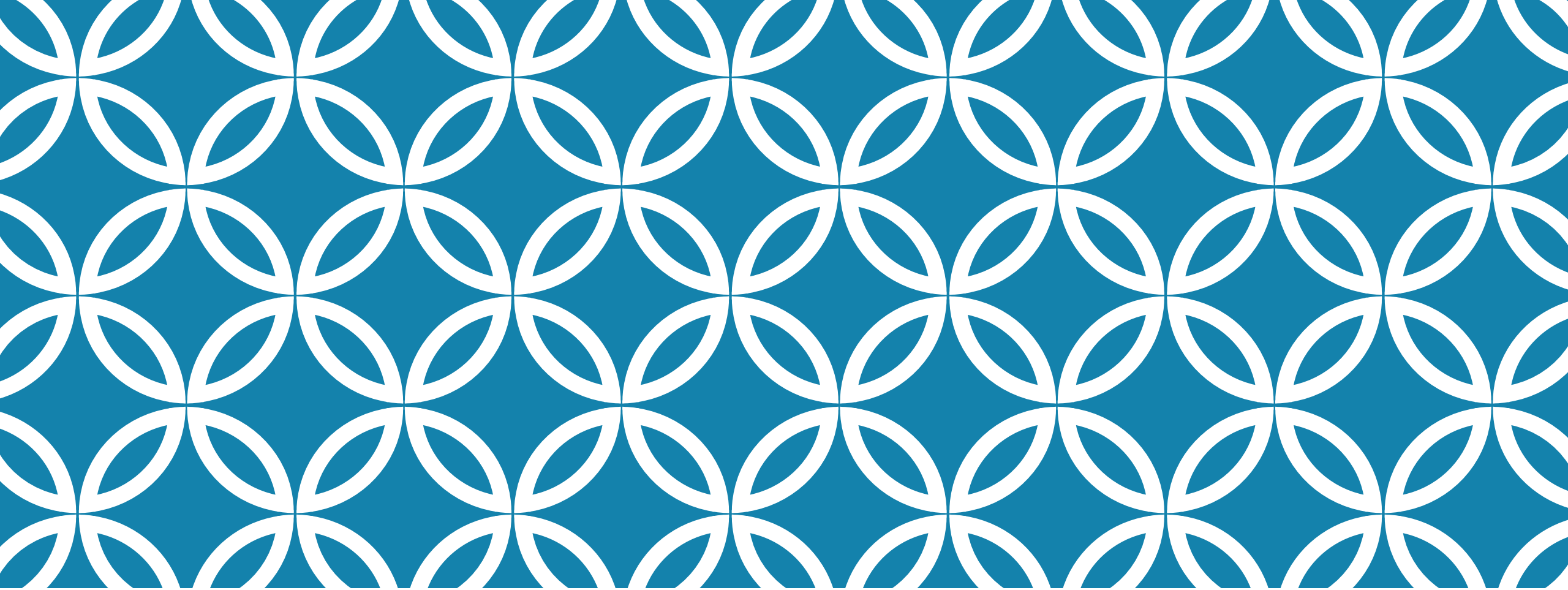


CS 5412/LECTURE 23 FAULT TOLERANCE THEORY

Ken Birman

Fall, 2022

BREAKDOWN OF OUR TOPIC

[**Today**] What does the theory tell us?

[**Wednesday**] How do systems like HDFS self-repair after failure?

- What happens when systems are built in layers?
- How can a "restarted" layer be sure that the storage layers it depends on are already fully repaired and correct?

SLOW NETWORK LINKS CAN MIMIC CRASHES

MIT Theoreticians Fischer, Lynch and Patterson modelled fault-tolerant agreement protocols (consensus on a single bit, 0/1).

This is easy with perfect failure detection, but if we aren't allowed to unplug and reboot machines, how 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.

THE FLP RESULT

Micheal Fischer

Nancy Lynch

Micheal Patterson

The impossibility of asynchronous consensus with one faulty process.
M. Fischer, N. Lynch, M. Patterson. JACM 32:2, April 1985, pp. 374-382.

This is one of the most famous theory results in modern distributed computing, and yet has some very peculiar aspects!

In particular their proof never considers any actual crashes! And "impossible" turns out to have a specific, somewhat unexpected definition…

HOW DOES THE "FLP" PROOF WORK?

They look at an agreement ("consensus" on **0** or **1**) in an asynchronous setting, with N initial processes of which at most one may crash.

They assume that the developer of the consensus algorithm claims:

- **Safety.** The outcome is said to be a "decision" value, and there can only be one outcome. The algorithm must decide a value that some process voted for.
- **Liveness.** Every non-crashed process eventually decides 0 or 1.

Goal: Show that if the protocol is safe, it cannot be live.

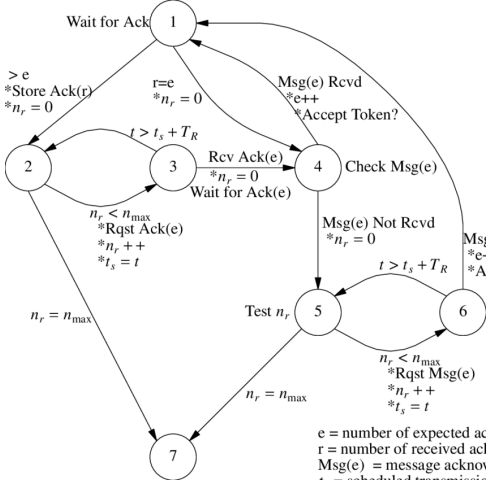
SOME IMPORTANT DEFINITIONS

The network is *reliable* but has no concept of time. No synchronized clocks or time bounds on message delays or process actions.

Messages always get there, but without clocks, timing is "arbitrary"

This is a standard way to define an asynchronous network and covers everything from RDMA and TCP all the way to communicating with satellites leaving the solar system on their way to other stars

SOME IMPORTANT DEFINITIONS



They assume that **membership is fixed** and that the consensus protocol runs as a **deterministic** state machine.

The code is organized as a graph. Nodes represent states, and edges are actions triggered by "events". An action can include sending a message.

An event is the delivery of a message or of a *null* message. The network might deliver a null even if there are messages in transit.

WHAT ABOUT NON-DETERMINISTIC PROTOCOLS?

Imagine that Don and Joe are running for President.

But now suppose the outcome is a perfect tie. How would you resolve this?

One option is to define a tie-resolution in advance. Eg: When the votes are all counted, if the outcome is a two-way tie, a coin toss decides the winner!

WHAT ABOUT NON-DETERMINISTIC CONSENSUS?

…. But a coin flip is non-deterministic. My challenge to you:

How would you "encode" non-determinism using the formalism that the FLP researchers adopted?

You don't get to change the rules.

- The code will still consist of deterministic state machines
- An event is still a message delivery or a null-message delivery.

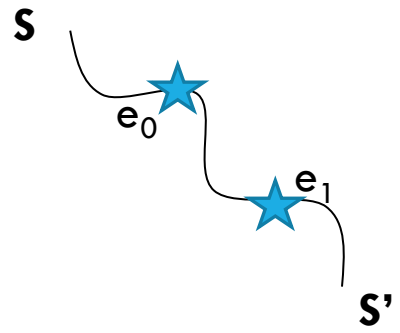
NEXT STEP… CONCEPT OF A SYSTEM STATE

A system is just a set of processes and pending messages.

Its state is the list of process states and the list of messages.

Often denoted by capital-S in FLP paper.

A RUN OF A SYSTEM



In FLP we think about sequences of states the system might pass through.

For this we need to know a starting state, call it S.

Then we are given the sequence of events.

- Call these e_0, …. e_k
- Each is a tuple: (p, m) or (p, ⊥)

NOTICE THAT A RUN IS DETERMINISTIC

The processes really have no freedom at all

The network decides the events (think of it as an active scheduler).

- Of course, it can't deliver a message that has never been sent.
- But can decide whether to deliver a pending message or a null.

Replaying the same event sequence gives the same final state.

STOPPING FAILURES

Processes fail by crashing: Some process suddenly ceases to do anything.

This is called a stopping failure. Messages to this process go into a black hole. Messages it sent before stopping do get delivered (eventually).

We lack a reliable means of detecting failures, yet because the consensus protocol is fault-tolerant, it still needs to reach a final decision state!

HOW DOES THE PROOF WORK?


Micheal Fischer

Nancy Lynch



Micheal Patterson

The authors first show that there must be some input states in which the processes votes include a mix of 0 and 1's, and where both are possible outcomes (either could be selected).

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

They show that we can force the protocol to do some work, but still end up back in a bivalent state. And then they repeat this… forever!

NON-TRIVIALITY

The FLP authors needed to rule out an algorithm like "no matter how people want to vote, the winner will always be 1"

So they require a *non-trivial* algorithm. Every process has its own vote.

- If all vote 1, the outcome must be 1. If all vote 0, it must pick 0.
- With a mixture of 0 and 1 votes, the algorithm can decide how to tabulate. It isn't required to go for a majority.

FINDING A BIVALENT INITIAL STATE

Consider a state that is all 0 but just one vote of 1.

The process voting 1 could fail before voting, and if so, the situation is identical to all 0's. So, the outcome must be 0.

Next, consider a state with two 1's, then with three 1's. Eventually they reach an initial state that can lead to both possible outcomes: 0 or 1, depending on whether a process fails.

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

INTUITION?

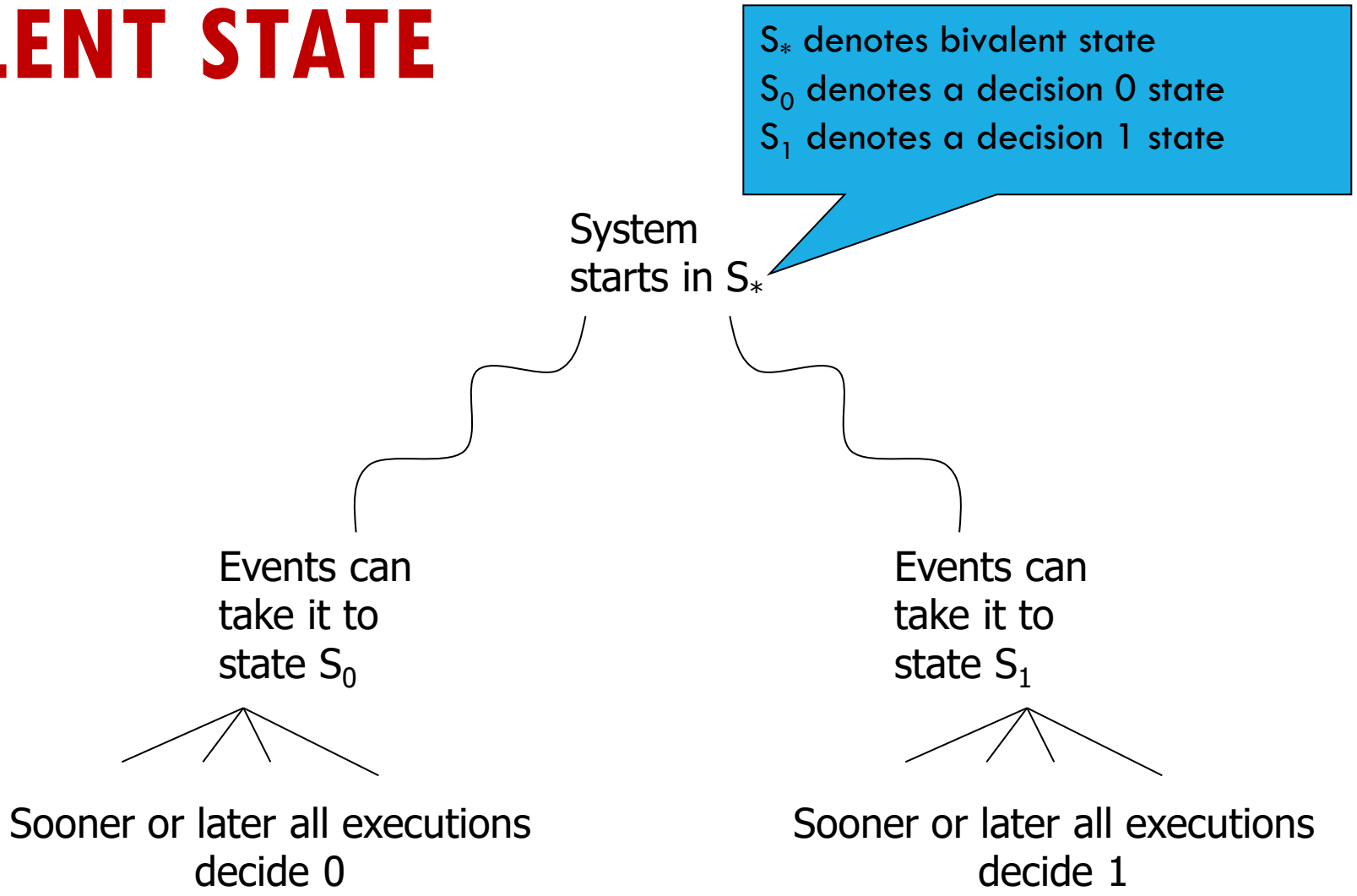
Think of N being odd, like 51. Imagine a vote of 26 0's and 25 1's.

Suppose the protocol picks the majority if possible. With no failures, it picks 0. But what if one of the 0 voters fails before voting?

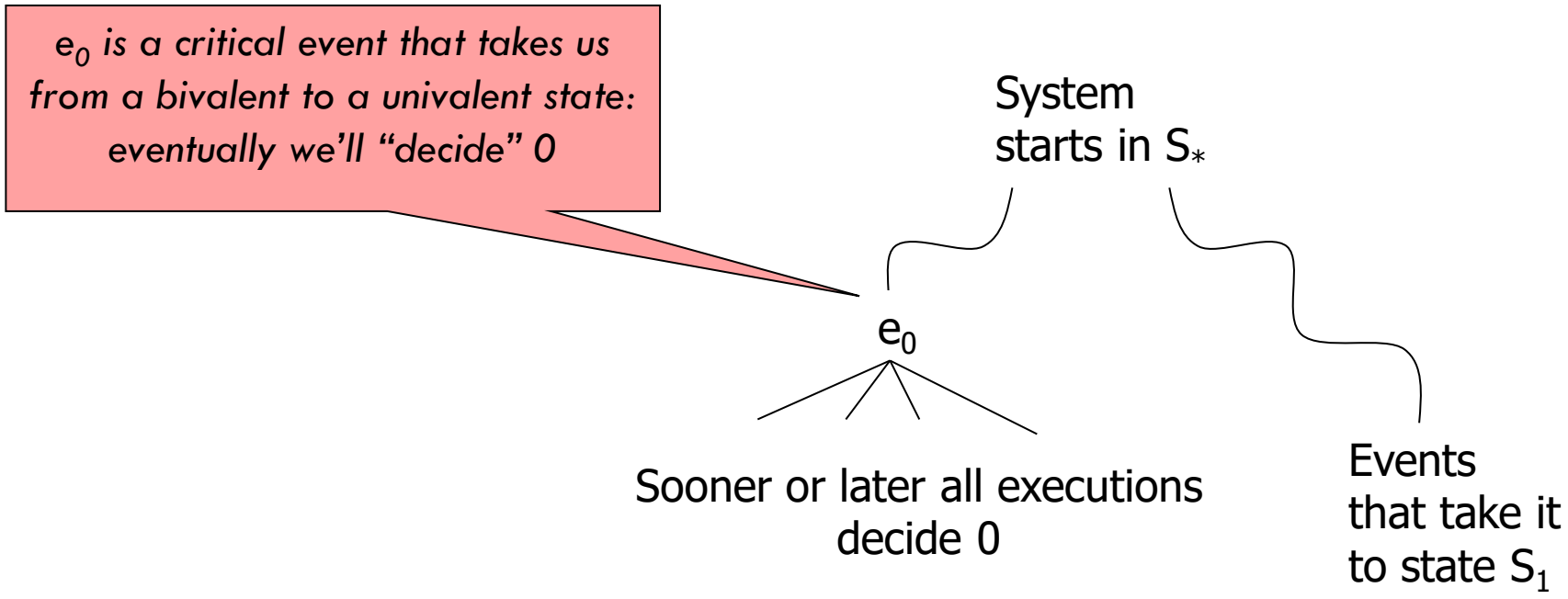
We get a tie! And in a tie, perhaps the tie-breaking rule is to pick 1.

So this is a bivalent initial state. If no failure occurs, it will decide 0. With a failure, it decides 1.

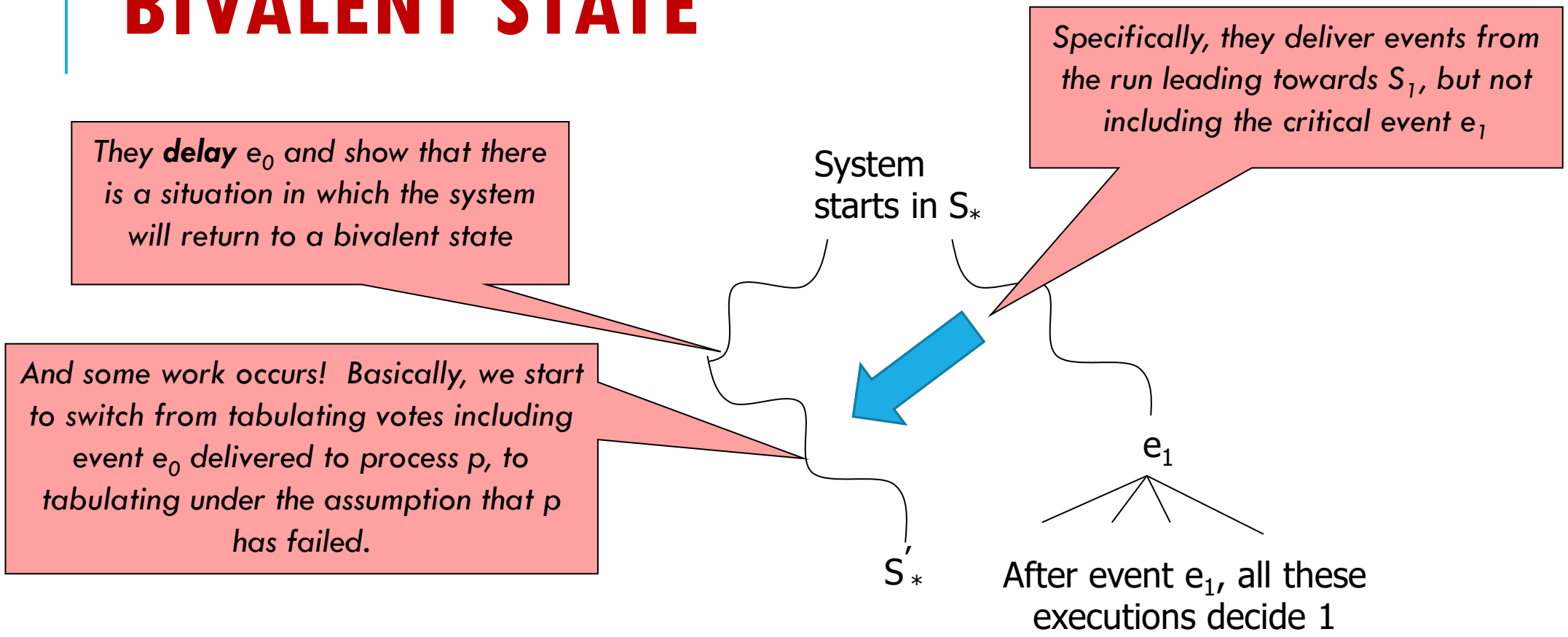
BIVALENT STATE



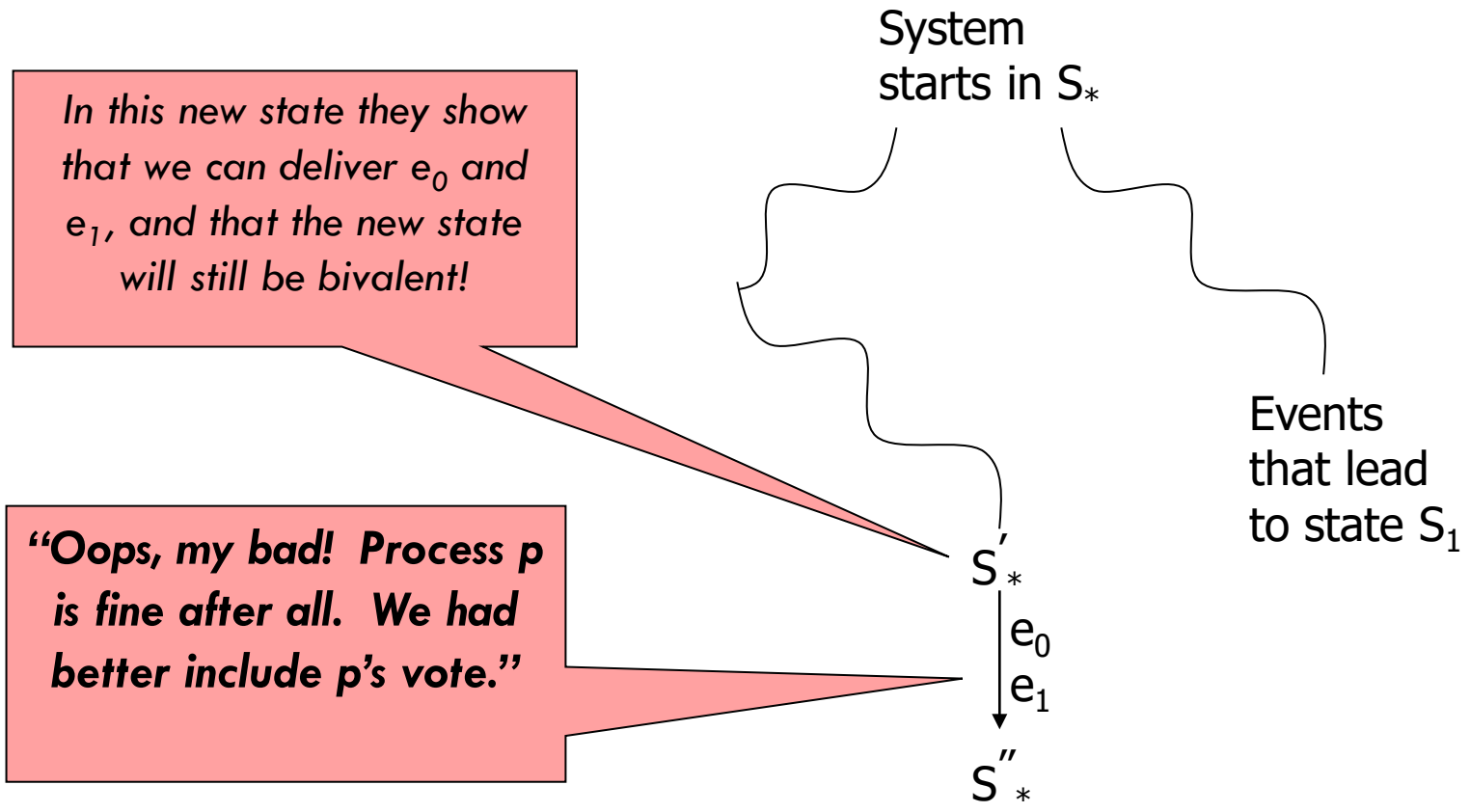
BIVALENT STATE



BIVALENT STATE



BIVALENT STATE

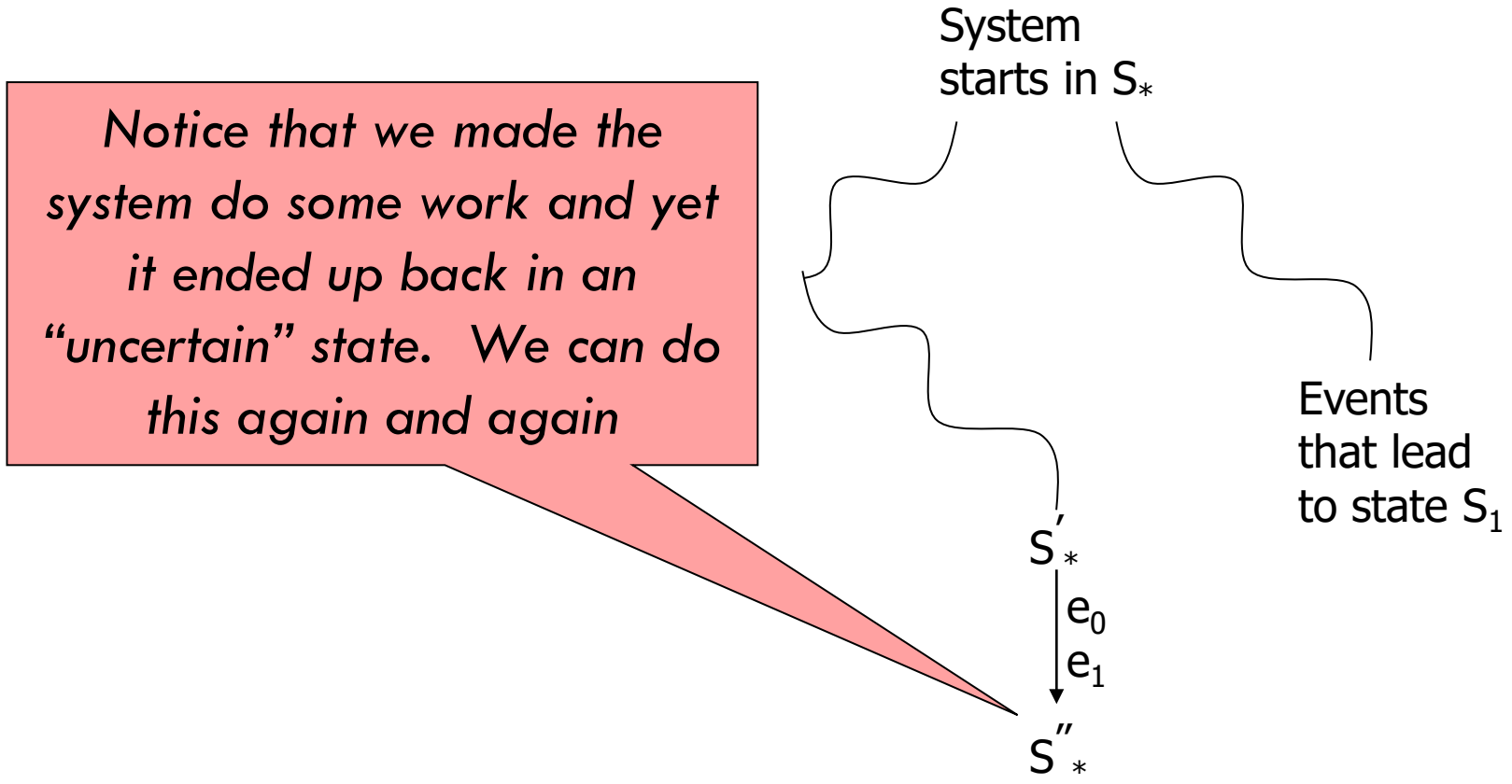


WHY INCLUDE P IF WE STARTED TO EXCLUDE IT?

In some settings, if anything goes wrong involving p, we just shut p off. James Hamilton said it best: "When suspicious, reboot, reimage, replace."

FLP only requires that protocols be able to tolerate one failure. It needs to reinclude p, in case some other process q might genuinely have failed!

BIVALENT STATE



CORE OF FLP RESULT IN WORDS

Thinking of an election, they focus on a near-tie situation.

By delaying one vote as if that voter didn't show up for the election, they push the system towards the tie-breaker. Nobody knows the outcome yet.

But before flipping the tie-breaking coin, the delayed vote turns up, which pushes the system back towards a full recount.

CORE OF FLP RESULT

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! It will be **safe** yet will not be **live**.

BUT WHAT DID "IMPOSSIBILITY" MEAN?

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

… and this was the title of their paper. But as you hear these words, 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 a real cloud setting, with a real system running something like Paxos or Derecho, how would the "network" ever know which message to delay, and when? In practice the FLP attack is infeasible!

SO…

…the FLP model gives too much power to the attacker. A system running Paxos or Derecho wouldn't ever experience an FLP attack.

Conversely, these systems can get stuck if they lose majority (partition). Yet FLP doesn't even consider this sort of networking problem.

In effect, real systems freeze up because of something FLP doesn't model, but never encounter the scenario FLP considers. Yet the conclusion is the same: real systems can't always guarantee progress (liveness), or at least to do that they need to make additional assumptions.

INTUITION FROM ELECTIONS?

Real elections are consensus algorithms, and they do need to tolerate crash failures (not every registered voter shows up to vote).

But they aren't asynchronous. A real election has a time window after which late votes aren't accepted.

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. But what if we can sense failures in a perfect way?

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! Then, like James said, we would force that process to restart.

HOW TO "WORK AROUND" FLP

You propose to solve consensus?

Inconceivable!

HOW DOES DERECHO DO IT?

It has a virtually synchronous self-managed membership service, sort of like Zookeeper. We learned about this back in lecture 7

Recall that we discussed the term virtual synchrony at the time: it centers on ordering of membership views (epochs), state transfers and multicast.

Originally used in Isis Distributed Toolkit in 1980's, but then explored in many papers and books.

HOW DOES DERECHO DO IT?

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.

At the core is a form of Paxos. It prevents split-brain behavior if leader failure is suspected. Zookeeper is very similar.

BUT CAN VIRTUAL SYNCHRONY AVOID THE FLP PROBLEM?

FLP is not directly applicable: in FLP, a healthy process *must be allowed to vote*. But Derecho might "kill" a non-failed process!

Derecho trusts its failure suspicion algorithm, even though it could be wrong

This leaves partitioning as a risk… and the "no split brain" logic could prevent progress. We traded FLP for a different problem!

NO SOLUTION CAN 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…)

REAL SYSTEMS DON'T AVOID FLP

The bottom line is that "fault tolerance is impossible" and yet "we solve it!"

It is almost as if FLP simply means "usually possible, but not always."

This is good enough because after all, the whole data center could have a leaky roof and shut down. Guaranteed progress isn't always meaningful.

ADDITIONAL WORK, BEYOND FLP

Over decades, the FLP result was shown to apply in many other settings

Problems like electing a leader, agreeing to switch from the primary server to a backup – all of these turn out to be "as strong as consensus" and any solution is subject to FLP

On the other hand, there has also been growing recognition that FLP doesn't matter very much!

THINK ABOUT THE TWO BIG THEORY RESULTS

For asynchronous networks, we have FLP. It says for a very broad class of problems, those problems can solve consensus, and are provably not live.

- Wouldn't we prefer to know **when we <u>can</u> definitely solve consensus**?
- We really want the weakest conditions for guaranteeing progress.

For Byzantine Agreement, we have the exact opposite situation!

- Byzantine Agreement systems assume an *unrealistic network model*.
- Then in this unimplementable network, the protocol solves the problem.
- Isn't this the situation where we would want impossibility results?

… FOLLOW-ON WORK HAS PURSUED THIS

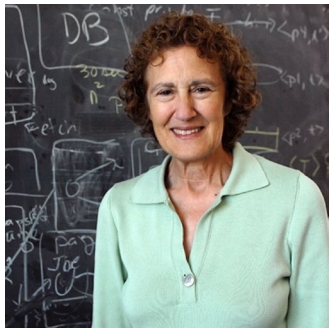
Today there is a theory for the best possible consensus solutions on realistic networks (asynchronous, could lose messages, etc).

The theory was created by Alex Shraer and Idit Keidar

The Derecho protocol is the first Paxos protocol to be optimal!
This also tells us that Cascade is optimal for consensus-like tasks

… FOLLOW-ON WORK HAS PURSUED THIS

Meanwhile, at MIT, Miguel Castro and Barbara Liskov explored practical algorithms for Byzantine Agreement in asynchronous networks



Their PRACTI solutions worked really well!

And over time, as blockchain rolled out, there has been growing excitement around other practical, Byzantine fault-tolerant solutions, too.

WHAT HAPPENED TO FLP???

Didn't we start by proving that fault tolerance is impossible?

… we did, and perfect solutions are not possible.

We should understand these results as telling us that under "suitable conditions" these kinds of solutions will be certain to work, but not under "all possible conditions"

EXAMPLE: CONDITIONS FOR DERECHO TO MAKE PROGRESS

must "vote" to accept the new view.

a failure: a slow process will be dropped.

Theory people would call this a ◊**S** requirement. Keidar proved that any practical consensus protocol needs ◊**S** , so this is another optimality result

Any practical system needs this, to avoid split-brain partitioning failures

- Derecho can only tolerate simultaneous failure of a minority of the view.
- The system must remain stable long enough to allow a leader to reconfigure into a new view, without any healthy process suspecting it

CONCLUSIONS

The FLP theorem is very broad and applies to any system that can solve consensus. Paxos, Zookeeper, Derecho… all are subject to FLP

But we can guarantee progress if we are willing to make stronger assumptions. Derecho, for example, assumes ◊**S** and also that a majority of the most recent members of the system remain available until we switch to the next epoch. Any practical system makes similar assumptions.

Derecho is optimal in this sense. No system can deliver messages sooner or make progress using weaker assumptions.