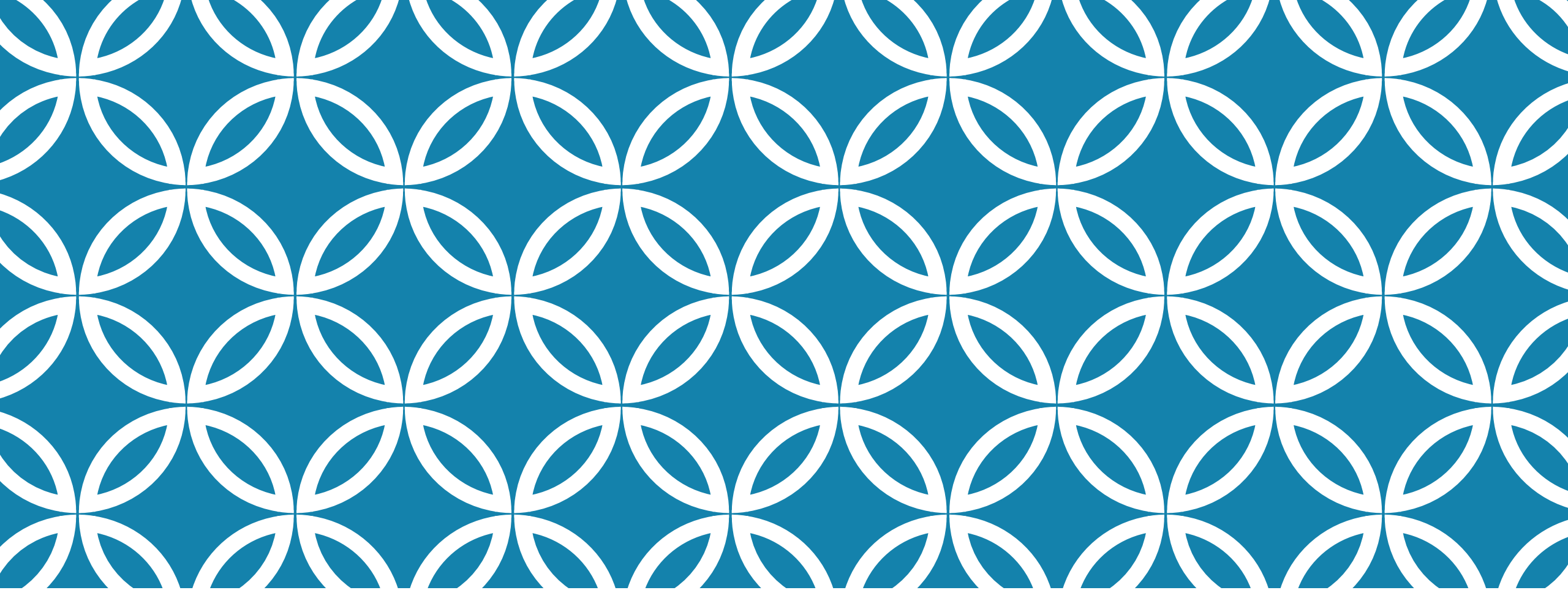


CS 5412/LECTURE 23 FAULT TOLERANCE

Ken Birman

Spring, 2021

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?
- What happened to Paxos and Derecho? Why isn't Apache using those ideas?

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. And Zookeeper trusts TCP: if a connection breaks, the client is considered to have failed. If a Zookeeper node crashes, the client must reconnect or it will be reported as faulty.

Then Apache "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 do you think it does this**?)

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?

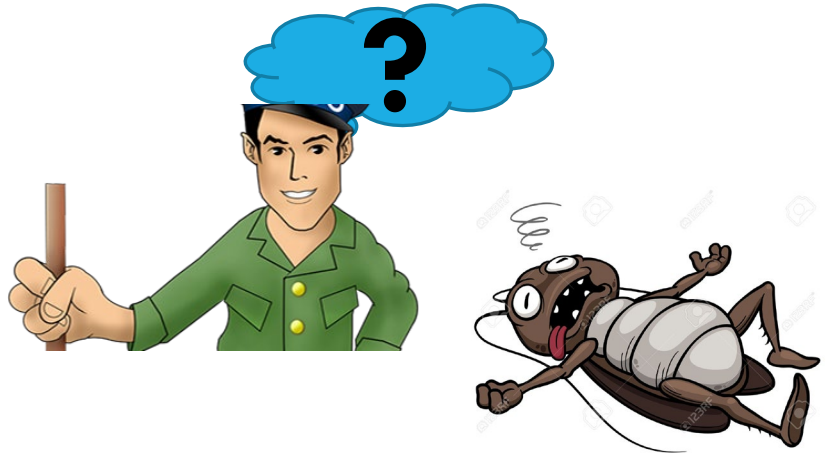
CORE OF THE PUZZLE

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

It never "talks to an ATM machine"! So the scenario of giving out money is an IoT example (an ATM is a sensor + actuator!) but not an Apache case.

Batch data analysis runs disconnected from the world! **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 have 2 connections to a process.

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

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

- What if A thinks B is down, and B thinks A is down?

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
- Can we safely make a decision here?

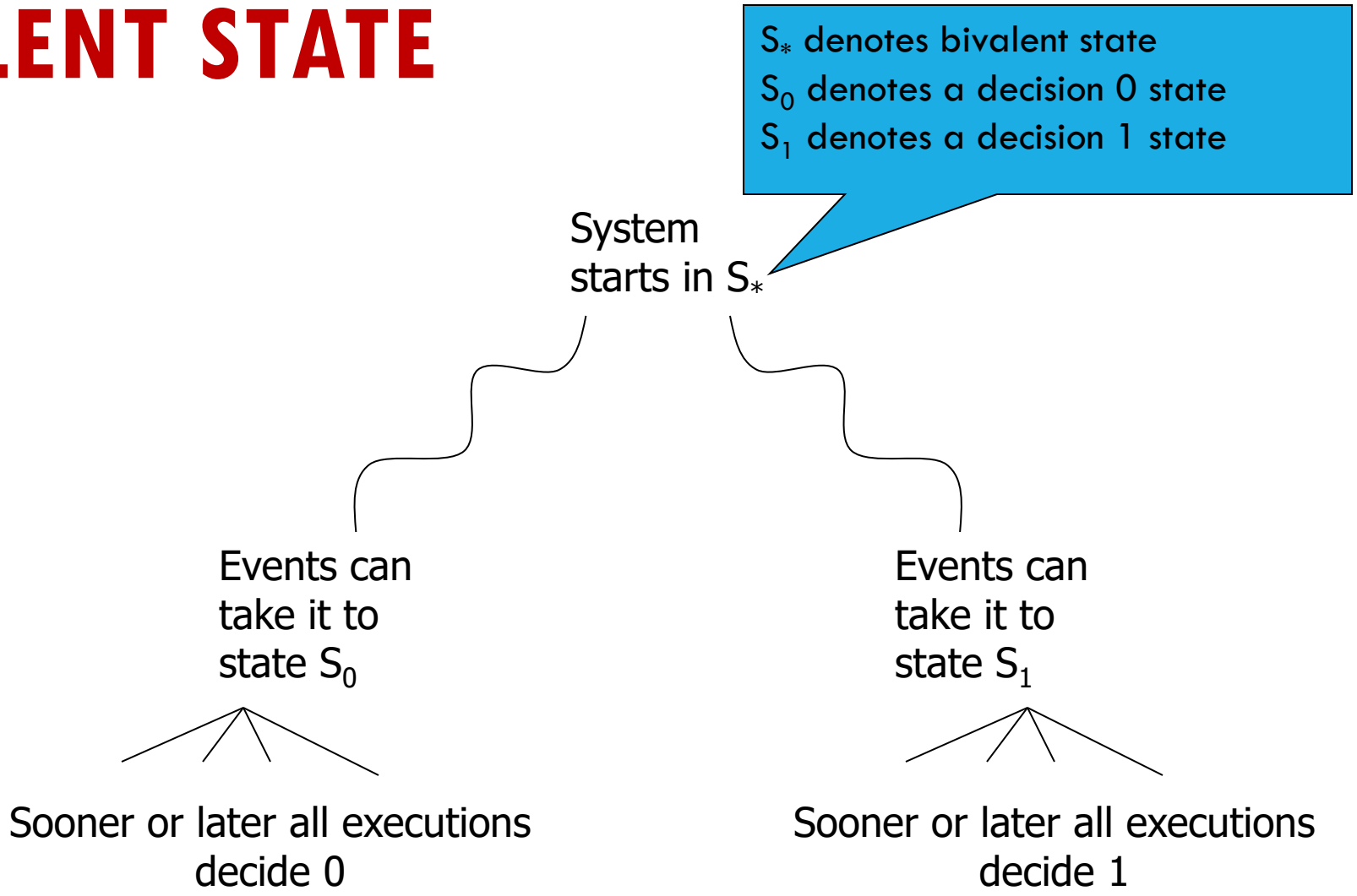
CORE OF FLP RESULT

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

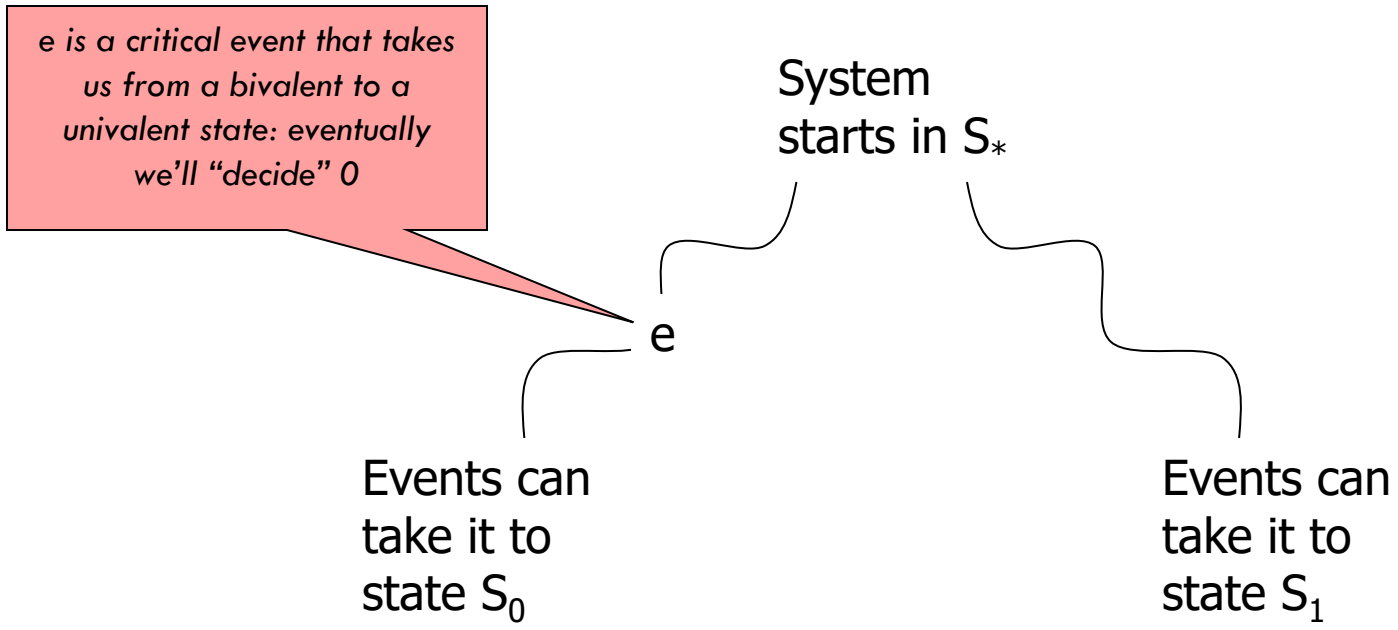
Effect is to force the system into an infinite loop!

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

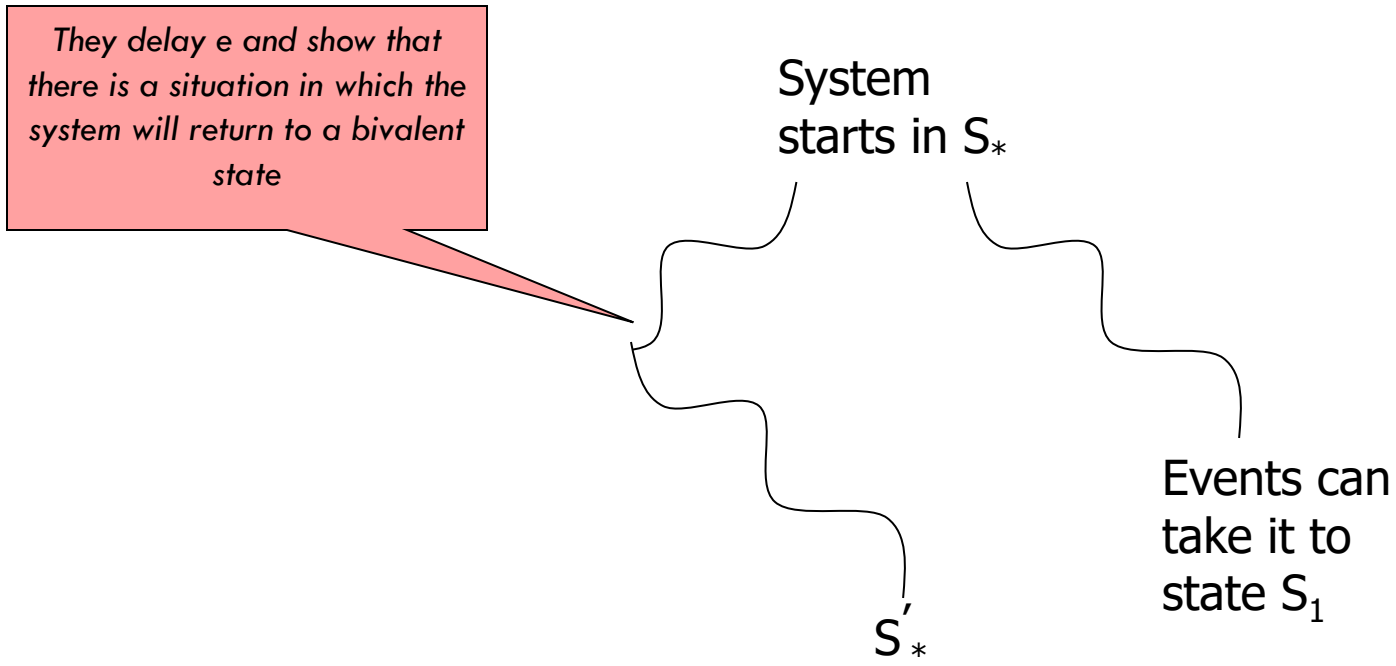
BIVALENT STATE



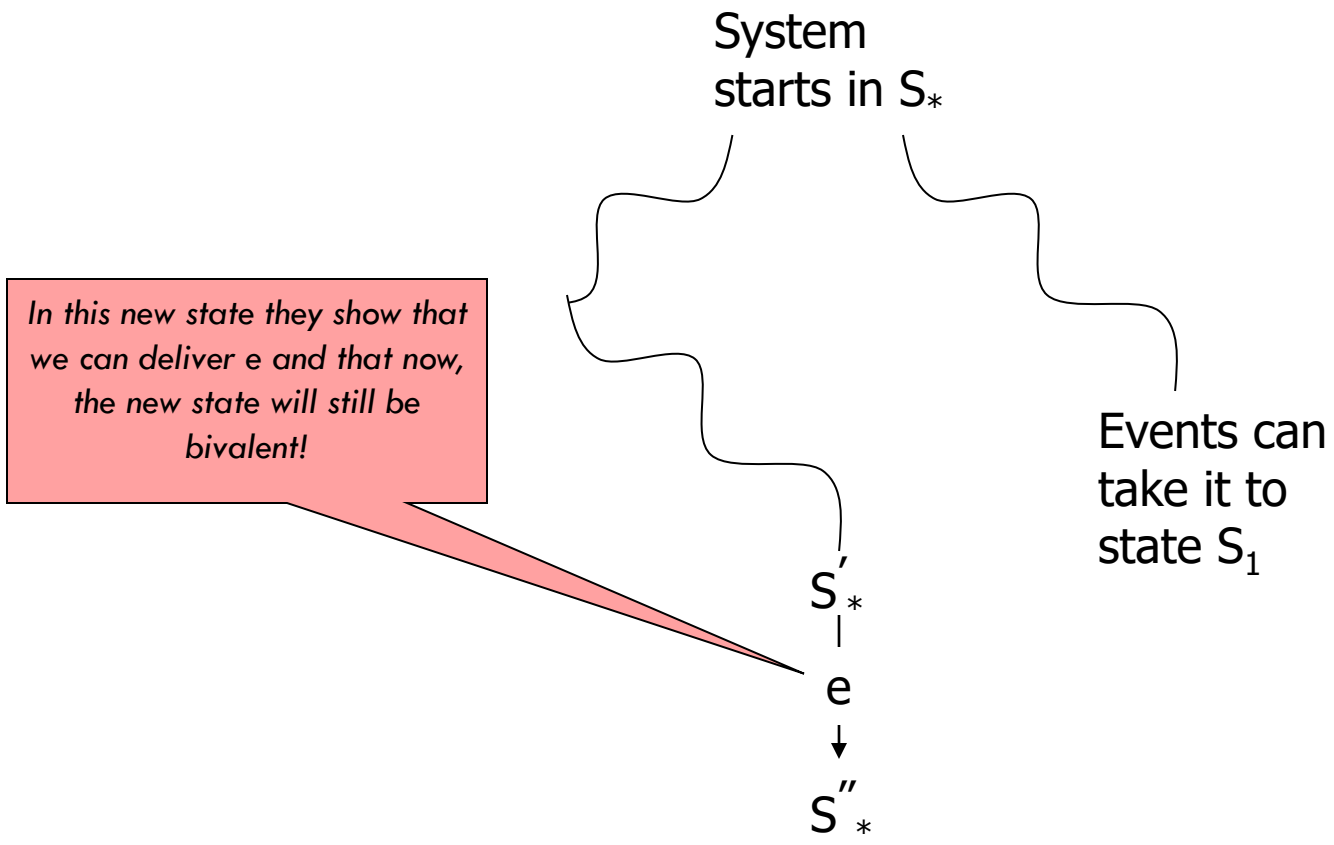
BIVALENT STATE



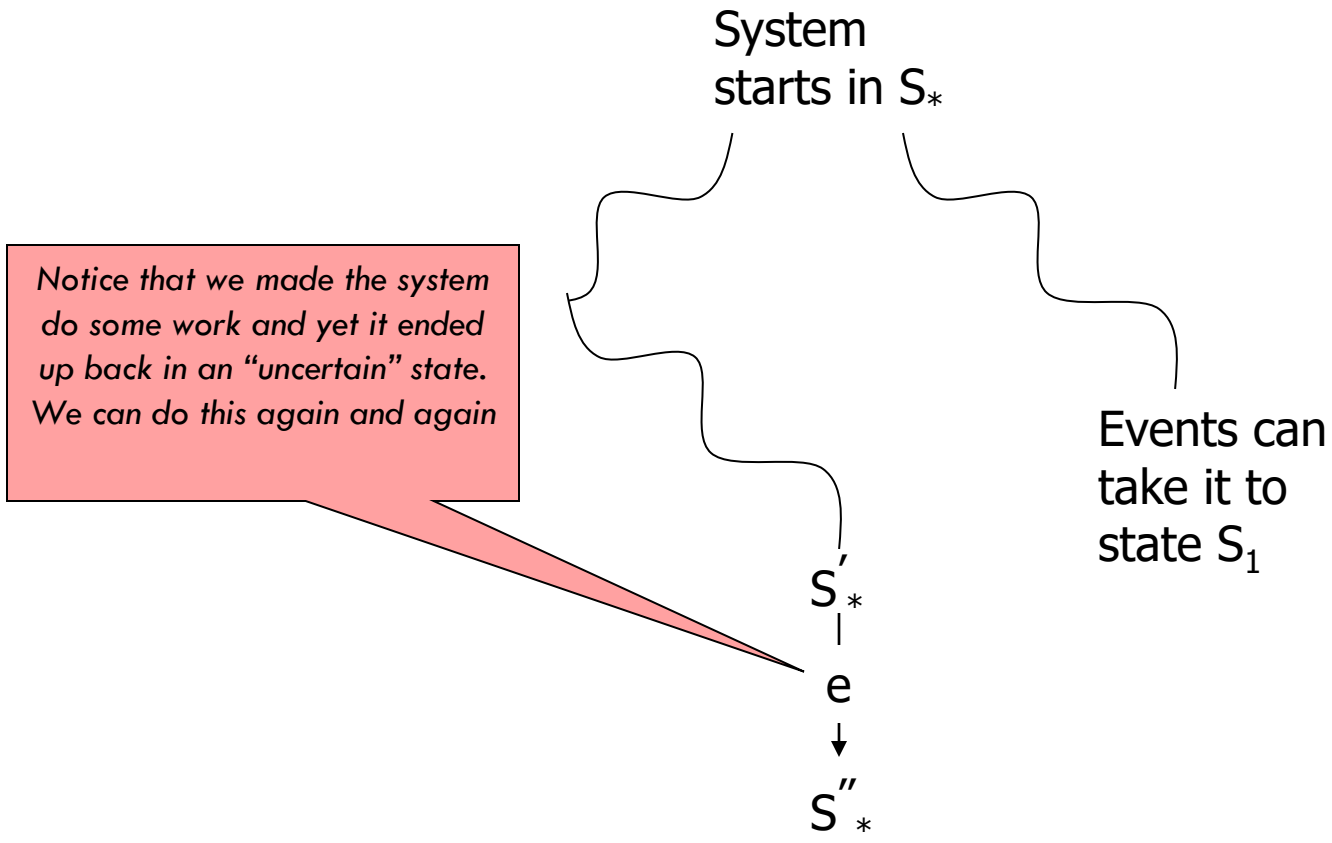
BIVALENT STATE



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

It turns out that if **m** is delayed, the system always reaches some other bivalent state before any decision can be reached.

CORE OF FLP RESULT

Now they show that there is actually a bivalent state in which they canb deliver **m**, the delayed message, *and no decision will occur*.

This form of delayed delivery

- Forced the system to do some work
- Left it in a bivalent state, just like it started.

They just loop and do this again and again. **No decision is ever reached!**

IMPLICATION?

If you have a fault-tolerant protocol able to solve consensus, like Derecho or Paxos or Chain Replication…

… and you have an all-powerful adversary who attacks the system

… it can be prevented from ever reaching a decision!

BUT WHAT DID "IMPOSSIBILITY" MEAN?

In effect, "*fault tolerant consensus is impossible*."

But do you believe this statement?

Or do you feel as if it is using a tortured concept of "possible" and "impossible" to come up with a cute claim?

AT THE CENTER OF IT: THE ADVERSARY

A very clever adversarial attack.

This is like one of those horror movies where the evil spirit can do the worst possible thing at the worst possible moment.

In practice, no adversary ever has this much control

WE LIVE IN THE REAL WORLD, NOT A MOVIE!

This is a problem!

FLP is clearly a “real” risk.

And yet the kind of attack it imagines cannot really arise! In fact it is easy to show that a system like Derecho or any Paxos protocol will make progress even with really simple added assumptions about message delays being “random” and not “controlled”

SO…

In fact the fault is in the theoretical model! It gives too much power to the attacker.

Yet at the same time, because partitioning failures can cause Paxos or Derecho to freeze up (they can lose quorum), in some sense a result similar to FLP applies in any case!

With real systems, freeze-up is a real risk… even if not due to FLP attacks!

WHAT DOES THIS SAY ABOUT ELECTIONS?

Think back to the John and Sally election scenario

In a real election, sooner or later we call a halt and count the votes that are in hand.

At that point the "votes" are immutable and the set of votes is known. It just becomes an exercise in counting – nothing more. Even the tie-breaking rule, and the outcome of the coin flip, become immutable.

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 already knew that, due to partitioning.

On the other hand, it also has a problematic "implication". It makes it very hard to prove the correctness of real systems using a pure logic formulation. We need probabilistic assumptions and goals, and only can show some high likelihood of progress.

IMPLICATION?

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

This ties back 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 DOES DERECHO DO IT?

It has a self-managed membership service, sort of like Zookeeper.

We used the term virtual synchrony for the resulting model, with membership epochs.

In fact, Zookeeper and Derecho both borrow this idea from an older system called Isis, built by Ken in the 1985-1987 period.

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?

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*).

Anyhow, this still leaves partitioning as a risk. We can't evade the risk of freezing up – we can only evade the FLP "scenario" for that happening.

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

The distributed systems theory community considers the FLP theorem to be the bottom line.

No system that can solve consensus is able to guarantee progress.

They also understand that in practical cloud settings, we may not be worried about the FLP scenario, or even the partitioning scenario (we can design a redundant network to minimize that risk…)

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

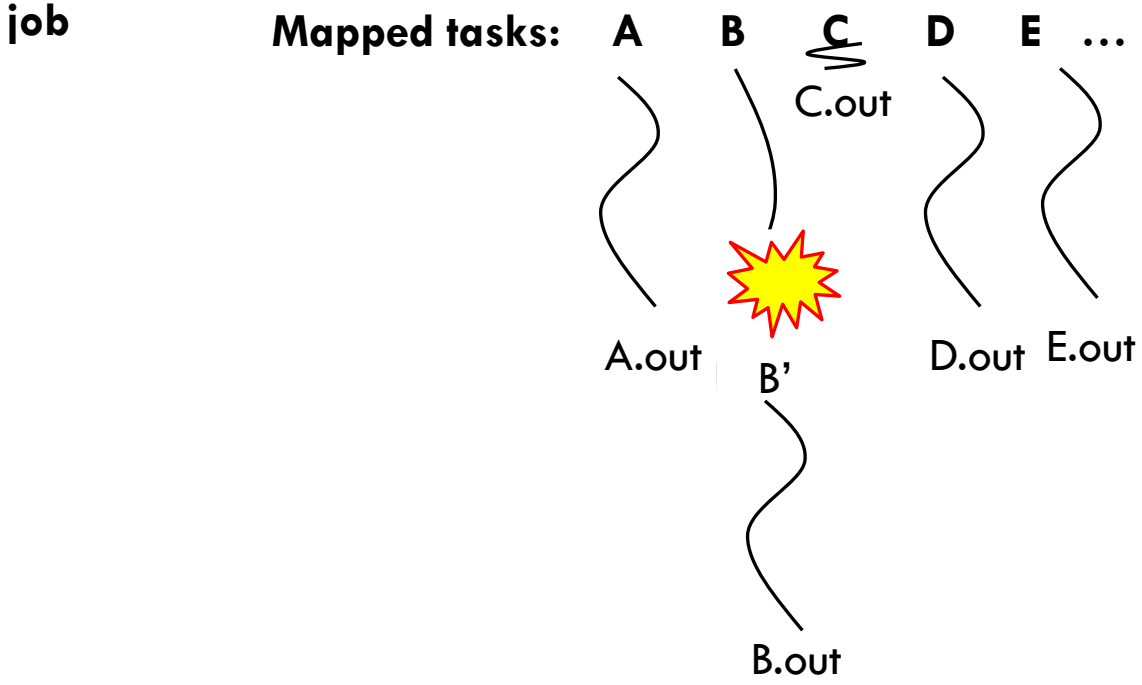
CHECKPOINTS

HDFS adds a "checkpoint" feature to what POSIX normally can support.

The checkpoint is just a file that contains the names, version numbers and lengths of the files your Hadoop application is using. To "roll back" it just truncates files back to the size they had and restores any deleted files.

This assumes that deleted file versions are kept around for a little while.

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).

BOTTOM LINE?

The cloud is highly available, because it has layers of backups – even backup datacenters and backups at geographic scale.

IoT data managed by the cloud c*an* be strongly consistent. This doesn't really reduce availability and in fact doesn't even reduce performance.

It leads to a style of coding in which membership is managed for you. But many parts of the existing cloud are using weaker consistency today, and you need to be aware of the risks when you use those tools.

EFFECTS OF BOTTOM LINE?

Today's existing cloud is weaker than it needs to be… but quite robust.

We often see stale data, especially with time-pressured updates. Solutions that use database styles of locking don't scale well.

Tools like Derecho (and soon, Cascade) offer stronger options but are not yet widely common, and you need to deploy them "by hand" to use them.