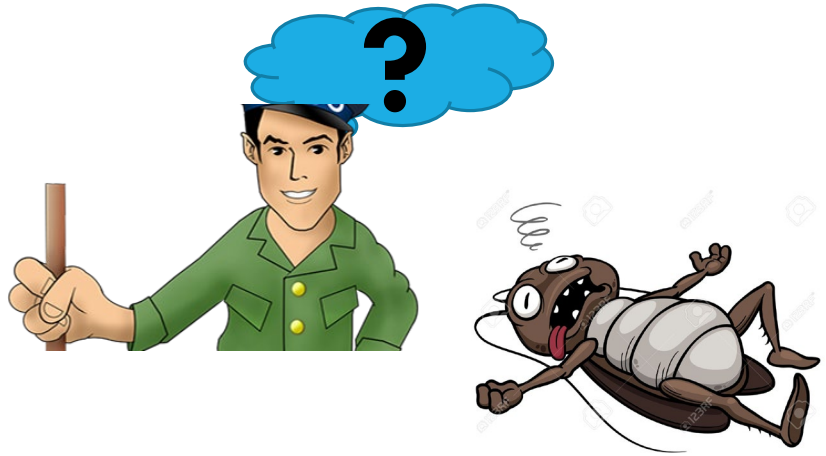
CORE OF THE PUZZLE

It is vitally important to realize that Apache big data tools don't run in an online manner!

They never "talk to an ATM machine"!

They run purely in the back end and purely in a batched context! **Why?**

WAYS TO DETECT FAILURES



Something segment faults or throws an exception, then exits

A process freezes up (like waiting on a lock) and never resumes

A machine crashes and reboots

SOME REALLY WEIRD EXAMPLES

Suppose we just trust TCP timeouts. But FTP and some applications have more than one TCP connection open between the same processes.

- What if one connection breaks but the other doesn't?

  … can you think of a way to easily cause this?

- What if process A in some pool of servers thinks S is down, but B is happily talking to S?

When clocks "resynchronize" they can jump ahead or backwards by many seconds or even several minutes.

- What would that do to timeouts?

SLOW NETWORK LINKS CAN MIMIC CRASHES

MIT Theoreticians Fischer, Lynch and Paterson modelled fault-tolerant agreement protocols (consensus on a single bit, 0/1). This is easy with perfect failure detection, but can we implement perfect detection?.

They proved that in an asynchronous network (like an ethernet), any consensus algorithm that is guaranteed to be correct (consistent) will run some tiny risk of indefinitely stalling and never picking an output value.

One implication: on an ethernet, perfect failure sensing is impossible!

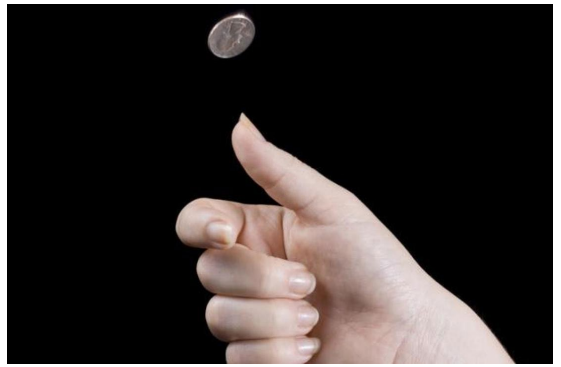
HOW DOES THE "FLP" PROOF WORK?

They look at agreeing on consensus via messages, with no deadlines on message delivery.

Their proof first shows that there must be some input states in which there is a mix of 0 and 1's proposed by the members, and where both are possible outcomes (thinking of an election, with two candidates).

They call this a "bivalent" state, meaning "two possible vote outcomes"

EXAMPLE OF A BIVALENT STATE



Suppose we are running an election and 0 represents voting for John Doe, whereas 1 represents a vote for Sally Smith. Majority wins. But N=50. To cover the risk of ties, we flipped a coin: in a tie, Sally wins.

- Suppose half vote John, half for Sally, but one voter has a "connectivity problem". If that vote isn't submitted on time, it won't be tallied.
- With 25 each, Sally is picked. But if just one Sally vote is delayed, then the exact same election comes out 25 for John, 24 for Sally… John wins

An algorithm that "tolerates failures" can't simply wait! It has to decide.

CORE OF FLP RESULT

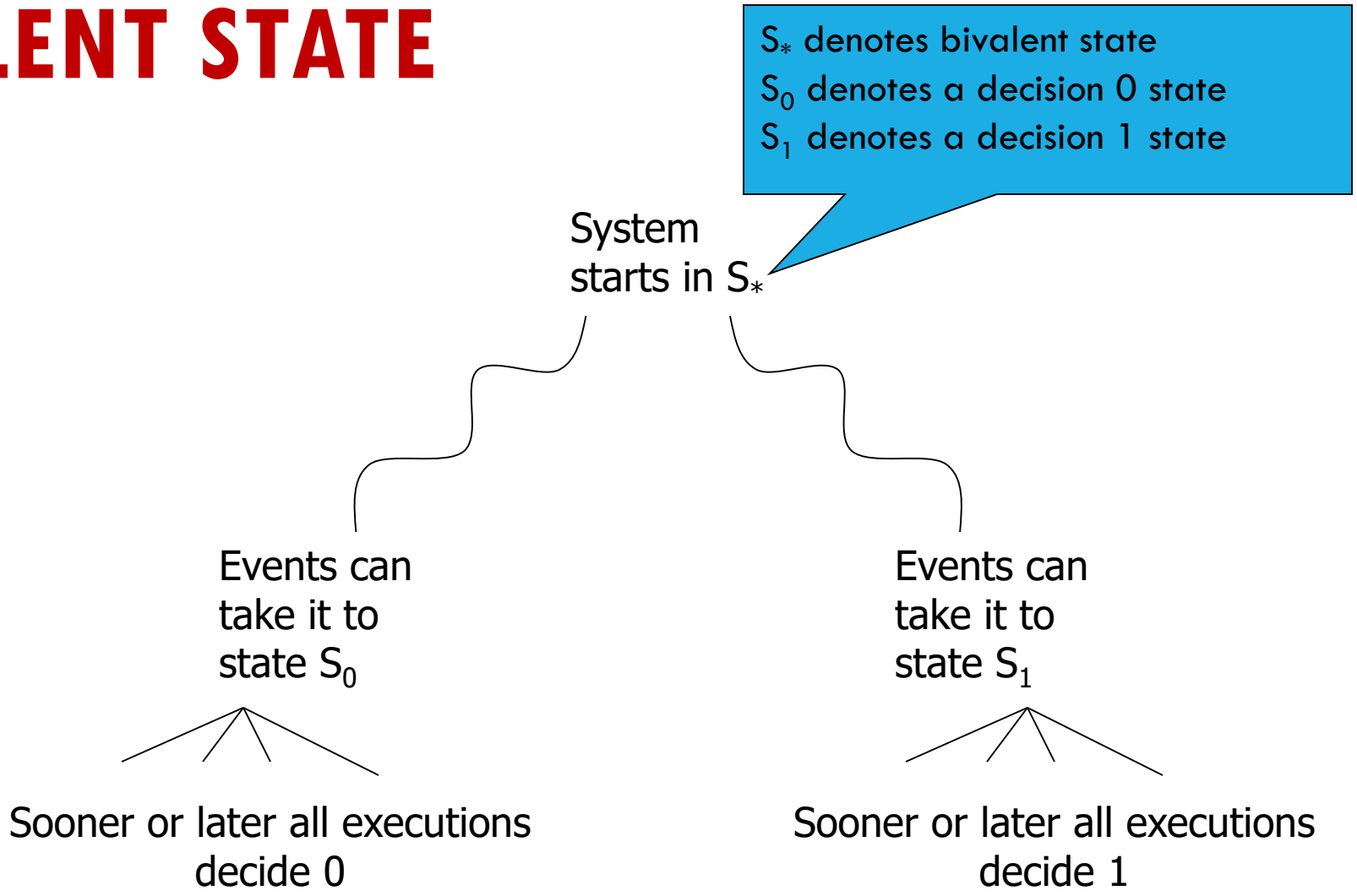
Now they will show that from this bivalent state we can force the system to do some work and yet still end up in an equivalent bivalent state

Then they repeat this procedure

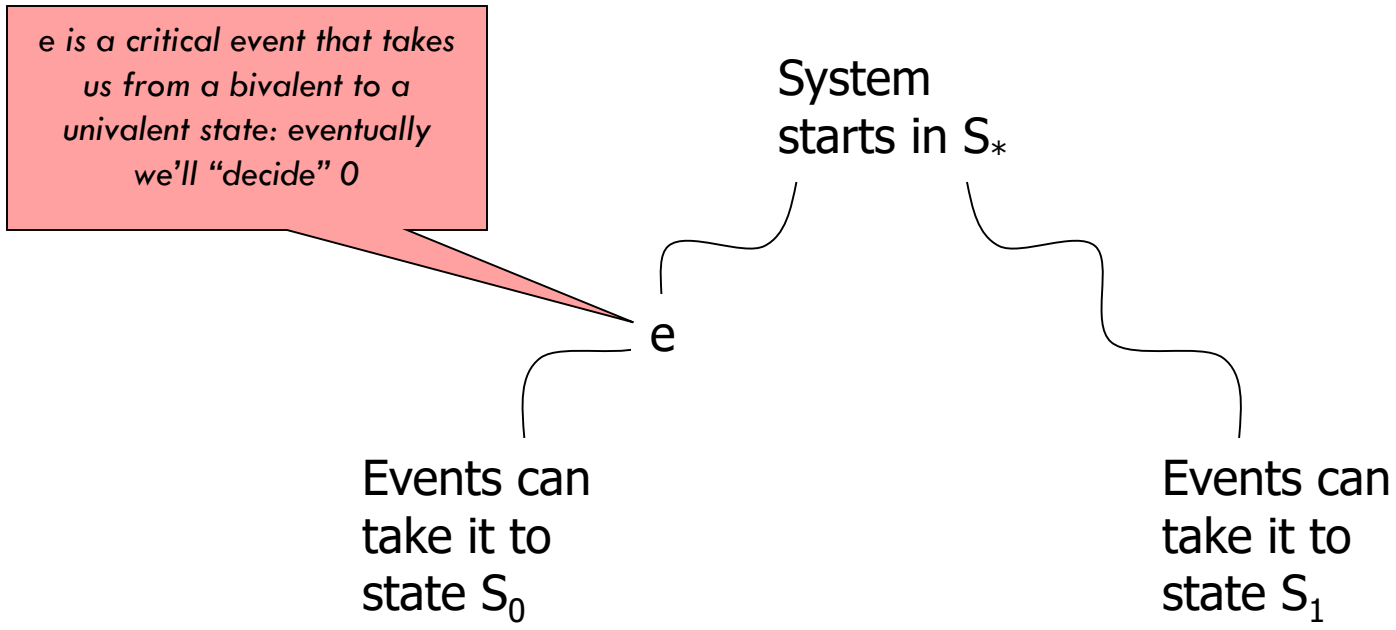
Effect is to force the system into an infinite loop!

- And it works *no matter what correct consensus protocol you started with.* This makes the result very general

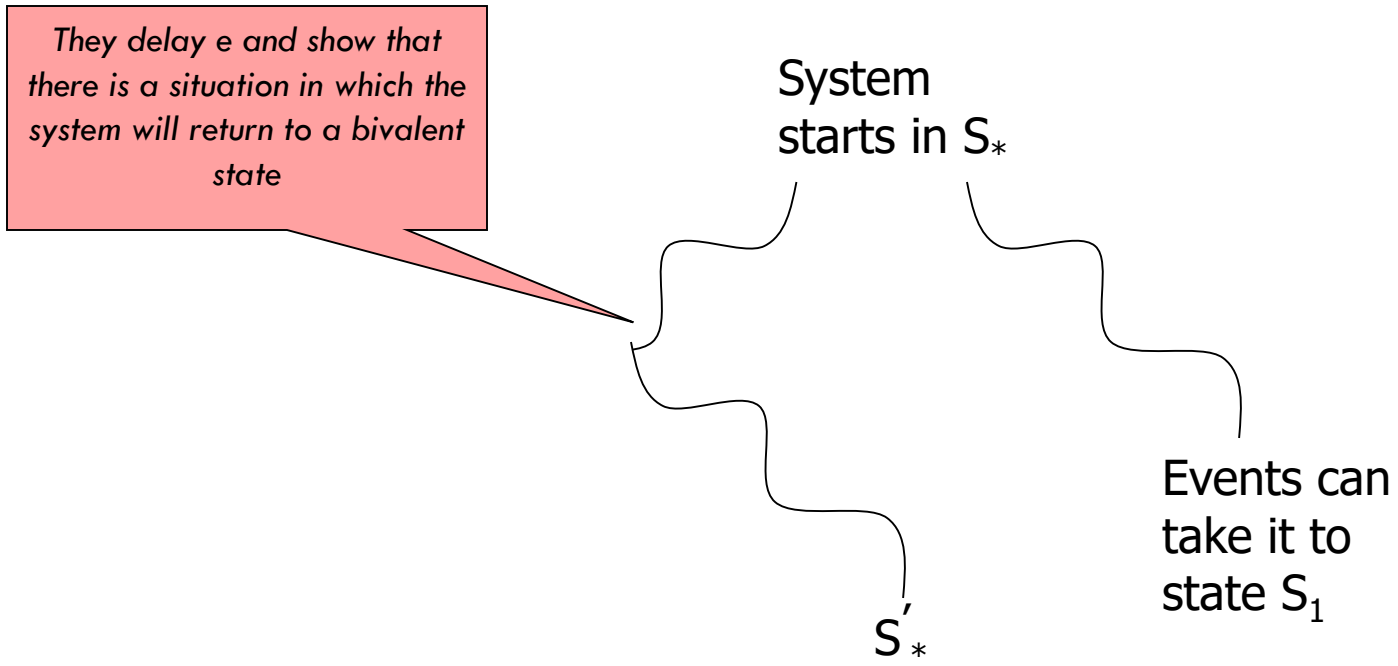
BIVALENT STATE



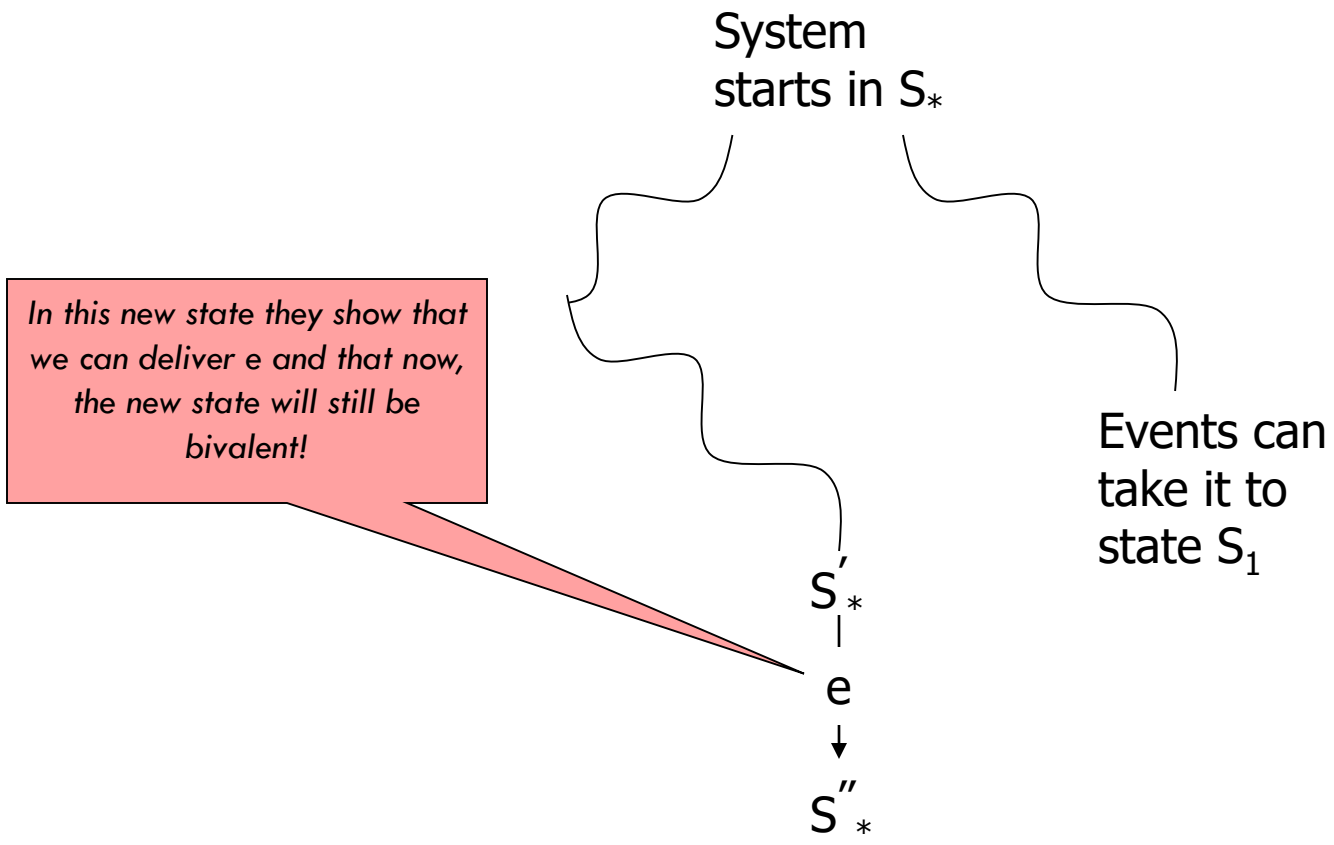
BIVALENT STATE



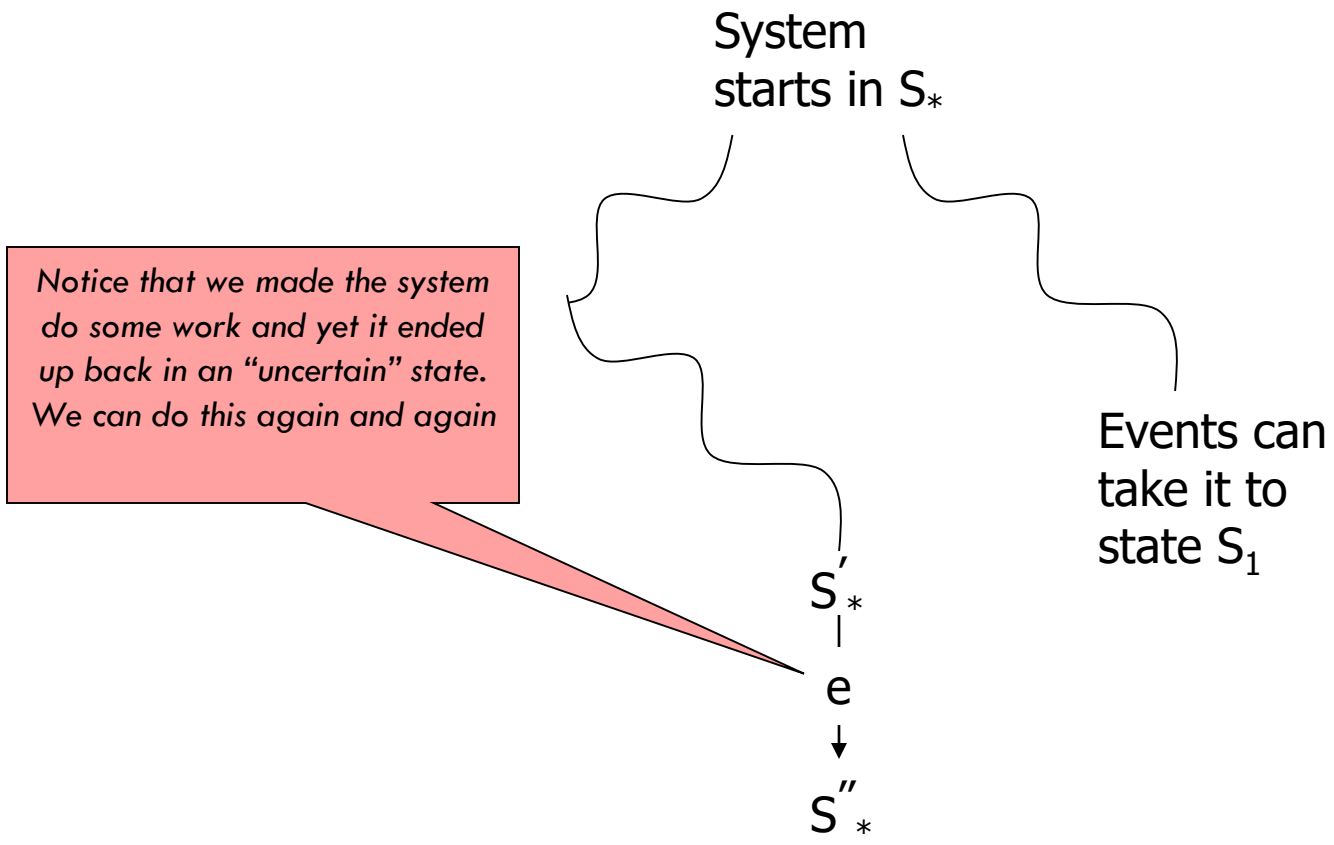
BIVALENT STATE



BIVALENT STATE



BIVALENT STATE



CORE OF FLP RESULT IN WORDS

In an initially bivalent state, they look at some execution that would lead to a decision state, say "0"

- At some step this run switches from bivalent to univalent, when some process receives some message m
- They now explore executions in which m is delayed

CORE OF FLP RESULT

So:

- Initially in a bivalent state
- Delivery of m would make us univalent but we delay m
- They show that if the protocol is fault-tolerant there must be a run that leads to the other univalent state
- And they show that you can deliver m in this run without a decision being made

This proves the result: they show that a bivalent system can be forced to do some work and yet remain in a bivalent state.

- If this is true once, it is true as often as we like
- In effect: we can delay decisions indefinitely

BUT HOW DID THEY "REALLY" DO IT?

Our picture just gives the basic idea

Their proof actually proves that there is a way to force the execution to follow this tortured path

But the result is very theoretical…

- … to much so for us in CS5412
- So we'll skip the real details

INTUITION BEHIND THIS RESULT?

Think of a real system trying to agree on something in which process p plays a key role

But the system is fault-tolerant: if p crashes it adapts and moves on

Their proof "tricks" the system into thinking p failed

- Then they allow p to resume execution, but make the system believe that perhaps q has failed
- The original protocol can only tolerate1 failure, not 2, so it needs to somehow let p rejoin in order to achieve progress

This takes time… and no real progress occurs

BUT WHAT DID "IMPOSSIBILITY" MEAN?

In formal proofs, an algorithm is totally correct if

- It computes the right thing
- And it always terminates

When we say something is possible, we mean "there is a totally correct algorithm" solving the problem

FLP proves that any fault-tolerant algorithm solving consensus has runs that never terminate

- These runs are extremely unlikely ("probability zero")
- Yet they imply that we can't find a totally correct solution
- And so "consensus is impossible" ("not always possible")

HOW DID THEY PULL THIS OFF?

A very clever adversarial attack

- They assume they have perfect control over which messages the system delivers, and when
- They can pick the exact state in which a message arrives in the protocol

They use this ultra-precise control to force the protocol to loop in the manner we've described

In practice, no adversary ever has this much control

IN THE REAL WORLD?

The FLP scenario "could happen"

- After all, it is a valid scenario.
- … And any valid scenario can happen

But step by step they take actions that are incredibly unlikely. For many to happen in a row is just impossible in practice

- A "probability zero" sequence of events
- Yet in a temporal logic sense, FLP shows that if we can prove correctness for a consensus protocol, we'll be unable to prove it live in a realistic network setting, like a cloud system

SO…

Fault-tolerant consensus is…

- Definitely possible (not even all that hard). Just vote!
- And we can prove protocols of this kind correct.

But we can't prove that they will terminate

- If our goal is just a probability-one guarantee, we actually *can* offer a proof of progress
- But in temporal logic settings we want perfect guarantees and we can't achieve that goal

NEXT STEP IN FLP PROOF

Inspired by this example, they consider patterns of message delays that are legal in an asynchronous network and that occur in a bivalent state.

They take a situation that leads to the "John wins" outcome.

Then they show that no matter what algorithm you use, there must be some message that, if we delay it, leads to a "Sally wins" outcome.

… and now they get very tricky.

FINAL PROOF STEP

In a very elegant (sophisticated) bit of mathematics they now show that if they briefly delay that "deciding" message but then allow it through, the voting protocol must end up back in a bivalent state!

They point out that this takes time (to send and receive the messages) and yet leads back to where they started. So by repeating this behavior, consensus is never actually reached!

It is as if we are endlessly arguing over which votes we should count, and never get to the point of actually tabulating the result.

DOES FLP MATTER?

FLP is often cited as a proof that "consistency is impossible" but in fact it only tells us that any digital system could run into conditions where it jams.

We knew that.

On the other hand, it also has a problematic "implication"

- No asynchronous system can accurately detect failures of its members. (if it were possible, that would contradict FLP).

IMPLICATION?

If we can't do perfect failure sensing, we need to make do with something imperfect.

This leads to the idea of a system that manages its own membership.

If the manager layer can't be sure that some process is healthy, it is allowed to just declare that the process has failed!

HOW DO APACHE TOOLS MANAGE THEIR OWN MEMBERSHIP? ZOOKEEPER!

Zookeeper acts like a single, fault-tolerance "decider".

It can keep a list of which processes are up, and which are down. It tells everyone when this changes (the model is sometimes called virtual synchrony and these lists are sometimes called "membership views"… Derecho uses this model, and it dates back to Ken's Isis system in 1987).

Then the whole application just trusts the views.

HOW DO APACHE TOOLS MANAGE THEIR OWN MEMBERSHIP? ZOOKEEPER!

Zookeeper has an elected leader, a set of "follower" members in theμ-service, and other "application" processes. There is a TCP connection from each application to some Zookeeper member, from members to the leader, and from the leader to members.

Periodic "heartbeat" messages are sent by healthy processes. Each process watches for these heartbeats. A timeout triggers "failure suspicion". Also, if a TCP connection breaks, the live process will immediately deem the other endpoint as having crashed.

A form of Paxos prevents split-brain behavior if leader failure is suspected.

BUT CAN THIS AVOID THE FLP PROBLEM?

Zookeeper didn't invent this idea (as mentioned, Ken's Isis Tookit was the first system to use this concept).

It treats slow processes like failed ones, even if the process itself wasn't actually the cause of the slowness (even if the network was at fault).

FLP is not directly applicable: in FLP, a healthy process *must be allowed to vote*. In systems like Zookeeper, *a healthy process might be "killed" by accident, but this keeps the system alive when it might otherwise freeze up*).

EVEN SO, A THEORY PERSON WOULD ARGUE THAT ZOOKEEPER CANNOT EVADE THE FLP THEOREM

Zookeeper has to manage itself. In doing that, it needs to run consensus and theoretically, one could use FLP to attack it! If you managed to do that (it would be very hard to do), Zookeeper itself freezes up.

The probability of this happening is *zero* unless the attacker can virtualize the entire distributed system and then can control everything.

So Apache applications simply use Zookeeper to track health of system, via a special type of file Zookeeper maintains listing system members.

WHAT ABOUT PAXOS? DERECHO?

They can't avoid the FLP limitation either: without extra assumption about the network, neither can guarantee progress except for "best effort, with high probability" kinds of promises.

This said, those promises are good enough in real systems.

One interesting side-remark: In fact Zookeeper is weaker than Paxos or Derecho (because it only checkpoints its state every 5 seconds).

HDFS USE OF ZOOKEEPER VIEWS

Recall that in HDFS every file has one or more replicas. It uses chain replication, with Zookeeper tracking the chain members!

If a chain member fails, HDFS still has a healthy replica and reads can continue. It restarts the failed member or launches some new node to take on the same role, and copies data from a healthy replica if needed to repair the failed replica.

If all replicas fail, HDFS will wait for recoveries. But in the normal case, HDFS itself stays available for reads.

BIG CHALLENGE: HADOOP (MAPREDUCE)

Failures could cause some tasks to disappear.

MapReduce and Hadoop will automatically restart the failed task on some other node (they will even run extra copies of very slow task, "just in case")

Whichever task finishes first, successfully, is considered to have completed that step and the others are terminated if any are still running. If they do produce output, it is ignored.

HADOOP IS "AT MOST ONCE" RELIABLE

This basic task fault handling ensures that each Hadoop task will be performed at least once, but at most one output will be preserved.

What if a task fails while writing files?

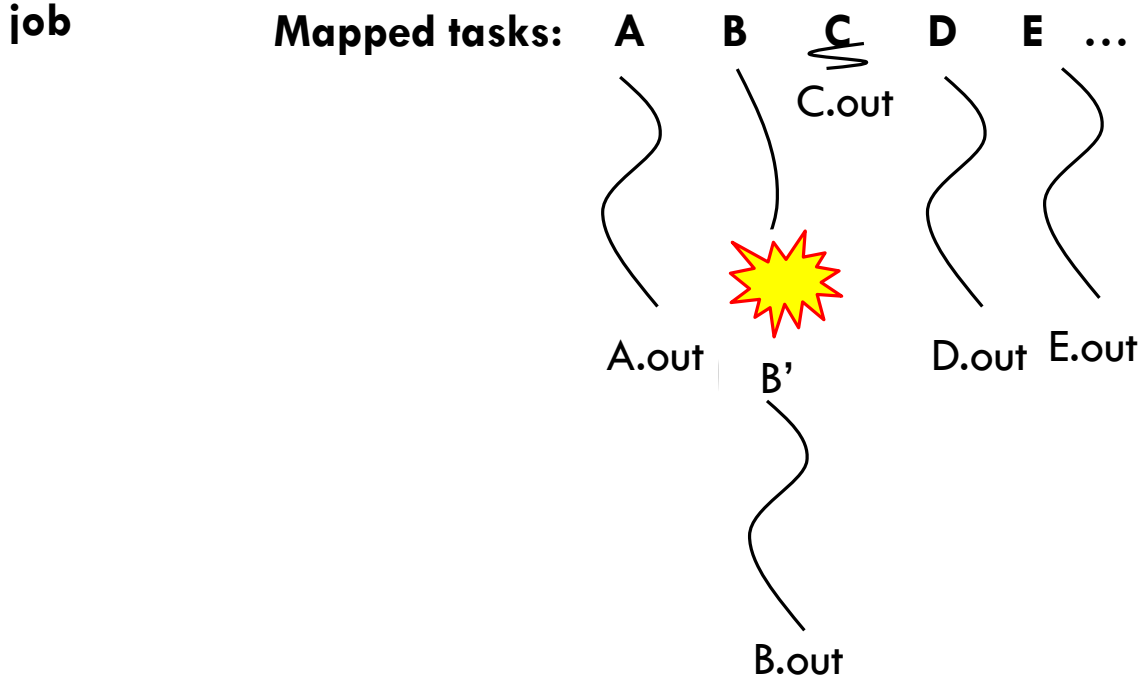
RECALL THAT HDFS IS APPEND-ONLY

We discussed the rule that HDFS uses for file updates: either create a whole new file version, or append to a file. You can't update the middle of a file – seek into the middle of an HDFS file will cause writes to fail.

… so, if some task has to be restarted, HDFS can just restore any files that task was writing to back to the length they had before the task started!

This works well because in Hadoop, every object can be constructed from some other object by some kind of repeatable ("idempotent") computation

VISUALIZING HADOOP FAILURES IN IMAGES



Normal case: A, B… E just run, create output (key,value collections in HDFS files), then the reduce step can run.

Failure case (B crashes). Now Hadoop just rolls back any files B was appending to and runs B', to repeat the task.

WHY SUCH A FOCUS ON FILES?

In fact everything in Hadoop is kept in files, even key,value tuples created by the tasks running on behalf of map, the shuffled data, the sorted version that are input to reduce, and the output from reduce.

This makes it much easier to deal with MapReduce cleanup after a failure: it just tracks what files are created by a task (it deletes the new version), and what files were extended (it restores the old length, truncating any extra data that was being written when the task failed).

THIS CONCEPT WORKS IN ALL OF APACHE

The whole Apache infrastructure centers on mapping all forms of failure handling to Zookeeper, HDFS files with this form of "rollback", and task restart!

It has similar effect to an abort/restart in a database system, but doesn't involve contention for locks and transactions, so Jim Gray's observations wouldn't apply. Apache tools scale well (except for Zookeeper itself, but it is fast enough for the ways it gets used).

DISCUSSION TOPIC (REST OF THE CLASS)

Can we employ the same Apache concept in Edge IoT systems?

To what degree does the edge need something more?

Are their edge situations where retry, or "rollback and retry" suffices?

Derecho offers a more sophisticated model of failure handling (virtual synchrony views of consistent process groups with identical replicas). What should be the rule for deciding which to use?