IMPOSSIBILITY OF CONSENSUS

Fall 2012

Ken Birman

Consensus… a classic problem

- Consensus abstraction underlies many distributed systems and protocols
  - N processes
  - They start execution with inputs $\in \{0,1\}$
  - Asynchronous, reliable network
  - At most 1 process fails by halting (crash)
  - Goal: protocol whereby all "decide" same value *v*, and *v* was an input

Distributed Consensus

Jenkins, if I want another yes-man, I'll build one!

Lee Lorenz, Brent Sheppard

Asynchronous networks

- No common clocks or shared notion of time (local ideas of time are fine, but different processes may have very different "clocks")
- No way to know how long a message will take to get from A to B
- Messages are never lost in the network

Quick comparison…

Asynchronous model	Real world
Reliable message passing, unbounded delays	Just resend until acknowledged; often have a delay model
No partitioning faults (“wait until over”)	May have to operate “during” partitioning
No clocks of any kinds	Clocks but limited sync
Crash failures, can’t detect reliably	Usually detect failures with timeout

Fault-tolerant protocol

- Collect votes from all N processes
  - At most one is faulty, so if one doesn't respond, count that vote as 0
- Compute majority
- Tell everyone the outcome
- They "decide" (they accept outcome)
- … *but this has a problem! Why?*

What makes consensus hard?

- Fundamentally, the issue revolves around membership
  - In an asynchronous environment, we can't detect failures reliably
  - A faulty process stops sending messages but a "slow" message might confuse us
- Yet when the vote is nearly a tie, this confusing situation really matters

Fischer, Lynch and Patterson

- A surprising result
  - *Impossibility of Asynchronous Distributed Consensus with a Single Faulty Process*
- They prove that no asynchronous algorithm for agreeing on a one-bit value can guarantee that it will terminate in the presence of crash faults
  - And this is true even if no crash actually occurs!
  - Proof constructs infinite non-terminating runs

Core of FLP result

- They start by looking at a system with inputs that are all the same
  - All 0's must decide 0, all 1's decides 1
- Now they explore mixtures of inputs and find some initial set of inputs with an uncertain ("bivalent") outcome
- They focus on this bivalent state

Self-Quiz questions

- When is a state “univalent” as opposed to “bivalent”?
- Can the system be in a univalent state if no process has actually decided?
- What “causes” a system to enter a univalent state?

Self-Quiz questions

- Suppose that event **e** moves us into a univalent state, and **e** happens at **p**.
  - Might **p** decide "immediately?
- Now sever communications from **p** to the rest of the system. Both event **e** and **p**'s decision are "hidden"
  - Does this matter in the FLP model?
  - Might it matter in real life?

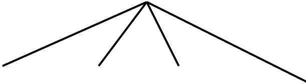
Bivalent state

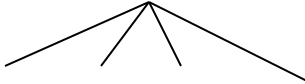
S_* denotes bivalent state
S_0 denotes a decision 0 state
S_1 denotes a decision 1 state

System starts in S_*

Events can take it to state S_0

Events can take it to state S_1

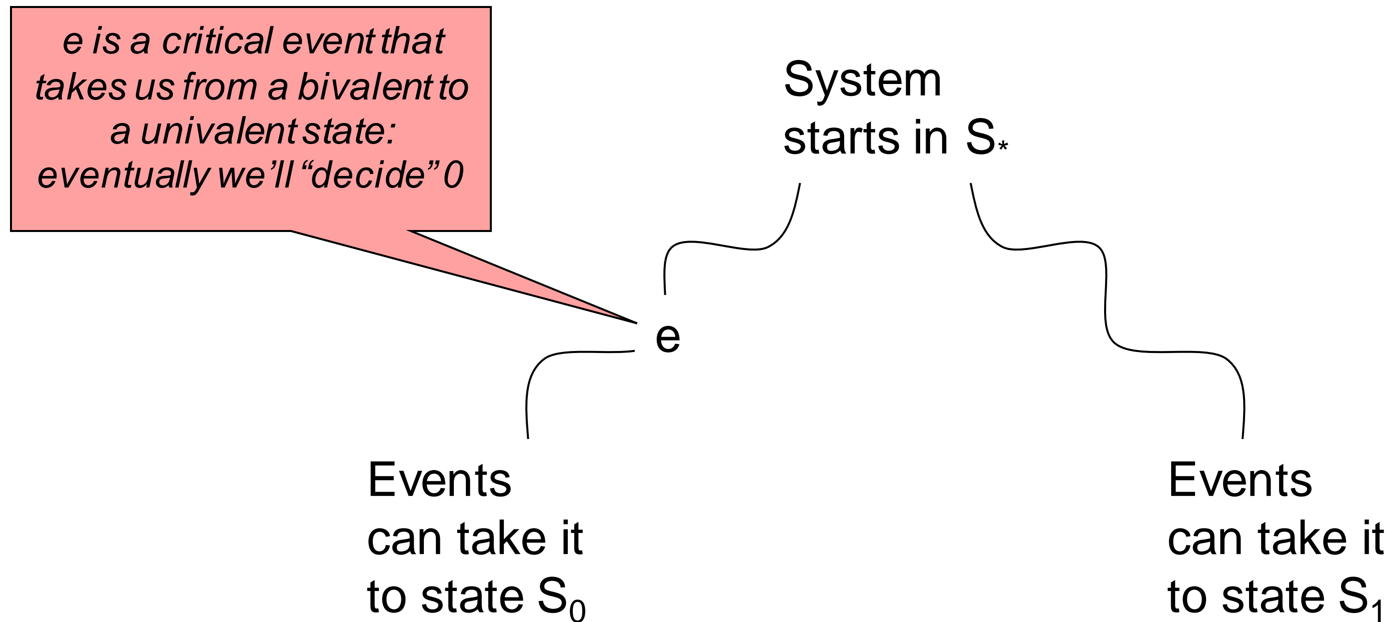




Sooner or later all executions decide 0

Sooner or later all executions decide 1

Bivalent state



Bivalent state

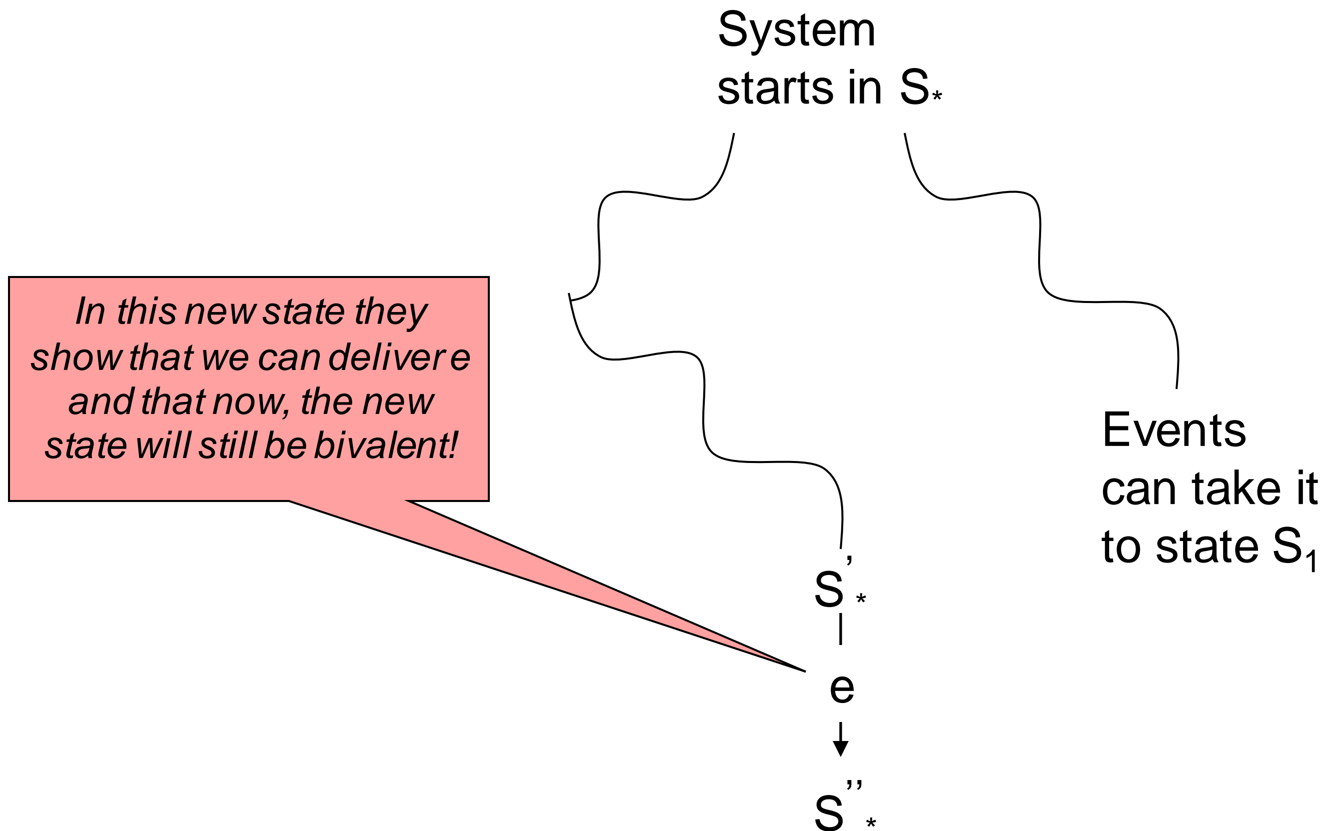
They delay e and show that there is a situation in which the system will return to a bivalent state

System starts in S_*

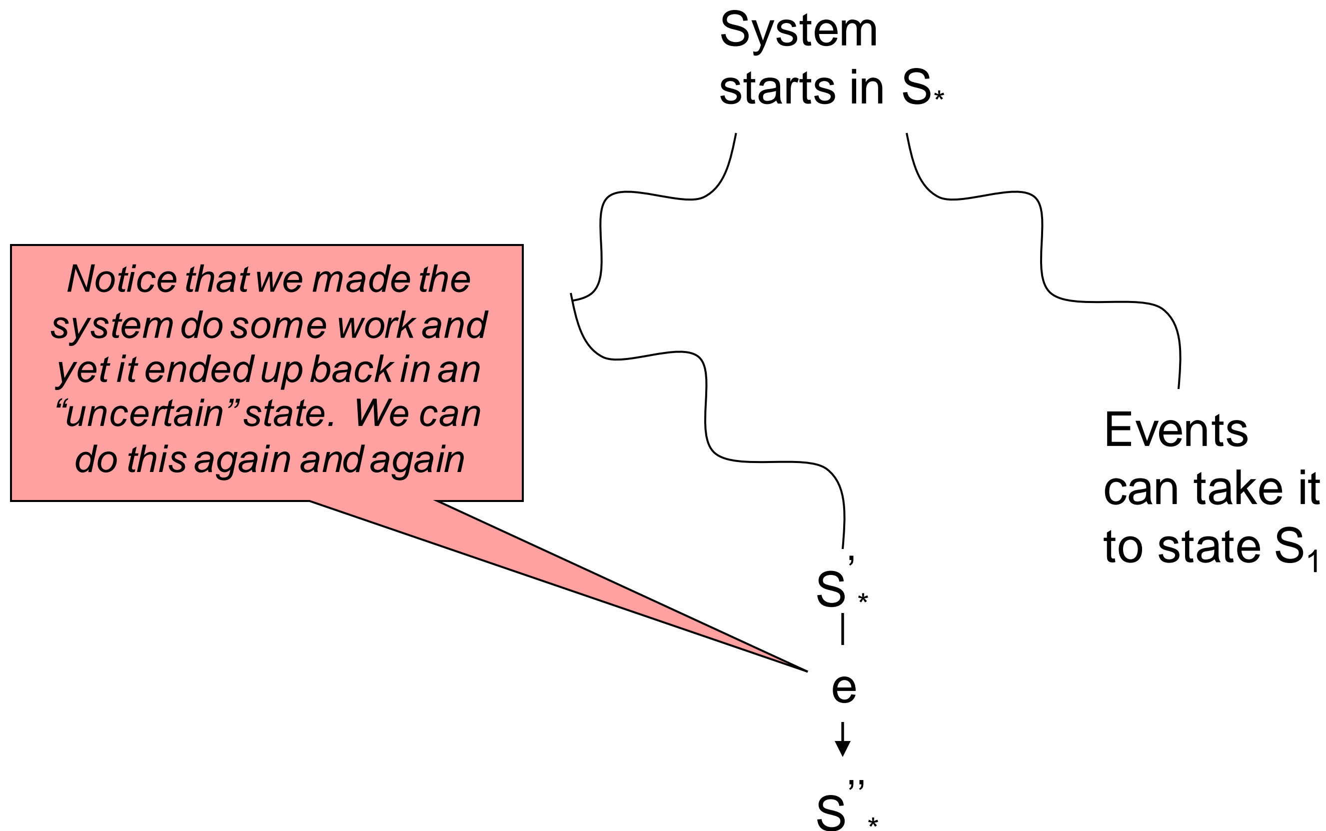
Events can take it to state S_1

S'_*

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

- Initially in a bivalent state
- Delivery of *m* would cause a decision, 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

Core of FLP result

- This proves the result: *a bivalent system can be forced to do some work and yet remain in a bivalent state.*
  - We can "pump" this to generate indefinite runs that never decide
  - Interesting insight: no failures actually occur (just delays). FLP attacks a fault-tolerant protocol using fault-free runs!

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 treating p as if it had failed, but then lets p resume execution and "rejoin"
- This takes time… and no real progress occurs

Constable's version of the FLP result

- He reworks the FLP proof, but using the NuPRL logic
  - A completely constructive ("intuitionist") logic
  - A proof takes the form of code that computes the property that was proved to hold

- In this constructive FLP proof, we actually see the system reconfigure to disseminate a kind of configuration: "Colin is faulty, don't count his vote"

Constable's version of the FLP result

- Now Colin resumes communication but Theo goes silent… we need to tolerate 1 failure (Theo) and are required to count Colin's vote
- Constable shows that FLP must reconfigure for this new state before it can decide
- These steps take time… and this proves the result!

But what did “impossibility” mean?

- So… consensus is impossible!
- 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

But what did “impossibility” mean?

- FLP proves that any fault-tolerant algorithm solving consensus has runs that never terminate
  - These runs are extremely unlikely (“probability zero”)
  - … but imply that we can’t find a totally correct solution
- “consensus is impossible” thus means “consensus is not always possible”

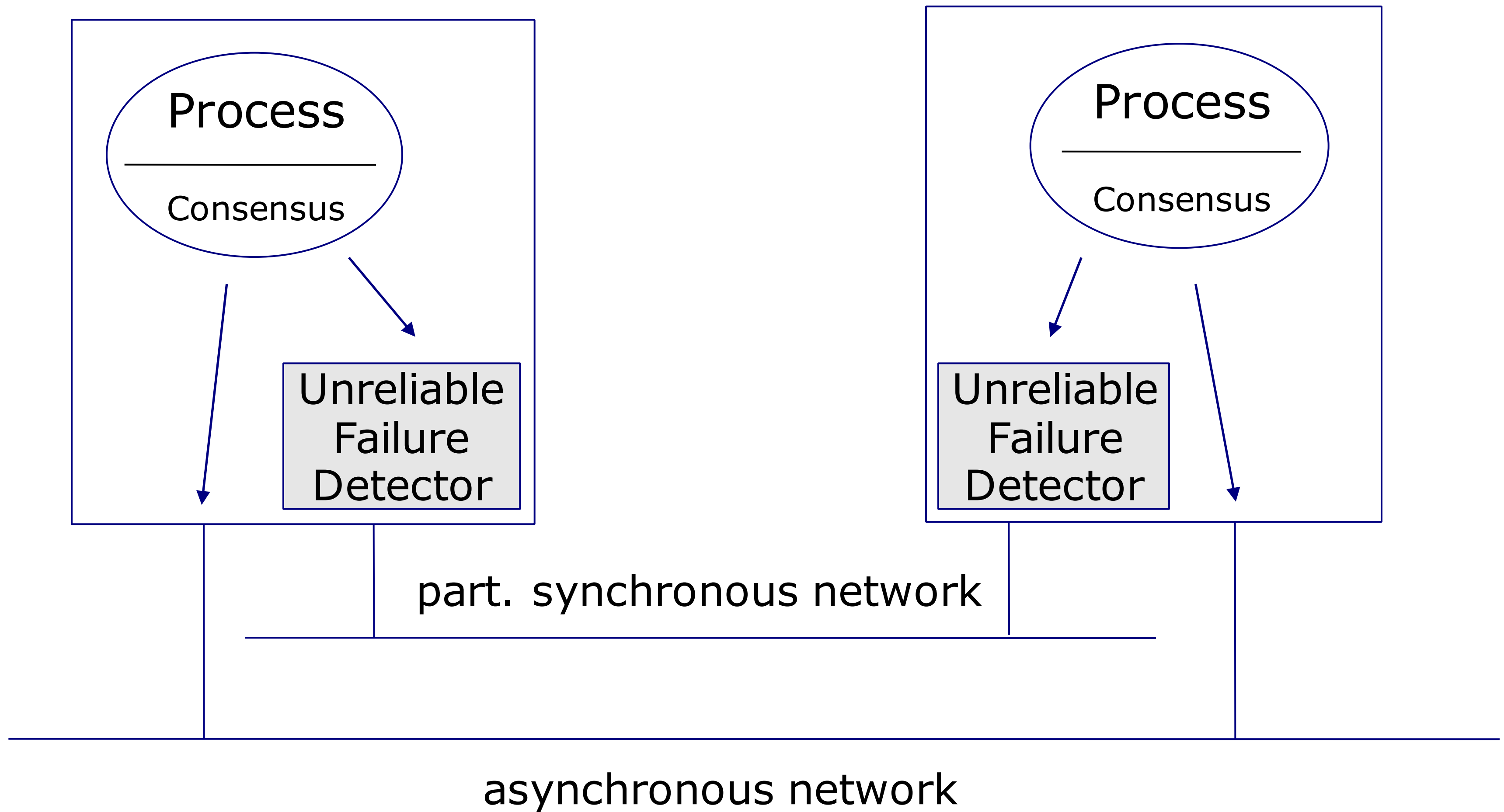
Solving consensus

- Systems that "solve" consensus often use a group membership service: a "GMS"
  - This GMS functions as an oracle, a trusted status reporting function
  - GMS service implements a protocol such as Paxos.
  - In the resulting virtual world, failure is a notification event reliably delivered by the GMS to the system members
- FLP still applies to the combined system

Chandra and Toueg

- This work formalizes the notion of a failure detection service
  - We have a failure detection component that reports on "suspected" failures. Implementation is a black box
  - Consensus protocol that consumes these events and seeks to achieve a consensus decision, fault-tolerantly
- Can we design a protocol that makes progress "whenever possible"?
- What is the weakest failure detector for which consensus is always achieved?

Motivation



Introduction and system model

- Unreliable Failure Detector: distributed oracle that provides (possibly incorrect) hints about the operational status of other processes

- Abstractly characterized in terms of two properties: completeness and accuracy
  - Completeness characterizes the degree to which failed processes are suspected by correct processes
  - Accuracy characterizes the degree to which correct processes are not suspected, i.e., restricts the false suspicions that a failure detector can make

Introduction and system model

Strong completeness: Eventually every process that crashes is permanently suspected by <u>every</u> correct process

Weak completeness: Eventually every process that crashes is permanently suspected by <u>some</u> correct process

Strong accuracy: Correct processes are never suspected

Weak accuracy: Some correct process is never suspected

Eventual strong accuracy: There is a time after which correct processes are not suspected by any correct process

Eventual weak accuracy: There is a time after which some correct process is not suspected by ane correct process

Completeness	**Accuracy**			
	Strong	Weak	Eventual strong	Eventual weak
Strong	*Perfect* *P*	*Strong* *S*	*Eventually Perfect* $\Diamond P$	*Eventually Strong* $\Diamond S$
Weak	*Quasi-Perfect* *Q*	*Weak* *W*	*Eventually Quasi-Perfect* $\Diamond Q$	*Eventually Weak* $\Diamond W$

Introduction and system model

- System model:
  - partially synchronous distributed system
  - finite set of processes $\Pi = \{p1, p2, ..., pn\}$
  - crash failure model (no recovery). A process is correct if it never crashes
  - communication only by message-passing (no shared memory)
  - reliable channel connecting every pair of processes (fully connected system)

Introduction and system

Completeness	Accuracy: Eventual strong	Accuracy: Eventual weak
Strong	*Eventually Perfect* $\Diamond P$	*Eventually Strong* $\Diamond S$
Weak	*Eventually Quasi-Perfect* $\Diamond Q$	*Eventually Weak* $\Diamond W$

- Chandra-Toueg's implementation of $\Diamond$P:
  - each process periodically sends an I-AM-ALIVE message to all the processes
  - upon timeout, suspect. If, later on, a message from a suspected process is received, then stop suspecting it and increase its timeout period
- Performance analysis (n processes, C correct):
  - Number of messages sent in a period: $n*(n-1)$
  - Size of messages: $\Theta(\log n)$ bits to represent id's
  - Information exchanged in a period: $\Theta(n^2 \log n)$ bits

Weaker detectors

- Core of result: Consensus can be solved with ◊W:
  - Form a ring of processes
  - Rotate role of being the leader (coordinator). Leader proposes a value, circulates token around the ring
  - If the token makes it around the ring twice, system becomes univalent. The leader is first to learn; others learn the outcome the next time they see a token
- Termination guaranteed if "eventually the leader is never suspected" but in fact the constraint on suspicions ends as soon as the decision is reached.

But can we implement ◊W?

- Not in an asynchronous network!
  - The network can always trigger false suspicions

- What about real networks?
  - In real networks we can talk about the probability of events, such as false suspicions, typical delays, etc
  - With this, if it is sufficiently unlikely that a false suspicion will occur, and sufficiently likely that messages are promptly delivered, ◊W is feasible w.h.p.

Real systems, like Paxos or Isis^2

- They use timeouts in various ways
- Paxos: Waits until it has a majority of responses
  - FLP attack: disrupts leader until a timeout causes a new one to take over
  - We end up with a mix of 2-phase and 3-phase rounds
- Isis^2: Runs a protocol called Gbcast in the GMS
  - Basically a strong leader selection and then a 2-phase commit, with a 3-phase commit if leader fails
  - FLP attack: causes repeated changes in leader role; old leader forced to rejoin

Summary

- Consensus is "impossible"
  - But this doesn't turn out to be a big obstacle
  - Can achieve consensus with probability 1.0 in practice
- Paxos and Isis^2 both support powerful consensus protocols that are very practical and useful
  - Neither really evades FLP… but FLP isn't a real issue
  - These systems are more worried about overcoming short-term failures. FLP is about eternity…