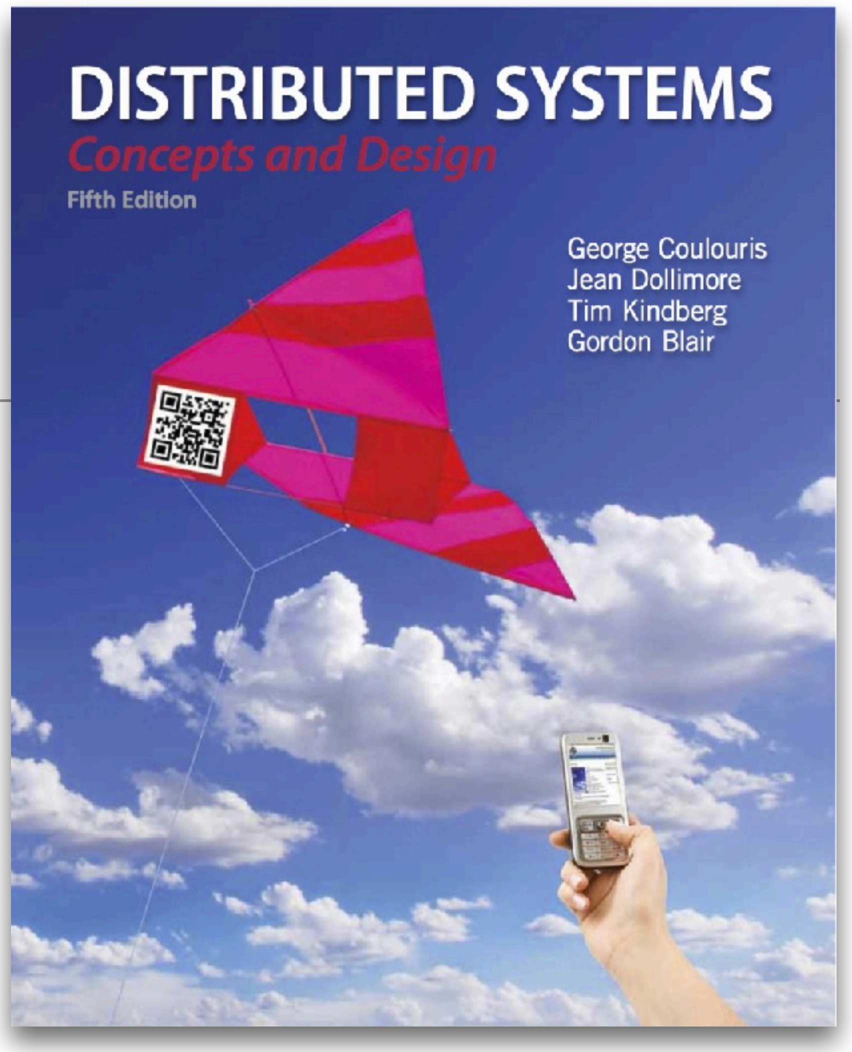
Distributed Systems 2023-2024: Consensus

Wouter Joosen & Tom Van Cutsem
DistriNet KU Leuven
November 2023

Outline

- **Consensus**: how to get a group of processes to all agree on the same value, even when networks are unreliable and processes may be faulty?
- Replicated State Machines
- System Models
- Defining Consensus
- Implementing Consensus (Consensus algorithms)
- Byzantine fault tolerance

Background reading



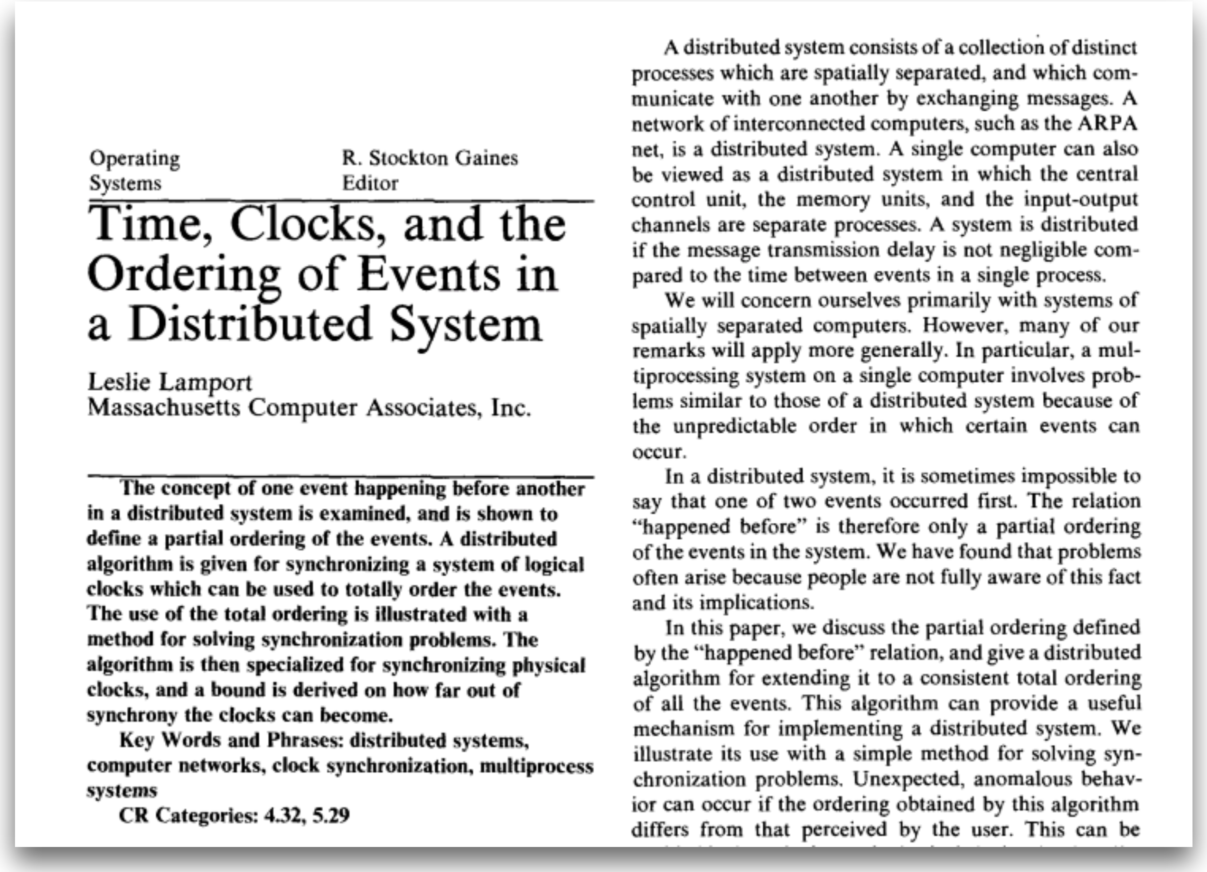
- CDK5 handbook
  - Chapter 15: section 15.5
  - Chapter 21: section 21.5.2

- Recommended course notes by prof. Martin Kleppmann (Cambridge University):
  - Sections 2.2, 2.3, 5.3 and 6.1
  - https://www.cl.cam.ac.uk/teaching/2223/ConcDisSys/dist-sys-notes.pdf

Replicated state machines

Problem

- Communication networks may fail
- Software processes may fail
- How can we still make a **reliable software system** from **unreliable parts?**
- Leslie Lamport: use replicated state machines (1978!)



Operating Systems — R. Stockton Gaines, Editor

Time, Clocks, and the Ordering of Events in a Distributed System

Leslie Lamport
Massachusetts Computer Associates, Inc.

The concept of one event happening before another in a distributed system is examined, and is shown to define a partial ordering of the events. A distributed algorithm is given for synchronizing a system of logical clocks which can be used to totally order the events. The use of the total ordering is illustrated with a method for solving synchronization problems. The algorithm is then specialized for synchronizing physical clocks, and a bound is derived on how far out of synchrony the clocks can become.

Key Words and Phrases: distributed systems, computer networks, clock synchronization, multiprocess systems

CR Categories: 4.32, 5.29

A distributed system consists of a collection of distinct processes which are spatially separated, and which communicate with one another by exchanging messages. A network of interconnected computers, such as the ARPA net, is a distributed system. A single computer can also be viewed as a distributed system in which the central control unit, the memory units, and the input-output channels are separate processes. A system is distributed if the message transmission delay is not negligible compared to the time between events in a single process.

We will concern ourselves primarily with systems of spatially separated computers. However, many of our remarks will apply more generally. In particular, a multiprocessing system on a single computer involves problems similar to those of a distributed system because of the unpredictable order in which certain events can occur.

In a distributed system, it is sometimes impossible to say that one of two events occurred first. The relation "happened before" is therefore only a partial ordering of the events in the system. We have found that problems often arise because people are not fully aware of this fact and its implications.

In this paper, we discuss the partial ordering defined by the "happened before" relation, and give a distributed algorithm for extending it to a consistent total ordering of all the events. This algorithm can provide a useful mechanism for implementing a distributed system. We illustrate its use with a simple method for solving synchronization problems. Unexpected, anomalous behavior can occur if the ordering obtained by this algorithm differs from that perceived by the user. This can be

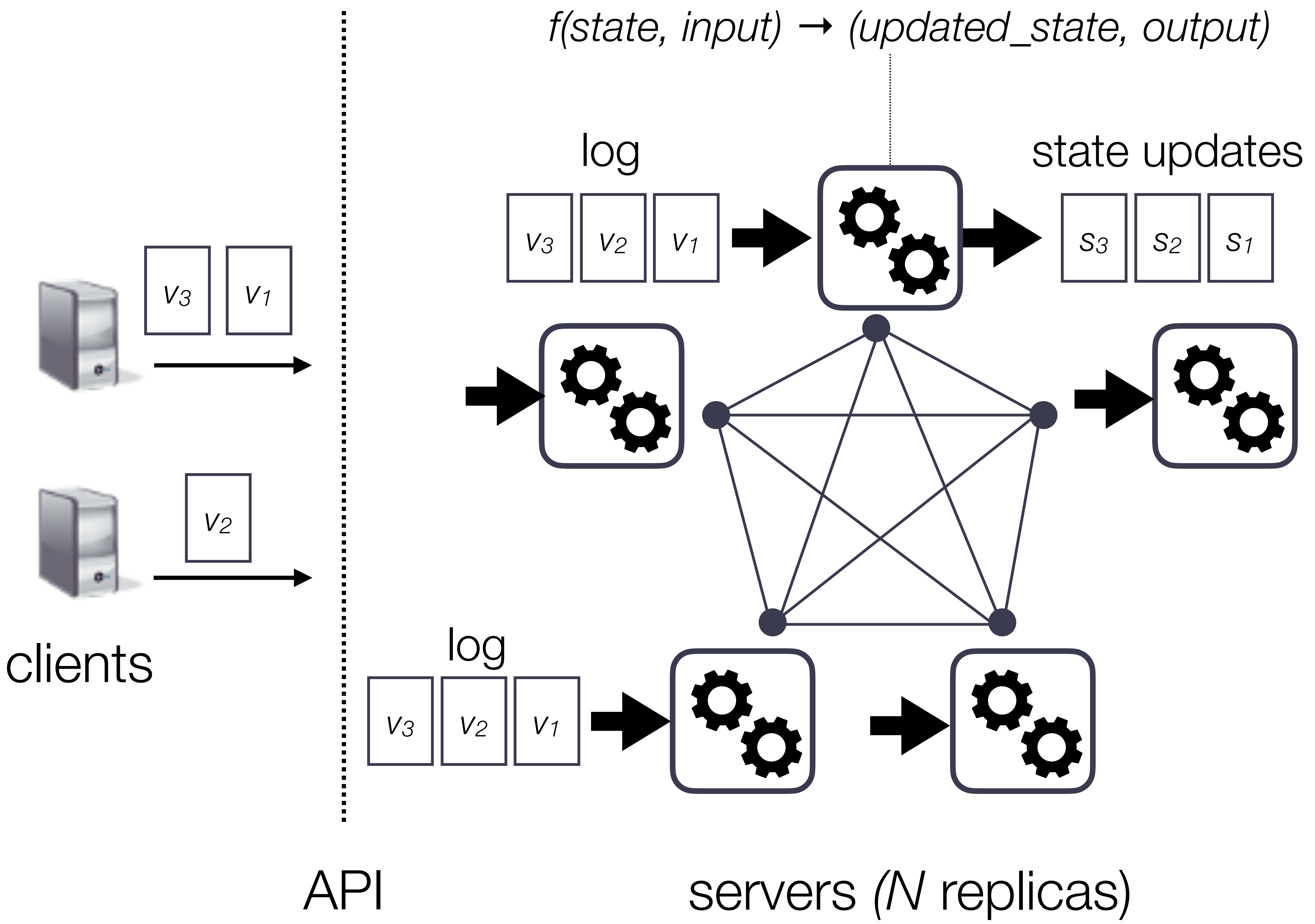
Communications of the ACM 1978

Leslie Lamport
(2013 Turing Award winner)

KU LEUVEN DistriNet

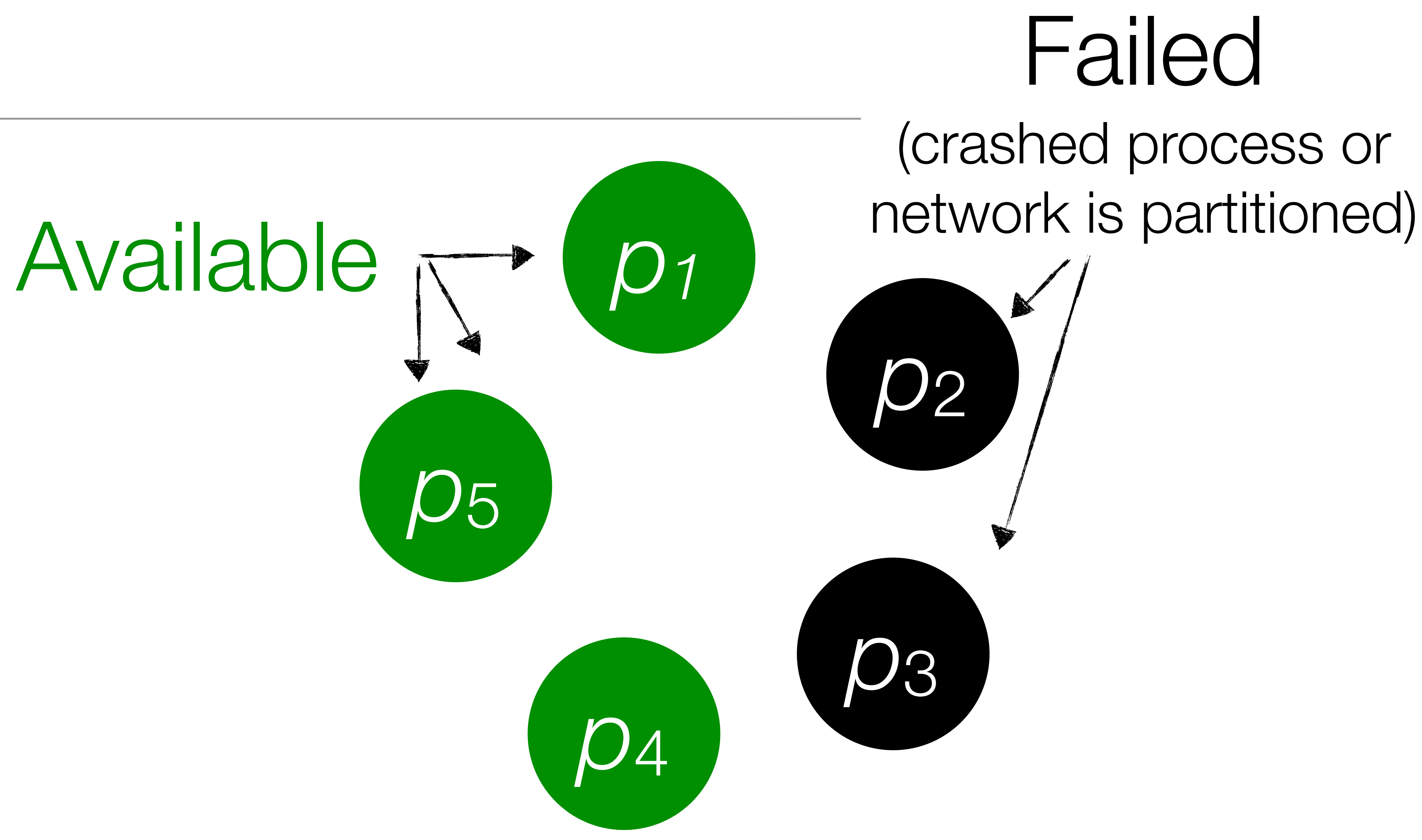
Replicated state machines: the key idea

- Model the service as a state machine with a <u>deterministic</u> *transition function:*
  f(state, input) ➝ (updated_state, output)
- Replicate this state machine *N* times on different processes. All processes read inputs from a *log*.
- If all state machines are initialized to the **same starting state and** the state machine function is **deterministic**, then **if the replicas process all the inputs in the same order** they will follow the same state transitions and produce exactly the same outputs.



Replicated state machines: the quorum

- To achieve fault-tolerance, the processes must be running on **different physical machines** in a cluster, or potentially even in different datacenters.
- Using a crash-fault tolerant consensus algorithm, the service can survive the failure of $f < N/2$ processes.
- In other words, **any majority** of processes can keep the service available. This is called a **quorum**.
- To tolerate f failures, set $N = 2f + 1$

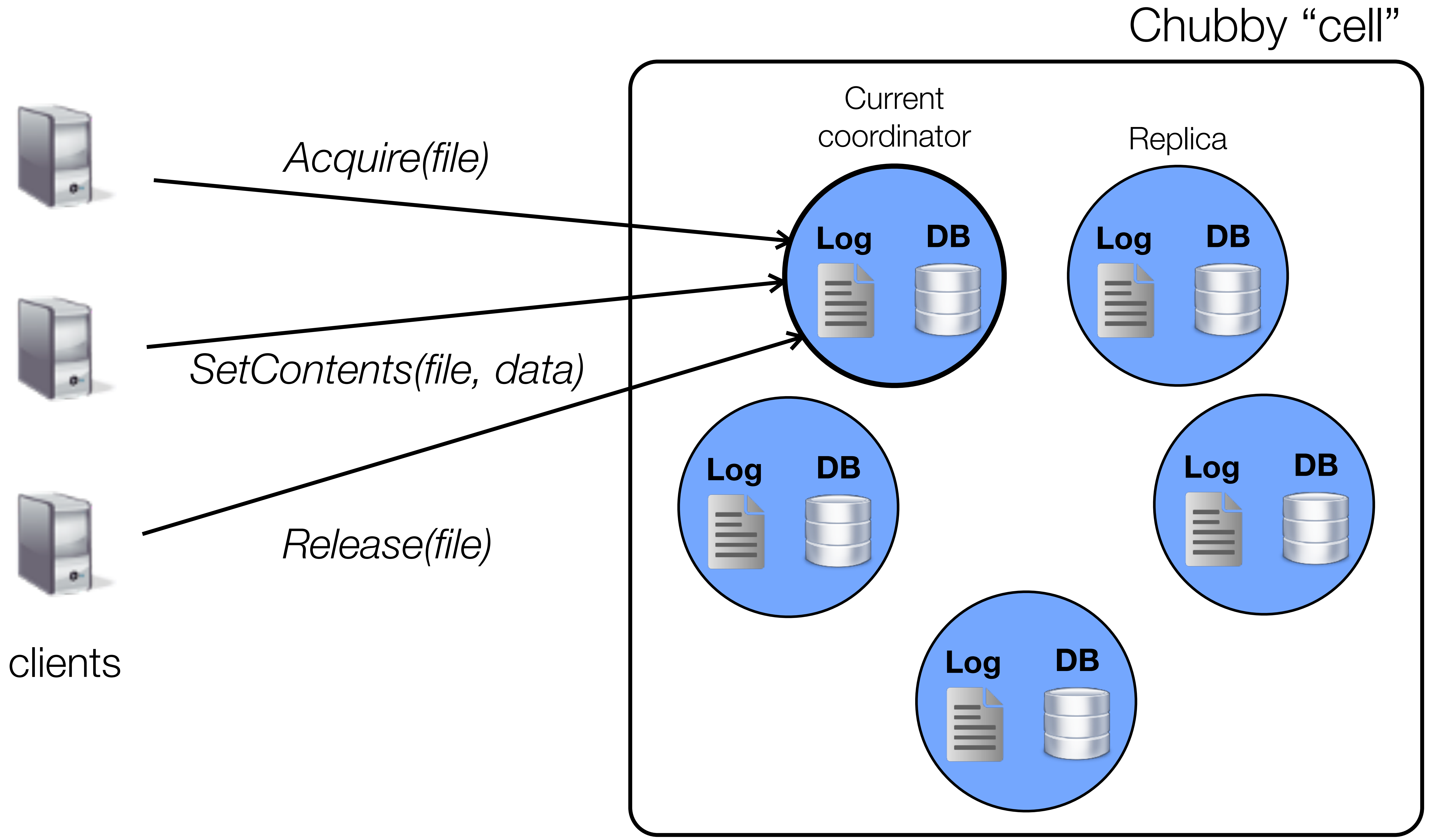


N	Quorum	Tolerated failures f
1	1	0
2	2	0
3	**2**	**1**
4	3	1
5	**3**	**2**
6	4	2

Common group size → row N = 5

Replicated state machines example: Google Chubby

- Chubby is a critical component in Google datacenter infrastructure implemented using the Replicated State Machine approach.



Replicated state machines example: Google Chubby

- Chubby's primary **use cases** in Google's datacenters:
  - **Lock service**: allow clients to acquire locks on files in a distributed file system
  - **Leader election**: allow election of a leader among a group of replicas (required in other Google services like BigTable). This can be easily built on top of the lock service:
    - All candidates attempt to lock a file, only 1 succeeds
    - The winner records its identity in the locked file and releases the lock
    - Other candidates can identify the leader by reading the contents of the file
- Chubby replicas keep their log consistent using the Paxos consensus algorithm (see later)

Open Source systems built using replicated state machines

- **Zookeeper:** a high-available configuration management, naming and locking service, inspired by Chubby
- **Consul:** a high-available service registry: register, lookup & configure services in a datacenter
- **etcd:** a "strongly consistent" replicated key-value store (a type of database)

System Models

Consensus algorithms: system models

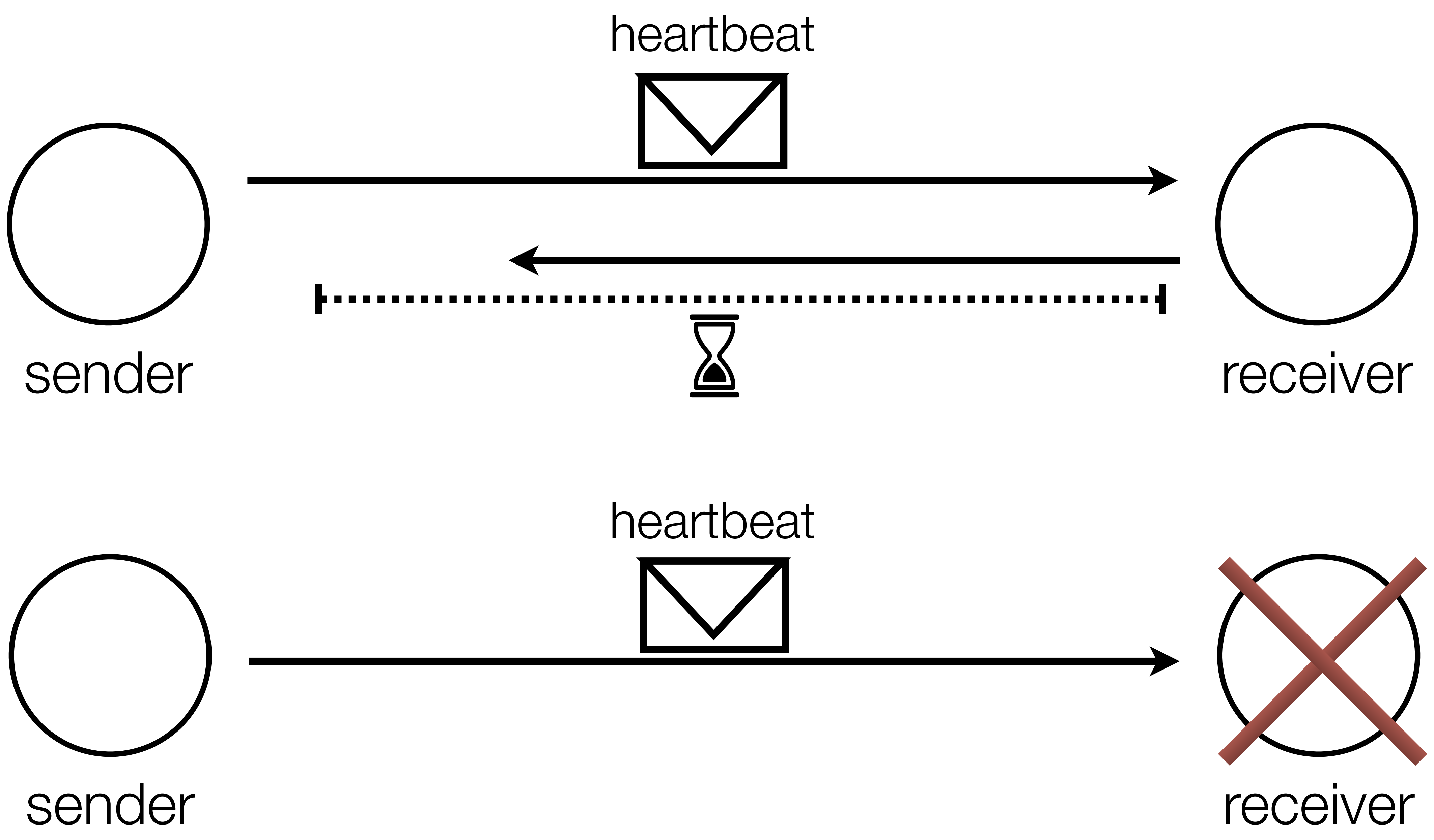
- To reason about consensus algorithms, we need a model of how a distributed system “behaves”
- This **makes the assumptions** that these algorithms rely on more **explicit**.
- The following system models are commonly used:
    - **Synchronous** system model
    - **Asynchronous** system model
    - **Partially synchronous** system model

Synchronous system model

- Assumes processes have **synchronized clocks** and that there is a maximum **upper bound** on message delivery across network links.
- This makes it possible to build a "perfect" failure detector that can accurately distinguish between a process failure and a network failure using **time-outs**

sender: "If I don't get a reply to my message before time-out, I assume the receiver has failed."

(We assume that case 1 can never happen)



Case 1:
slow or disconnected network

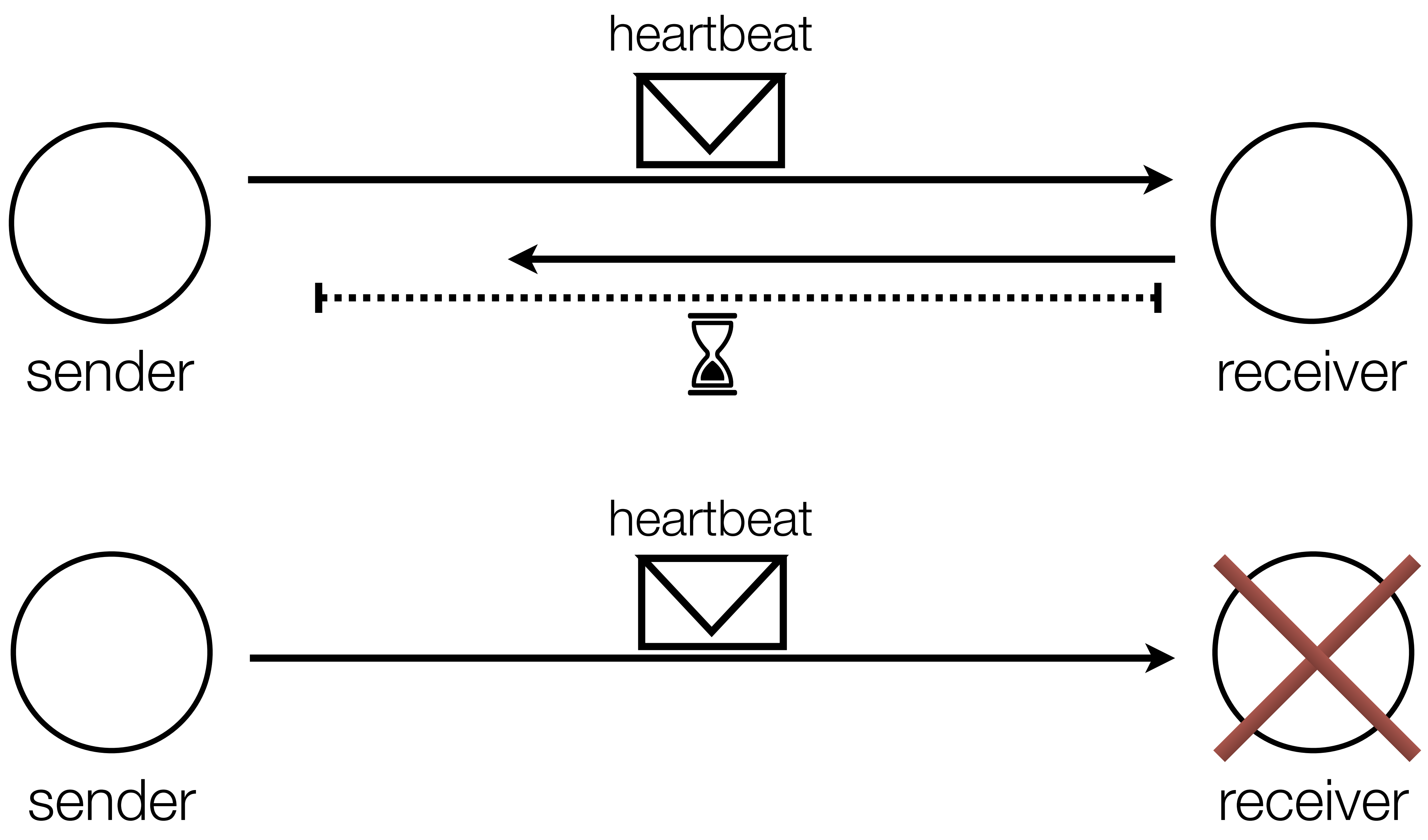
Case 2:
sender has crashed

Asynchronous system model

- Assumes **no synchronized clock** among processes, and that message delivery across network links is **unbounded** (it can take arbitrarily long for a message to be delivered)
- This makes it **impossible** to accurately distinguish between a node failure and a network failure (failure detectors will be imperfect)

sender: “If I don’t get a reply to my message before time-out, did the sender fail or did the message just get delayed?”

(We cannot distinguish case 1 from case 2)



Case 1:
slow or disconnected network

Case 2:
sender has crashed

Partially synchronous System Model

- Assume the **system behaves synchronously most of the time**. Most messages are delivered in a timely manner (in bounded time).

- But, there may be **bounded periods** (of finite but unknown duration) **where the system behaves asynchronously**. During this period, some messages may not be delivered in a timely manner. Processes may fail or network links may fail, but we assume they will *eventually* recover.

- In practice, the synchronous model is overly optimistic and the asynchronous model is overly pessimistic. The **partially synchronous model more closely approximates the behaviour of real-world distributed systems**.

- Most practical consensus algorithms (see later) assume this model.

Defining Consensus

The **consensus** problem in distributed systems

- How to get a group of distributed processes to all **agree** on the same value, even when network links are unreliable and processes may crash.

- **Examples** of values to agree on:

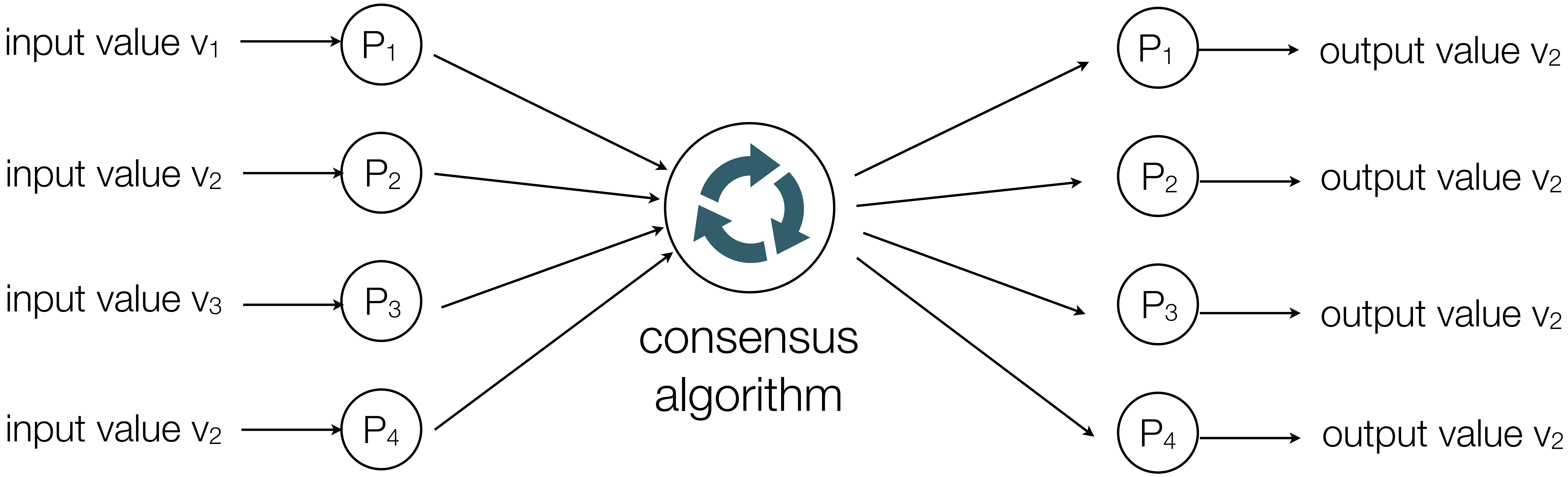
    - **Distributed mutual exclusion**: all processes in a group should **agree what process has acquired the lock** at any given time (see lecture on Lamport Clocks)

    - **Total-Order broadcast**: all processes in a group should **agree on the next message to deliver** to the group (see lecture on Group Communication)

    - **Replicated state machines:** all processes in a group should **agree on the next state update** to make to their state machine

Consensus: general problem formulation

- One or more processes **propose** a value. All correct processes must eventually **agree** on the **same** value

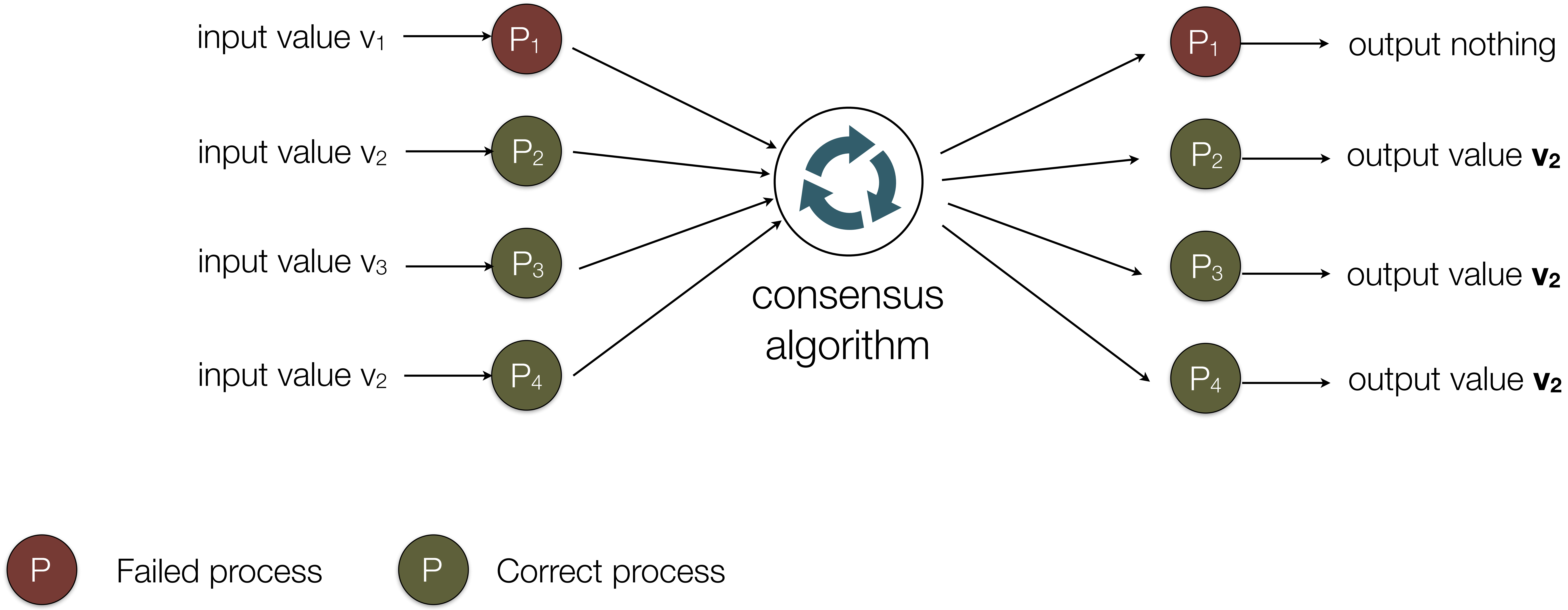


Consensus: desirable properties

- Agreement
- Validity
- Termination

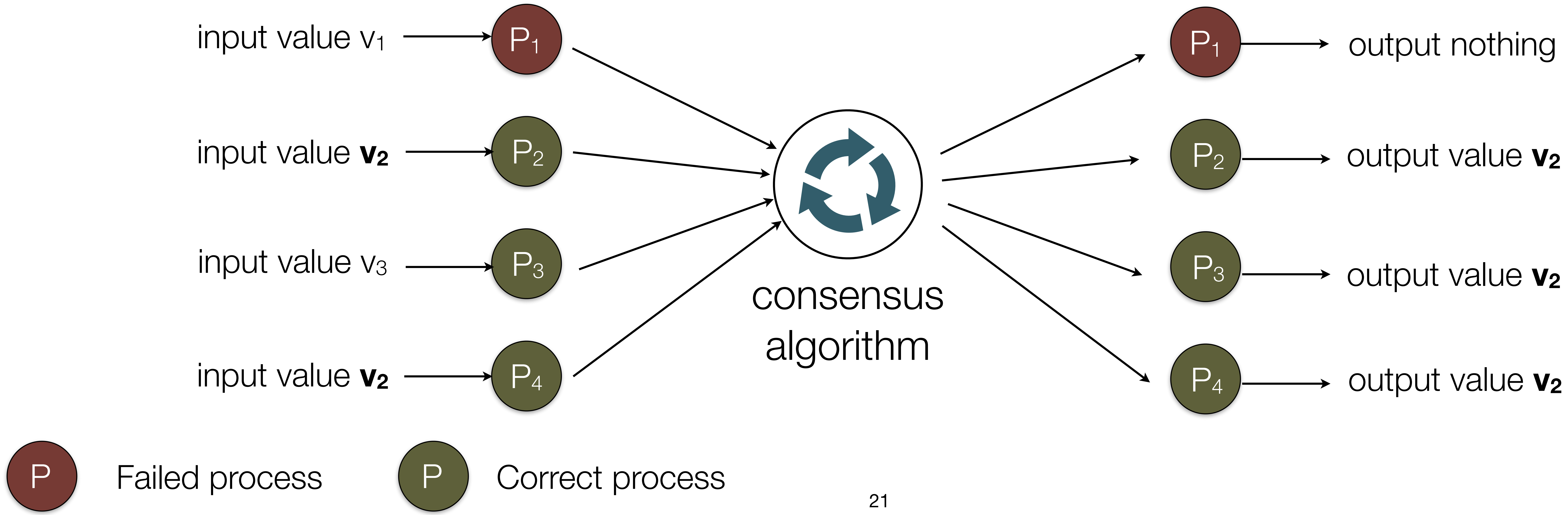
Consensus: desirable properties

- **Agreement**: all correct processes must output (= “decide on”) the same value



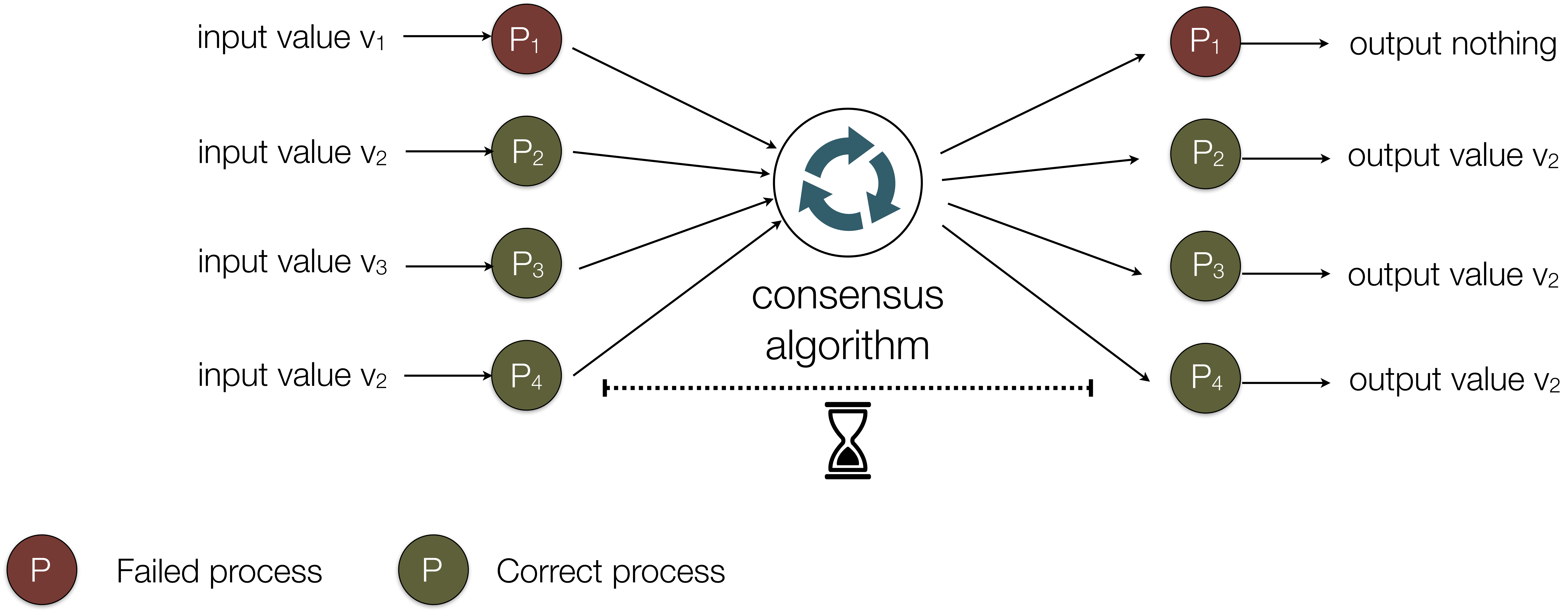
Consensus: desirable properties

- **Validity**: the output value for *all* correct processes must have been provided as the input value for *some* correct process
- If all processes propose the same input value, that value can be the *only* possible output that is decided on



Consensus: desirable properties

- **Termination**: every *correct* process *eventually* outputs some value



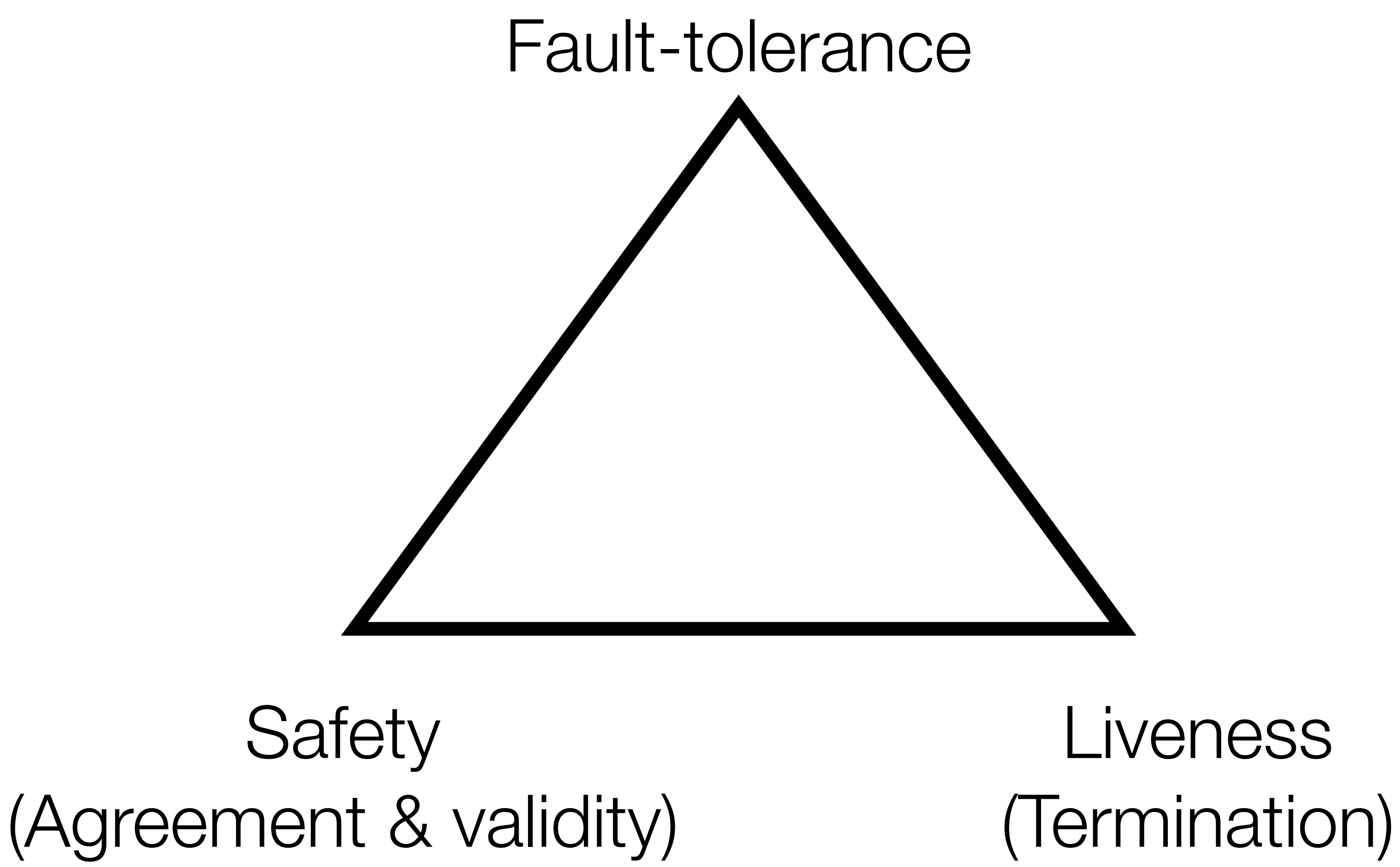
Consensus algorithms: desirable properties

- Agreement and validity are **safety** properties
- Termination is a **liveness** property

- **Safety** properties guarantee that “nothing ***bad*** will ***ever*** happen”
- **Liveness** properties guarantee that “something ***good*** will ***eventually*** happen”

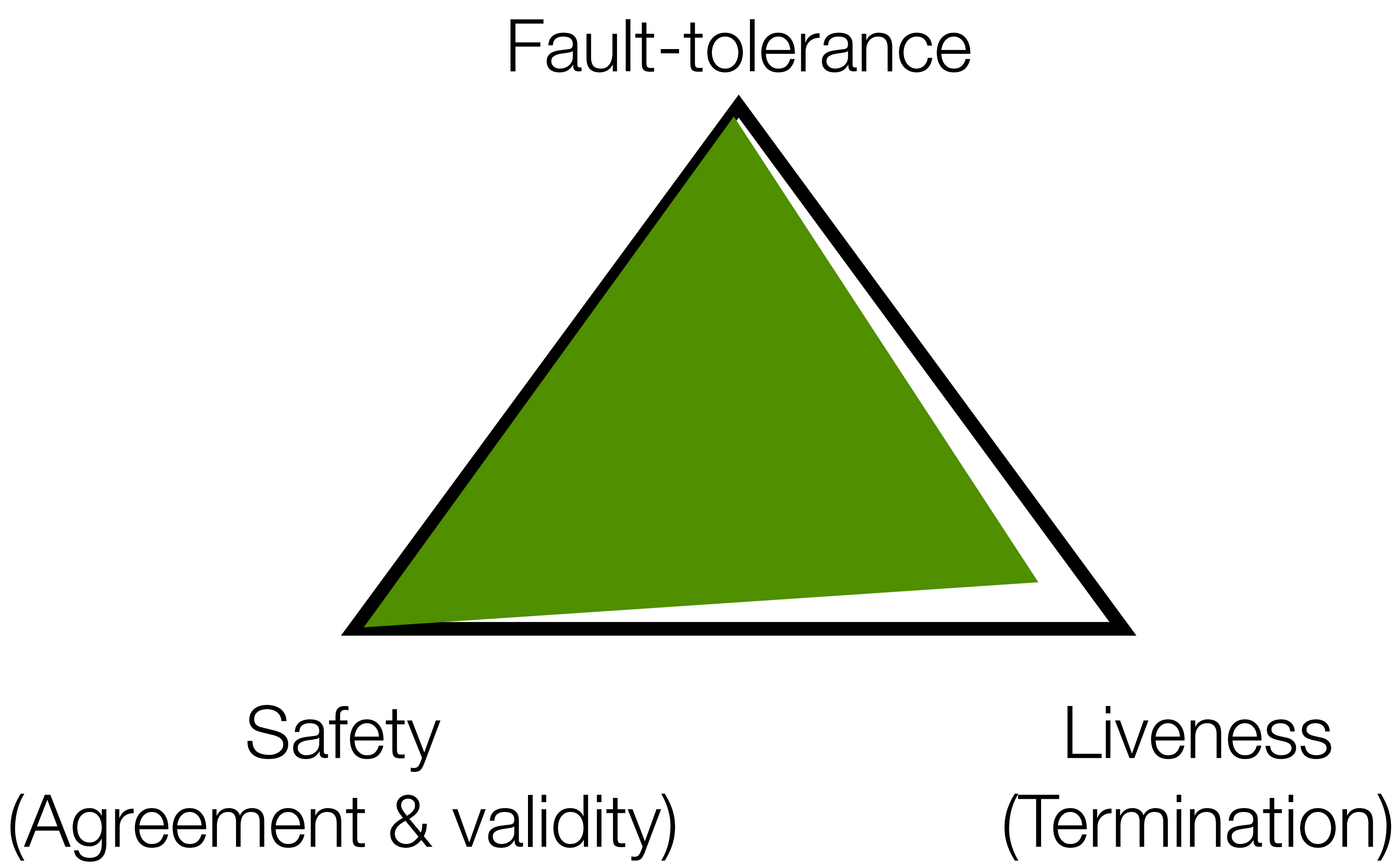
Impossibility of consensus (in theory)

- The “**FLP** impossibility result” after Fischer, Lynch and Paterson (1985) states that consensus cannot be *guaranteed* to be achieved *in bounded time* if there is even a single faulty process, *assuming an asynchronous network model*.
- If messages keep on being delayed due to failures, processes may remain forever undecided.
- In simplified terms: “agreement, termination and fault-tolerance: choose two”
- For a detailed explanation, see: https://www.the-paper-trail.org/post/2008-08-13-a-brief-tour-of-flp-impossibility/



Impossibility of consensus (in theory)

- The "FLP impossibility result" does *not* state that reaching consensus is always impossible.
- The result only states that in an asynchronous system there is **no guarantee that consensus can always be achieved in bounded time**
- We give up on one of the 3 desirable properties: we can no longer *guarantee* liveness, but we can achieve it in practice *with high probability*
- How? By assuming a partially synchronous system model (assume an upper bound on message delivery to detect and react to failed processes)
- In practice we can **detect failures** using time-outs (with limited clock synchronisation), we can **checkpoint/restore** crashed processes and we can **apply randomness** to avoid electing the same failing process as a leader over and over again.



Side-note: Consensus versus Atomic Commit

- Recall lecture on Distributed transactions: processes must **agree** consistently on whether **to commit or abort** a transaction using an Atomic Commit Protocol such as Two-phase Commit (2PC).
- Is this the same problem as consensus? Similar, but not the same:

Atomic Commit	Consensus
Every process votes whether to commit or abort	One or more processes propose a value
Must commit if *all* processes vote to commit; must abort if *at least one* node votes to abort	Any one of the proposed values is decided
Must abort if *any* participating process crashes	Crashed processes can be tolerated, as long as a quorum (majority) is still available

Implementing Consensus (Consensus Algorithms)

Consensus, total order broadcast and replicated state machines

- The consensus problem is equivalent to the reliable Total-Order broadcast problem, but formulated more broadly.
- If we can reliably broadcast updates to each replicated state machine using FIFO-Total Order broadcast, then all replicas will process the updates in the same order!
- Then we can implement a replicated state machine as follows:

on request to perform update u **do**
    send u via FIFO-total order broadcast
end on

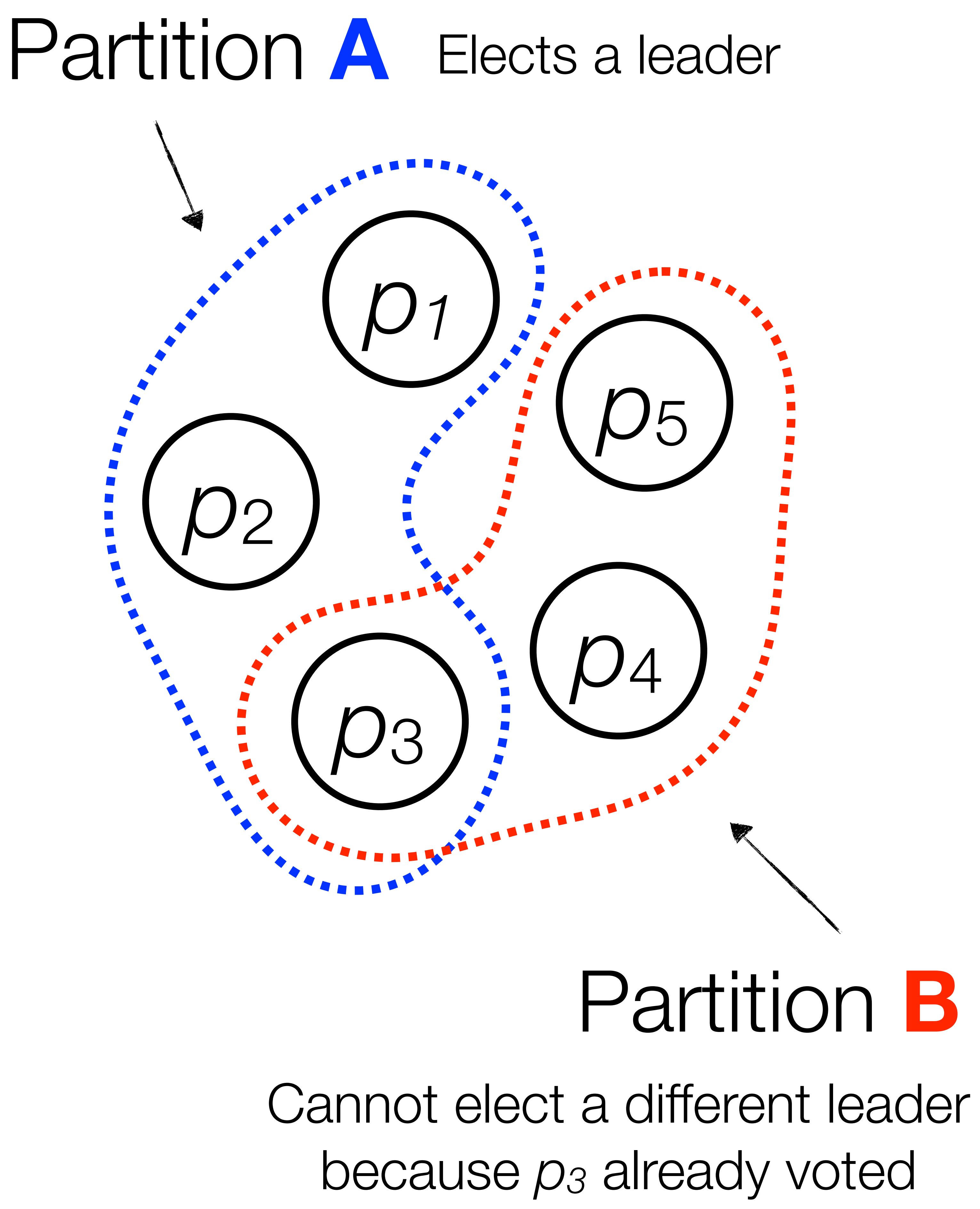
on delivering u through FIFO-total order broadcast **do**
    update state using arbitrary deterministic logic!
end on

Implementing consensus using reliable Total-Order broadcast

- **If we can implement reliable Total-Order broadcast, we can implement Consensus**.
- Recall the algorithms for Total-Order broadcast we discussed earlier:
  - **Single-leader:** route all messages through a single leader who then decides on the order of delivery
  - **Lamport timestamps:** attach Lamport timestamps to all messages and deliver them in timestamp order. Ensure that all processes have advanced up to at least the message timestamp before delivering the message (by waiting for timestamped acknowledgements, as in Lamport's Distributed Mutual Exclusion algorithm)
- **Neither implementation tolerates failures!**
  - The single-leader algorithm fails to make progress if the leader crashes or becomes otherwise unavailable.
  - Lamport's algorithm fails to make progress if any process fails to send an acknowledgement.
- Can we make the single-leader algorithm fault tolerant, e.g. by **automatically choosing a new leader** from the group?

Leader election

- Consensus algorithms use a leader to sequence messages.
- Use a **failure detector** (timeout) to determine *suspected* crash or unavailability of leader.
  - On suspected leader crash, **elect a new one**.
  - Prevent **two leaders at the same time** ("split-brain")!
- Ensure ≤ 1 leader per **term**:
  - Term is incremented every time a leader election is started
  - A node can only vote once per term
  - Require a **quorum** of nodes to elect a leader in a term



Consensus algorithms: academic literature

- **Paxos** (Lamport, 1989): initially often misunderstood, later widely influential (cfr. its use in Google Chubby). The standard Paxos algorithm only provides agreement on a *single* value. An extension is needed for agreement on *sequences* of values (called **Multi-Paxos**).

- **Raft** (Ongaro and Ousterhout, 2014): a consensus algorithm designed specifically for log replication. Considered easier to understand and implement than Paxos. Supports agreement on sequences of values. Used in systems like Consul and etcd.

- **Viewstamped Replication** (Oki and Liskov, 1988): designed specifically for consensus on message delivery in group communication (Total-Order broadcast).

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.

Categories and Subject Descriptors: C2.4 [**Computer-Communications Networks**]: Distributed Systems—*Network operating systems*; D4.5 [**Operating Systems**]: Reliability—*Fault-tolerance*; J.1 [**Administrative Data Processing**]: Government

General Terms: Design, Reliability

Additional Key Words and Phrases: State machines, three-phase commit, voting

In Search of an Understandable Consensus Algorithm (Extended Version)

Diego Ongaro and John Ousterhout
Stanford University

Abstract

Raft is a consensus algorithm for managing a replicated log. It produces a result equivalent to (multi-)Paxos, and it is as efficient as Paxos, but its structure is different from Paxos; this makes Raft more understandable than Paxos and also provides a better foundation for building practical systems. In order to enhance understandability, Raft separates the key elements of consensus, such as leader election, log replication, and safety, and it enforces a stronger degree of coherency to reduce the number of states that must be considered. Results from a user study demonstrate that Raft is easier for students to learn than Paxos. Raft also includes a new mechanism for changing the cluster membership, which uses overlapping majorities to guarantee safety.

state space reduction (relative to Paxos, Raft reduces the degree of nondeterminism and the ways servers can be inconsistent with each other). A user study with 43 students at two universities shows that Raft is significantly easier to understand than Paxos: after learning both algorithms, 33 of these students were able to answer questions about Raft better than questions about Paxos.

Raft is similar in many ways to existing consensus algorithms (most notably, Oki and Liskov's Viewstamped Replication [29, 22]), but it has several novel features:

- **Strong leader:** Raft uses a stronger form of leadership than other consensus algorithms. For example, log entries only flow from the leader to other servers. This simplifies the management of the replicated log and makes Raft easier to understand.

Viewstamped Replication: A New Primary Copy Method to Support Highly-Available Distributed Systems

Brian M. Oki
Barbara H. Liskov

Massachusetts Institute of Technology
Laboratory for Computer Science
Cambridge, MA 02139

Abstract

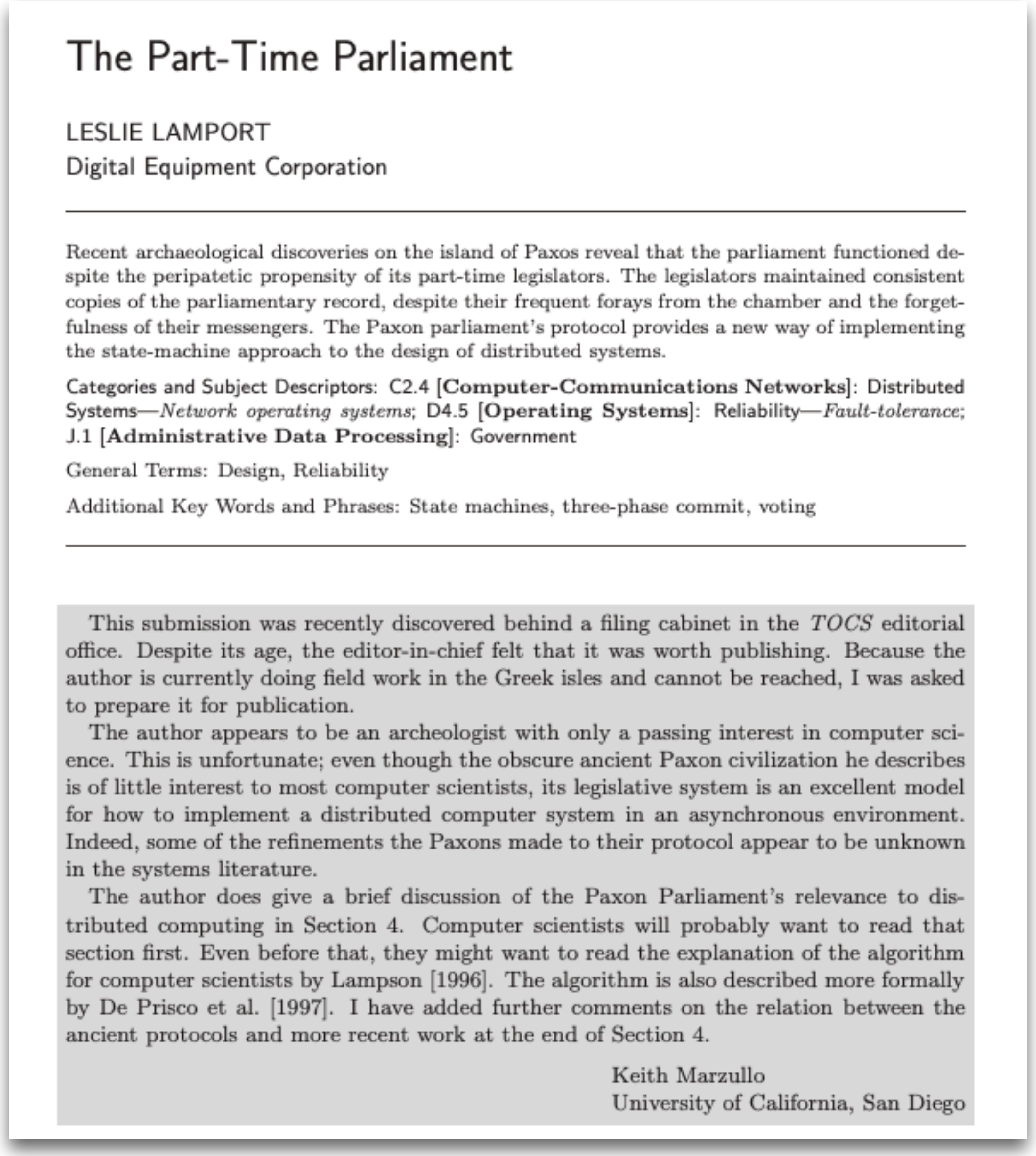
One of the potential benefits of distributed systems is their use in providing highly-available services that are likely to be usable when needed. Availabilty is achieved through replication. By having more than one copy of information, a service continues to be usable even when some copies are inaccessible, for example, because of a crash of the computer where a copy was stored. This paper presents a new replication algorithm that has desirable performance properties. Our approach is based on the primary copy technique. Computations run at a primary, which notifies its backups of what it has done. If the primary crashes, the backups are reorganized, and one of the backups becomes the new primary. Our method works in a general network with both node crashes and partitions. Replication causes little delay in user computations and little information is lost in a reorganization; we use a special kind of timestamp called a viewstamp to detect lost information.

Our algorithm runs on a system consisting of nodes connected by a communication network. Nodes are independent computers that communicate with each other only by sending messages over the network. Although both nodes and the network may fail, we assume these failures are not byzantine [24]. Nodes can crash, but we assume they are failstop processors [34]. The network may lose, delay, and duplicate messages, or deliver messages out of order. Link failures may cause the network to partition into subnetworks that are unable to communicate with each other. We assume that nodes eventually recover from crashes and partitions are eventually repaired.

Our replication method assumes a model of computation in which a distributed program consists of *modules*, each of which resides at a single node of the network. Each module contains within it both data objects and code that manipulates the objects; modules can recover from crashes with some of their state intact. No other module can access the data objects of another module directly.

Paxos: a fault-tolerant consensus algorithm

- Invented by Leslie Lamport in 1989.
- Widely considered one of the most important algorithms in distributed systems, but with a reputation for being difficult to understand and implement.
- Of **practical importance**: forms the basis for building fault-tolerant services (e.g. it is used as part of Google's Chubby to implement a reliable distributed locking service)
- Why is it called Paxos? To illustrate the algorithm, Lamport used the example of a fictional parliament on the Greek Island of Paxos



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.

Categories and Subject Descriptors: C2.4 [**Computer-Communications Networks**]: Distributed Systems—*Network operating systems*; D4.5 [**Operating Systems**]: Reliability—*Fault-tolerance*; J.1 [**Administrative Data Processing**]: Government

General Terms: Design, Reliability

Additional Key Words and Phrases: State machines, three-phase commit, voting

This submission was recently discovered behind a filing cabinet in the *TOCS* editorial office. Despite its age, the editor-in-chief felt that it was worth publishing. Because the author is currently doing field work in the Greek isles and cannot be reached, I was asked to prepare it for publication.

The author appears to be an archeologist with only a passing interest in computer science. This is unfortunate; even though the obscure ancient Paxon civilization he describes is of little interest to most computer scientists, its legislative system is an excellent model for how to implement a distributed computer system in an asynchronous environment. Indeed, some of the refinements the Paxons made to their protocol appear to be unknown in the systems literature.

The author does give a brief discussion of the Paxon Parliament's relevance to distributed computing in Section 4. Computer scientists will probably want to read that section first. Even before that, they might want to read the explanation of the algorithm for computer scientists by Lampson [1996]. The algorithm is also described more formally by De Prisco et al. [1997]. I have added further comments on the relation between the ancient protocols and more recent work at the end of Section 4.

Keith Marzullo
University of California, San Diego

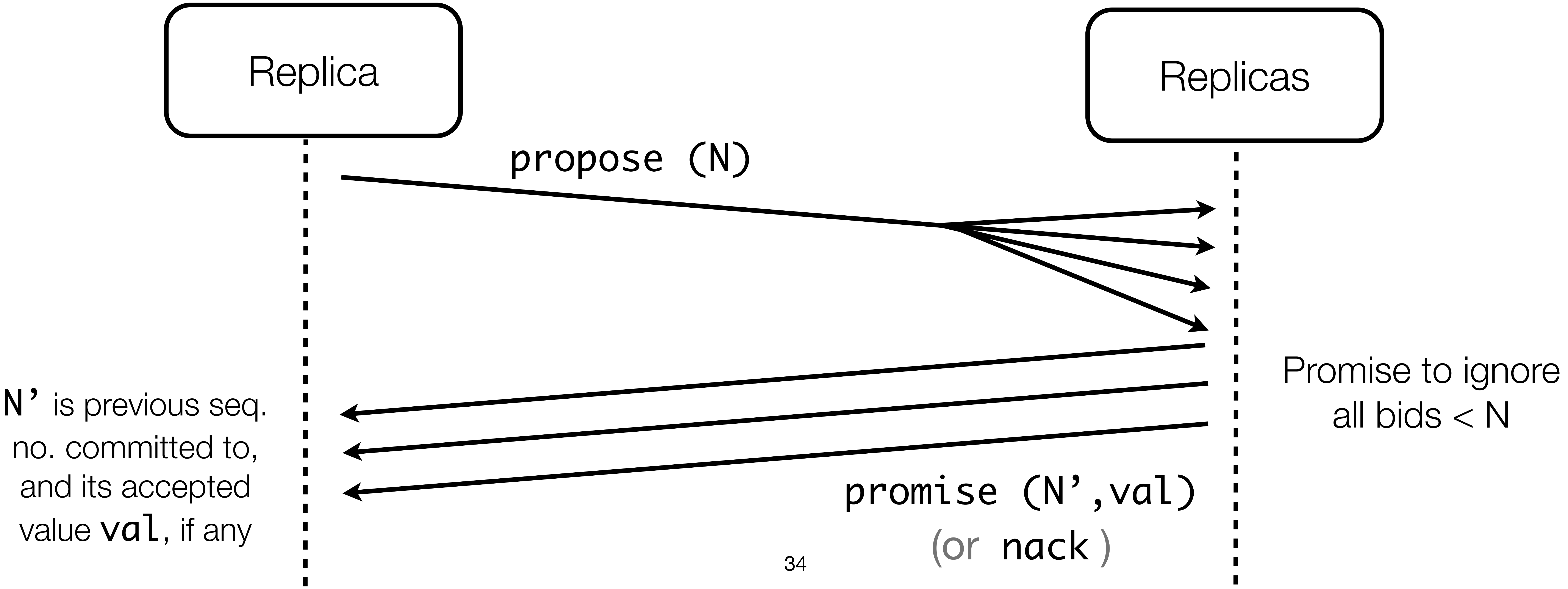
Leslie Lamport: "The Part-time Parliament"
(originally published in 1989)

Paxos: assumptions

- Paxos **does not assume synchronized clocks**. Processes operate at their own speed.
- **Processes may fail** (and subsequently recover). Processes have access to stable, persistent storage that survives crashes.
- **The network may fail.** Messages may take an arbitrarily long time to be delivered.
- Paxos assumes processes are **cooperative** (i.e. processes will follow the algorithm truthfully). Paxos does *not* deal with “byzantine failures” (see later)

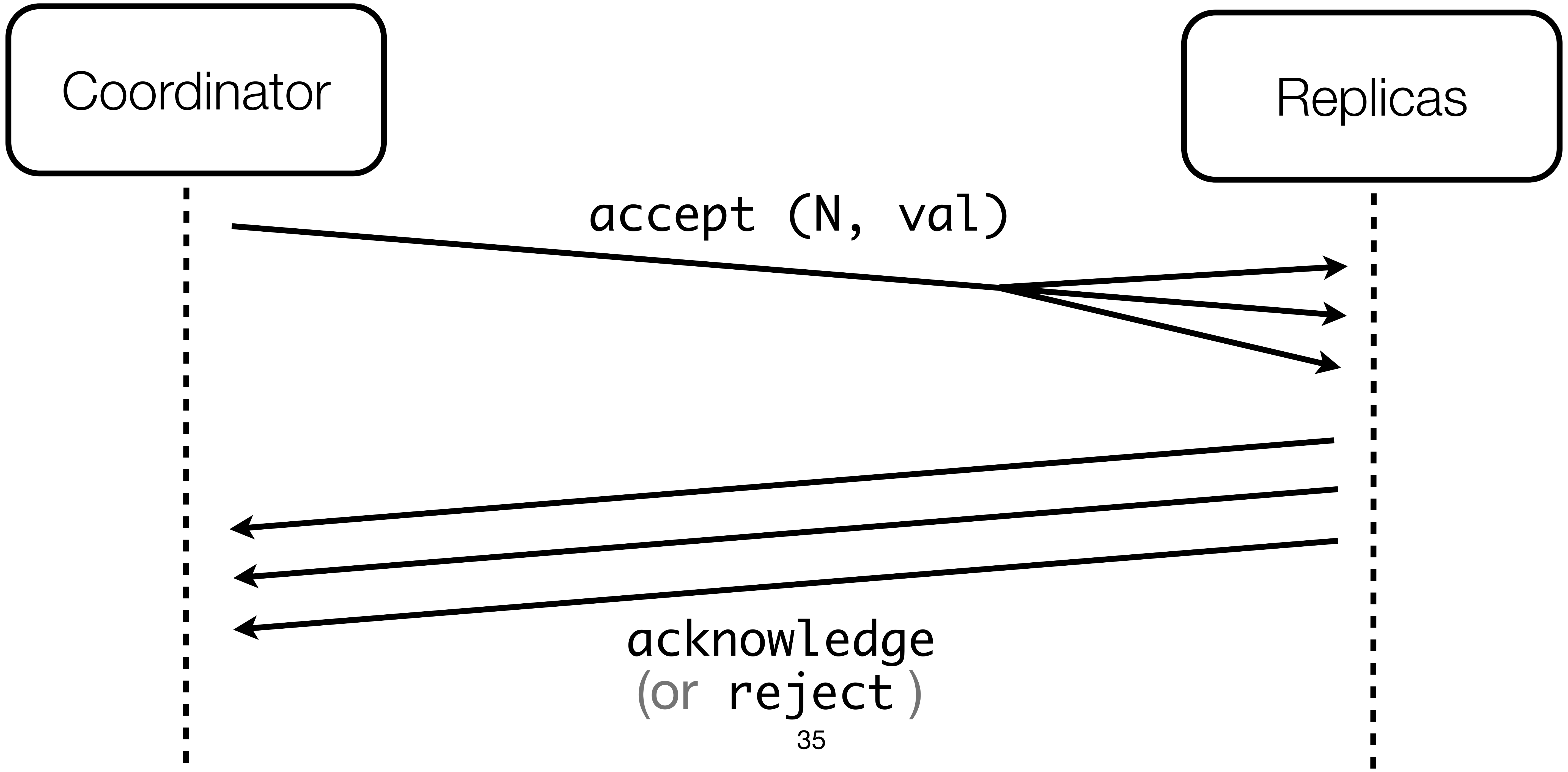
Paxos in action. Step 1: electing a coordinator.

- Flexible election process: any replica can bid to become “coordinator” (= leader) by broadcasting a `propose` message with a *unique* higher sequence number `N`
- On receiving a `propose(N)` message, if `N` is the largest number the replica has seen so far, **the replica promises to ignore all other (older) coordinators with lower sequence numbers**
- The bidding replica becomes coordinator if it receives promises from a **majority** of replicas



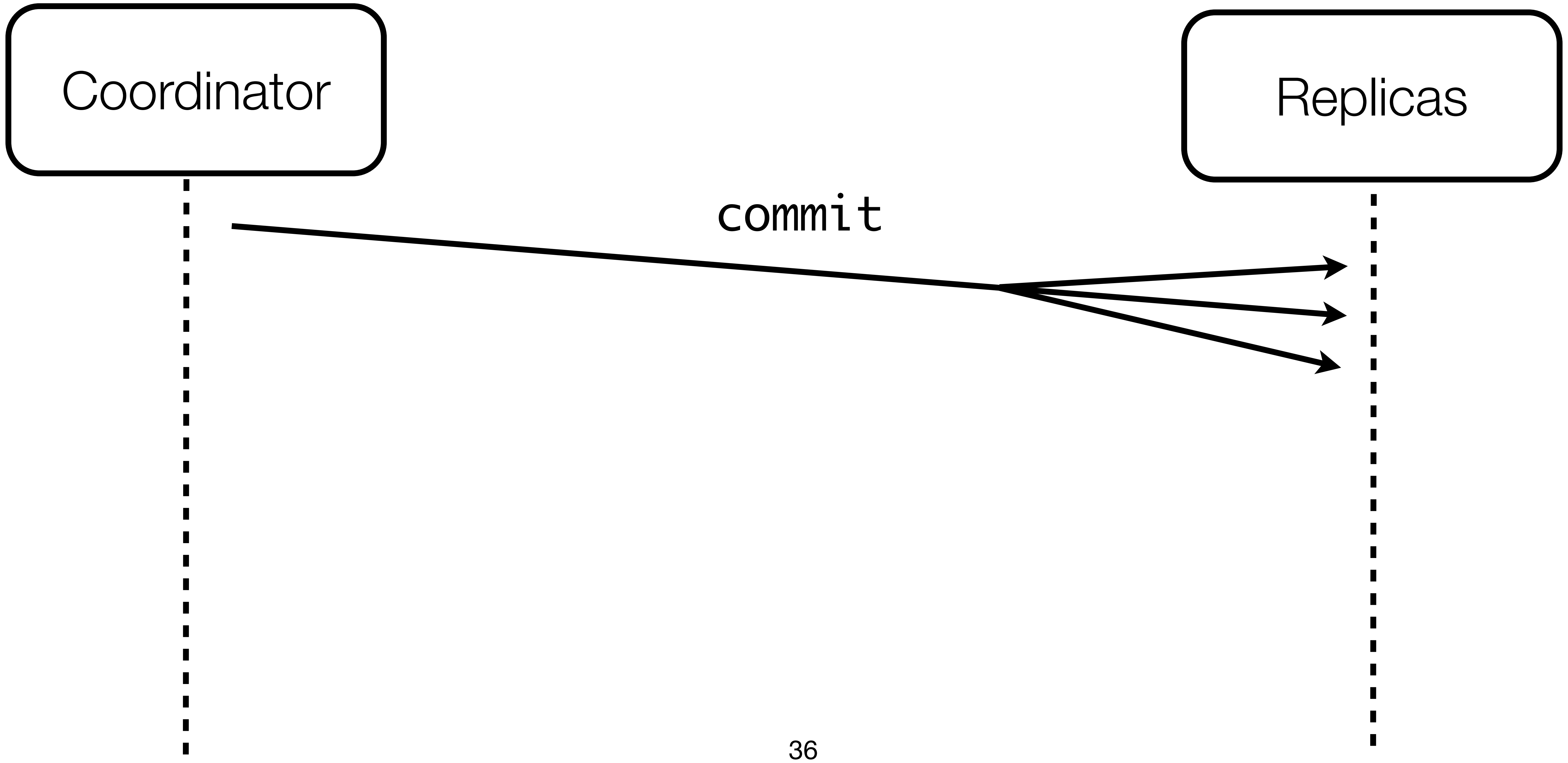
Paxos in action. Step 2: seeking consensus.

- The elected coordinator **proposes a value**. It must pick the most recent value from the set of values it has received; otherwise, the coordinator is free to select its own value.
- The coordinator sends an `accept(N,val)` message, then waits for **replies from a majority** of replicas.
- Replicas reply with an `acknowledge` message *only* if `N` is still the highest sequence number they have heard of so far. Otherwise there exists a higher bidding coordinator, and they will reply with a `reject` message



Paxos in action. Step 3: achieving consensus.

- **If a majority acknowledges**, consensus on the value has been achieved.
- The coordinator then broadcasts a `commit` message to notify replicas of this agreement. Once a commit for the value is received, the replica can safely process the value.
- **If no majority acknowledges**, the coordinator can abandon the proposal and start a new term (with a new unique *higher* sequence number)



Paxos: Keep trying

- A proposal may fail:
  - because a replica may have made a new promise to ignore all sequence numbers less than some value > N
  - because two or more coordinators outbid each other
  - because a coordinator does not receive a quorum of responses: either in step 1 (*propose*) or in step 2 (*accept*)
- Algorithm then has to be **restarted** with a higher bid (sequence number)

Paxos: guarantees

- Paxos ensures agreement and validity (**safety**): if the algorithm terminates, all processes have agreed on the same input value.
- Paxos does **not** guarantee termination (**liveness**): in theory, the algorithm may never terminate.
- Due to the FLP impossibility result, Paxos cannot *guarantee* liveness, but in practice the algorithm will frequently terminate after a short number of rounds.
- The algorithm needs ($2f$+1) processes to survive the simultaneous failure of f processes.
    - E.g. tolerating 2 simultaneous failures requires at least 5 processes
    - In other words: a majority of processes must remain alive

Multi-Paxos

- A single run of the Paxos algorithm decides on a single value *v*
- We often want to decide on a sequence of values v_1, v_2, v_3, … (cfr. operations submitted to a replicated state machine, or messages delivered using total-order broadcast)
- It is possible to "chain" multiple runs of the Paxos algorithm: **Multi-Paxos**
- Optimization: if coordinator doesn't change between runs, we can skip step 1.
    - Try not to let the coordinator change too much: after initial election, the coordinator is the only one to propose values.
    - If the coordinator is suspected to have failed (detected using time-outs), any other process can start bidding and take over as the new coordinator.

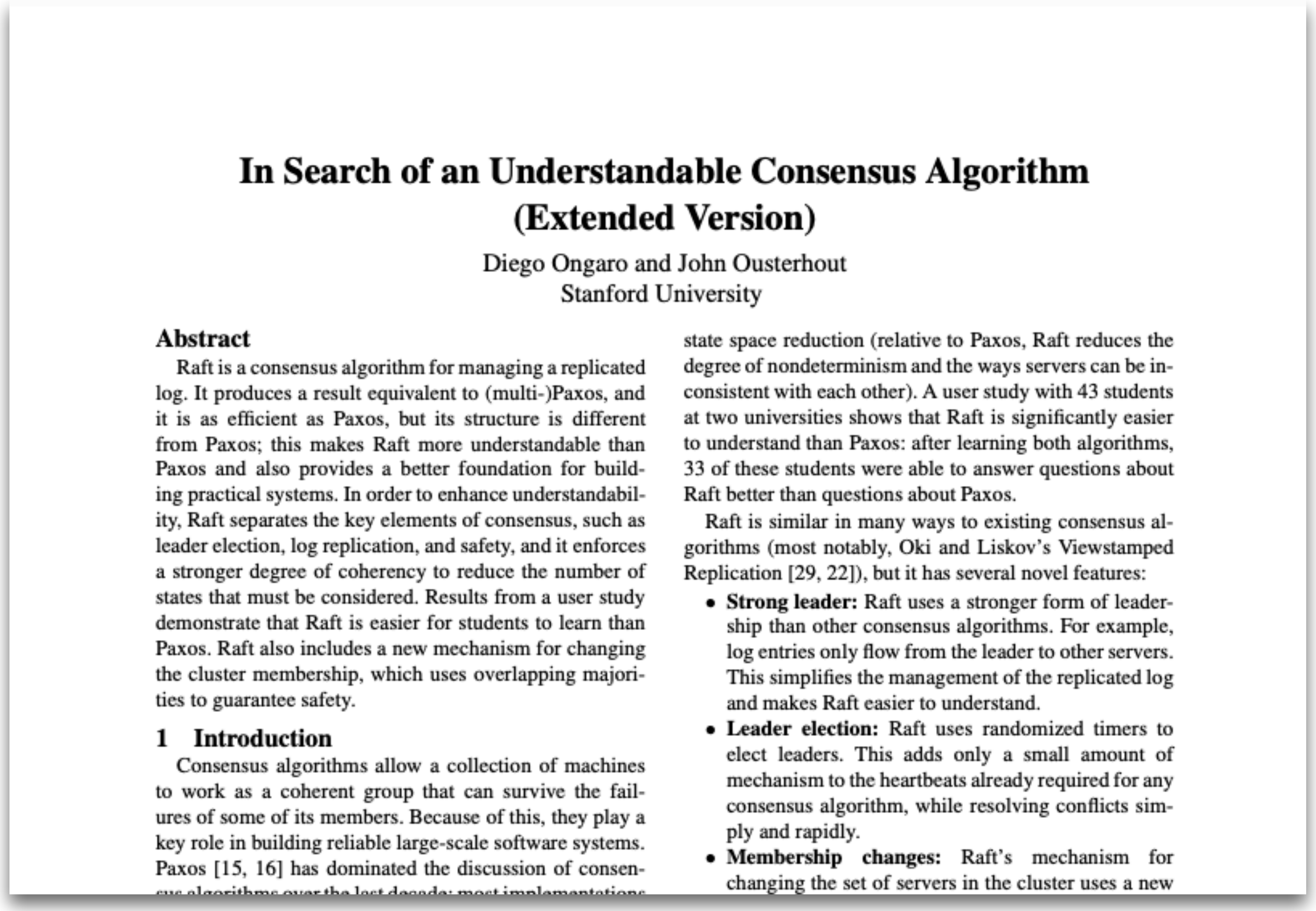
Raft: in search for an understandable consensus algorithm

- Raft was born out of the frustrations in trying to understand and implement the Paxos algorithm

- Raft simplifies the logic, but largely follows the same principles (elect a leader, follow the leader's proposals, re-elect a leader on time-out)

- We will not cover the details of the algorithm. See Prof. Kleppmann's Lecture notes (section 6.2) if interested to learn more.

- A step-by-step visualisation of the algorithm: http://thesecretlivesofdata.com/raft/



In Search of an Understandable Consensus Algorithm (Extended Version)

Diego Ongaro and John Ousterhout
Stanford University

Abstract

Raft is a consensus algorithm for managing a replicated log. It produces a result equivalent to (multi-)Paxos, and it is as efficient as Paxos, but its structure is different from Paxos; this makes Raft more understandable than Paxos and also provides a better foundation for building practical systems. In order to enhance understandability, Raft separates the key elements of consensus, such as leader election, log replication, and safety, and it enforces a stronger degree of coherency to reduce the number of states that must be considered. Results from a user study demonstrate that Raft is easier for students to learn than Paxos. Raft also includes a new mechanism for changing the cluster membership, which uses overlapping majorities to guarantee safety.

1 Introduction

Consensus algorithms allow a collection of machines to work as a coherent group that can survive the failures of some of its members. Because of this, they play a key role in building reliable large-scale software systems. Paxos [15, 16] has dominated the discussion of consensus algorithms over the last decade;

state space reduction (relative to Paxos, Raft reduces the degree of nondeterminism and the ways servers can be inconsistent with each other). A user study with 43 students at two universities shows that Raft is significantly easier to understand than Paxos: after learning both algorithms, 33 of these students were able to answer questions about Raft better than questions about Paxos.

Raft is similar in many ways to existing consensus algorithms (most notably, Oki and Liskov's Viewstamped Replication [29, 22]), but it has several novel features:

- **Strong leader:** Raft uses a stronger form of leadership than other consensus algorithms. For example, log entries only flow from the leader to other servers. This simplifies the management of the replicated log and makes Raft easier to understand.
- **Leader election:** Raft uses randomized timers to elect leaders. This adds only a small amount of mechanism to the heartbeats already required for any consensus algorithm, while resolving conflicts simply and rapidly.
- **Membership changes:** Raft's mechanism for changing the set of servers in the cluster uses a new

(Ongaro and Ousterhout, 2014)
USENIX Annual Technical Conference 2014 paper: https://raft.github.io/raft.pdf

Byzantine Fault Tolerance

Two families of consensus algorithms

- **Crash fault-tolerant (CFT) consensus**: assume processes may fail due to crashes or network failures, but also assume all processes implement the consensus algorithm correctly and strictly follow the rules of the algorithm.
    - Tolerate up to (but not including) 1/2 failing processes
- **Byzantine fault-tolerant (BFT) consensus**: assume processes may fail due to crashes or network failures, but make no additional assumptions. In particular, processes may incorrectly implement the consensus algorithm and may deviate from the algorithm in arbitrary ways.
    - Tolerate up to (but not including) 1/3 failing processes
- All algorithms discussed so far (Paxos, Raft, Viewstamped Replication) are CFT algorithms.

Byzantine Fault Tolerance

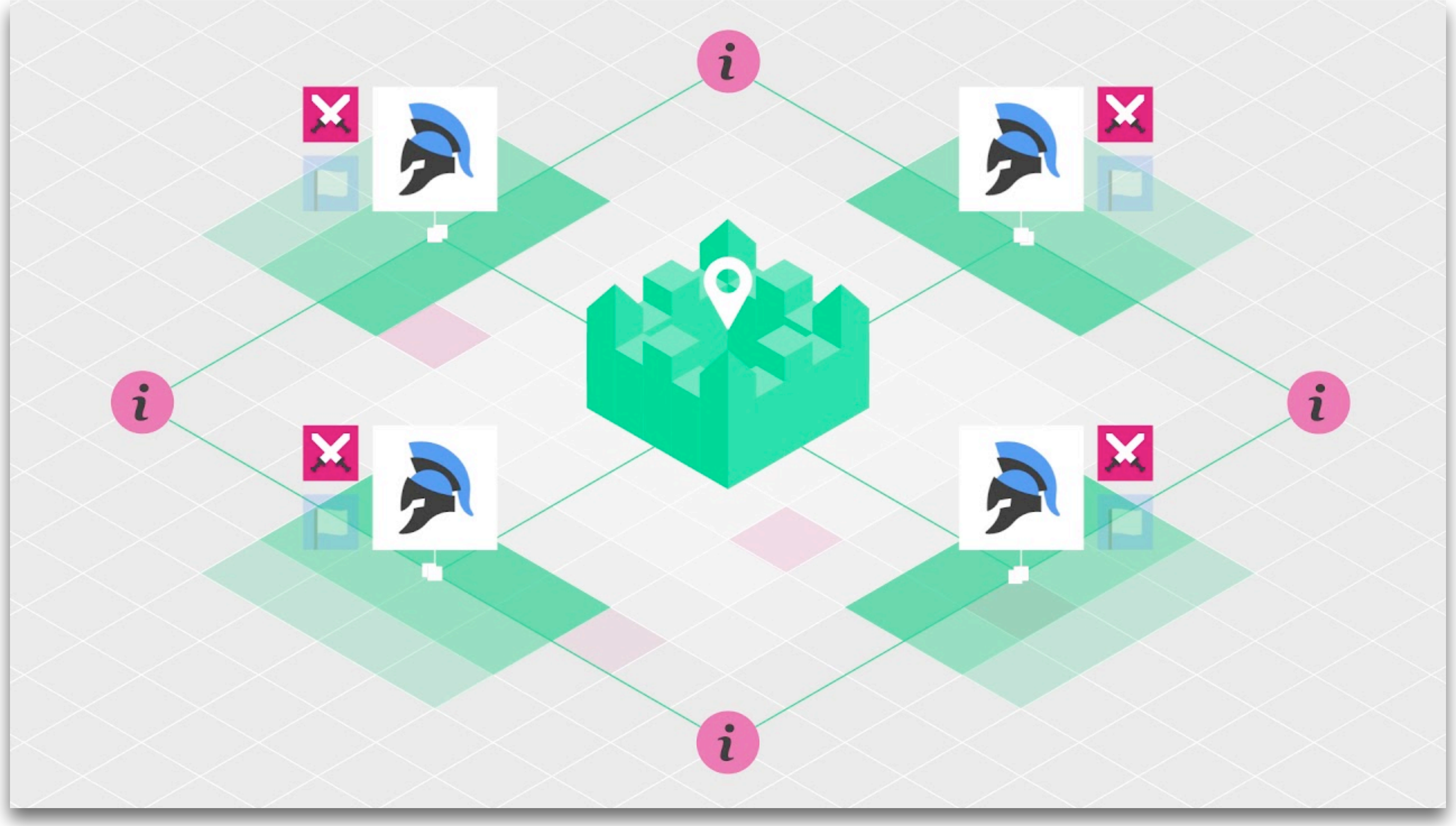
- In a distributed system, a “**byzantine failure**” is used to describe a process that may **fail** in **totally arbitrary** ways, including:
  - Failing to respond to messages
  - Returning incorrect results from messages
  - Returning deliberately misleading results from messages
  - Returning a different result for the same request to different processes (!)

Byzantine Fault Tolerance

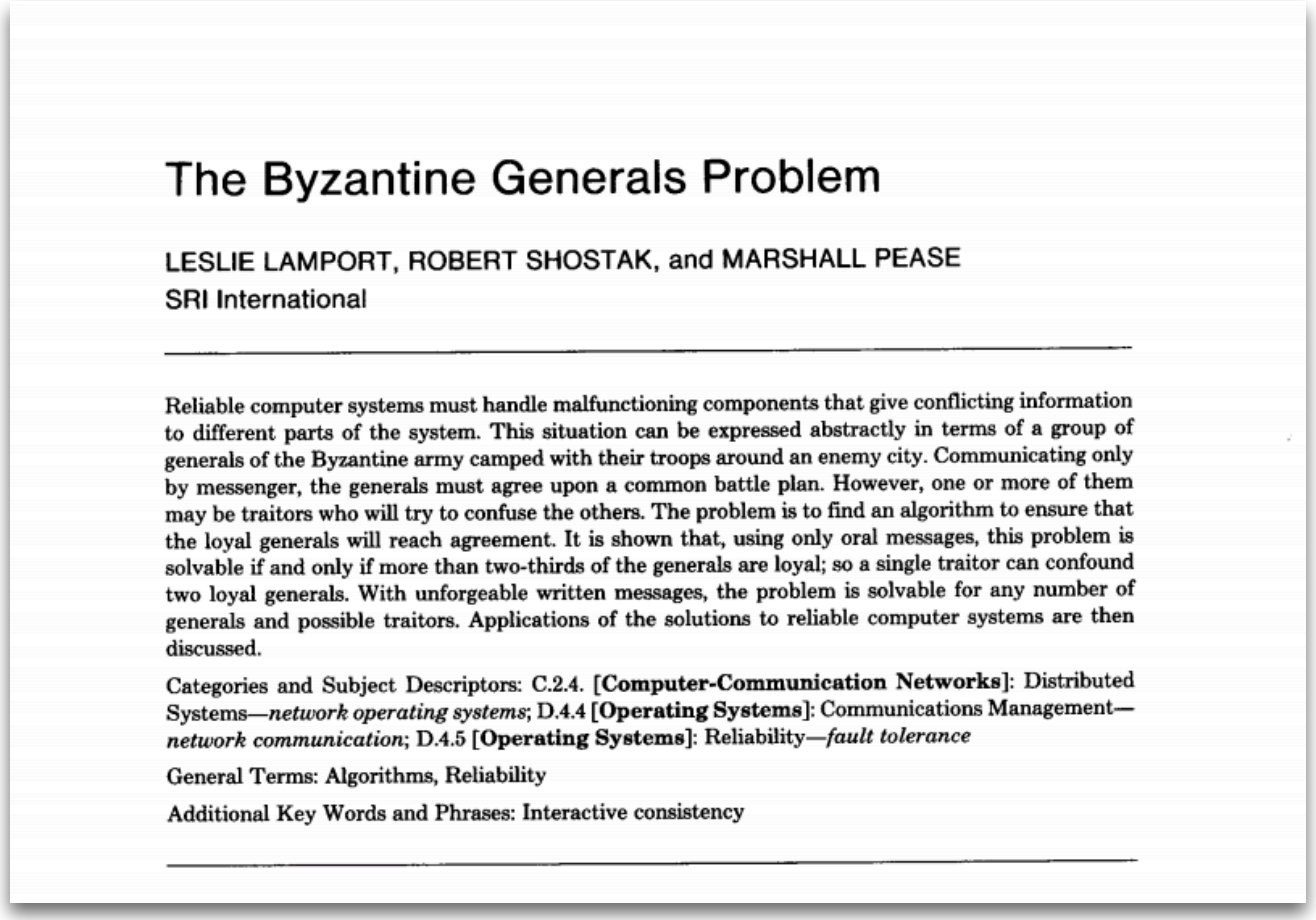
- In a distributed system, a “**byzantine failure**” is used to describe a process that may **fail** in **totally arbitrary** ways
- Often a good assumption to make in an **adversarial** context where processes may be taken over by attackers that want to deliberately subvert the system (e.g. Blockchains)
  - Model *attacks* as byzantine failures
- Also a good assumption to make in **real-world** deployments where both hardware and software mail fail in unexpected ways (e.g. corrupted files or network packets, faulty device drivers, partially updated software, …)
  - Model *bugs* as byzantine failures

The Byzantine Generals Problem

Commander and his lieutenants need to agree to <u>attack</u> or <u>retreat</u>.
But the commander and/or the lieutenants may be traitors
that deliberately spread a false decision to their peers.



(image credit: <u>binance.com</u>)



The Byzantine Generals Problem

LESLIE LAMPORT, ROBERT SHOSTAK, and MARSHALL PEASE
SRI International

Reliable computer systems must handle malfunctioning components that give conflicting information to different parts of the system. This situation can be expressed abstractly in terms of a group of generals of the Byzantine army camped with their troops around an enemy city. Communicating only by messenger, the generals must agree upon a common battle plan. However, one or more of them may be traitors who will try to confuse the others. The problem is to find an algorithm to ensure that the loyal generals will reach agreement. It is shown that, using only oral messages, this problem is solvable if and only if more than two-thirds of the generals are loyal; so a single traitor can confound two loyal generals. With unforgeable written messages, the problem is solvable for any number of generals and possible traitors. Applications of the solutions to reliable computer systems are then discussed.

Categories and Subject Descriptors: C.2.4. [**Computer-Communication Networks**]: Distributed Systems—*network operating systems*; D.4.4 [**Operating Systems**]: Communications Management—*network communication*; D.4.5 [**Operating Systems**]: Reliability—*fault tolerance*

General Terms: Algorithms, Reliability

Additional Key Words and Phrases: Interactive consistency

Lamport, Shostak and Pease, The Byzantine Generals Problem
ACM Transactions on Programming Languages and Systems, 1982

KU LEUVEN DistriNet

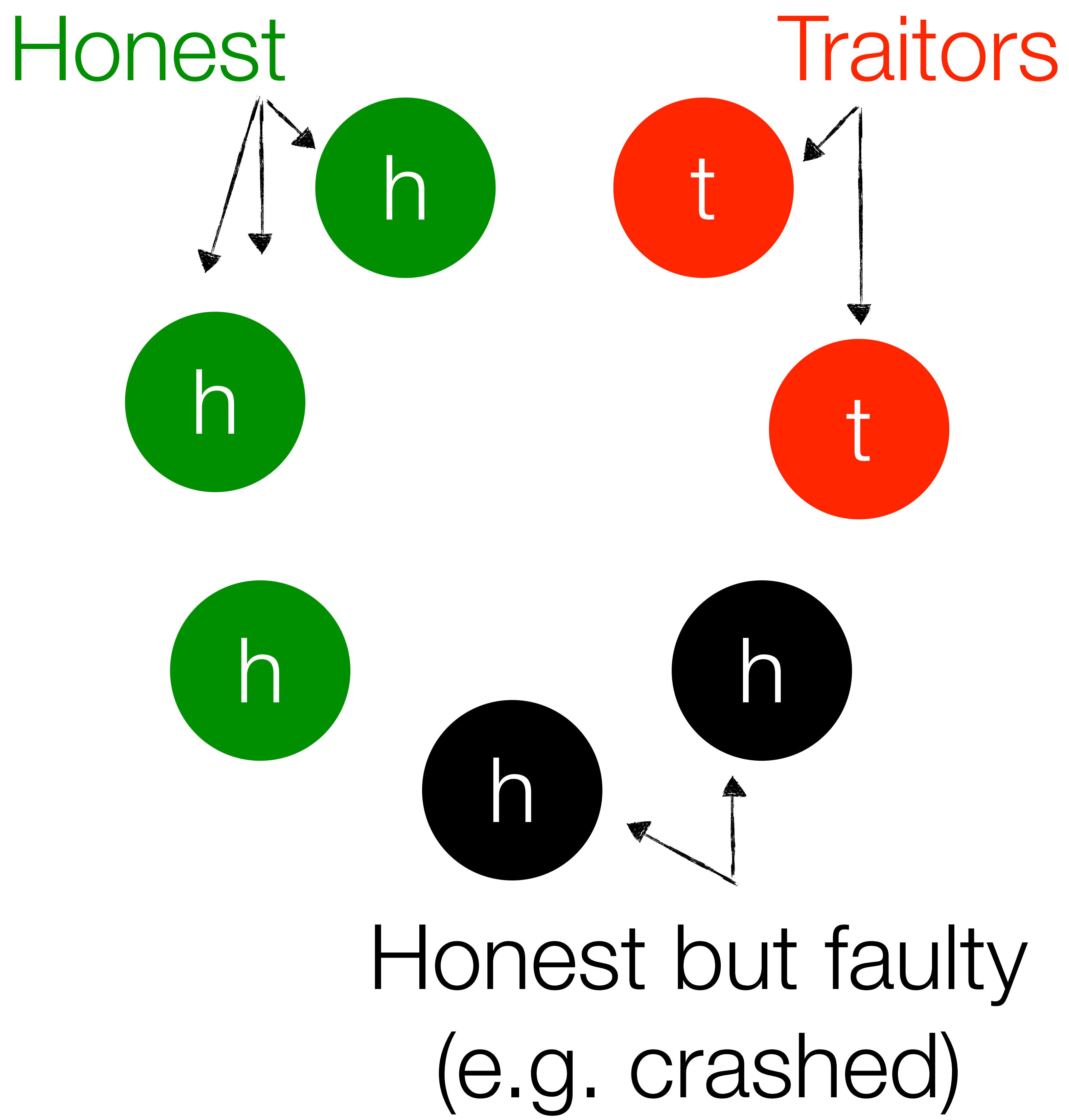
The Byzantine Generals Problem: fundamental result

- Need at least $N = 3f + 1$ processes to tolerate f “traitors” or faulty processes

- In other words, need at least a strictly 2/3 majority of honest (correct) processes

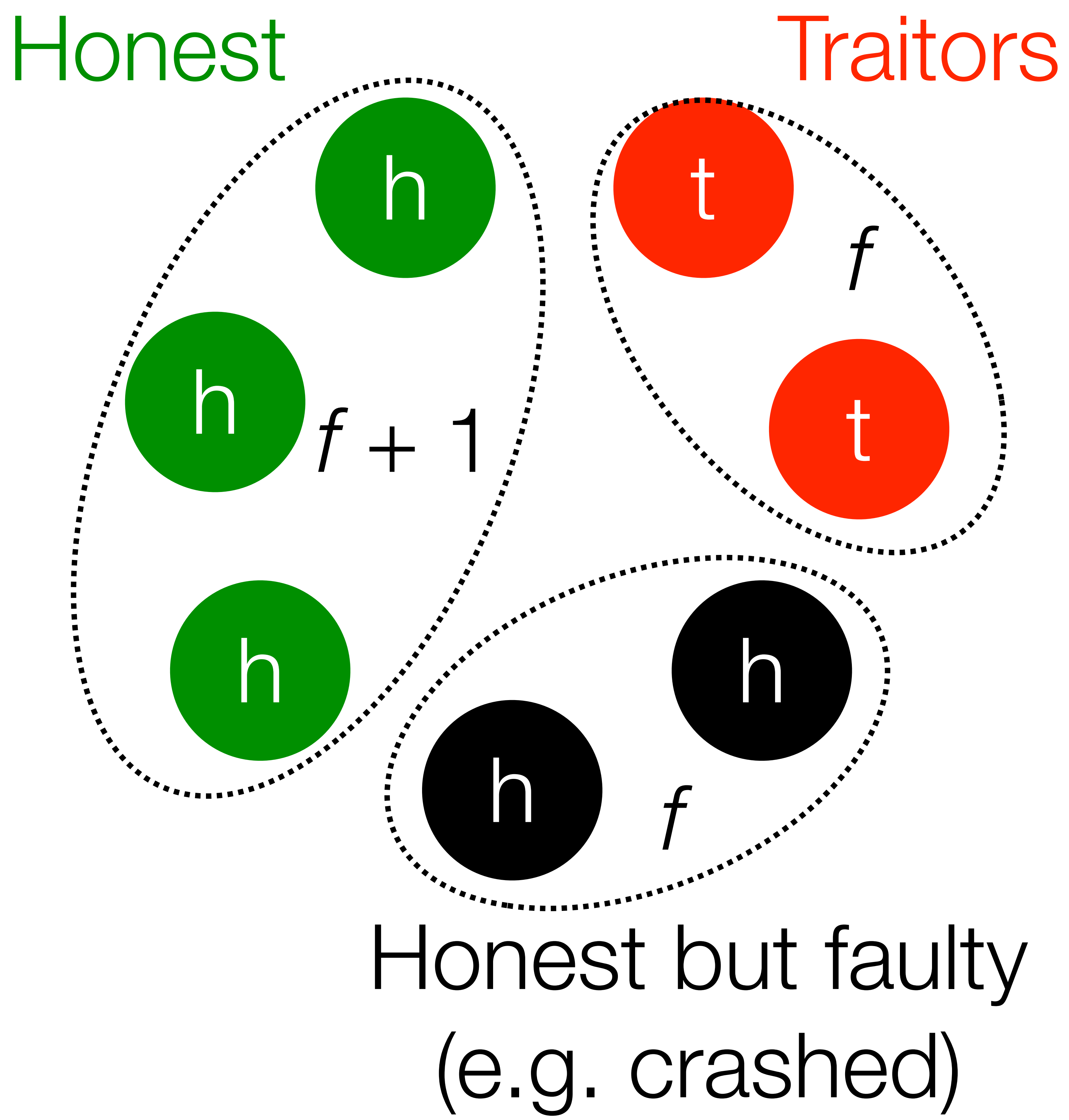
- Intuition: assume f processes are unresponsive (but honest), e.g. due to network or device failure. Of the remaining $N - f$ processes, another f could be traitors. To ensure enough responses from honest participants, the available honest participants need to outnumber the traitors, i.e. $N - 2f > f$, therefore $N > 3f$

- So $N = 3f + 1$ is the optimal number of processes needed to tolerate f traitors and/or faulty processes



The Byzantine Generals Problem: fundamental result

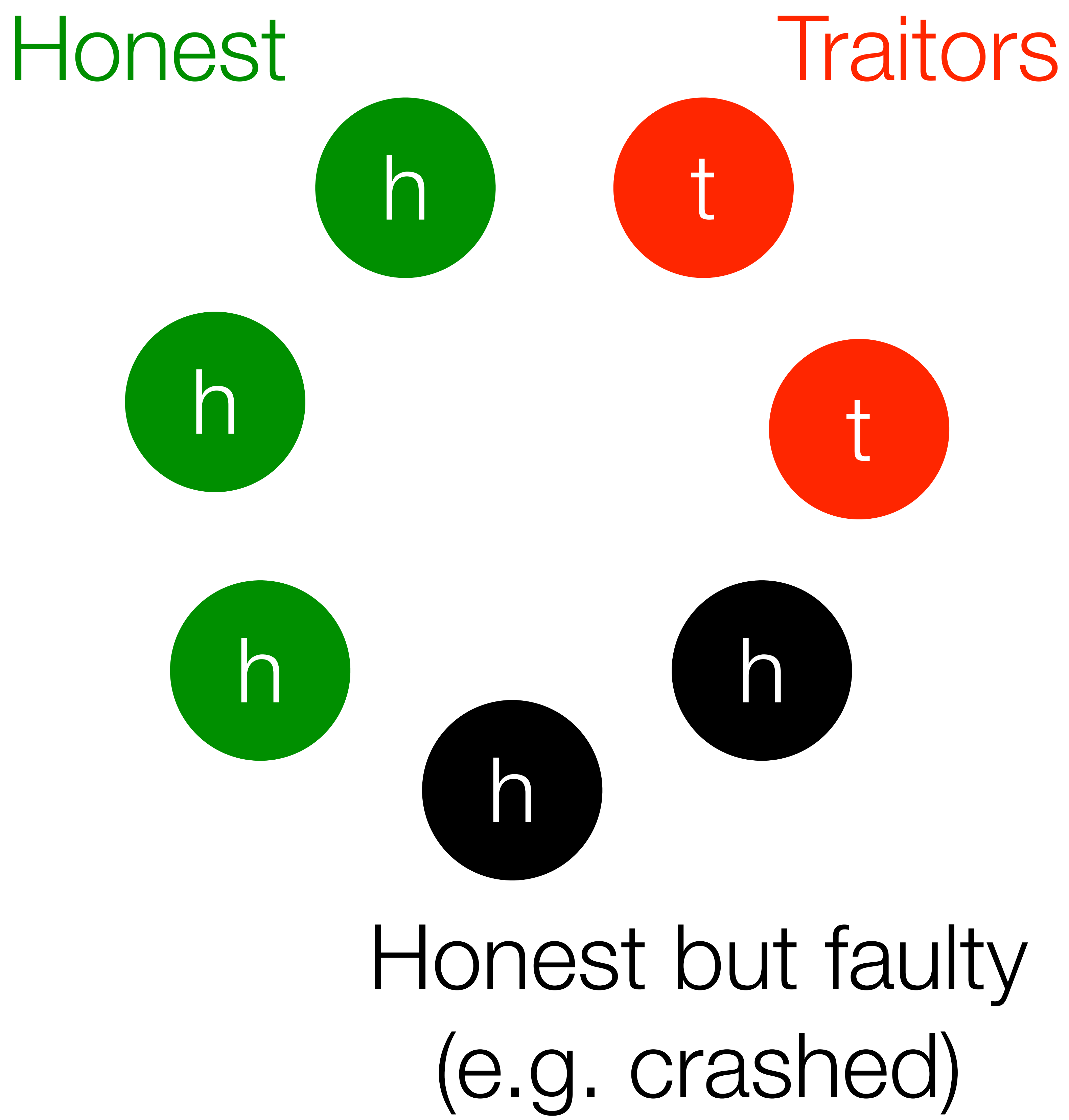
- Need at least $N = 3f + 1$ processes to tolerate f “traitors” or faulty processes
- In other words, need at least a strictly 2/3 majority of honest (correct) processes
- Intuition: assume f processes are unresponsive (but honest), e.g. due to network or device failure. Of the remaining $N - f$ processes, another f could be traitors. To ensure enough responses from honest participants, the available honest participants need to outnumber the traitors, i.e. $N - 2f > f$, therefore $N > 3f$
- So $N = 3f + 1$ is the optimal number of processes needed to tolerate f traitors and/or faulty processes



The Byzantine Generals Problem: quorum

- $N = 3f + 1$ or f = floor((N-1)/3)

N	Honest quorum $f+1$	Faulty or traitors f
1	-	0
2	-	0
3	-	0
4	**2**	**1**
5	2	1
6	2	1
7	**3**	**2**
8	3	2



Byzantine fault-tolerant (BFT) consensus algorithms

- **BFT consensus algorithms exist**, e.g. the Practical Byzantine Fault Tolerance (PBFT) algorithm by Castro and Liskov (1999)
- They are **more complex** than CFT algorithms. They use digital signatures and cryptographic hash functions to ensure that communicated decisions are unforgeable and irrefutable, to avoid spreading false information.
- We will not cover these algorithms here.
- Applications of BFT consensus?
  - **Blockchain**: Blockchain networks require *byzantine* consensus in an *open* and *adversarial* environment to agree on an order of transactions. See later lecture.

Consensus: Summary

- **Consensus**: how to get a group of processes to all agree on the same value, even when networks are unreliable and processes may be faulty?
- **Replicated State Machines:** can be used to build highly reliable systems. A consensus algorithm is needed to **agree on the order of updates** to the state machine.
- **System Models:** To reason about consensus algorithms, we need a model of how a distributed system "behaves". This makes the assumptions that these algorithms rely on more explicit.
- **Defining Consensus:** safety and liveness properties: agreement, validity and termination.
- **Implementing Consensus** (Consensus algorithms): elect a leader, then let the leader broadcast a proposal. The difficulty is in ensuring there is only ever a single leader.
    - **Paxos**: a widely influential crash-fault tolerant (CFT) consensus algorithm
- **Byzantine fault-tolerant (BFT) consensus:** processes must all agree on the same value, even when networks are unreliable and processes may fail in totally arbitrary ways or are actively malicious.