Distributed Computing

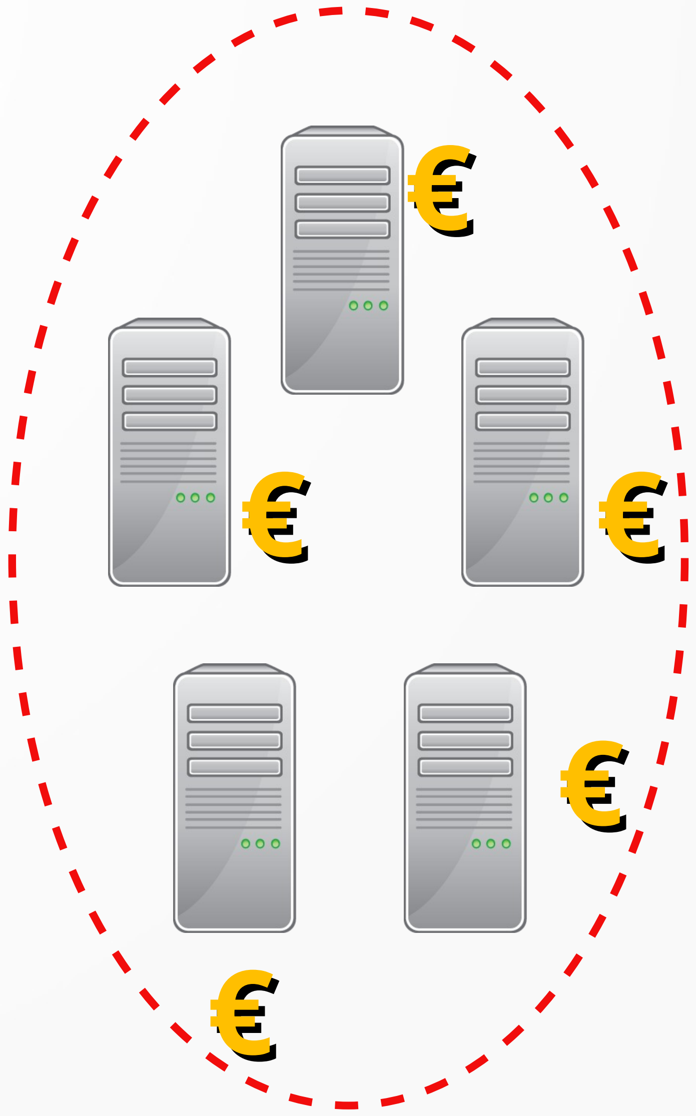
A-03. Consensus

Scaling Vertically

- Your (videogame, finance service, etc.) start-up needs a server
  - You run one (or use the cloud). All is fine.
- Congratulations, you're successful! You need a bigger server!
  - To do twice as many operations, you need to spend more than twice, but it's ok
- At some point the cost is not manageable, or a single-server solution still doesn't work
- Also, now that you're a huge business, you can't afford to lose money because there's a black-out or a flood

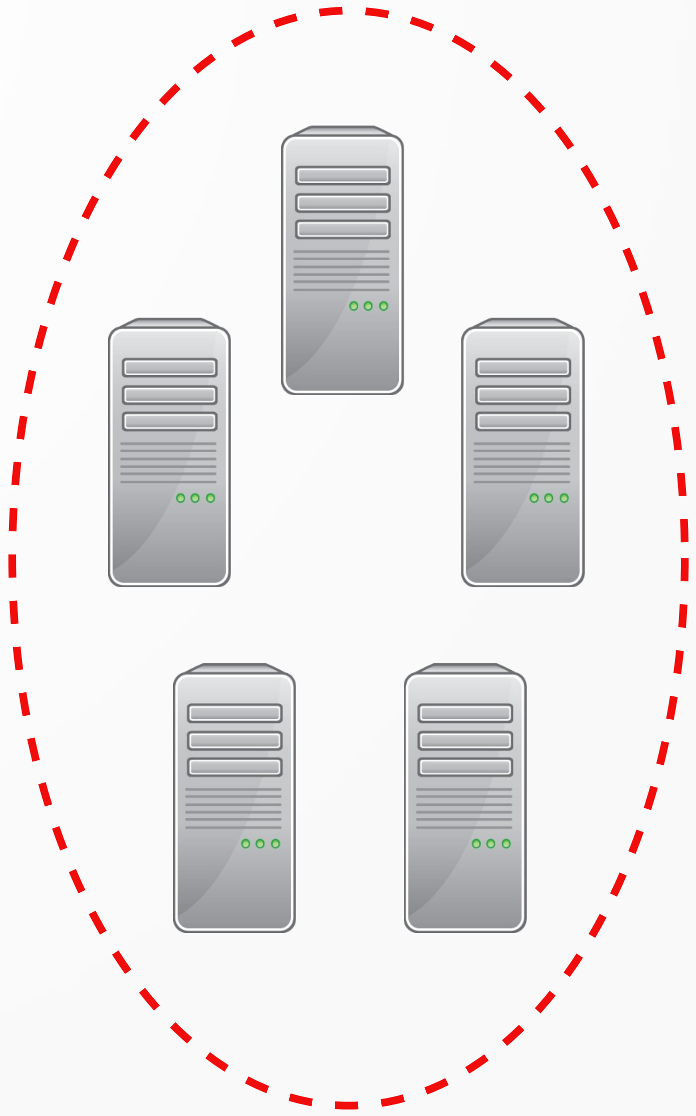
Scaling Horizontally

- A desireable alternative would be to **divide work between multiple servers**
  - We can then buy the servers with the best performance/price ratio
  - We can distribute them geographically for better redundancy (e.g., catastrophes)
- Two problems:
  - Coordination: making them work together **as if they were a single computer**
  - Scalability: make it so that **costs of coordination** are not huge



Remember ACID?

- Transaction properties
  - Atomicity
  - Consistency
  - Isolation
  - Durability
- Taken together, our system behaves “as if it is a single always-on machine”, processing all transactions one after the other
- How to pull this off?



State Machine Replication

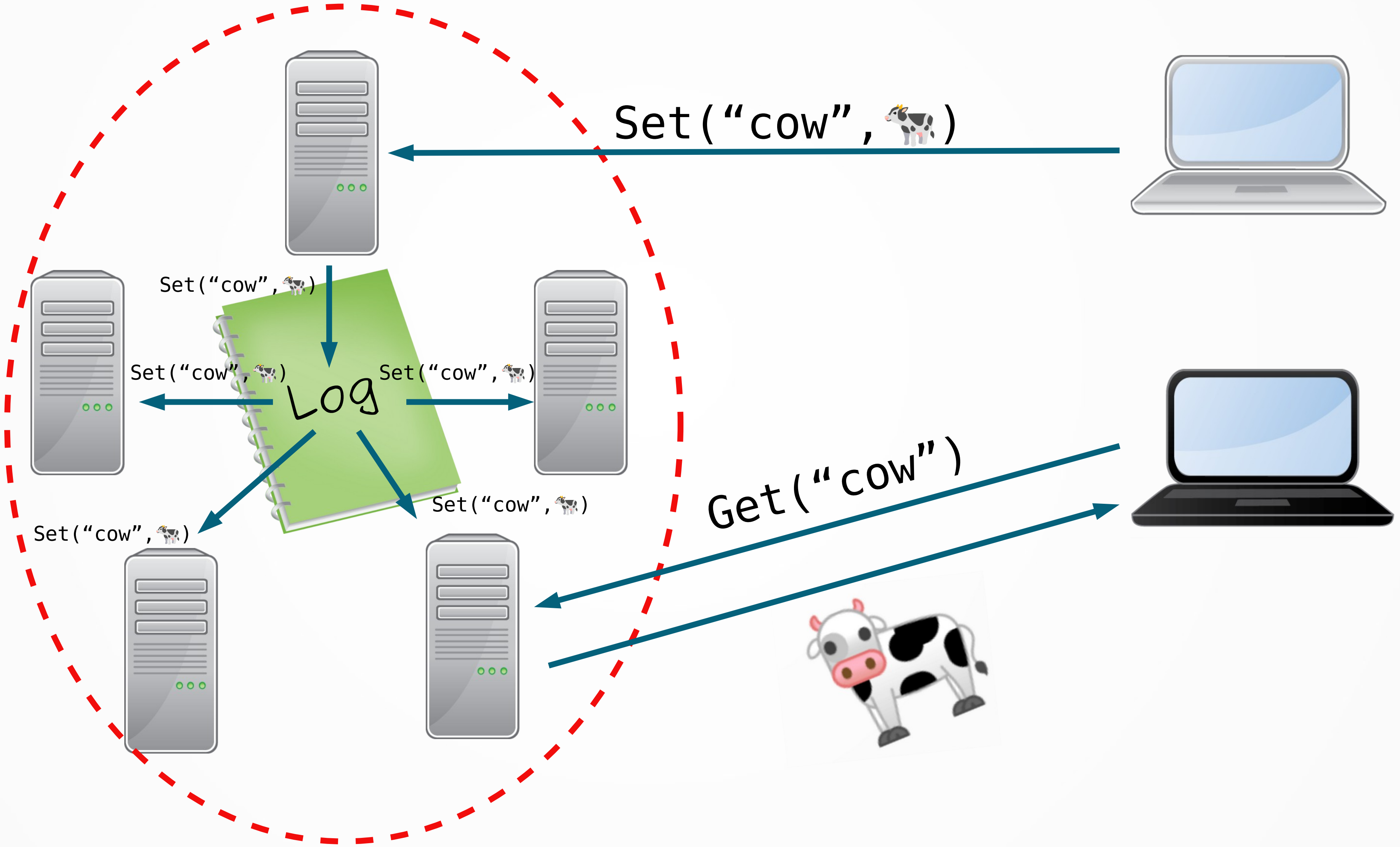
Deterministic State Machines

- Each machine has a **state** that determines its future behavior
  - Represents e.g. memory, registers, configuration
- Each machine has the same **starting state**
- Receving an **input** changes the state (and may produce output) **deterministically**
  - I.e., there is a transition function (Input × State → State × Output)
- Hence, **two machines** are in the same state **if they have received the same input in the same order**

Read-Intensive Workloads

- Input as a **list (log) of commands**
- In many architectures, **most commands don’t change the state**—e.g.:
  - Read a file from a distributed filesystem
  - Perform a database query
  - Get a webpage
- Our distributed system can be based on servers that
  - Work like deterministic **state machines**
  - Put “write” commands that change the state in a **shared log**, and execute them all on all machines
  - Run “read” commands **locally**
- If most of the workload is made of “read”, our system will scale well

An Example Key-Value Store



How To Implement a Shared Log?

- The **fail-stop** scenario:
  - One-to-one messages only
  - They can take arbitrary time
  - They can **be lost**
  - Computers can **stop for an arbitrary time**
  - But they **don’t lose data**
- Kind of like having to communicate only via SMS when you have bad coverage

Note: Other Consensus Problems

- There is a worse failure model than fail-stop
- Broken nodes may send **any** message, maybe even acting **maliciously**
- This is the kind of scenario you need in **cryptocurrencies**
- This is called **byzantine** consensus
  - Solved by classical algorithms and blockchains

Paxos

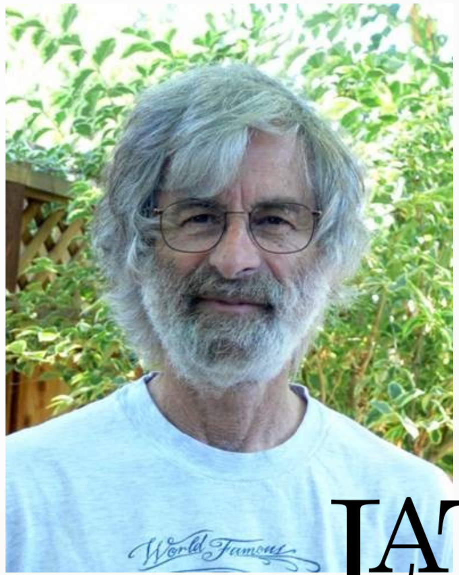
(or *How to confuse people for a quarter of a century*)

See Also

- Lesson by Chris Colohan
- Talk by Luis Quesada Torres

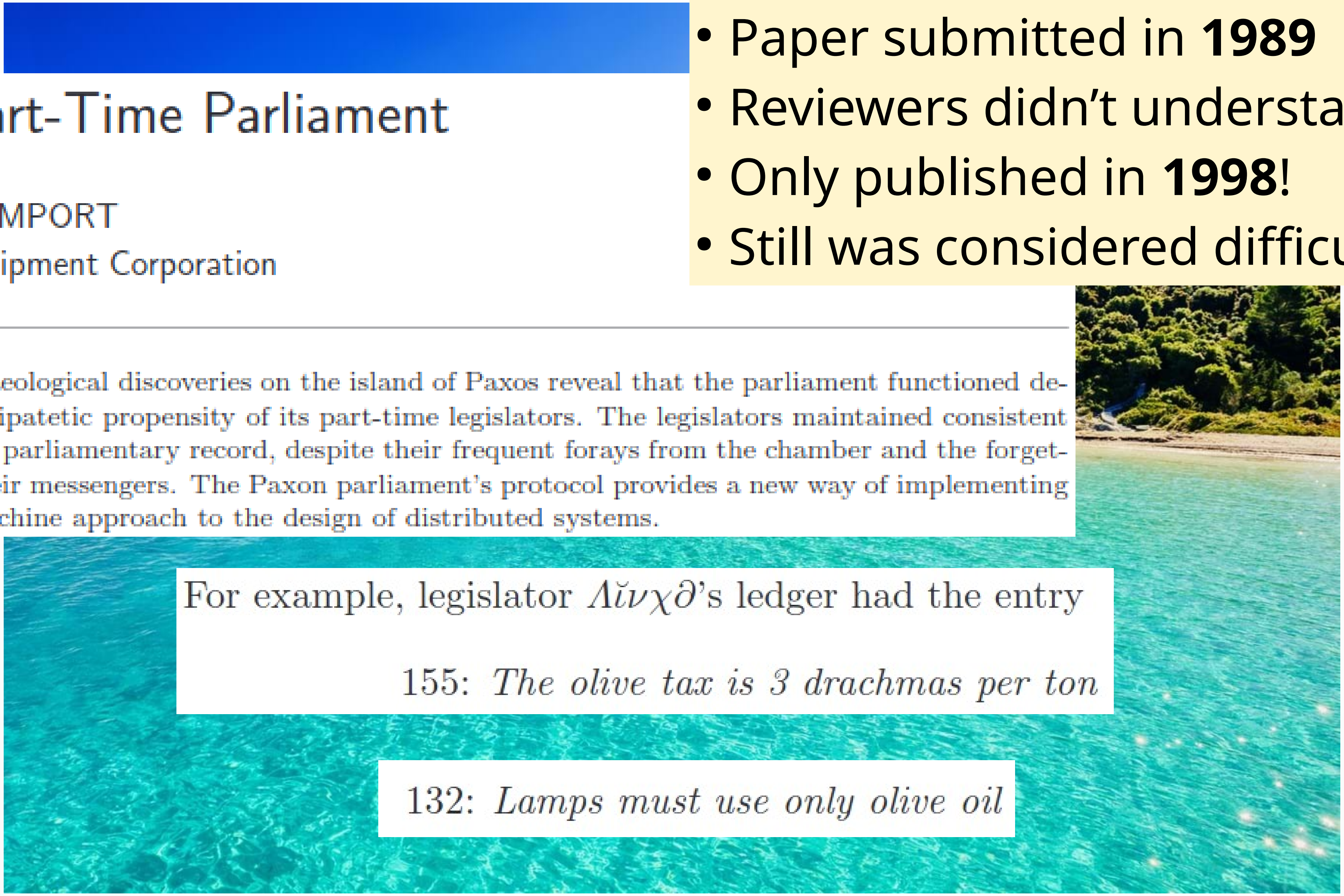
The Context

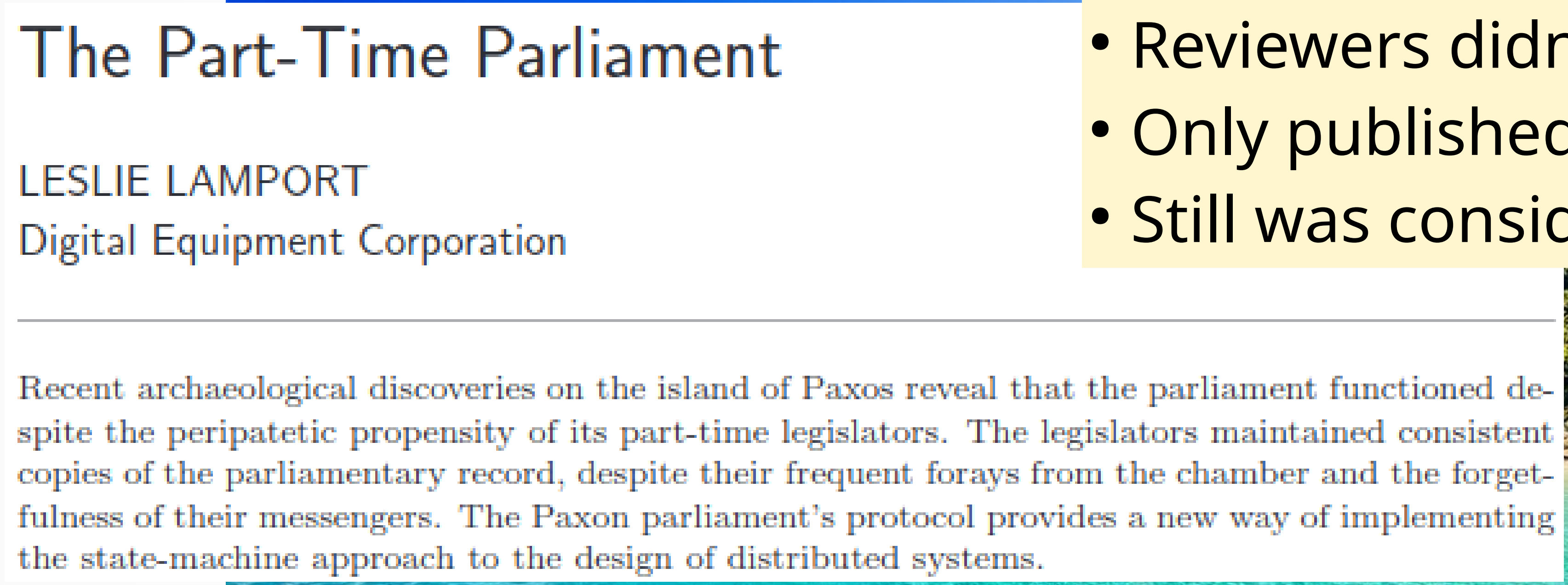
- In the ‘80s, Nancy Lynch and Barbara Liskov built a reliable redundant distributed system
  - No proof their system was correct for every single corner case
- Leslie Lamport wanted to show that they made mistakes, and their goal was impossible
  - He failed in that, because he found an algorithm that **could** do it: Paxos



The Confusing Story of Paxos

- Paper submitted in **1989**
- Reviewers didn’t understand it
- Only published in **1998**!
- Still was considered difficult





The Part-Time Parliament

LESLIE LAMPORT
Digital Equipment Corporation

Recent archaeological discoveries on the island of Paxos reveal that the parliament functioned despite the peripatetic propensity of its part-time legislators. The legislators maintained consistent copies of the parliamentary record, despite their frequent forays from the chamber and the forgetfulness of their messengers. The Paxon parliament’s protocol provides a new way of implementing the state-machine approach to the design of distributed systems.

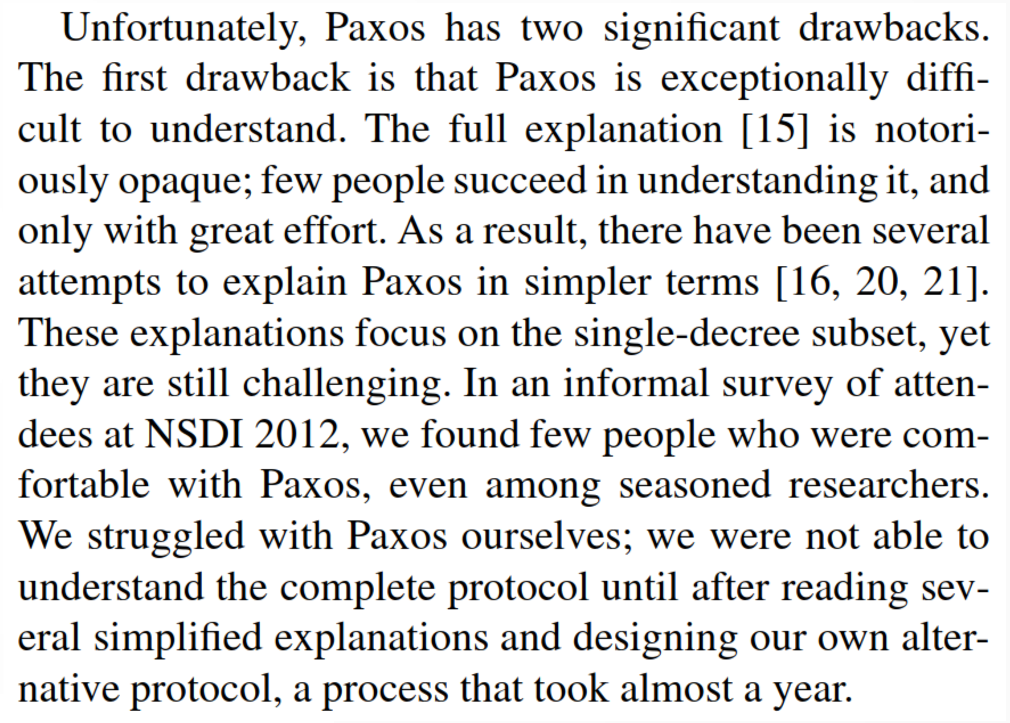
For example, legislator $\Lambda\breve{\iota}\nu\chi\partial$’s ledger had the entry

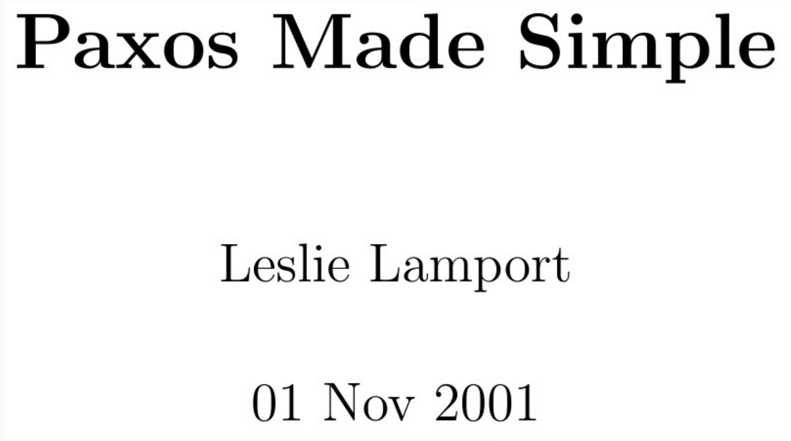
155: *The olive tax is 3 drachmas per ton*

132: *Lamps must use only olive oil*

Paxos Made Simple (?)

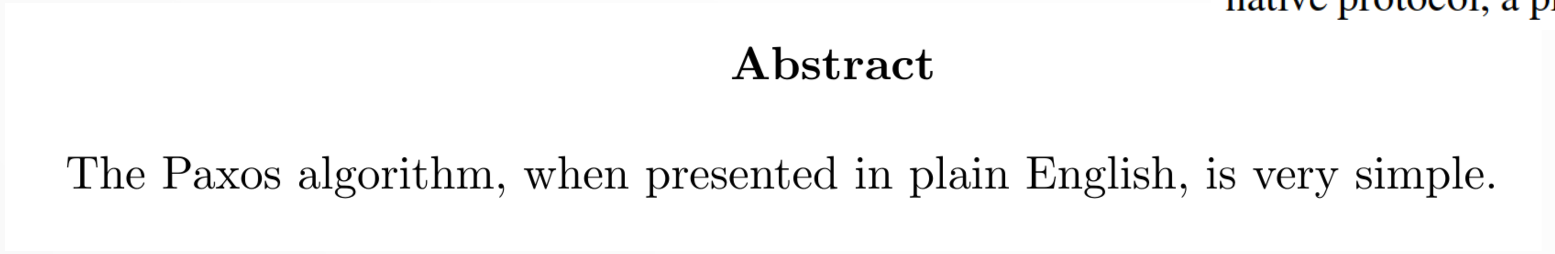
However… (Ongaro and Ousterhout 2014)


Unfortunately, Paxos has two significant drawbacks. The first drawback is that Paxos is exceptionally difficult to understand. The full explanation [15] is notoriously opaque; few people succeed in understanding it, and only with great effort. As a result, there have been several attempts to explain Paxos in simpler terms [16, 20, 21]. These explanations focus on the single-decree subset, yet they are still challenging. In an informal survey of attendees at NSDI 2012, we found few people who were comfortable with Paxos, even among seasoned researchers. We struggled with Paxos ourselves; we were not able to understand the complete protocol until after reading several simplified explanations and designing our own alternative protocol, a process that took almost a year.


Paxos Made Simple

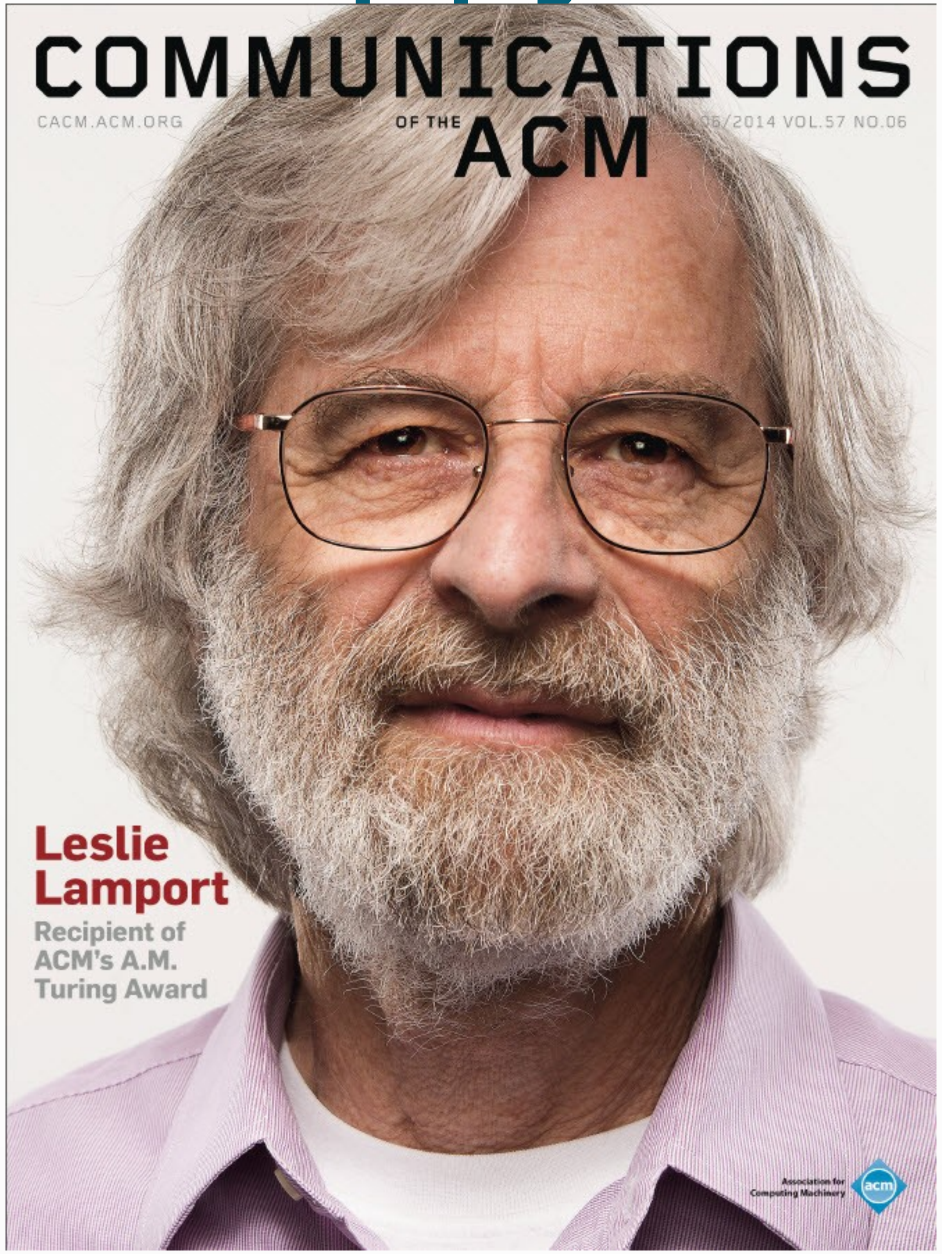
Leslie Lamport

01 Nov 2001


Abstract

The Paxos algorithm, when presented in plain English, is very simple.

Lamport's Happy Ending


COMMUNICATIONS OF THE ACM
CACM.ACM.ORG
06/2014 VOL.57 NO.06
Leslie Lamport
Recipient of ACM's A.M. Turing Award
Association for Computing Machinery
acm

So, What Is Paxos About?

- **Consensus**: creating a **shared log** in an asynchronous distributed system
- More precisely, this problem is solved by *Multi*-Paxos
- The basic Paxos algorithm finds consensus on **one** log entry
- Basic Paxos is *not so* difficult…

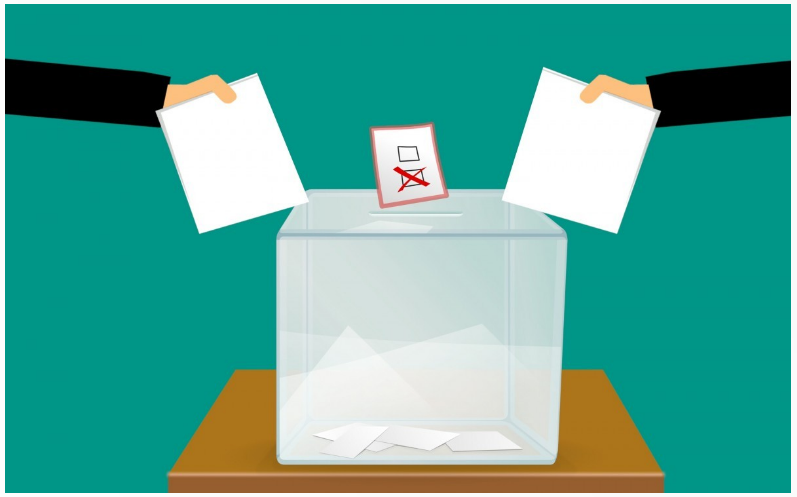
How Does Paxos Work?

- Say you have the old mobile phone we've seen before, and want to decide with your friends whether to go for pizza or sushi
  - You want to all agree and go to the same place
- Works like this:
  1) Send to all *"Hey, what are we doing tonight?"*
  2) If most of your friends answered and nobody has plans, propose (say, "*pizza*") and send to all
  3) If most of your friends agree, then it's decided.
- Doesn't sound very complicated, of course the devil is in the details...

Fail-Stop Failure Model

- *Failed* nodes stop working (e.g., your mobile has no connectivity)
  - They may resume (e.g., you eventually get signal)
  - Any message may get lost or arrive even after days… (you know, SMS…)
- **But**
  - Nobody loses their state (Nokia 3310 lasts forever)
  - Everybody is honest and makes no mistakes
  - To deal with malicious behavior, you'll need **byzantine** failure models

Majority Wins

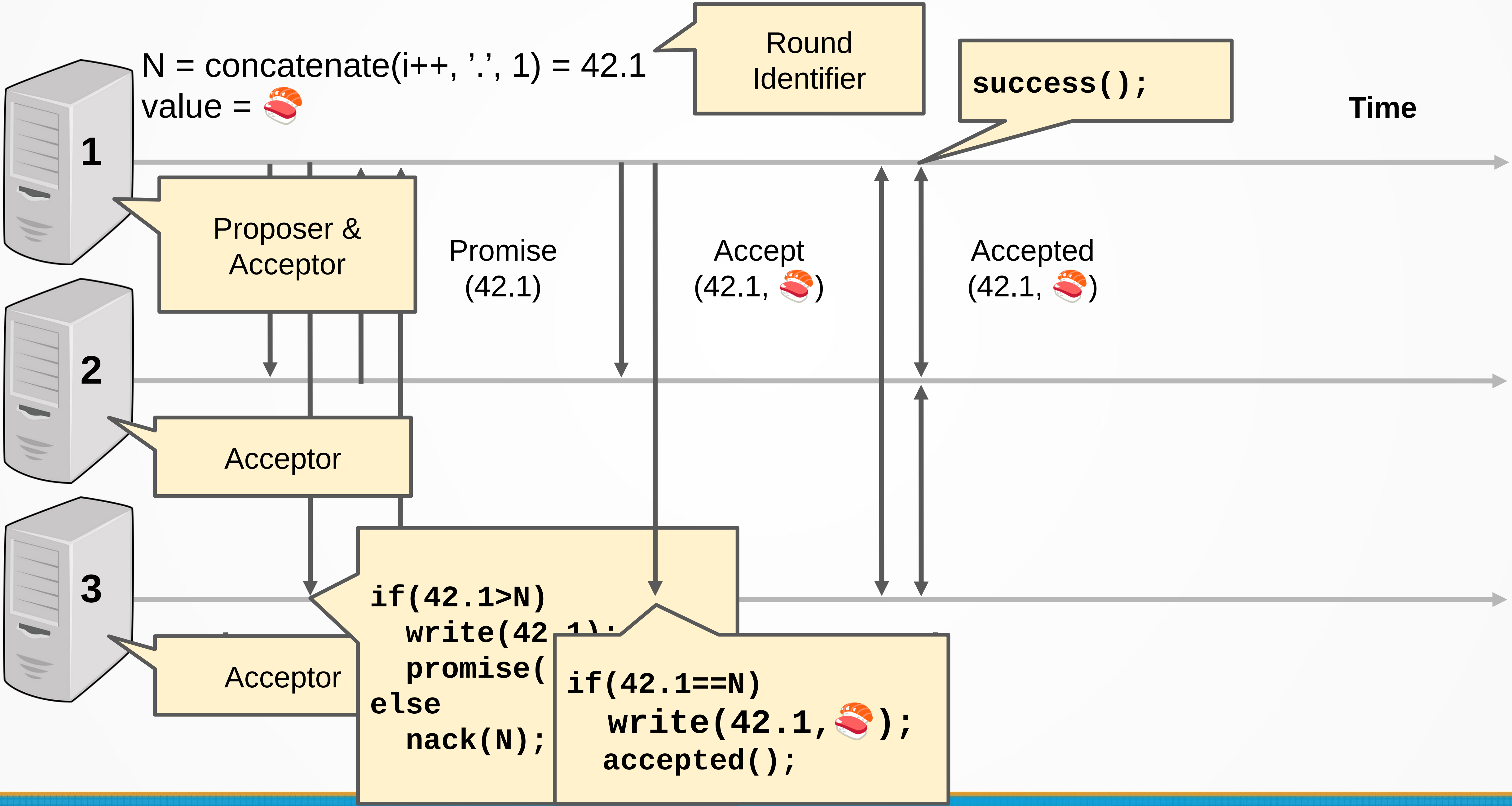


- We want to **tolerate *n* failures**
- We'll need **2n+1** servers
- A decision is taken when a majority of **n+1** nodes agree on it
  - Key reason: **two majorities must intersect**
  - Hence, **at least one participant will see a conflict** and avoid it
- Note: unlike regular elections, there are **no conflicting interests here**
  - Participants just want to **agree on something**

Thanks!

- The next slides are coming, with minor modifications and with permission, from the course by Chris Colohan (http://www.distributedsystemscourse.com/). Check it out!

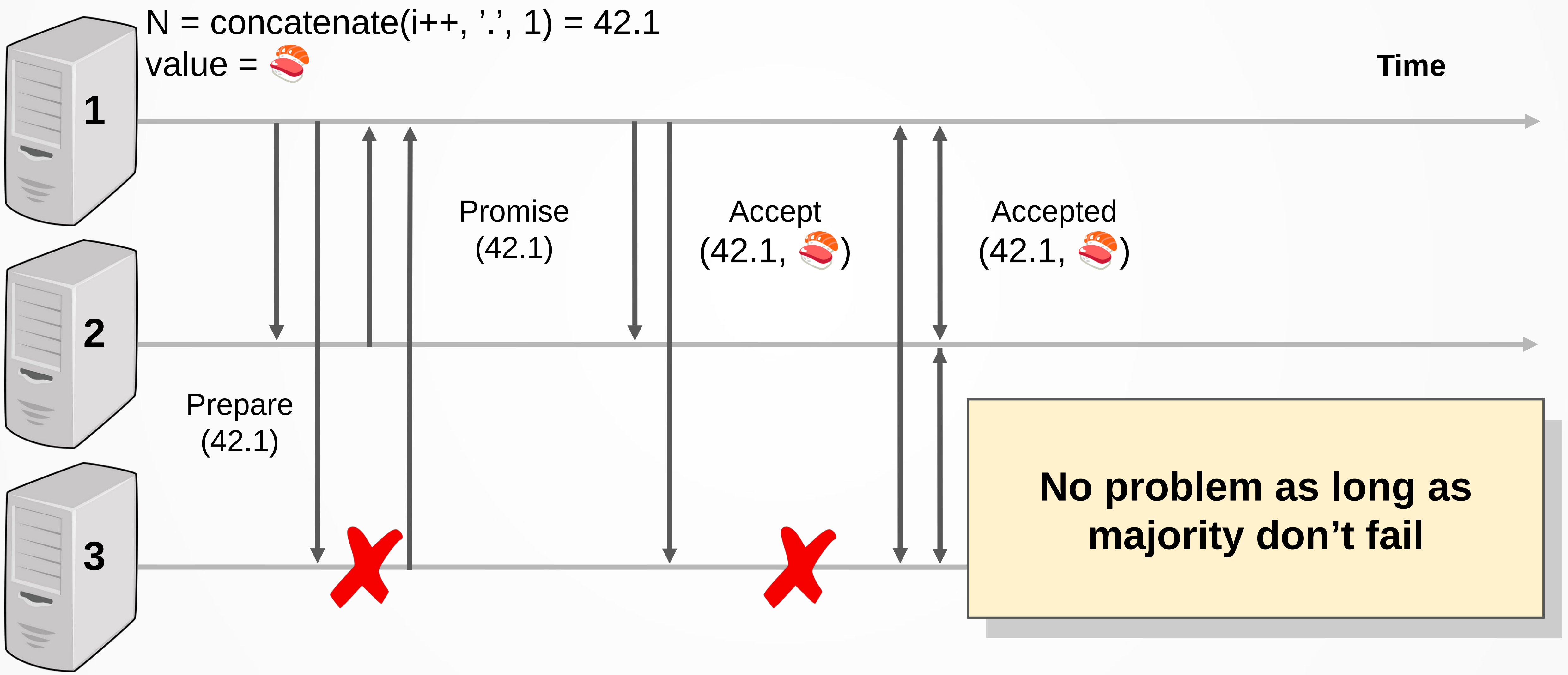
Basic Paxos



```
if(42.1>N)
  write(42
  promise(
else
  nack(N);
```

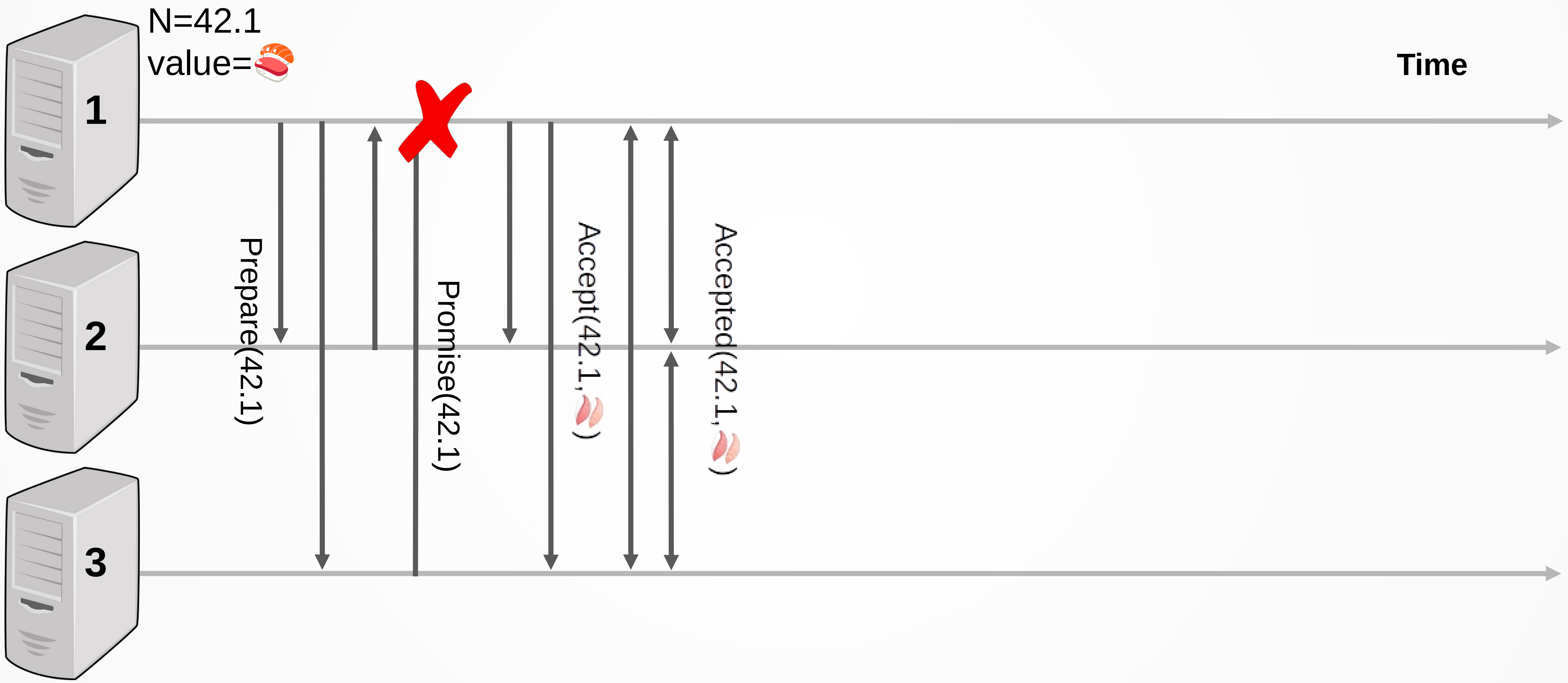
```
if(42.1==N)
  write(42.1,🍣);
  accepted();
```

Failures: Acceptor

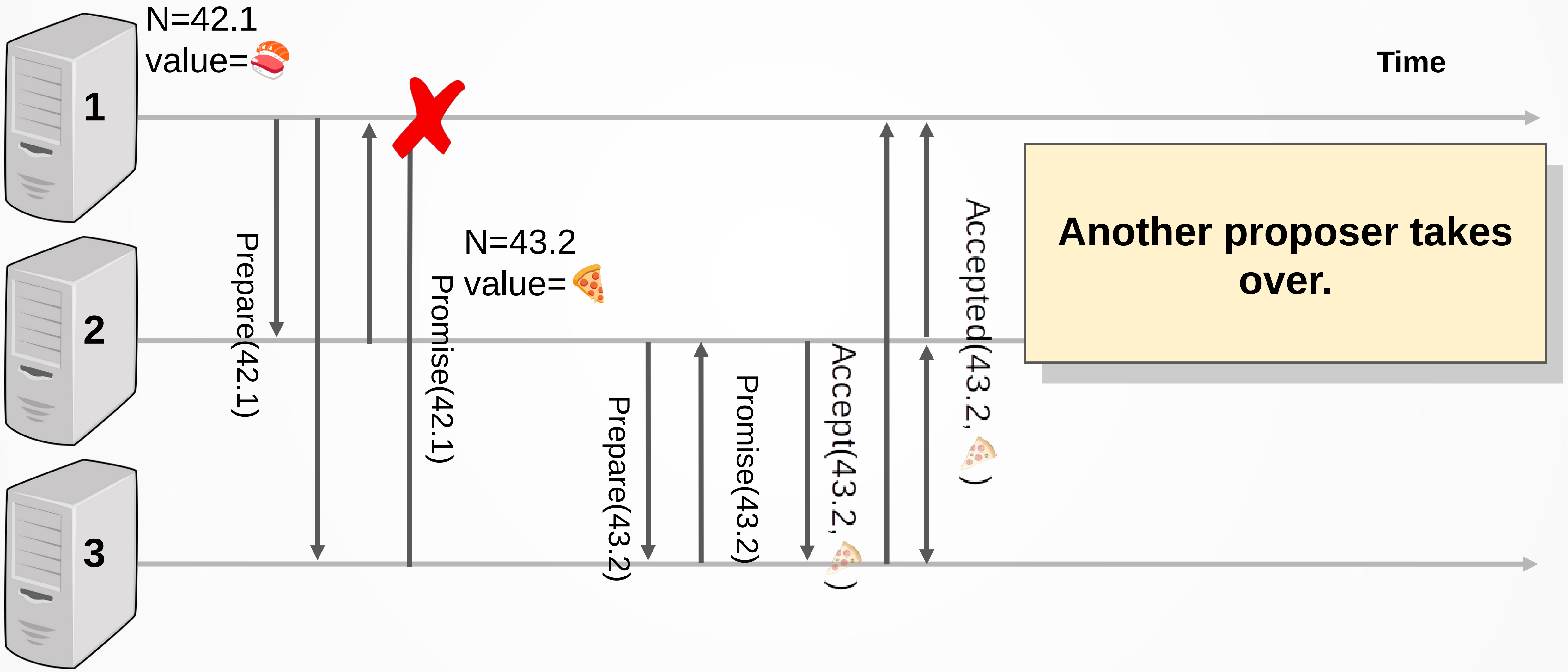


No problem as long as majority don't fail

Failures: Proposer in Prepare Phase

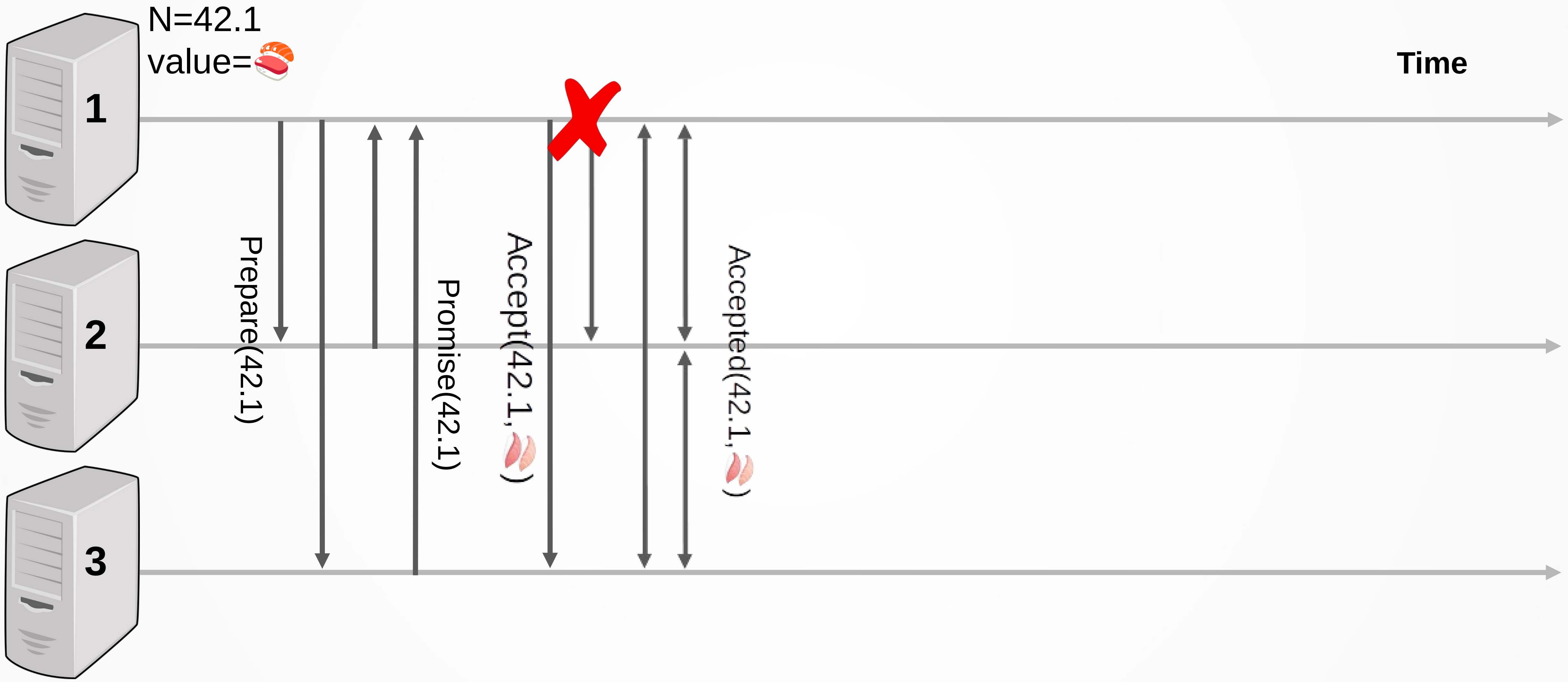


Failures: Proposer in Prepare Phase

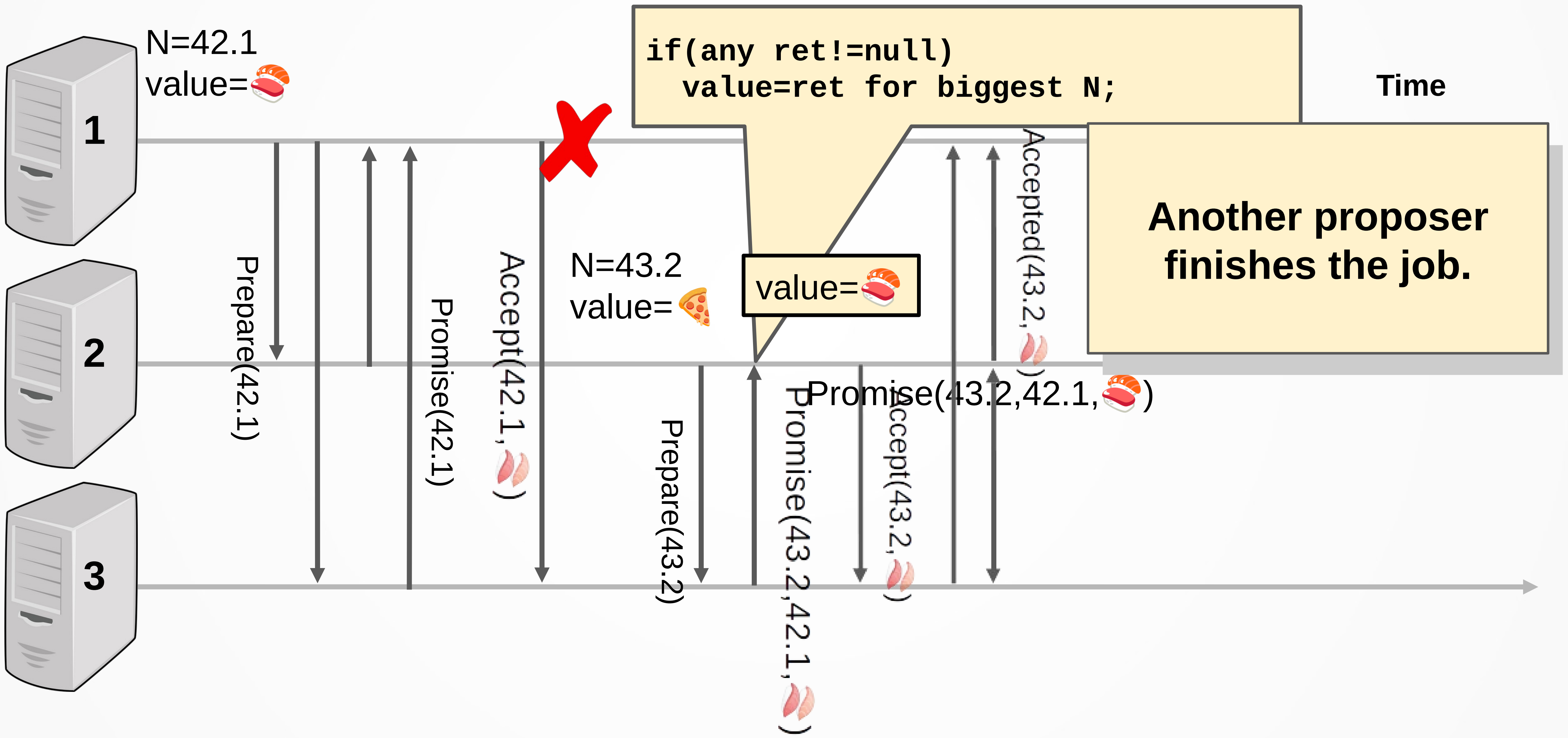


Another proposer takes over.

Failures: Proposer in Accept Phase



Failures: Proposer in Accept Phase



```
if(any ret!=null)
  value=ret for biggest N;
```

Another proposer finishes the job.

What could go wrong?

- If you have just one proposer
  - One or more acceptors fail
    - Still works as long as the majority is up
  - A proposer fails in the “prepare” phase
    - Nothing happens, another proposer should eventually show up and start the algorithm
  - A proposer fails in the “accept” phase
    - Either another proposer overwrites the decision
    - Or it gets notified of what’s been done until now, and finishes the job
- Two or more proposers
  - The algorithm will never result in a wrong output
  - But (in theory) it can result in a *livelock*--an infinite loop of messages

In the Real World

- Leader election
  - Only have a proposer at a time
  - You can do it with Paxos itself!
- Multi-Paxos
  - Build a full log of decisions
  - Additional complexity
  - Faster: one round-trip per transaction
- Cluster management
- Developing, testing and debugging is hard!

Raft

Credits Again

- Slides by Ben B. Johnson, CC BY 4.0 license
  - With very minor edits
- See also the original paper by Ongaro and Ousterhout

Conclusions

The Cost of Consensus

- Very difficult to implement and use correctly
  - Luckily, there are libraries!
- All machines responsible for consensus risk being a bottleneck
  - Risk of going at the speed of the slowest
- In common cases, a transaction takes a network round-trip to complete
  - If machines are all close, it’s quick **but** what about correlated failures?
  - Trade-off between reliability and latency!