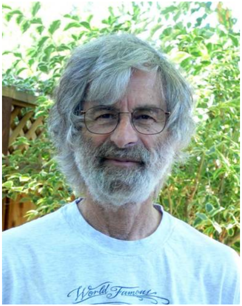
Consensus in Distributed Systems

Impossibility of Distributed Consensus with One Faulty Process

and

Paxos Made Simple

Paxos Made Simple



Leslie Lamport

The Plain English Definition of Consensus

- A group of agents want to agree on a common value
- They are cautious because they might have different opinions
- Consequently, they don't write down a firm value until agreement
- Once written down, value can't be changed
- We want all agents to eventually write down the same value

Consensus is Everywhere

Lots of examples of consensus in distributed systems:

- Primary replica for data object in Bayou
- Distributed primary in Pond
- Chain replication
- Any time you determine a consistent commit ordering
- CAP Theorem: Consistency requires consensus

Formal Definition of Consensus

- Group of agents: proposers, acceptors, learners
- Agents have input/output registers for value
- Formal specification for protocol and communication
- Safety: no contradiction
- Termination: doesn't run forever

Simple Consensus Protocols

Examples

- Single leader
- Majority

Node failure can break protocol

Need a failure resilient consensus algorithm!

Possible Failures in Paxos

Asynchronous execution:

- Agents operate at arbitrary speed
- May fail by stopping and may restart later
- Messages can be delayed arbitrarily and delivered out of order
- No arbitrary failures

Some other requirements

Want Paxos to be as general as possible, but

- Mostly no hardware clocks, but may need these to fix some issues later.
- Need some permanent store across fail/restart.

Safety in Paxos

The criteria for safety in the Paxos algorithm are:

- Only a value that has been proposed may be chosen
- Only a single value is chosen
- A process never learns that a value has been chosen unless it actually has been

Note the distinction between **proposed** and **chosen**.

Requirements

If only a single proposal gets through due to failures, still want consensus

P1: An acceptor must accept the first proposal it receives

Requirements

If only a single proposal gets through due to failures, still want consensus

P1: An acceptor must accept the first proposal it receives

Note the distinction between **proposed**, **accepted**, and **chosen**.

Rely on overlapping majority to make consistent choice.

Numbering Proposals

- To distinguish proposals, give every proposal a number.
- Numbers have a total order
- No two proposals share a number

Requirements

How can we make sure only one value is chosen?

Requirements

P2: If a proposal with value v is chosen, every higher numbered proposal **that is chosen** has value v

Requirements

P2: If a proposal with value v is chosen, every higher numbered proposal **that is chosen** has value v

P2a: If a proposal with value v is chosen, every higher numbered proposal **accepted by any acceptor** has value v

Requirements

P2: If a proposal with value v is chosen, every higher numbered proposal **that is chosen** has value v

P2a: If a proposal with value v is chosen, every higher numbered proposal **accepted by any acceptor** has value v

P2b: If a proposal with value v is chosen, every higher numbered proposal **issued by any proposer** has value v.

Requirements

P2: If a proposal with value v is chosen, every higher numbered proposal **that is chosen** has value v

P2a: If a proposal with value v is chosen, every higher numbered proposal **accepted by any acceptor** has value v

$$\Uparrow$$

P2b: If a proposal with value v is chosen, every higher numbered proposal **issued by any proposer** has value v.

Requirements

P2: If a proposal with value v is chosen, every higher numbered proposal **that is chosen** has value v

$\Uparrow$

P2a: If a proposal with value v is chosen, every higher numbered proposal **accepted by any acceptor** has value v

$\Uparrow$

P2b: If a proposal with value v is chosen, every higher numbered proposal **issued by any proposer** has value v.

Requirements

Safety: Only one value is chosen

$\Uparrow$

P2: If a proposal with value *v* is chosen, every higher numbered proposal **that is chosen** has value *v*

$\Uparrow$

P2a: If a proposal with value *v* is chosen, every higher numbered proposal **accepted by any acceptor** has value *v*

$\Uparrow$

P2b: If a proposal with value *v* is chosen, every higher numbered proposal **issued by any proposer** has value *v*.

Towards an algorithm

How could we make P2b hold?

- Start with a proposal with value v and number m that gets chosen.

Towards an algorithm

How could we make P2b hold?

- Start with a proposal with value v and number m that gets chosen.
- Let's think about the next highest proposal that comes in.

Towards an algorithm

How could we make P2b hold?

- Start with a proposal with value v and number m that gets chosen.
- Let's think about the next highest proposal that comes in.
- It needs to have value v.

Towards an algorithm

How could we make P2b hold?

- Start with a proposal with value v and number m that gets chosen.
- Let's think about the next highest proposal that comes in.
- It needs to have value v.
- Since m chosen, there's a majority of acceptors accepting it.

Towards an algorithm

How could we make P2b hold?

- Start with a proposal with value v and number m that gets chosen.
- Let's think about the next highest proposal that comes in.
- It needs to have value v.
- Since m chosen, there's a majority of acceptors accepting it.
- If the proposer hears from a majority of acceptors, it will hear about v.

Towards an algorithm

How could we make P2b hold?

- Start with a proposal with value v and number m that gets chosen.
- Let's think about the next highest proposal that comes in.
- It needs to have value v.
- Since m chosen, there's a majority of acceptors accepting it.
- If the proposer hears from a majority of acceptors, it will hear about v.
- If it hears about v, it needs to propose it to maintain P2b.

More Properties

P2c: To make a proposal numbered n with value v, a proposer must know of a majority S of acceptors such that either

- No acceptor in S has accepted anything numbered less than n
- v is the value of the highest numbered proposal less than n accepted by somebody in S

More Properties

P2c: To make a proposal numbered n with value v, a proposer must know of a majority S of acceptors such that either

- No acceptor in S **will ever accept** anything numbered less than n
- v is the value of the highest numbered proposal less than n **ever accepted** by somebody in S

Promises

Instead of trying to predict the future, use promises.

In learning about acceptors, extract a promise that the information it learns will **always** be true.

A request for information about requests numbered less than n is also a contract to never accept any proposals numbered less than n.

Behavior of Proposer

Now, protocol for Proposers/Acceptors is basically fixed:

- Proposer picks a proposal number n
- Sends *prepare request* asking for information about proposals less than n
- This is also a promise not to accept new proposals less than n

Behavior of Proposer

- If it hears back from a majority, it knows enough information to apply P2c
- If majority hasn't ever accepted proposals, can pick any value
- If someone accepted with some value v, has to re-propose v
- The message with a proposal is an *accept request*

Behavior of Acceptor

An acceptor can do anything it hasn't promised not to do

- It can ignore any request without compromising safety
- Can always respond to prepare request
- Can respond to accept request if it hasn't promised otherwise

Note on Failures

P2c needs to be maintained across failure/restart

Solution: Acceptor remembers highest numbered proposal it has ever accepted, and highest numbered promise it has made, on permanent storage.

Learning the chosen value

- Since value chosen is consistent, learning is easy.
- Somehow broadcast acceptances to all learners.
  - Acceptors could all inform distinguished learner.
  - Better: acceptors inform small set of learners.

Progress

Suppose we have two proposers, p_1, p_2

- p_1 sends prepare request numbered n_1

Progress

Suppose we have two proposers, p_1, p_2

- p_1 sends prepare request numbered n_1
- p_2 sends prepare request numbered $n_2 > n_1$

Progress

Suppose we have two proposers, p_1, p_2

- p_1 sends prepare request numbered n_1
- p_2 sends prepare request numbered $n_2 > n_1$
- p_1 sends proposal numbered n_1, rejected

Progress

Suppose we have two proposers, p_1, p_2

- p_1 sends prepare request numbered n_1
- p_2 sends prepare request numbered $n_2 > n_1$
- p_1 sends proposal numbered n_1, rejected
  It starts again with prepare request numbered $n_3 > n_2$

Progress

Suppose we have two proposers, p_1, p_2

- p_1 sends prepare request numbered n_1
- p_2 sends prepare request numbered $n_2 > n_1$
- p_1 sends proposal numbered n_1, rejected
  It starts again with prepare request numbered $n_3 > n_2$
- p_2 sends proposal numbered n_2, rejected

Progress

Suppose we have two proposers, p_1, p_2

- p_1 sends prepare request numbered n_1
- p_2 sends prepare request numbered $n_2 > n_1$
- p_1 sends proposal numbered n_1, rejected
  It starts again with prepare request numbered $n_3 > n_2$
- p_2 sends proposal numbered n_2, rejected
  It starts again with prepare request numbered $n_4 > n_3$

Progress

Suppose we have two proposers, p_1, p_2

- p_1 sends prepare request numbered n_1
- p_2 sends prepare request numbered $n_2 > n_1$
- p_1 sends proposal numbered n_1, rejected
  It starts again with prepare request numbered $n_3 > n_2$
- p_2 sends proposal numbered n_2, rejected
  It starts again with prepare request numbered $n_4 > n_3$
- p_1 sends proposal numbered n_1, rejected
  It starts again with prepare request numbered $n_5 > n_4$
- p_2 sends proposal numbered n_4, rejected
  It starts again with prepare request numbered $n_6 > n_5$
- ...

Distinguished Proposer

- Solve the competing proposer problem by having a *distinguished proposer*
- Single proposer always makes progress if enough components working.
- Single point of failure, so need a way of electing a new proposer

Wait...

Isn’t that consensus again?

Wishy Washy Solutions

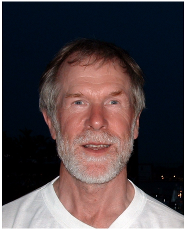
Lamport punts, says to use timeouts, or failure detectors.

There's actually a good reason for this

Impossibility of Distributed Consensus with One Faulty Process

Michael Fisher

Nancy Lynch



Michael Paterson

Motivation

"Window of Vulnerability"

- Delay at the wrong time stalls system
- Distinguishing between failed process and temporarily slow process is difficult

The Model: the Good

Want as general a model as possible

- Processes are state machines, possibly infinite states
- Deterministic
- Fail by stopping
- Can send arbitrary messages to other machines
- Messages uncorrupted

The Model: the Bad

But system is asynchronous

- Could take arbitrarily long between actions for process
- Could take arbitrarily long to deliver message
- Only guarantee delivery if process tries to receive infinitely many times
- Processes have no physical clocks

Bunch of Definitions

- Configuration: Process states and undelivered messages

Bunch of Definitions

- Configuration: Process states and undelivered messages
- Event: Process-message pair. Message can be $\emptyset$

Bunch of Definitions

- Configuration: Process states and undelivered messages
- Event: Process-message pair. Message can be $\emptyset$
- Run: Sequence of applicable events, possibly infinite

Bunch of Definitions

- Configuration: Process states and undelivered messages
- Event: Process-message pair. Message can be $\emptyset$
- Run: Sequence of applicable events, possibly infinite
- Deciding run: Some process writes to output register

Bunch of Definitions

- Configuration: Process states and undelivered messages
- Event: Process-message pair. Message can be $\emptyset$
- Run: Sequence of applicable events, possibly infinite
- Deciding run: Some process writes to output register
- Partial Correctness: Only one decision value, nontrivial

Bunch of Definitions

- Configuration: Process states and undelivered messages
- Event: Process-message pair. Message can be $\emptyset$
- Run: Sequence of applicable events, possibly infinite
- Deciding run: Some process writes to output register
- Partial Correctness: Only one decision value, nontrivial
- Faulty process: In a run, only takes finitely many steps

Bunch of Definitions

- Configuration: Process states and undelivered messages
- Event: Process-message pair. Message can be $\emptyset$
- Run: Sequence of applicable events, possibly infinite
- Deciding run: Some process writes to output register
- Partial Correctness: Only one decision value, nontrivial
- Faulty process: In a run, only takes finitely many steps
- Admissible run: One fault, messages eventually delivered

Bunch of Definitions

- Configuration: Process states and undelivered messages
- Event: Process-message pair. Message can be $\emptyset$
- Run: Sequence of applicable events, possibly infinite
- Deciding run: Some process writes to output register
- Partial Correctness: Only one decision value, nontrivial
- Faulty process: In a run, only takes finitely many steps
- Admissible run: One fault, messages eventually delivered
- *Totally correct in spite of one fault*: Protocol is partially correct, and every admissible run is deciding

Result

Theorem (Impossibility)

No consensus protocol is totally correct in spite of one fault

Bivalent configurations

- *Bivalent configuration*: Different runs cause protocol to decide 0 and 1
- These are 'indecisive' configurations
- Window of Vulnerability: bad run can be continually indecisive

Lemma 1

There is a bivalent initial configuration

Lemma 2

If we're at a bivalent configuration C, e is an event we can apply, and $\mathscr{D}$ are the configurations reachable from C doing e last...

... then $\mathscr{D}$ has a bivalent configuration.

Meaning of Lemma 2

If we're sitting at C, and e is about to take us to a non-bivalent configuration:

- e 'decides' the protocol (though output may be delayed)
- Message or process for e gets delayed
- Instead, we do some other events, eventually doing e
- Now, we just did e, so we're in $\mathscr{D}$
- Pick 'other stuff' appropriately, so that we're still undecided

That's the window of vulnerability.

Proof of Theorem

Know that we have to remain undecided. Want an admissible run.

- Construct the run by stages

Proof of Theorem

Know that we have to remain undecided. Want an admissible run.

- Construct the run by stages
- Put the processes in a rotating queue

Proof of Theorem

Know that we have to remain undecided. Want an admissible run.

- Construct the run by stages
- Put the processes in a rotating queue
- Grab first process. Look at earliest message (possibly none)

Proof of Theorem

Know that we have to remain undecided. Want an admissible run.

- Construct the run by stages
- Put the processes in a rotating queue
- Grab first process. Look at earliest message (possibly none)
- Hit the event e with Lemma 2

Proof of Theorem

Know that we have to remain undecided. Want an admissible run.

- Construct the run by stages
- Put the processes in a rotating queue
- Grab first process. Look at earliest message (possibly none)
- Hit the event e with Lemma 2
- End up having delivered message, but still indecisive

Proof of Theorem

Know that we have to remain undecided. Want an admissible run.

- Construct the run by stages
- Put the processes in a rotating queue
- Grab first process. Look at earliest message (possibly none)
- Hit the event e with Lemma 2
- End up having delivered message, but still indecisive
- Put p at back of queue, repeat

No faulty processes, all messages delivered, but no decision

Notes on Theorem

Just constructed indecisive run with no faults.

So, it isn't the faults themselves, but the protocol being correct in spite of faults, that causes indecision.

Thus: difficulty is in distinguishing fault from temporary delay.

More Notes on Theorem

Also, we only showed the existence of *one* bad run.

May be exceedingly unlikely.

Only applies to truly asynchronous systems.

Indecision could be resolved with physical clocks or failure detectors

What Have We Learned?

- Consensus is everywhere
- Paxos safe against failures, maybe even terminates (does it?)
- Everything has a window of vulnerability
- If we strengthen our model, may not apply

The Weakest Failure Detector for Solving Consensus

Tushar Chandra, Vassos Hadzilacos and Sam Toueg

The Failure Detector

Simplest failure detector necessary:

- There is a time after which every failed process is suspected by some correct process
- There is a time after which some correct process is never suspected by any correct process

Can use, for instance, timeouts to give you this