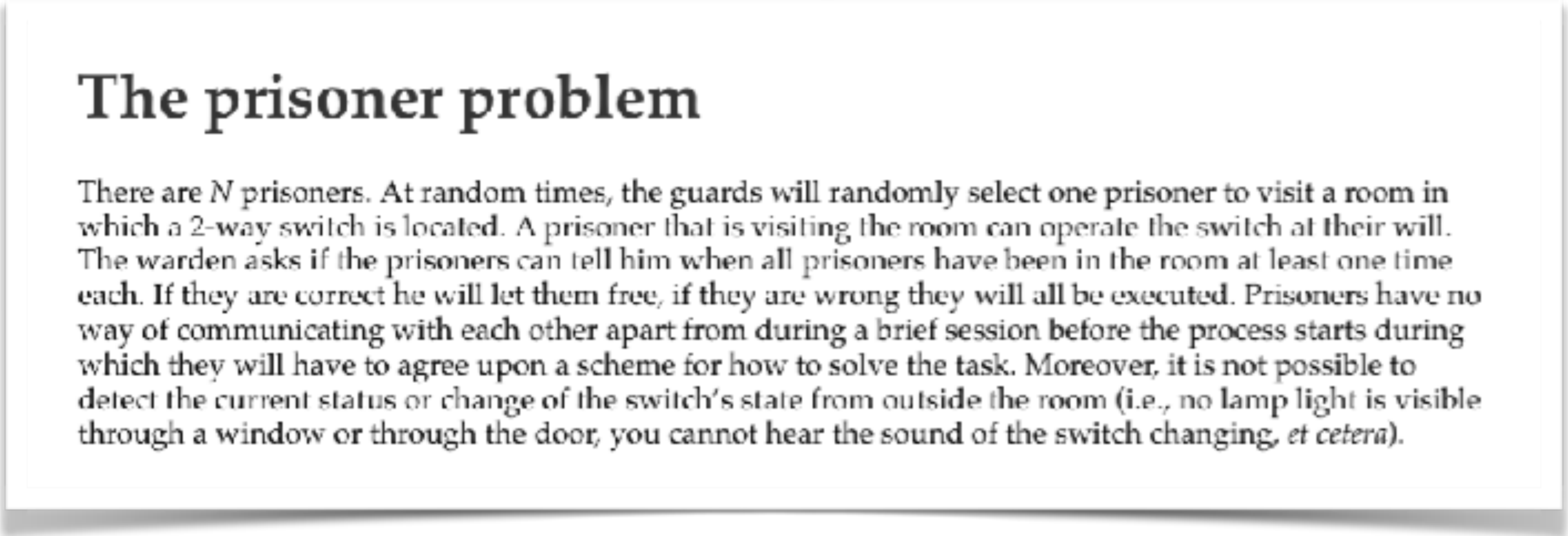
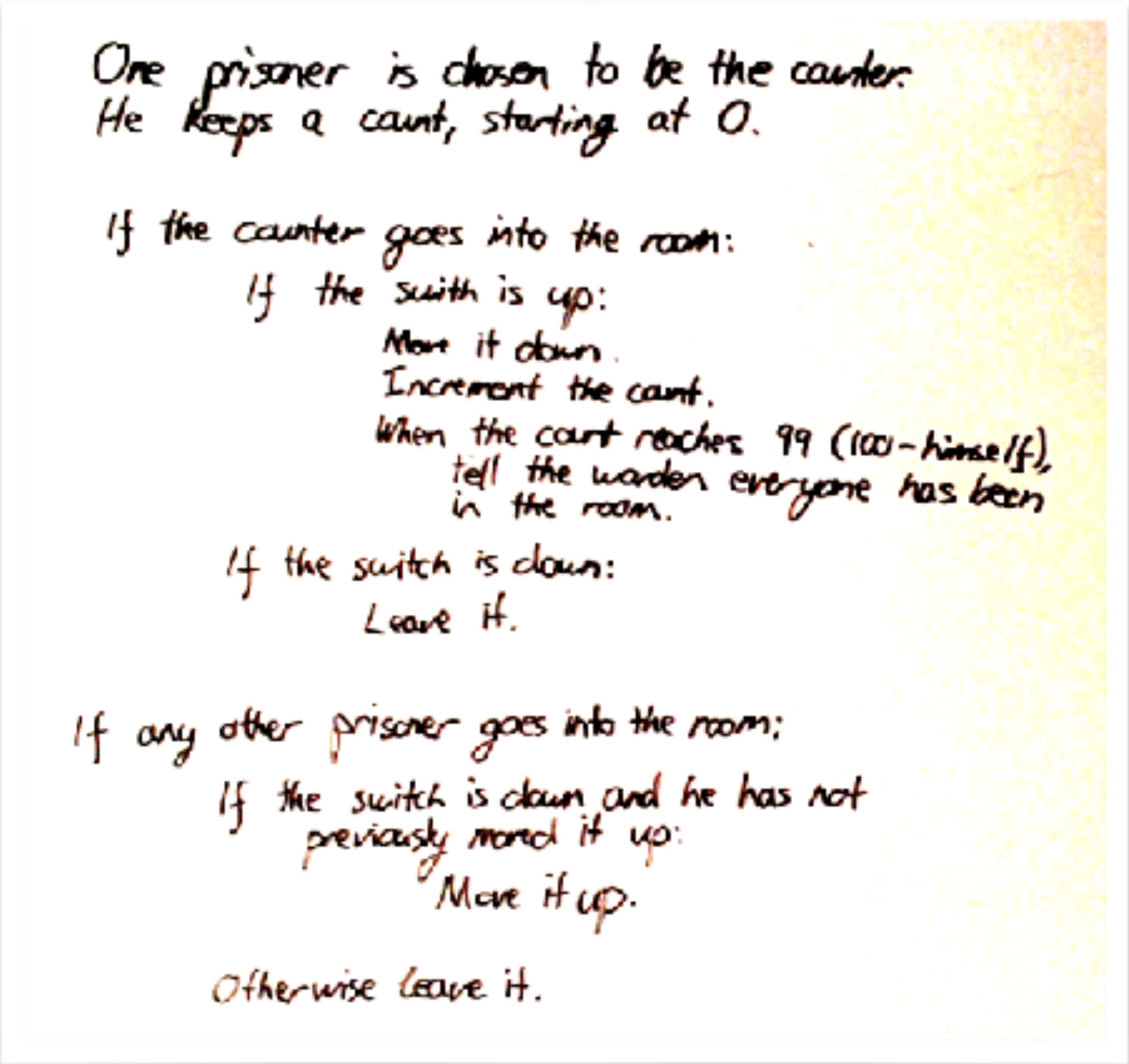
Cloud Computing
#6 - Distributed computing topics & cloud

Homework #1



The prisoner problem

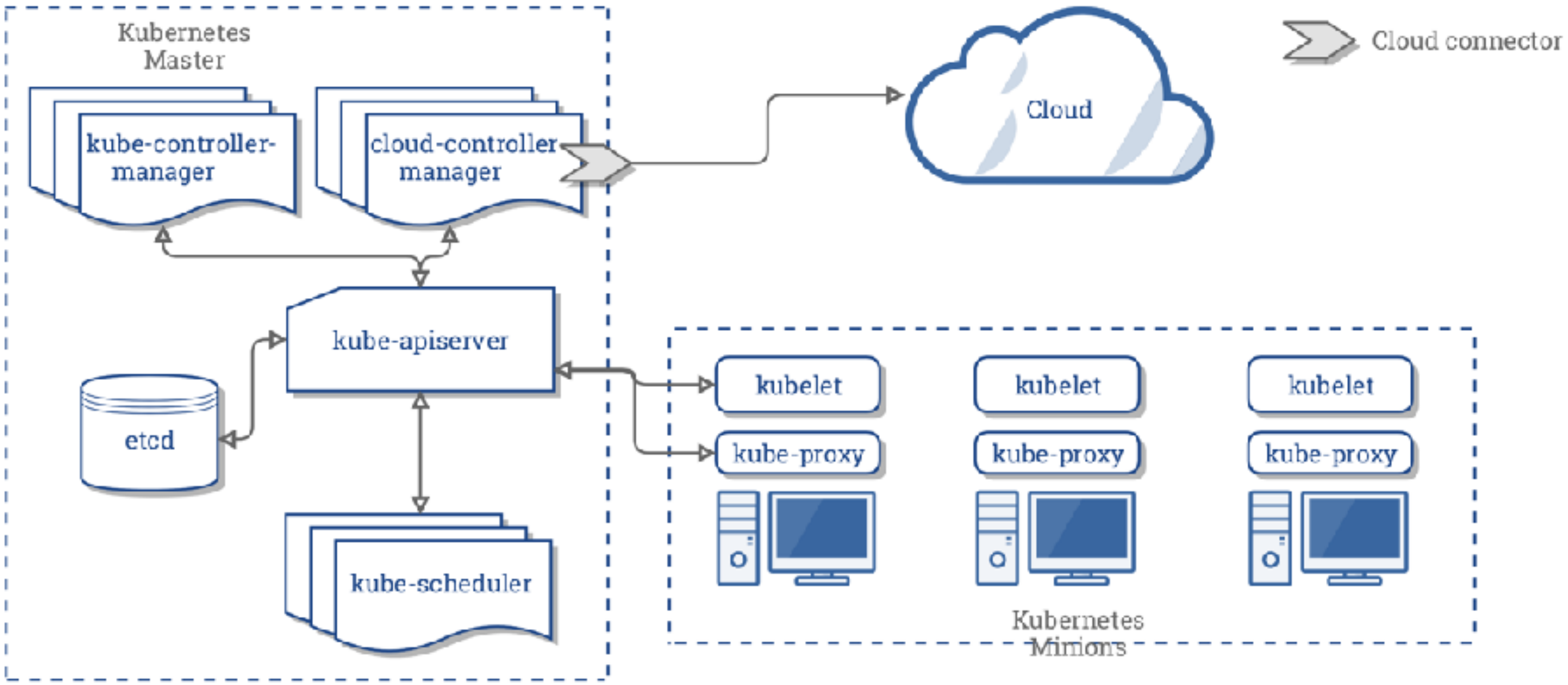
There are *N* prisoners. At random times, the guards will randomly select one prisoner to visit a room in which a 2-way switch is located. A prisoner that is visiting the room can operate the switch at their will. The warden asks if the prisoners can tell him when all prisoners have been in the room at least one time each. If they are correct he will let them free, if they are wrong they will all be executed. Prisoners have no way of communicating with each other apart from during a brief session before the process starts during which they will have to agree upon a scheme for how to solve the task. Moreover, it is not possible to detect the current status or change of the switch's state from outside the room (i.e., no lamp light is visible through a window or through the door, you cannot hear the sound of the switch changing, *et cetera*).



One prisoner is chosen to be the counter.
He keeps a count, starting at 0.

```
If the counter goes into the room:
    If the swith is up:
        Move it down.
        Increment the count.
        When the count reaches 99 (100-himself),
            tell the warden everyone has been
            in the room.
    If the switch is down:
        Leave it.

If any other prisoner goes into the room:
    If the switch is down and he has not
      previously moved it up:
        Move it up.
    Otherwise leave it.
```

Last week: k8 architecture



Kubernetes Master

kube-controller-manager

cloud-controller-manager

Cloud

Cloud connector

kube-apiserver

etcd

kube-scheduler

kubelet

kube-proxy

kubelet

kube-proxy

kubelet

kube-proxy

Kubernetes Minions

Distributed Computing

A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable.

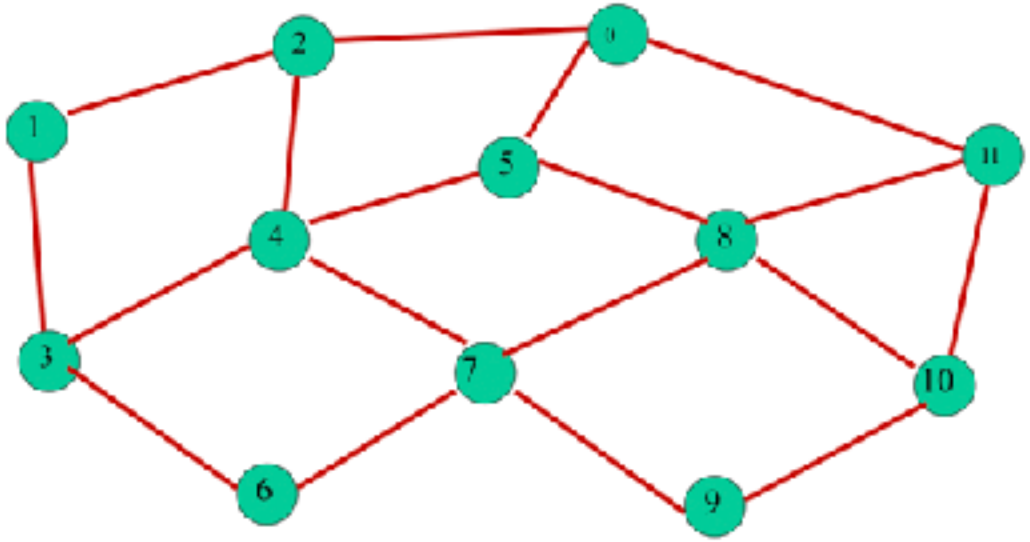
Leslie Lamport
email communication
1987

Distributed computations are concurrent programs
in which processes communicate by message passing.

Gregory R. Andrews
“Paradigms for Process Interaction in Distributed Programs”
ACM Computing Surveys 23(1), 1991

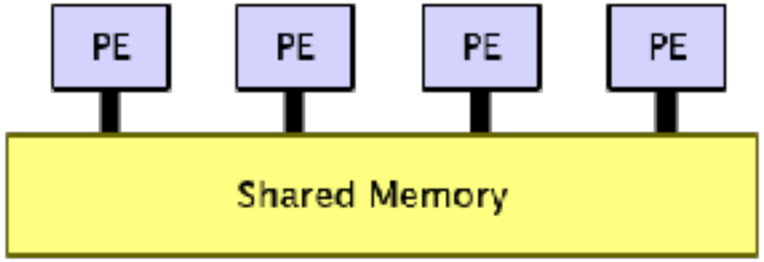
A Distributed System

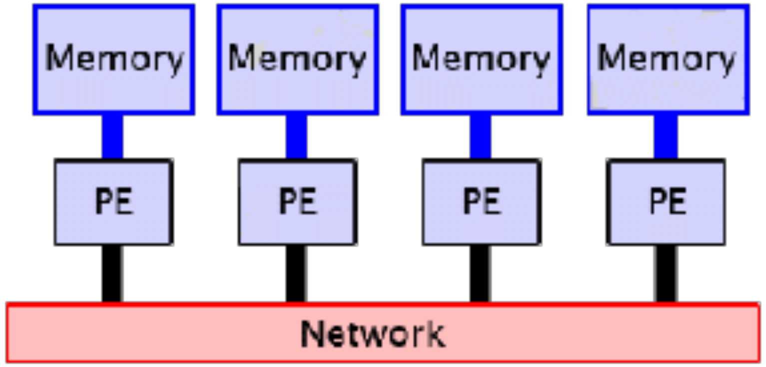
- Distributed system is composed of n processes
- A process executes a sequence of events
    - Local computation
    - Sending a message m
    - Receiving a message m



- A distributed algorithm is an algorithm that runs on more than one process

Inter-Process Communication Models





Distributed Computing

Things being looked at

- the algorithm(s) of the processes
- the messages
- order, causality
- whether delivery is reliable
- whether processes crash (and how)
- whether processes are “nice”

Things that usually aren't

- the nature of the interconnect
- time / speed
- location of processes
- data formats

Examples of distributed algorithms

- Synchronizers (time & order)
- Resource allocation, mutual exclusion
- Data Consistency
- Failure detectors
- Consensus and agreement
    - Leader election

Synchronous vs Asynchronous

Synchronous systems:

known upper bounds on time for computation and message delivery
or access to global clock
or execution in synchronized rounds (not realistic, unfortunately)

Asynchronous systems:

no upper bounds on time for computation and message delivery

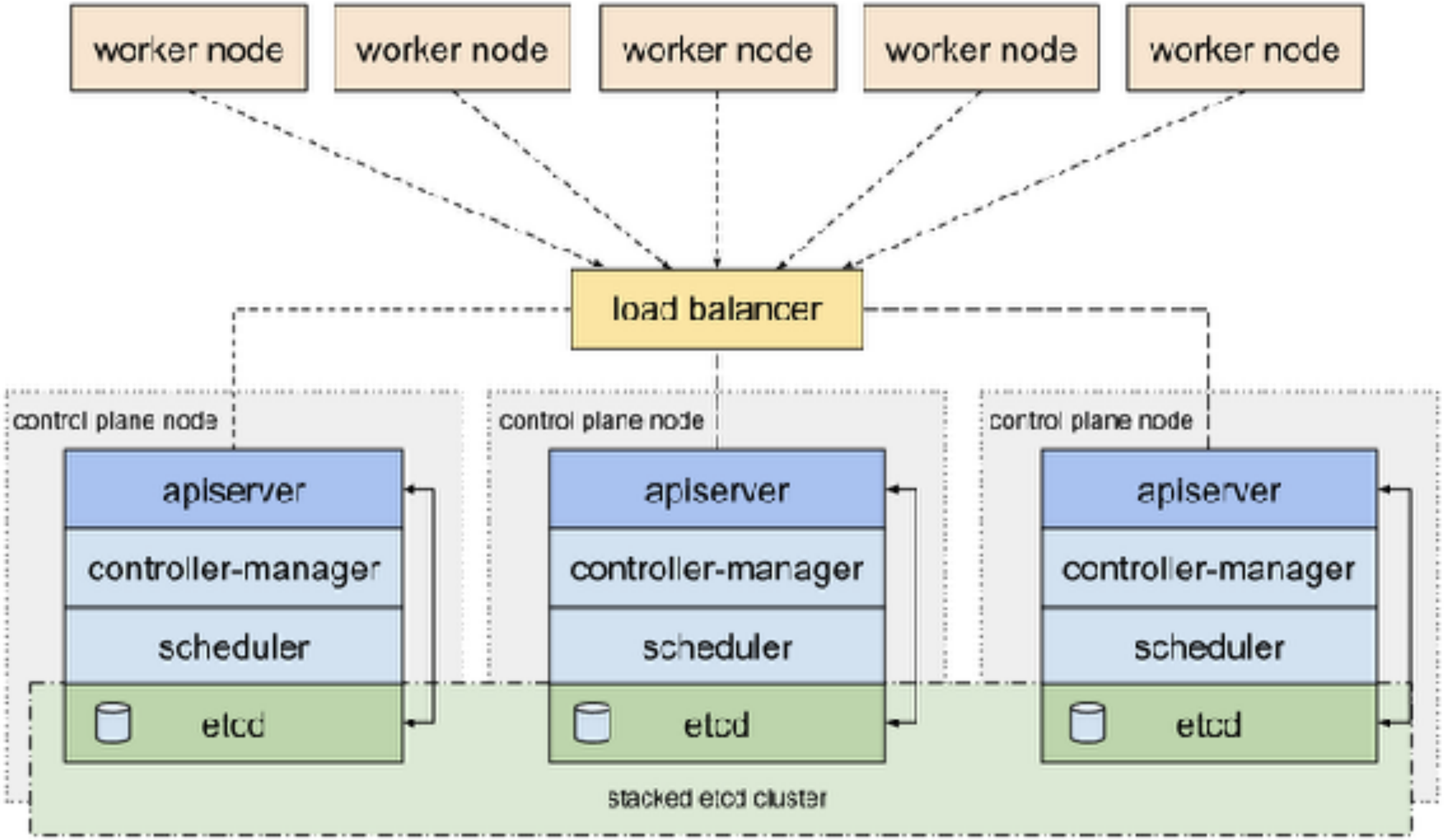
Partially synchronous systems:

anything in between, e.g.

- unknown upper bounds on time for computation and message delivery
- almost-synchronized clocks
- bounded-drift local clocks
- approximate bounds (on execution/message delivery time)
- bound on message delay, bound on relative process speeds
- bound on the delay ratio between fastest and slowest message at any time

etcd

- Distributed key-value store
- **The** core component in Kubernetes
  - Single source of truth
  - Stores state of all API objects and all events that occur
- Based on the Raft consensus protocol
- Leader selection
  - only one controller-manager, scheduler, apiserver active

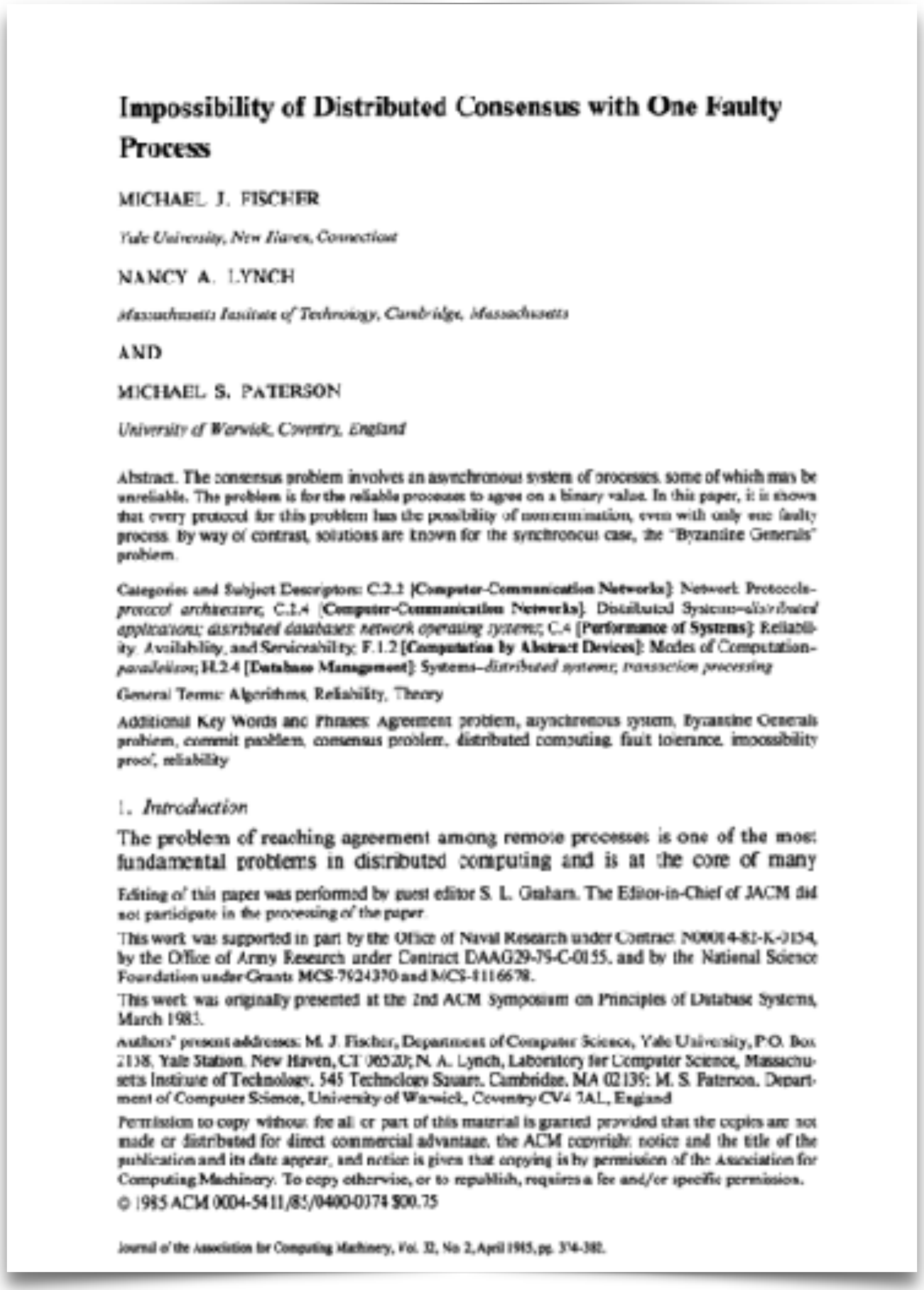


Why is this even hard?

source

It is not only hard, it is in general impossible

Enter: the FLP theorem



Impossibility of Distributed Consensus with One Faulty Process

MICHAEL J. FISCHER

Yale University, New Haven, Connecticut

NANCY A. LYNCH

Massachusetts Institute of Technology, Cambridge, Massachusetts

AND

MICHAEL S. PATERSON

University of Warwick, Coventry, England

Abstract. The consensus problem involves an asynchronous system of processes, some of which may be unreliable. The problem is for the reliable processes to agree on a binary value. In this paper, it is shown that every protocol for this problem has the possibility of nontermination, even with only one faulty process. By way of contrast, solutions are known for the synchronous case, the "Byzantine Generals" problem.

Categories and Subject Descriptors: C.2.2 [**Computer-Communication Networks**]: Network Protocols–*protocol architecture*; C.2.4 [**Computer-Communication Networks**]: Distributed Systems–*distributed applications; distributed databases; network operating systems*; C.4 [**Performance of Systems**]: Reliability, Availability, and Serviceability; F.1.2 [**Computation by Abstract Devices**]: Modes of Computation–*parallelism*; H.2.4 [**Database Management**]: Systems–*distributed systems; transaction processing*

General Terms: Algorithms, Reliability, Theory

Additional Key Words and Phrases: Agreement problem, asynchronous system, Byzantine Generals problem, commit problem, consensus problem, distributed computing, fault tolerance, impossibility proof, reliability

1. *Introduction*

The problem of reaching agreement among remote processes is one of the most fundamental problems in distributed computing and is at the core of many

Editing of this paper was performed by guest editor S. L. Graham. The Editor-in-Chief of JACM did not participate in the processing of the paper.

This work was supported in part by the Office of Naval Research under Contract N00014-82-K-0154, by the Office of Army Research under Contract DAAG29-79-C-0155, and by the National Science Foundation under Grants MCS-7924370 and MCS-8116678.

This work was originally presented at the 2nd ACM Symposium on Principles of Database Systems, March 1983.

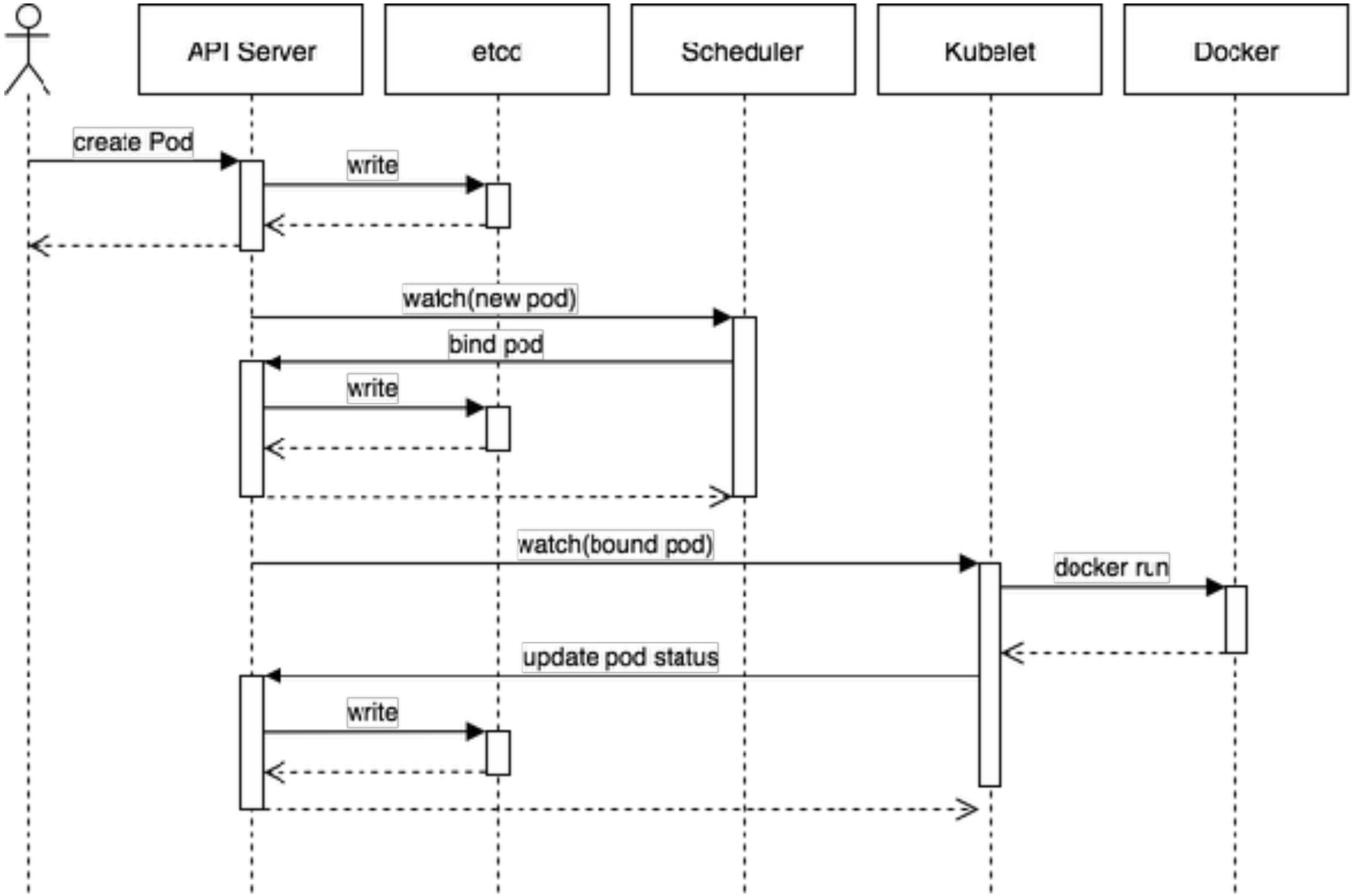
Authors' present addresses: M. J. Fischer, Department of Computer Science, Yale University, P.O. Box 2158, Yale Station, New Haven, CT 06520; N. A. Lynch, Laboratory for Computer Science, Massachusetts Institute of Technology, 545 Technology Square, Cambridge, MA 02139; M. S. Paterson, Department of Computer Science, University of Warwick, Coventry CV4 7AL, England.

Journal of the Association for Computing Machinery, Vol. 32, No. 2, April 1985, pp. 374-382.

etcd

- Pod activation example:

- All persistent data is stored in etcd
- Distributed data base



Why is this even hard?

source

More bad news: The CAP Theorem

Consistency

All clients shall see the same data at any given time.

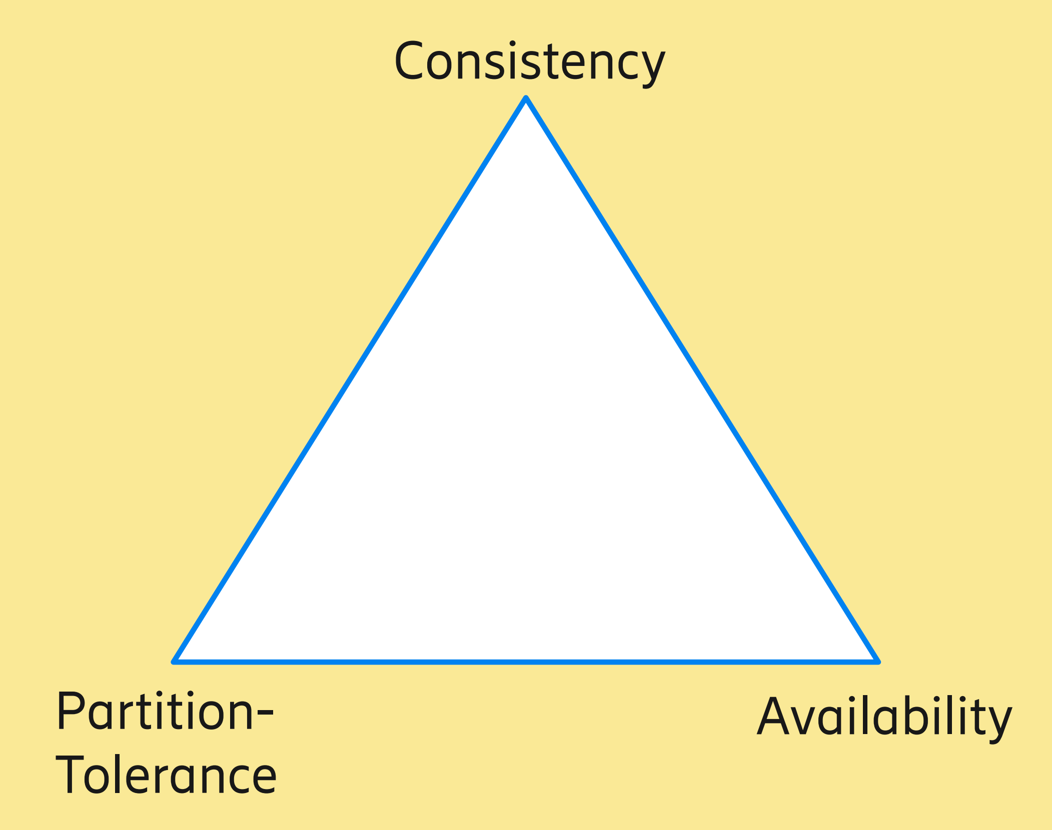
A read must return the latest written value by any client.

Availability

The system allows read and write operations all the time, and these operations return within a reasonable time.

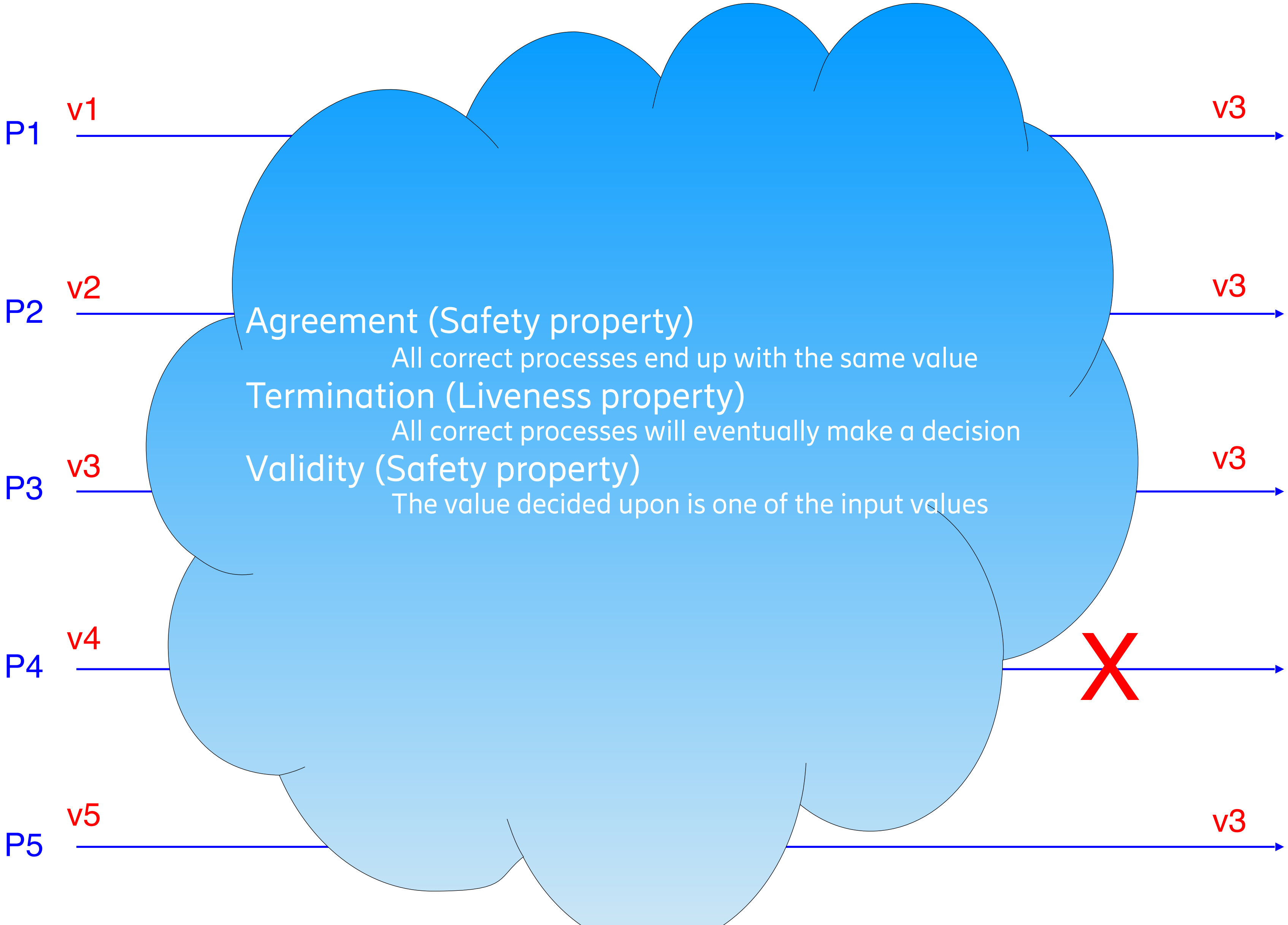
Partition-Tolerance

The system continues to work normally even if network partitions occur.



The CAP theorem was proposed by Eric Brewer from UC Berkeley, and was proved theoretically by Gilbert and Lynch from NUS and MIT.

The Consensus Problem



Agreement (Safety property)

All correct processes end up with the same value

Termination (Liveness property)

All correct processes will eventually make a decision

Validity (Safety property)

The value decided upon is one of the input values

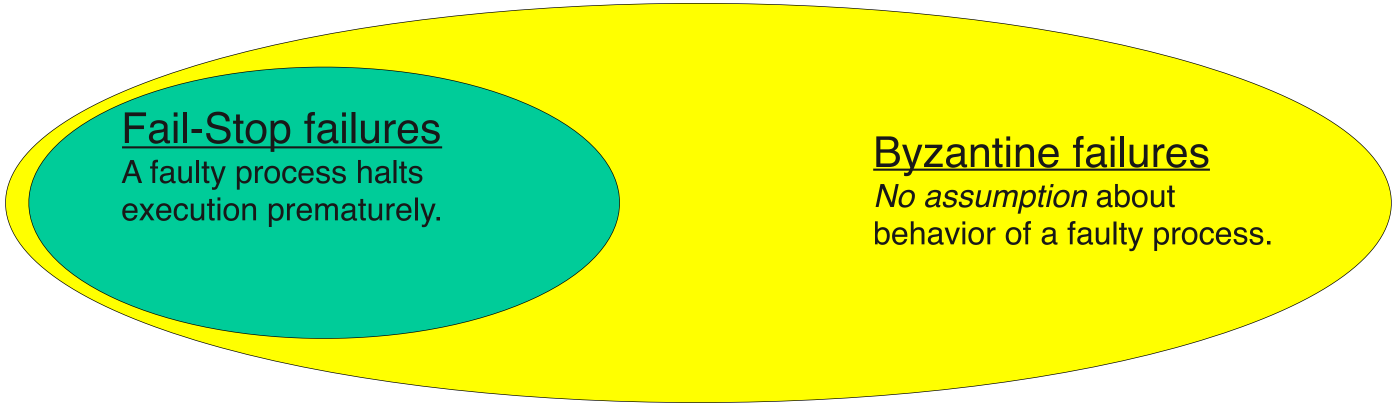
A Naïve Protocol

- 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?

A Naïve Protocol

- In an asynchronous environment, we can’t detect failures reliably
- A faulty process stops sending messages but a “slow” message might confuse us
- When the vote is nearly a tie, this confusing situation really matters

Failure models



Fail-Stop failures

A faulty process halts execution prematurely.

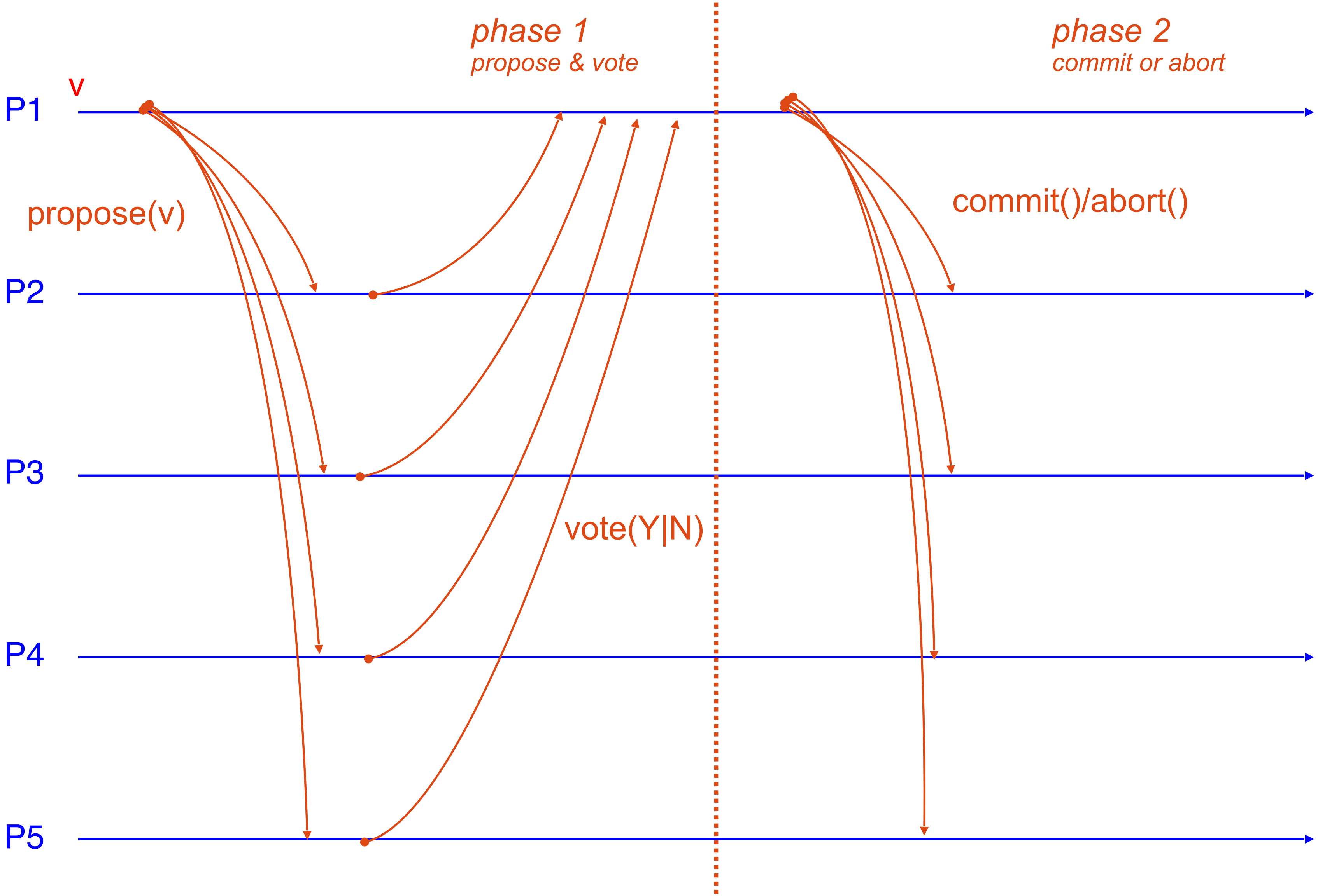
Byzantine failures

No assumption about behavior of a faulty process.

Some failure models (not a complete list)

- Fail-Stop (Crash-stop)
  - A processor stops, and never starts again
- Byzantine
  - A processor behaves adversarially, maliciously.
- Crash-recover
  - Well, it crashes and then restarts sometime later
- Omission
  - Doesn’t respond to input (or infinitely late)
- Timing
  - Correct response, but outside required time window

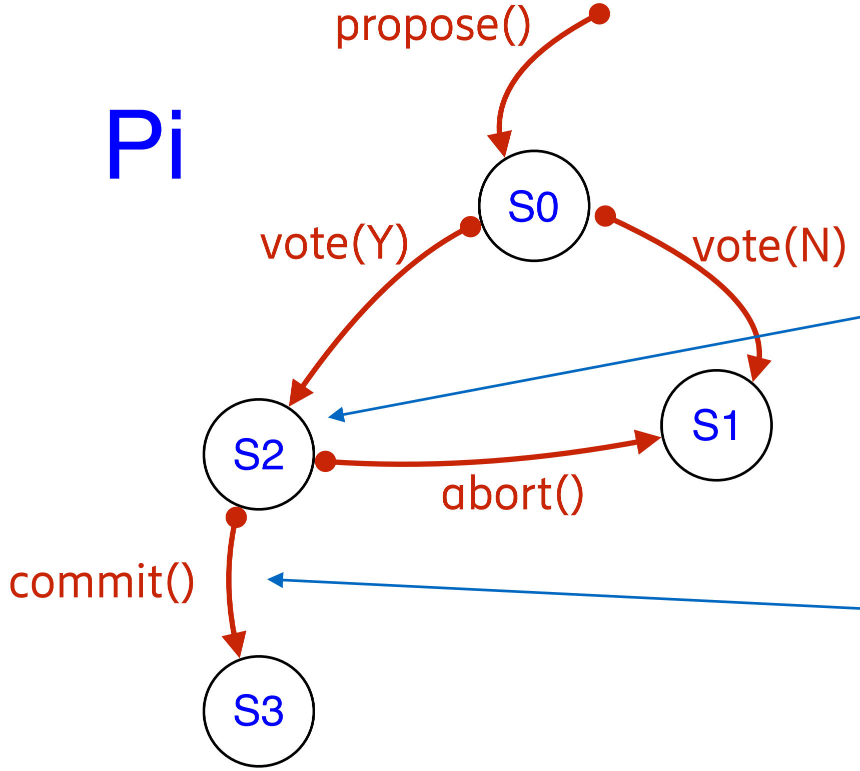
2PC (Two-Phase Commit)



agreement?
validity?
termination?

robustness?

2PC (Two-Phase Commit)



What if proposer fails here?

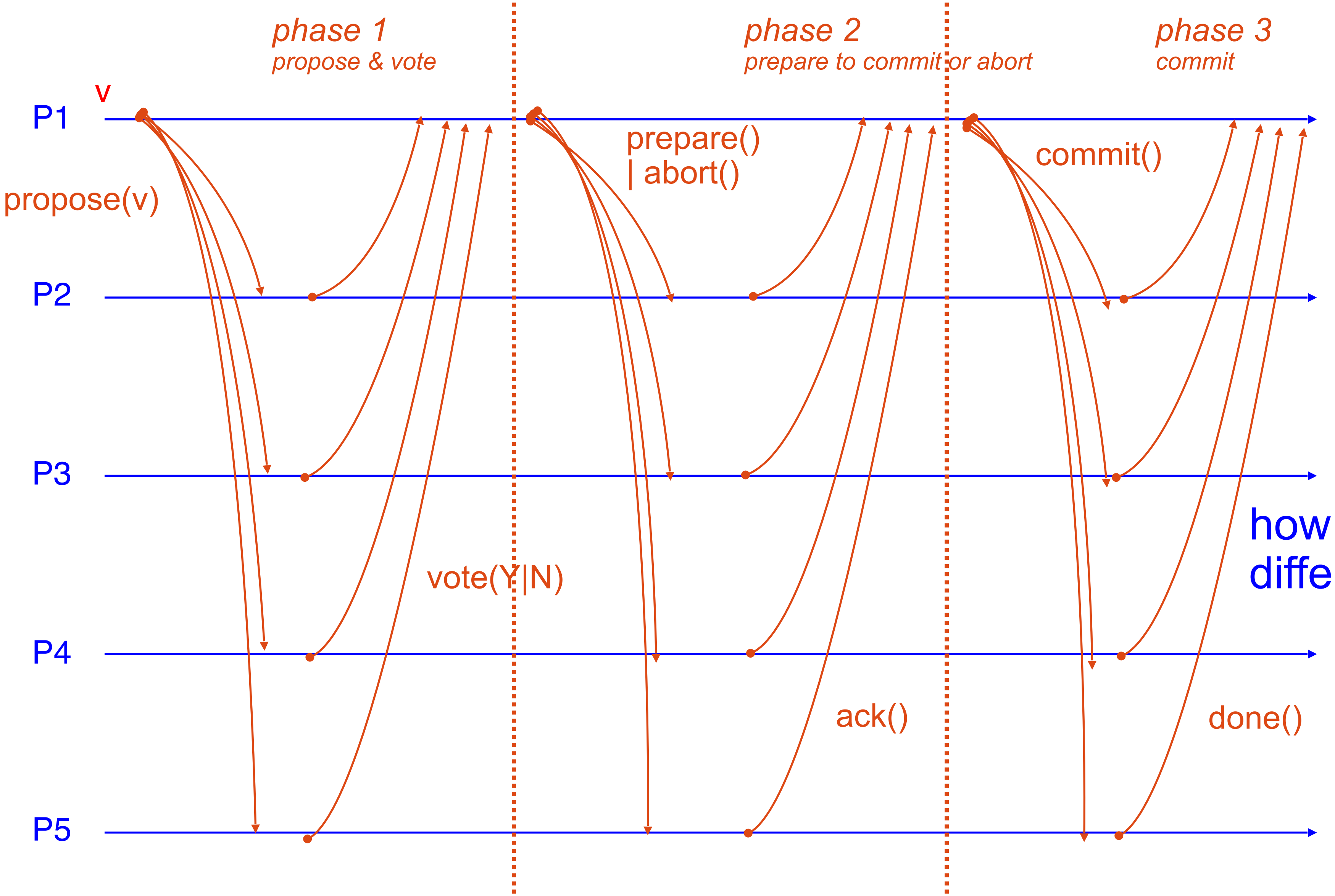
What if both Pi & the proposer fails here?

S1 - **aborted**: Voted N, received abort()

S2 - **uncertain**: Voted Y, not received commit()

S3 - **committed**: Received commit()

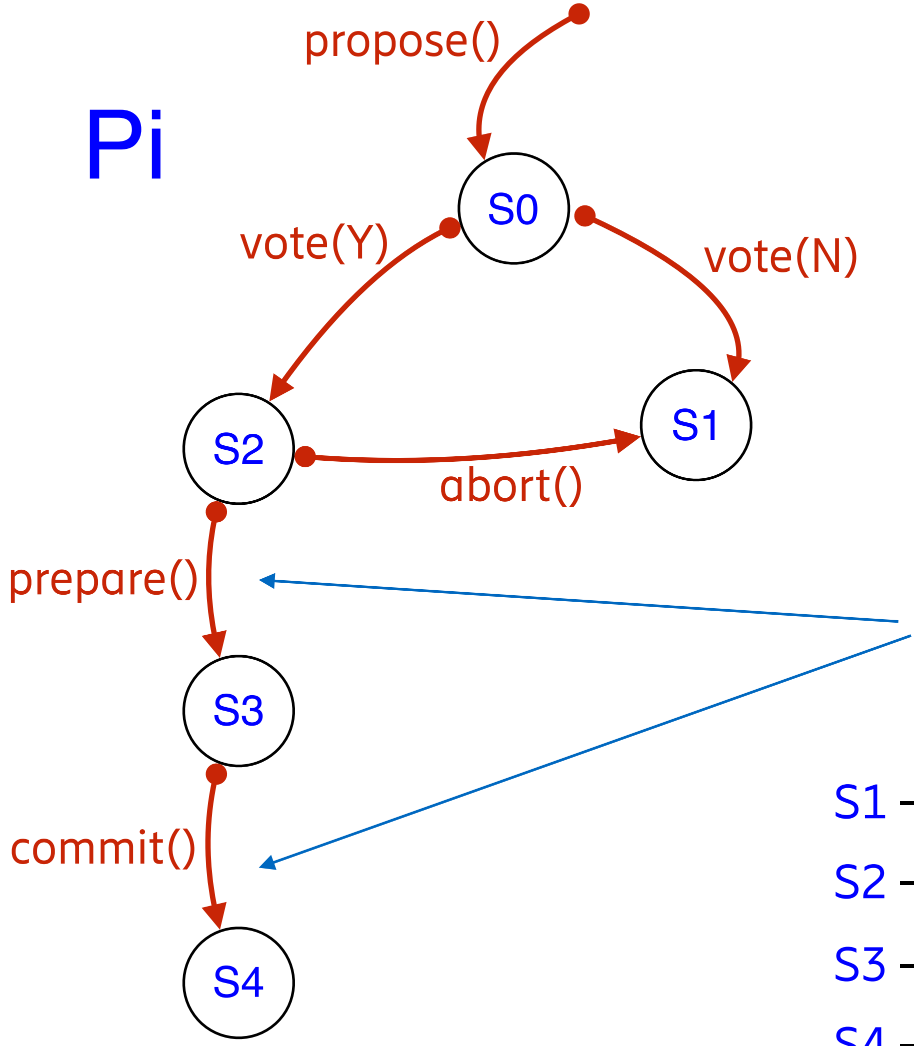
3PC (Three-Phase Commit)



agreement?
validity?
termination?

how does robustness differ from 2PC?

3PC (Three-Phase Commit)



What if Pi & the proposer fails here?

S1 - **aborted**: voted N, received abort()

S2 - **uncertain**: Voted Y, not received prepare()

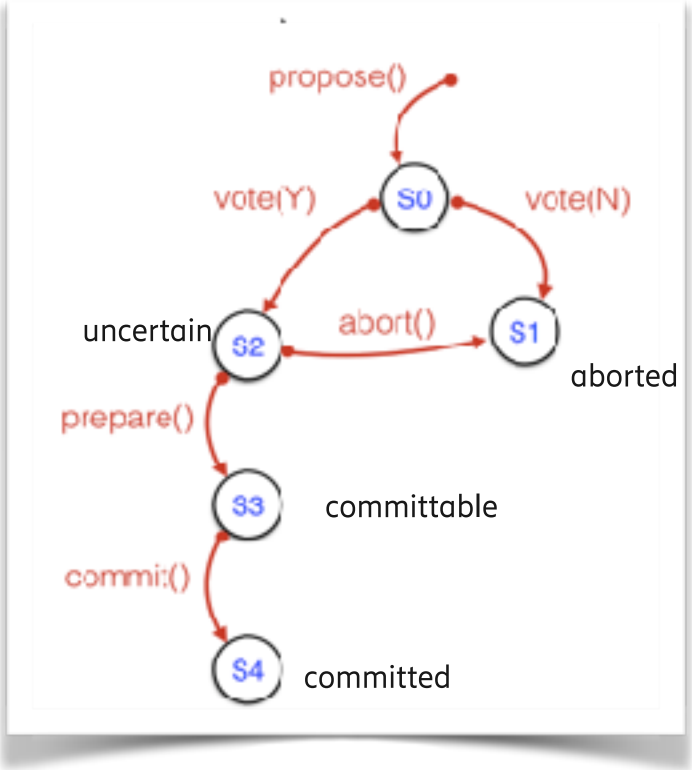
S3 - **committable**: Received prepare(), not commit()

S4 - **committed**: Received commit()

3PC (Three-Phase Commit)

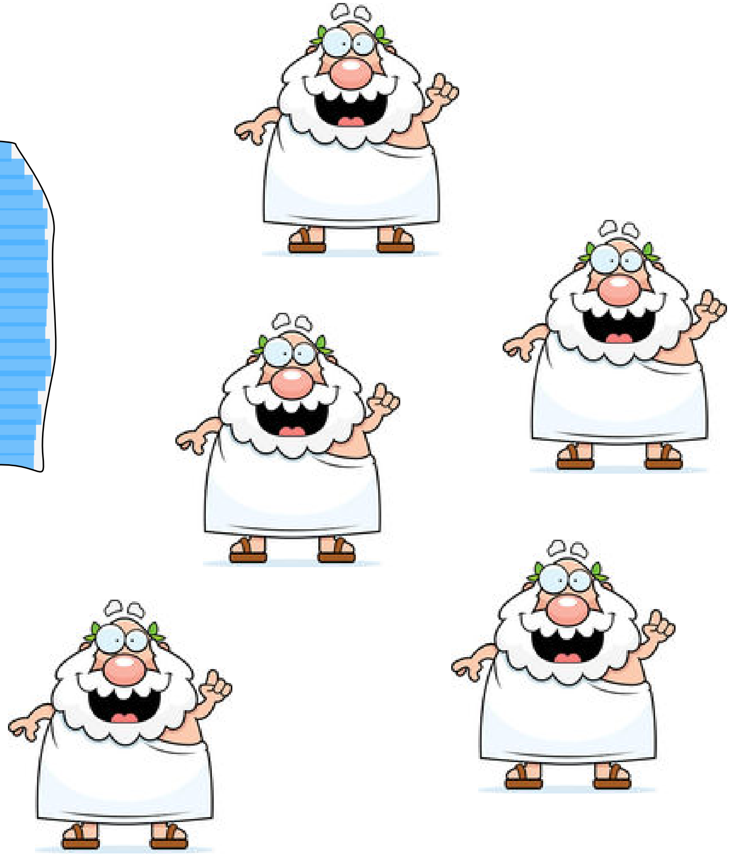
Recovery rules (run by a elected node)

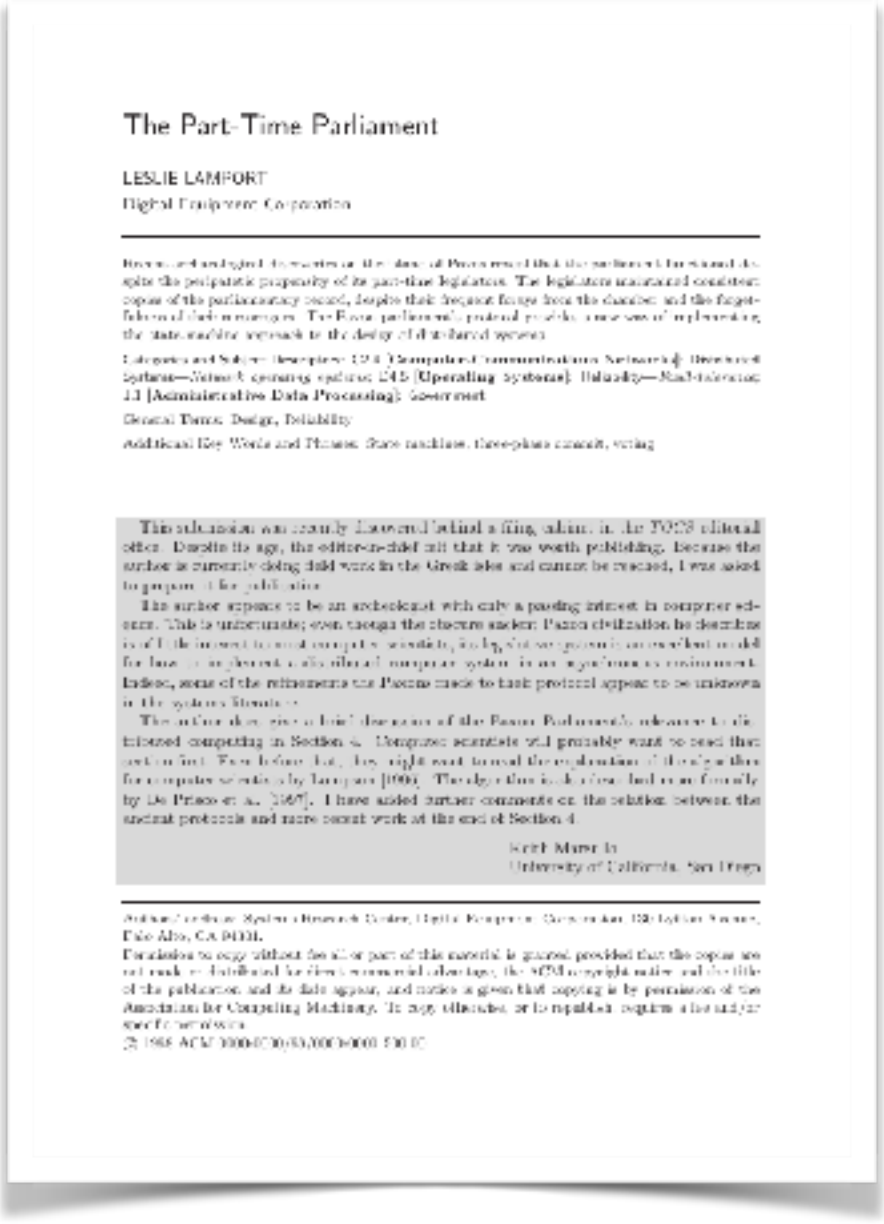
```
If some process in state aborted
    send abort() to all
else if some process in state committed
    send commit() to all
else if all processes in state uncertain
    send abort() to all
else if some process in state commitable
  send prepare() to all process in state uncertain
    wait for ack() and then send commit() to all
```




WE WERE ON A BREAK!

Paxos and Chubby





The Part-Time Parliament

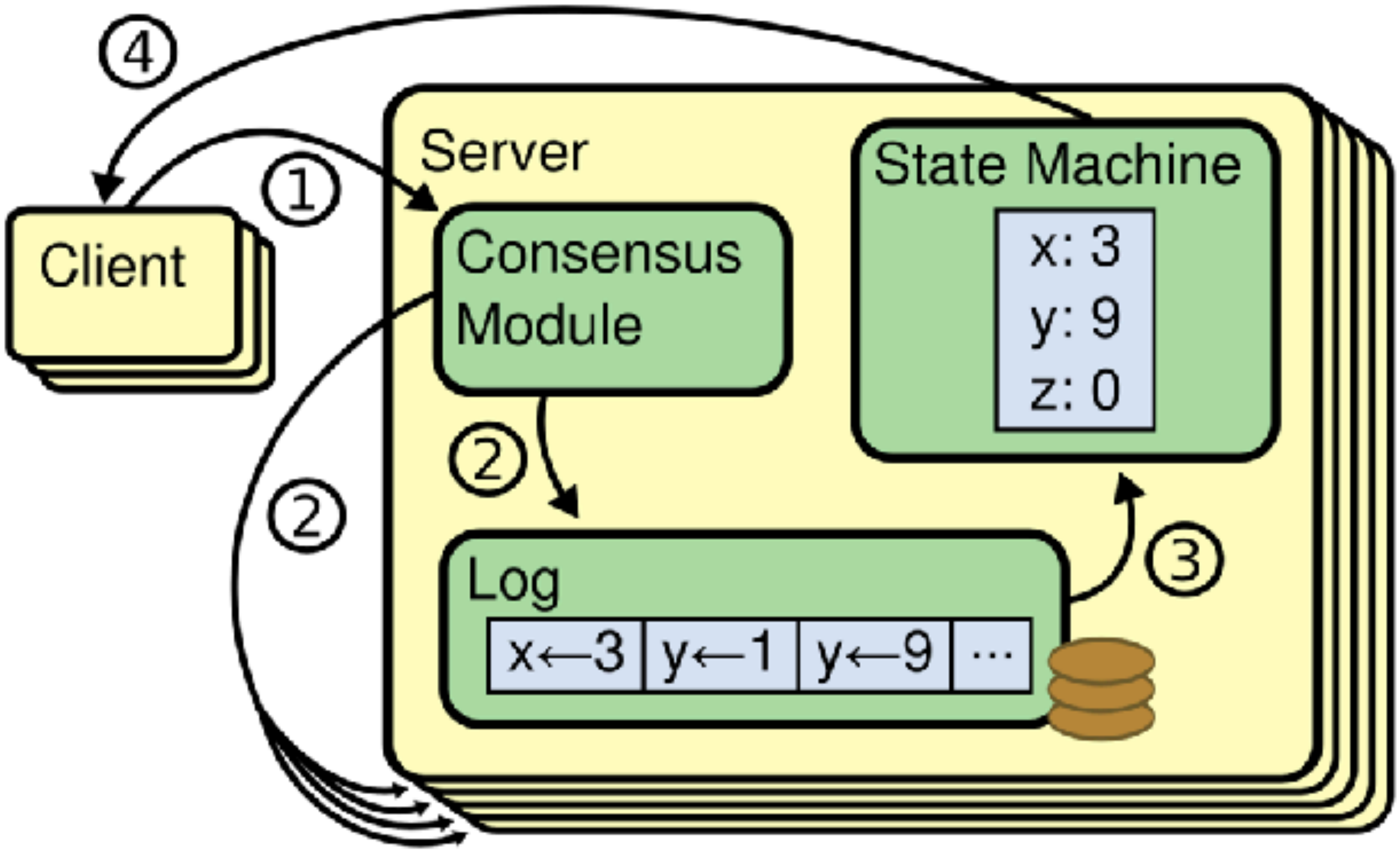
LESLIE LAMPORT

Digital Equipment Corporation

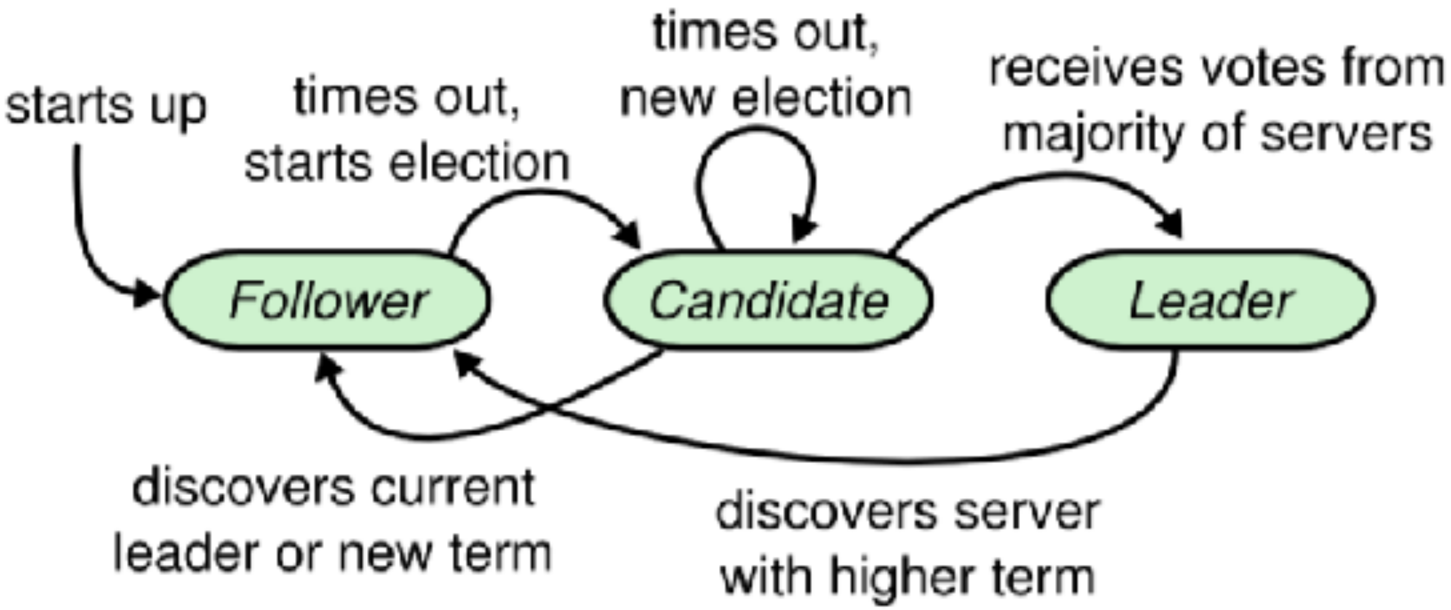
Took 8 years to review...

Raft

- Designed to be understood



Raft


starts up
times out,
starts election
times out,
new election
receives votes from
majority of servers
Follower
Candidate
Leader
discovers current
leader or new term
discovers server
with higher term

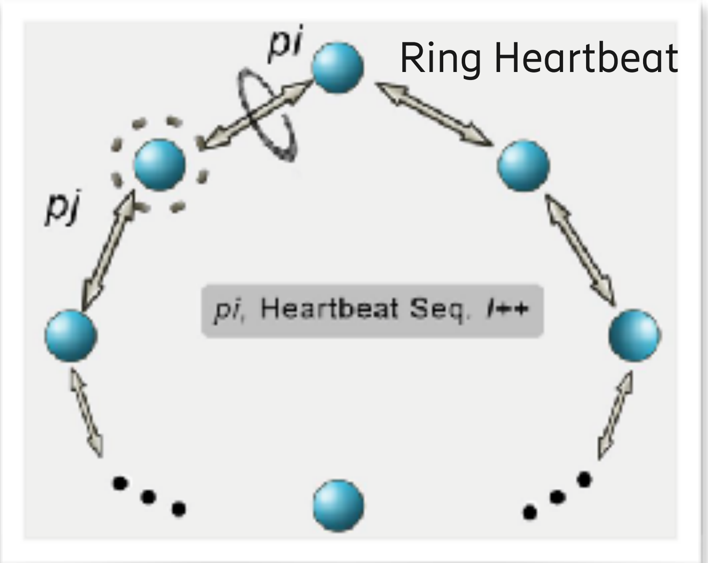
Consensus using etcd

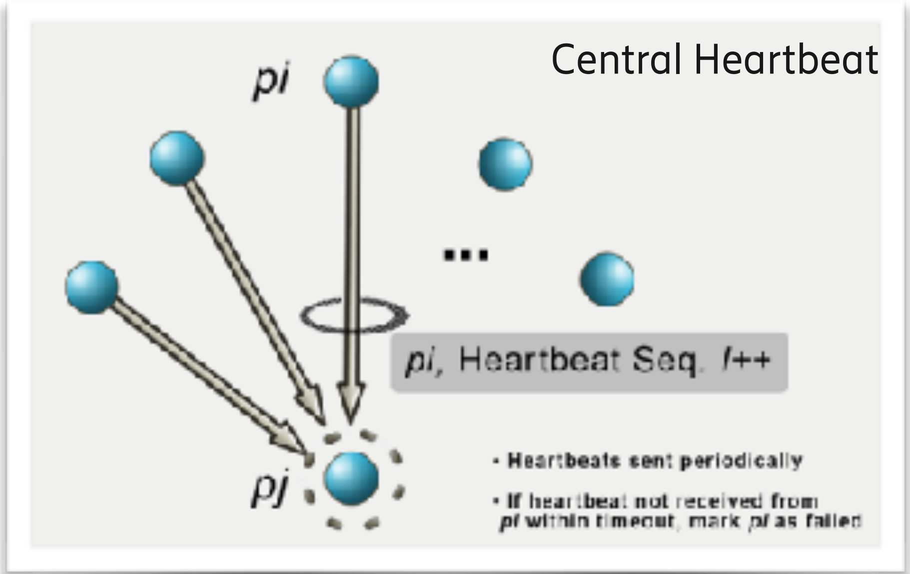
```python
"""etcd3 Leader election."""
import sys
import time
from threading import Event

import etcd3

LEADER_KEY = '/leader'
LEASE_TTL = 5
SLEEP = 1


def put_not_exist(client, key, value, lease=None):
    status, _ = client.transaction(
        compare=[client.transactions.version(key) == 0],
        success=[client.transactions.put(key, value, lease)],
        failure=[],
    )
    return status

def leader_election(client, me):
    try:
        lease = client.lease(LEASE_TTL)
        status = put_not_exist(client, LEADER_KEY, me, lease)
        return status, lease
    except Exception:
        status = False
        return status, None
```

```python
def main(me):
   client = etcd3.client()
    while True:
        print('leader election')
        leader, lease = leader_election(client, me)

        if leader:
            print(me + ': leader')
            try:
                while True:
                    # do work
                    lease.refresh()
                    time.sleep(SLEEP)
            except (Exception, KeyboardInterrupt):
                return
            finally:
                lease.revoke()
        else:
            print('follower; standby')

            election_event = Event()
            def watch_cb(event):
                if isinstance(event, etcd3.events.DeleteEvent):
                    election_event.set()
            watch_id = client.add_watch_callback(LEADER_KEY, watch_cb)

            try:
                while not election_event.is_set():
                    time.sleep(SLEEP)
                print('new election')
            except (Exception, KeyboardInterrupt):
                return
            finally:
                client.cancel_watch(watch_id)

if __name__ == '__main__':
    me = sys.argv[1]
    main(me)
```

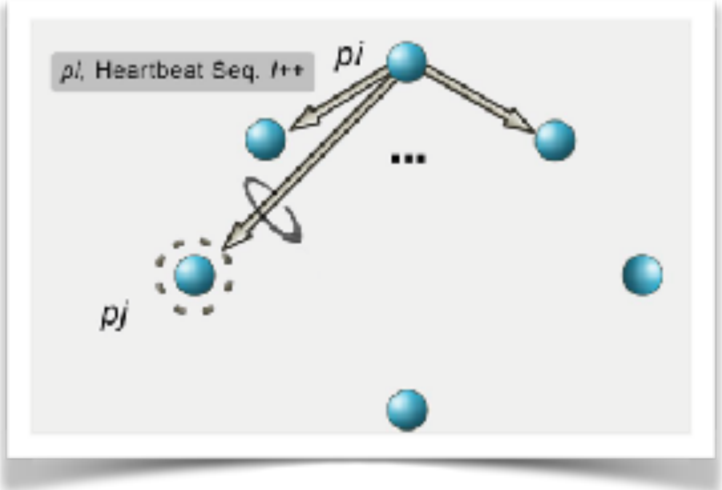
Heartbeating

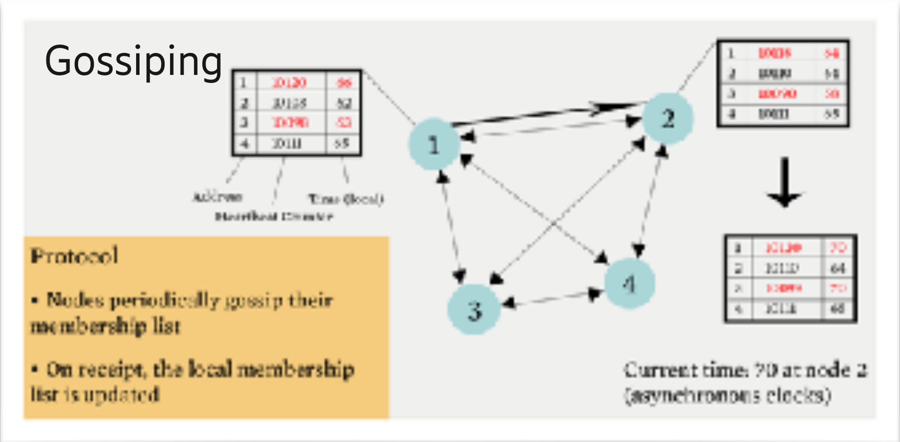
Some simple failure detectors





- Heartbeats sent periodically
- If heartbeat not received from *pi* within timeout, mark *pi* as failed





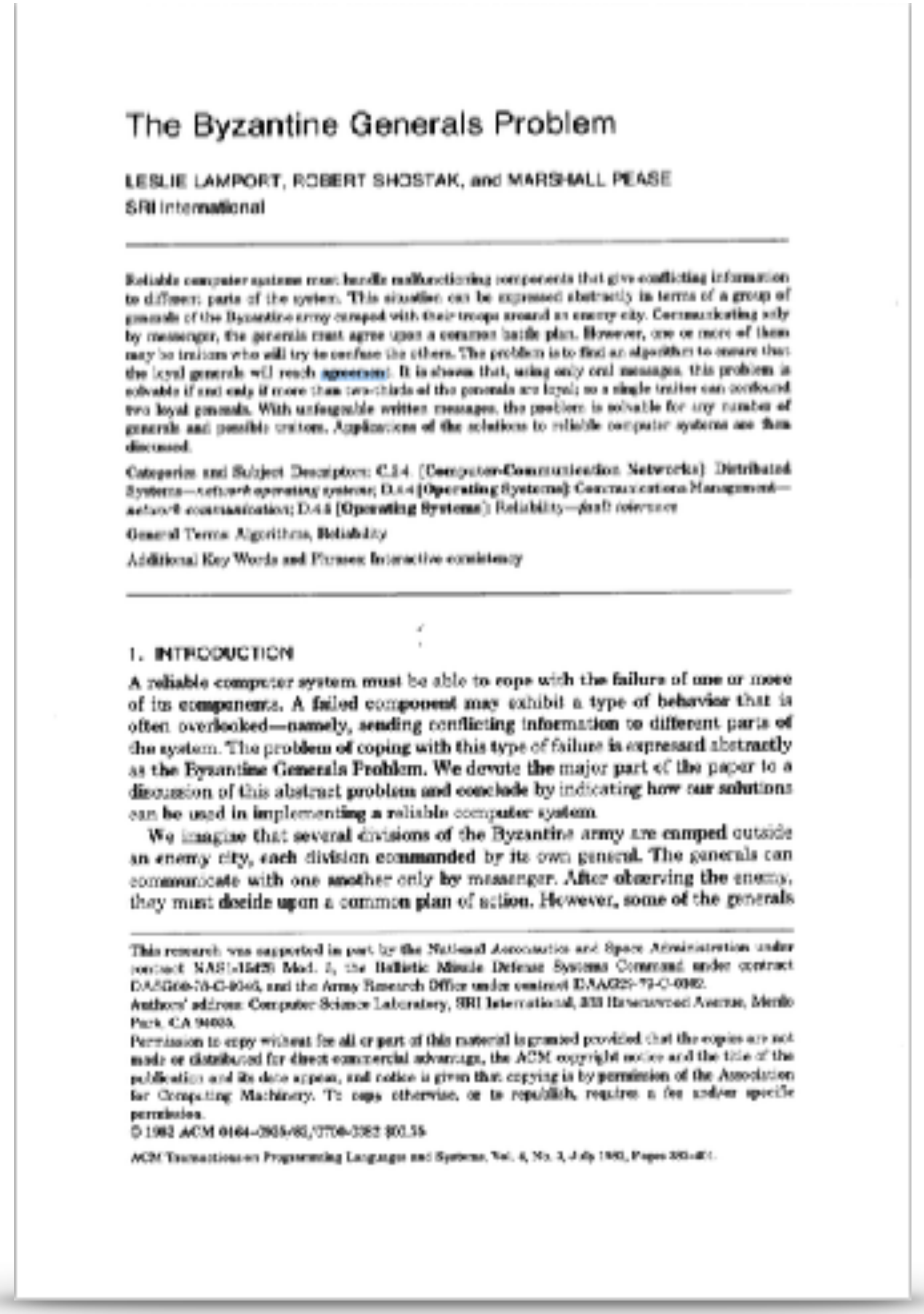
Protocol

- Nodes periodically gossip their membership list
- On receipt, the local membership list is updated

Current time: 70 at node 2
(asynchronous clocks)

The Byzantine Generals

- How can we handle faulty, malicious or incomplete messages?
  - Failed broadcasts for example
  - May actively try to trick other processes, eg fake message or not sending
- Is synchronous and we can detect missing message



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

1. INTRODUCTION

A reliable computer system must be able to cope with the failure of one or more of its components. A failed component may exhibit a type of behavior that is often overlooked—namely, sending conflicting information to different parts of the system. The problem of coping with this type of failure is expressed abstractly as the Byzantine Generals Problem. We devote the major part of the paper to a discussion of this abstract problem and conclude by indicating how our solutions can be used in implementing a reliable computer system.

We imagine that several divisions of the Byzantine army are camped outside an enemy city, each division commanded by its own general. The generals can communicate with one another only by messenger. After observing the enemy, they must decide upon a common plan of action. However, some of the generals

The Byzantine Generals

- A city under siege by several divisions of the Byzantine army
- Each division is commanded by its own general.
- The generals communicate only using messengers.
- After observing the enemy, they must decide upon a common plan of action.
- However, some of the generals may be traitors, trying to prevent the loyal generals from reaching agreement.
- We will assume that there is a single commanding general (Commander), and the rest of the generals are his subordinates (Lieutenants)

Objective

- All loyal generals decide upon the same plan of action
- A small number of traitors will not cause the loyal generals to adopt a bad plan

Formally rephrased as

Byzantine General Problems

1. All loyal lieutenants obey the same order
2. If the commander is loyal, then every loyal lieutenant obeys the order he sends

The Byzantine Generals

1. All loyal lieutenants obey the same order
2. If the commander is loyal, then every loyal lieutenant obeys the order he sends

Impossible for L1 to distinguish between the two cases! ⇒ impossible to fulfil requirements 1 & 2

Desired behaviour (condition #2) is
Attack!

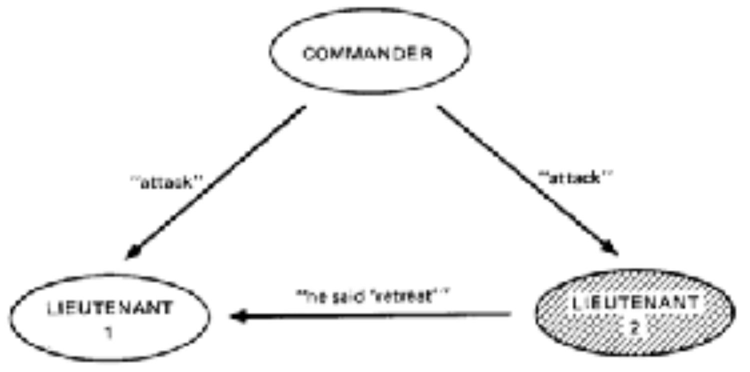


Fig. 1. Lieutenant 2 a traitor.

Desired behaviour (condition #1) is
Retreat!

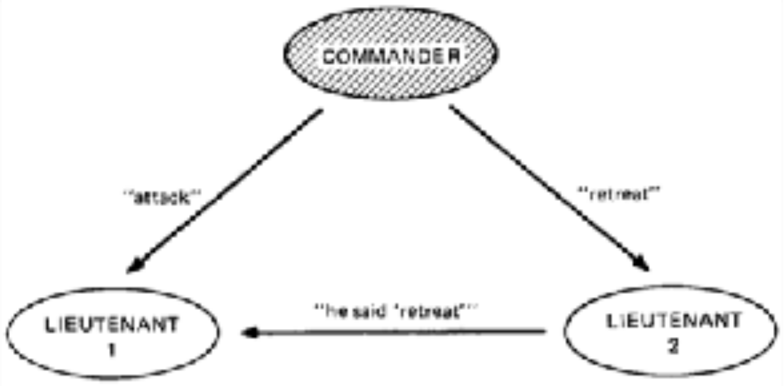


Fig. 2. The commander a traitor.

Minimal Bound on Traitors

There is no algorithm to reach consensus unless more than two thirds of the generals are loyal. In other words, impossible if $n \leq 3m$ for n processes, m of which are faulty

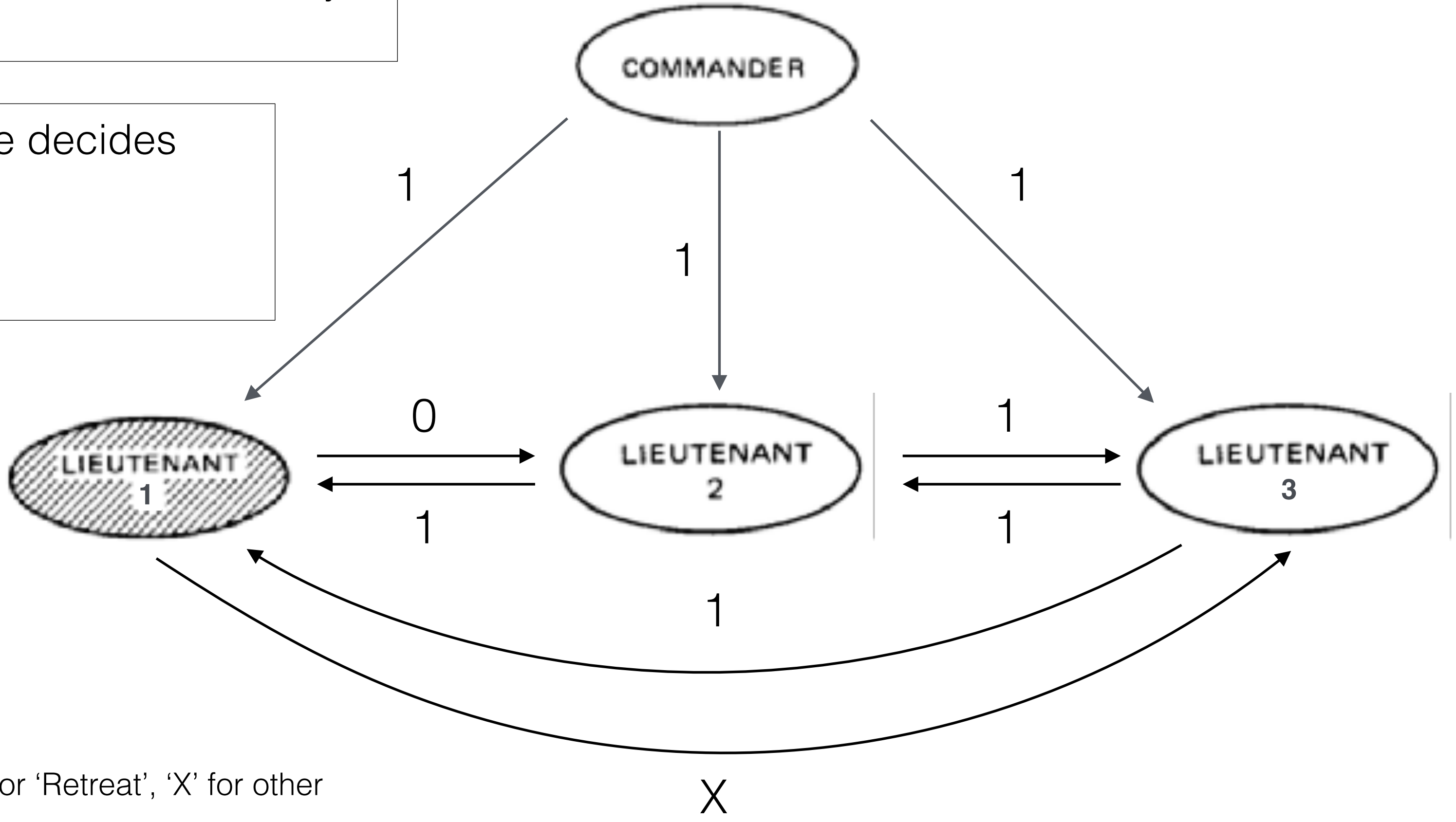
$\Rightarrow n > 3m + 1$ for all algorithms that solve the Byzantine Generals problem

A Loyal Commander and One Traitor Lieutenant

Step 1: Commander sends same value *1* to all

Step 2: Each of L1, L2, L3 forwards the message, but L1 sends arbitrary values

Step 3: Each node decides
L2 has {1,1,0},
L3 has {1,1,X},
Both choose 1.



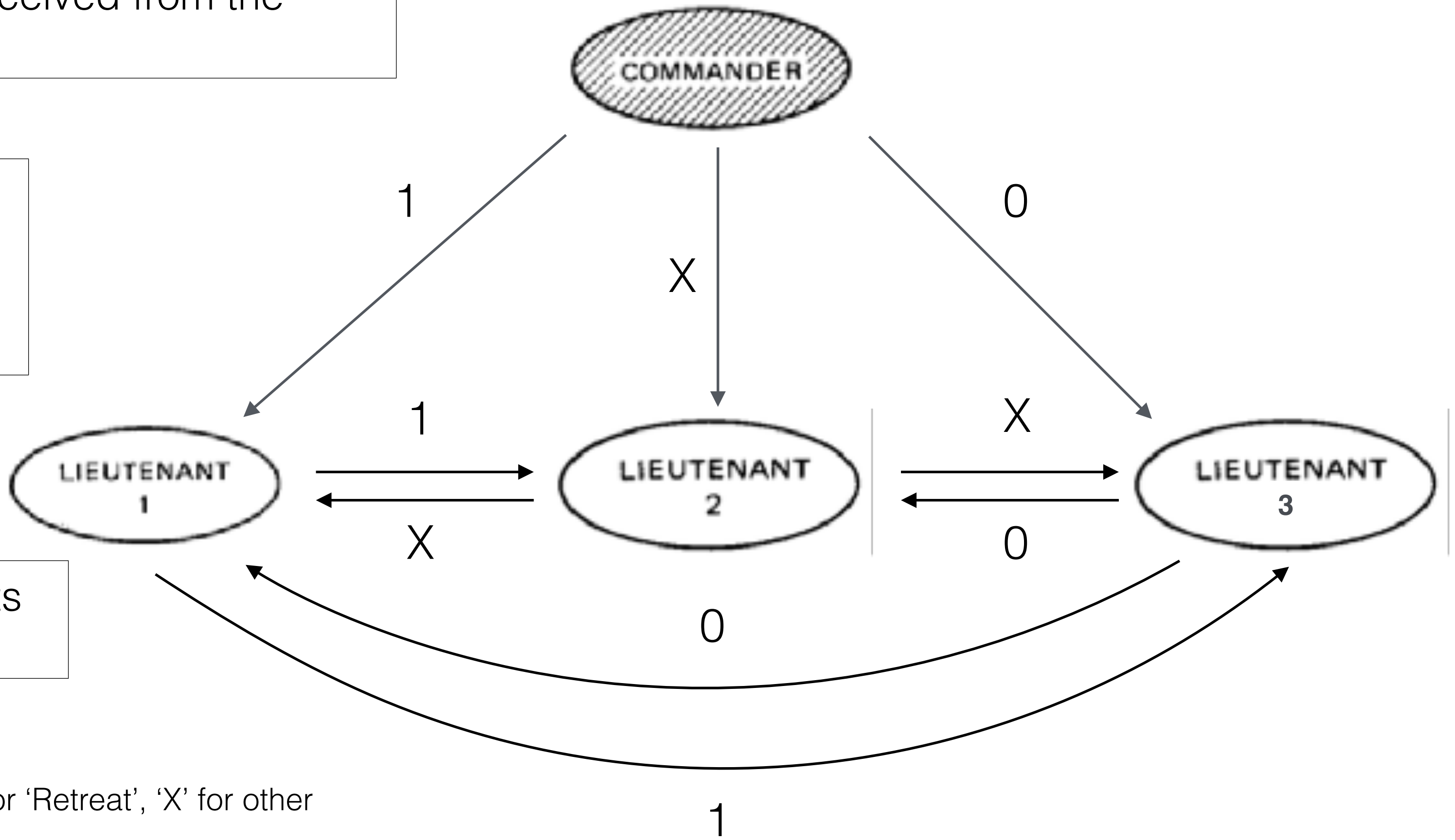
Use ‘1’ for ‘Attack’, or ‘0’ for ‘Retreat’, ‘X’ for other

A Traitor Commander with Loyal Lieutenants

Step 1: Commander sends different values to all

Step 2: Each of L1, L2, L3 forwards the values they received from the commander

Step 3: Decide
L1 has {1, 0, X},
L2 has {1, 0, X},
L3 has {1, 0, X}

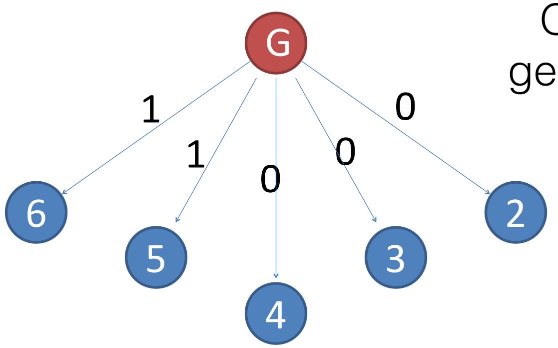


All loyal lieutenants get same result!

Use ‘1’ for ‘Attack’, or ‘0’ for ‘Retreat’, ‘X’ for other

Let's try this live

The commander sends a message to all lieutenants



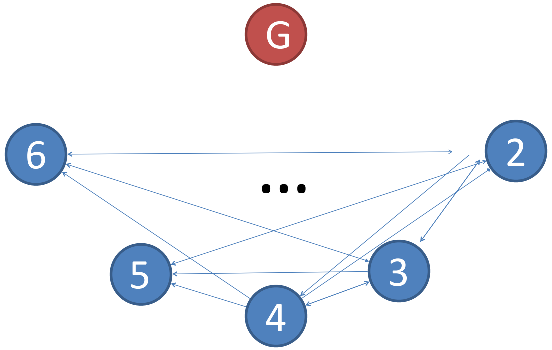
Only the general is a traitor

Use a single bit: ‘1’ for ‘Attack’, or ‘0’ for ‘Retreat’.

Each lieutenant sends the message he received to all other lieutenants

Sender=P_2	Sender=P_3	Sender=P_4	Sender=P_5	Sender=P_6
{0,12}	{0,13}	{0,14}	{1,15}	{1,16}

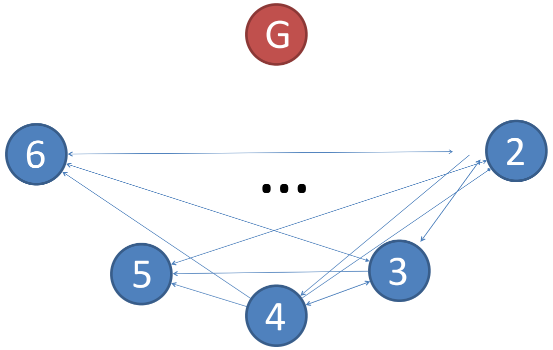
The message {0,12} is sent to all other nodes by node 2, and so on



Each lieutenant forwards all the messages he received to all other lieutenants

Sender=P_2	Sender=P_3	Sender=P_4	Sender=P_5	Sender=P_6
{0,132} {0,142} {1,152} {1,162}	{0,123} {0,143} {1,153} {1,163}	{0,124} {0,134} {1,154} {1,164}	{0,125} {0,135} {0,145} {1,165}	{0,126} {0,136} {0,146} {1,156}

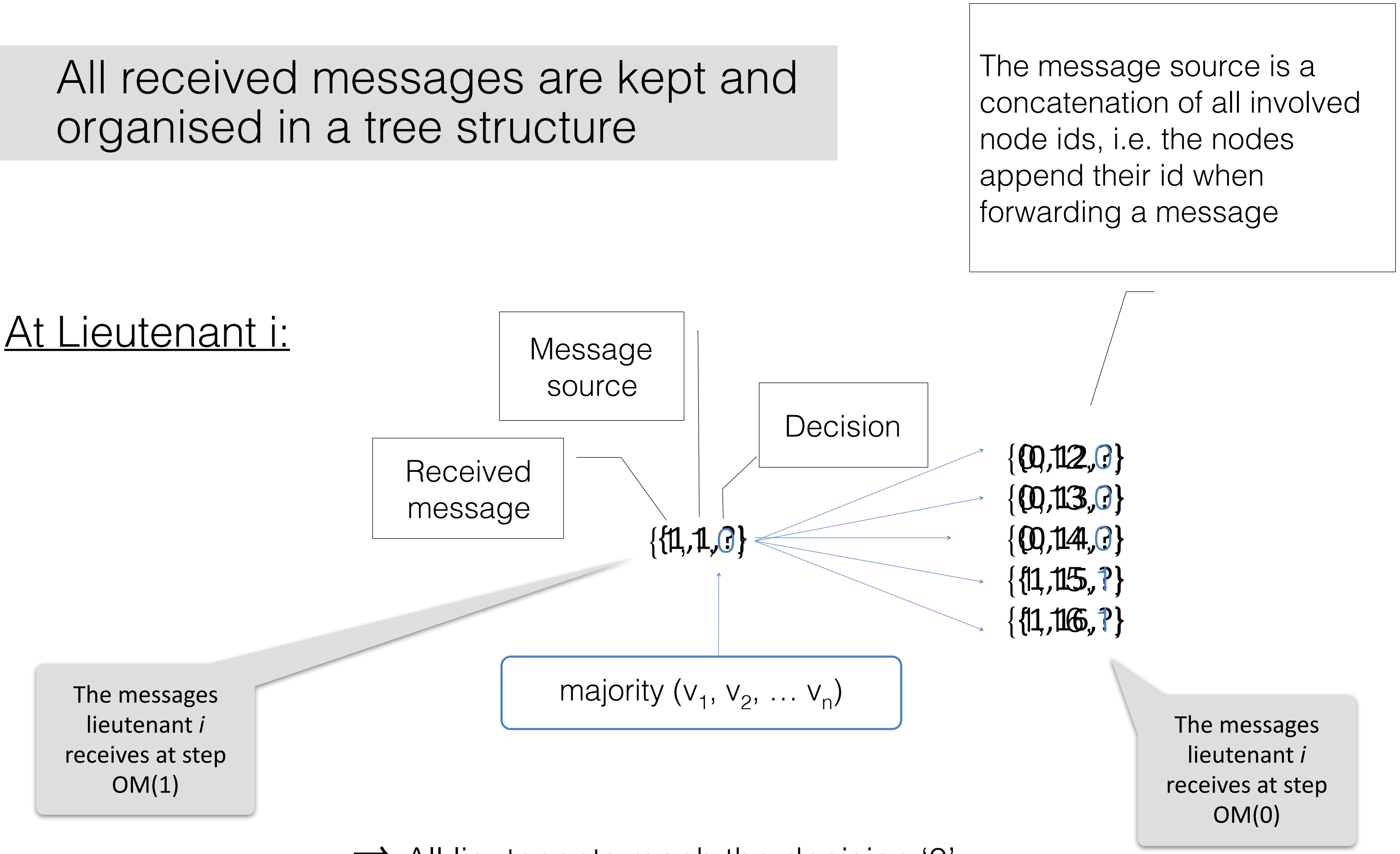
(This round is not necessary when we only have one traitor)



The Decision Making

All received messages are kept and organised in a tree structure

At Lieutenant i:



The message source is a concatenation of all involved node ids, i.e. the nodes append their id when forwarding a message

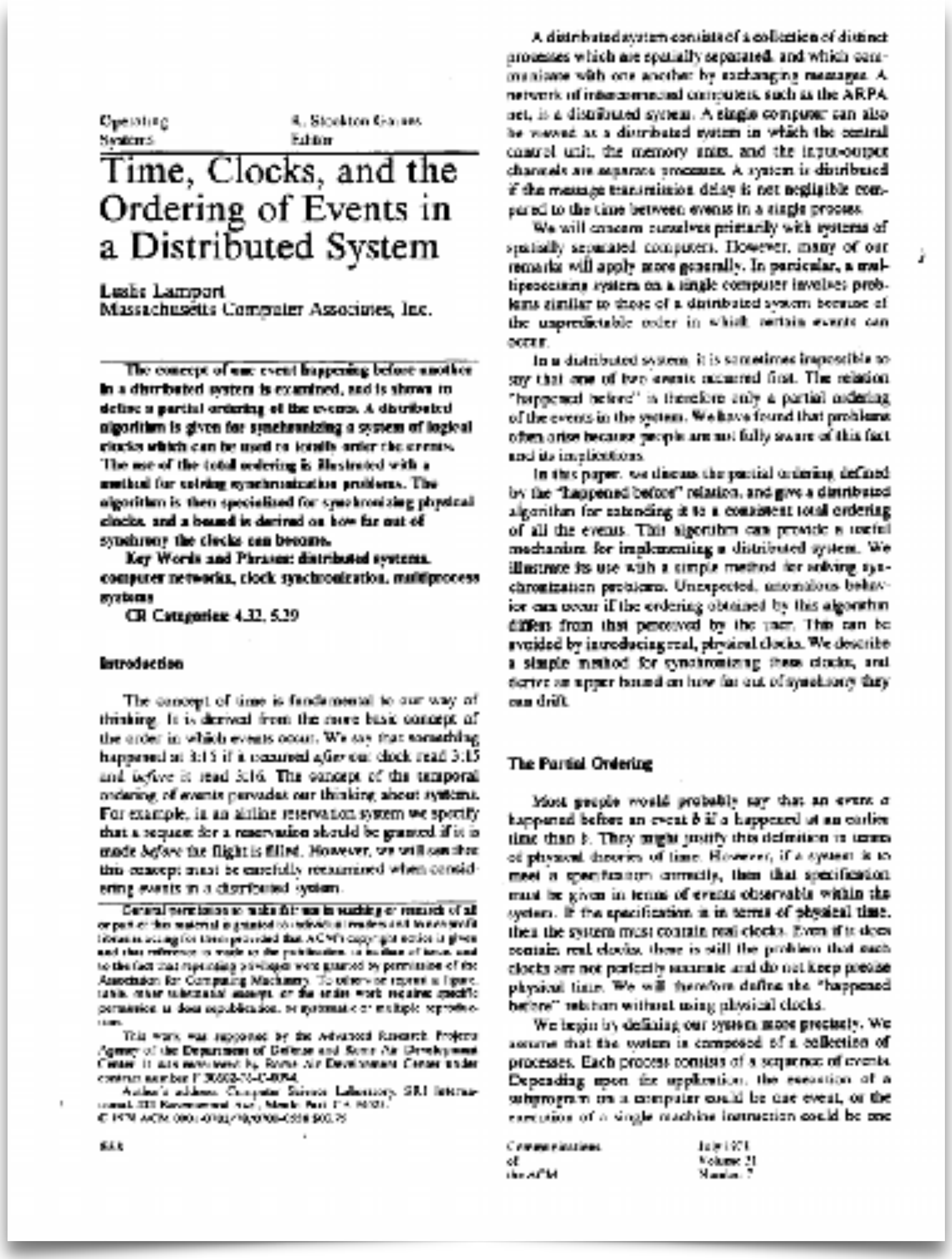
The messages lieutenant *i* receives at step OM(1)

The messages lieutenant *i* receives at step OM(0)

⇒ All lieutenants reach the decision ‘0’

Logical Clocks

- "Time, Clocks, and the Ordering of Events in a Distributed System", Leslie Lamport, 1978
- A distributed algorithm for find a total order of events in a distributed system



Operating Systems

R. Stockton Gaines Editor

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

Leslie Lamport
Massachusetts Computer Associates, Inc.

The ‘Happened-before’ Relation

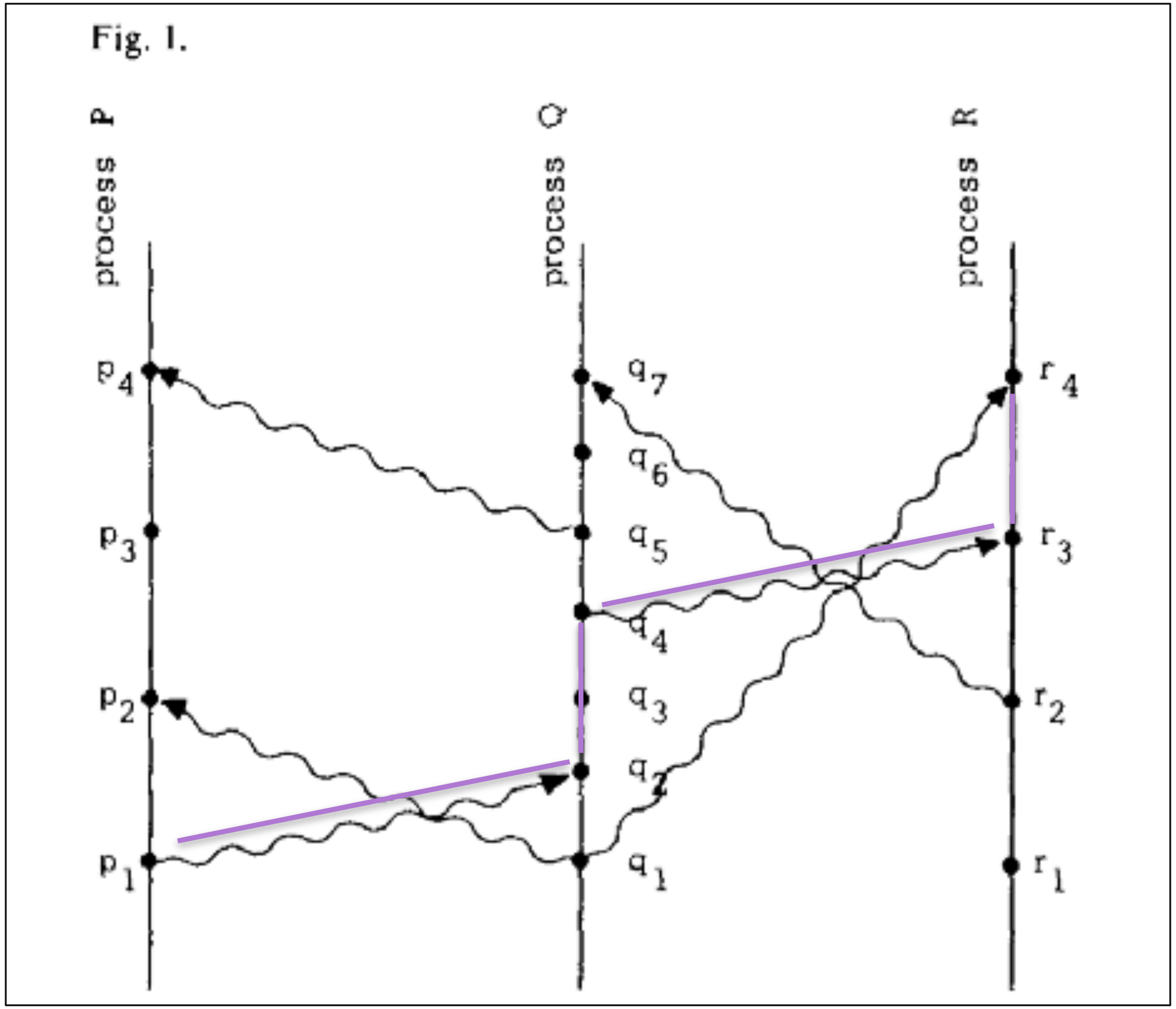


Fig. 1.

For example:

$p_1 \longrightarrow r_4$

path exists

p_2, q_3 : concurrent

p_3, q_3 : concurrent

Permits out of order message arrival

Logical Clocks

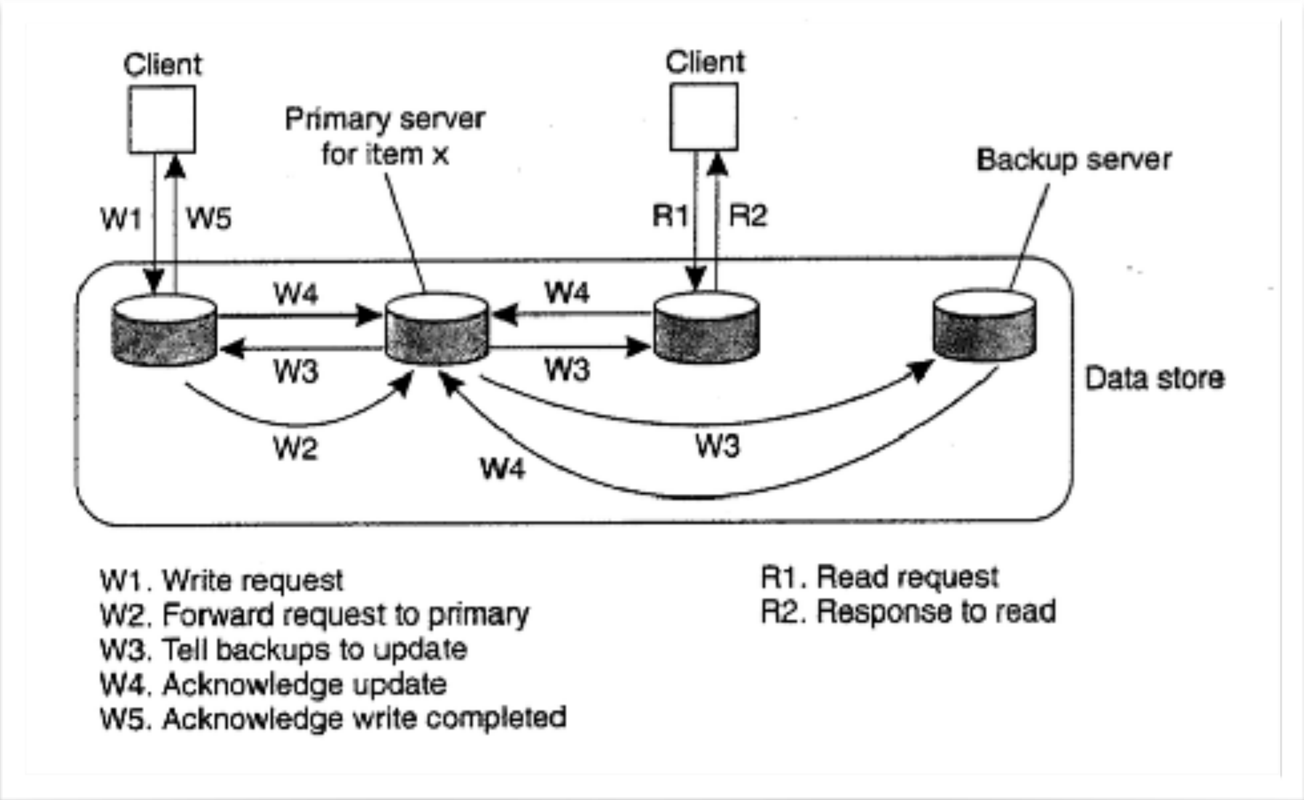
- Logical clocks – abstract way of assigning a number to an event where the number denotes the time of occurrence of the event
- A clock $C_i\langle a\rangle$ assigns a number to an event a in process P_i
  - Could be a number or actual time
    - The Clock Condition: $a \longrightarrow b \Longrightarrow C\langle a\rangle < C\langle b\rangle$ (the opposite is not true, why?)

Total Ordering

- Use system of clocks satisfying the Clock Condition to place a total ordering of all events denoted "$\Rightarrow$"
    - Simply order events by the their time $C\langle a\rangle$ and break ties with any arbitrary total order (alphabetical, etc)
- We can define a total ordering on the set of all system events
    - $a \Rightarrow b$ if either $C_i\langle a\rangle < C_j\langle b\rangle$ or $C_i\langle a\rangle = C_j\langle b\rangle$ and $P_i < P_j$
- This ordering is not unique, but well defined

Data Replication

- There are two primary reasons for replicating data
  - Reliability
  - Performance.
- Three classic approaches to replicated data
  - primary copy
  - multi-master
  - quorum consensus

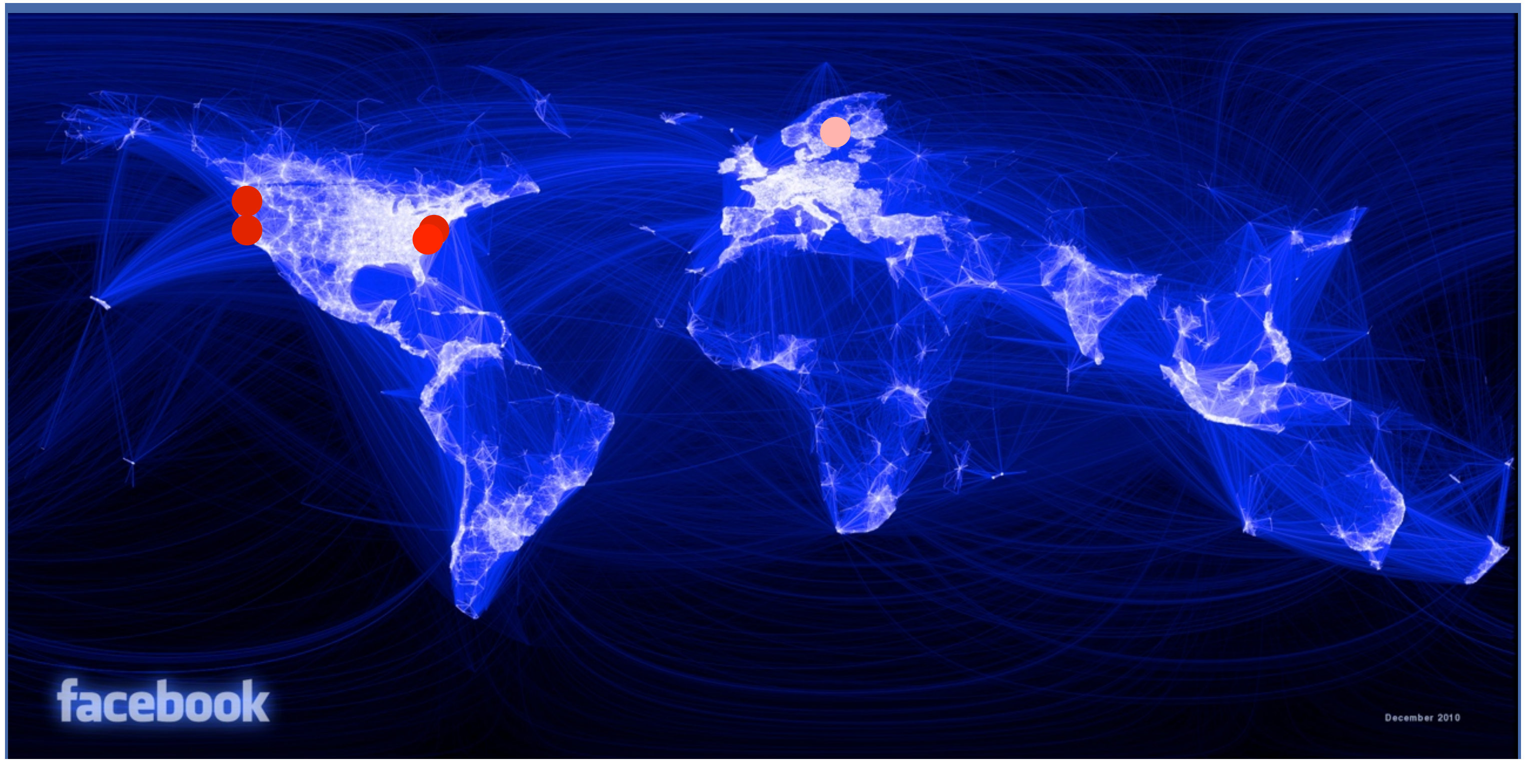


W1. Write request
W2. Forward request to primary
W3. Tell backups to update
W4. Acknowledge update
W5. Acknowledge write completed

R1. Read request
R2. Response to read

Example: Facebook

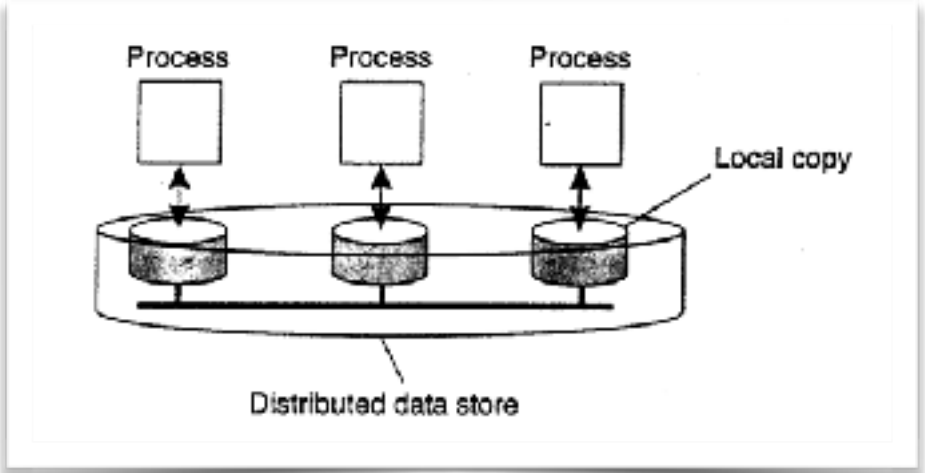
- Read from closest server
- Write to California
- Other servers update cache every 15 minutes
- After write: read from CA for 15 minutes



Slide: Marc Shapiro

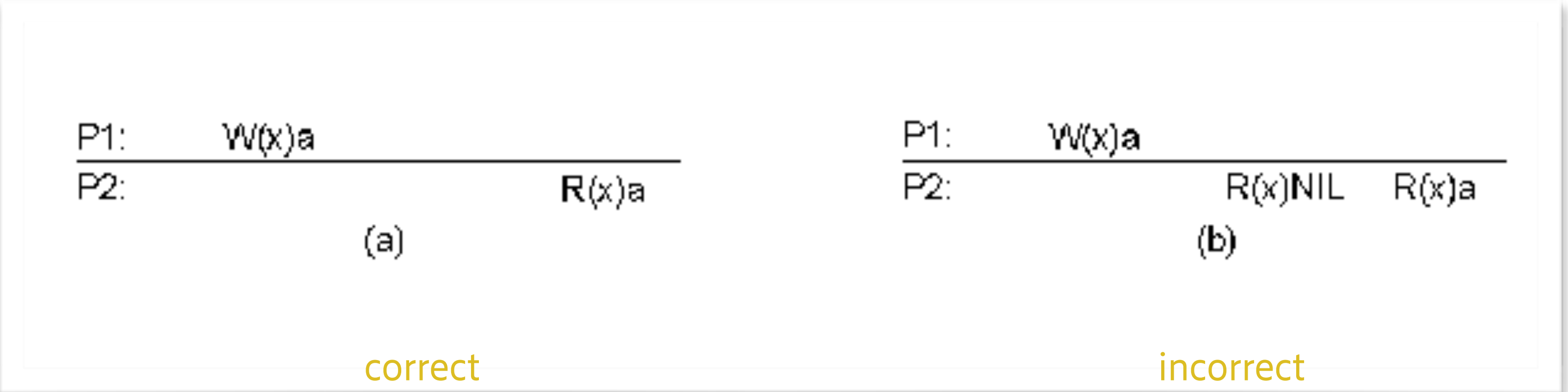
The Consistency Problem

- How is data replicated across a distributed system?
- The client side view :
  - A set of process that reads & writes
  - A distributes data store that is treated as a black box
  - How & when do updates become observable?
- The server side view:
  - The inside of the distributed data store
  - How is data propagated and replicated between storage nodes?



Strict consistency

- All writes are instantaneously visible to all processes and absolute global time order is maintained
- Hard to achieve!

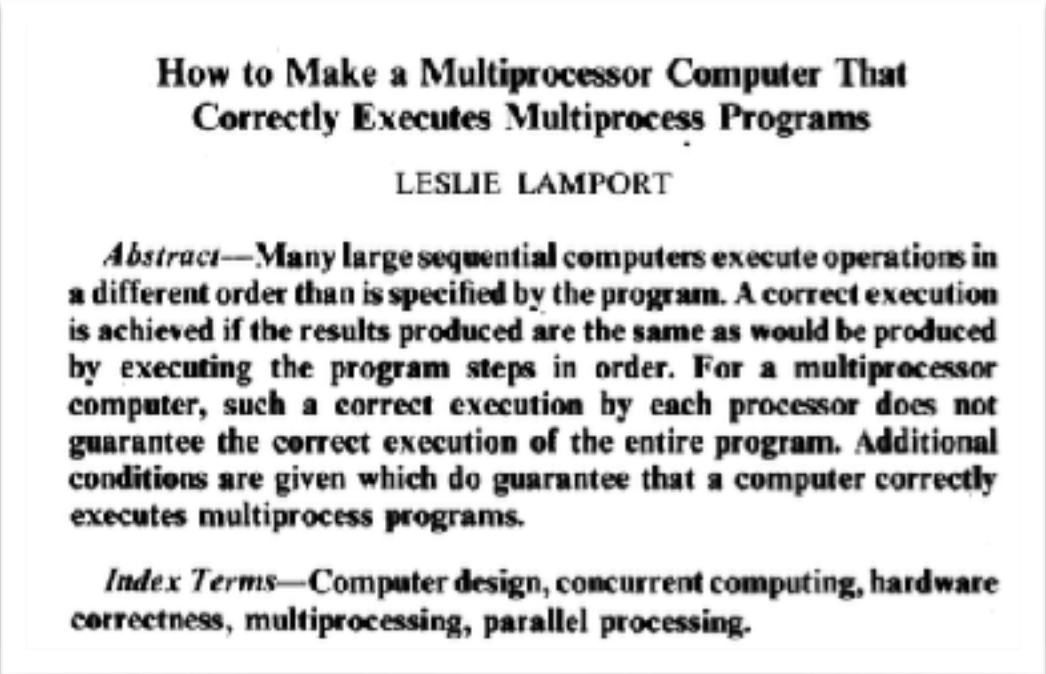


correct

incorrect

Sequential consistency

- Operations on each process must be in order
- All processes see the same interleaving set of operations, regardless of what that interleaving is.
- Cares about **program order,** not time



How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs

LESLIE LAMPORT

Abstract—Many large sequential computers execute operations in a different order than is specified by the program. A correct execution is achieved if the results produced are the same as would be produced by executing the program steps in order. For a multiprocessor computer, such a correct execution by each processor does not guarantee the correct execution of the entire program. Additional conditions are given which do guarantee that a computer correctly executes multiprocess programs.

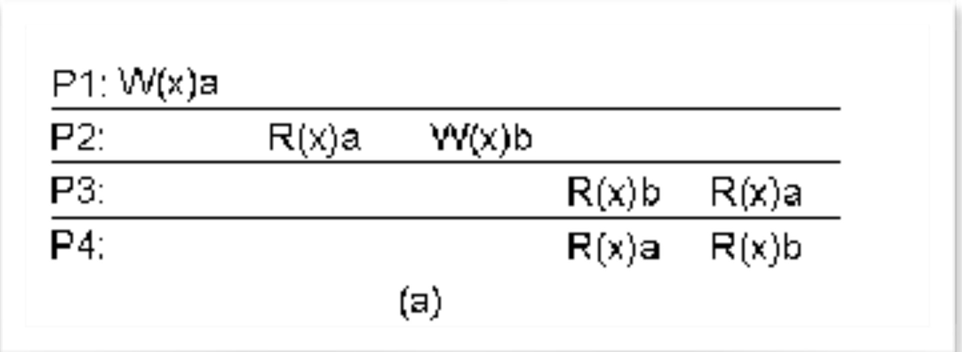
Index Terms—Computer design, concurrent computing, hardware correctness, multiprocessing, parallel processing.

(a)
correct

(b)
incorrect

Causal consistency

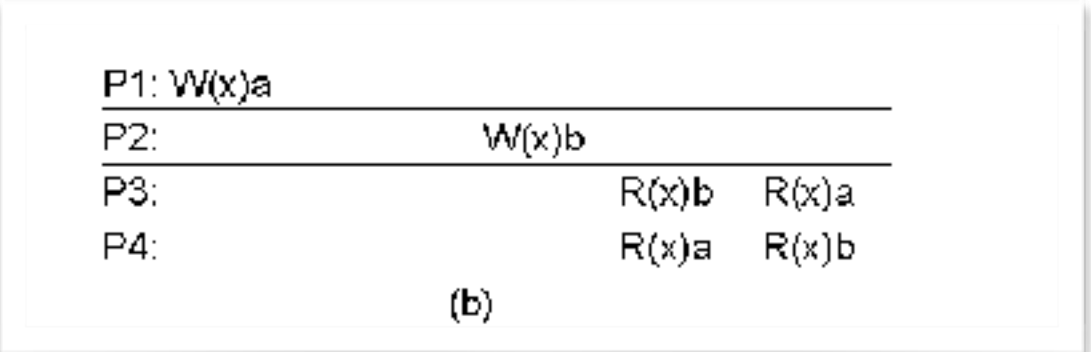
Writes that are *potentially* causally related must be seen by all processes in the same order.
Concurrent writes may be seen in a different order by different processes.



P1:	W(x)a			
P2:	R(x)a	W(x)b		
P3:			R(x)b	R(x)a
P4:			R(x)a	R(x)b

(a)

incorrect



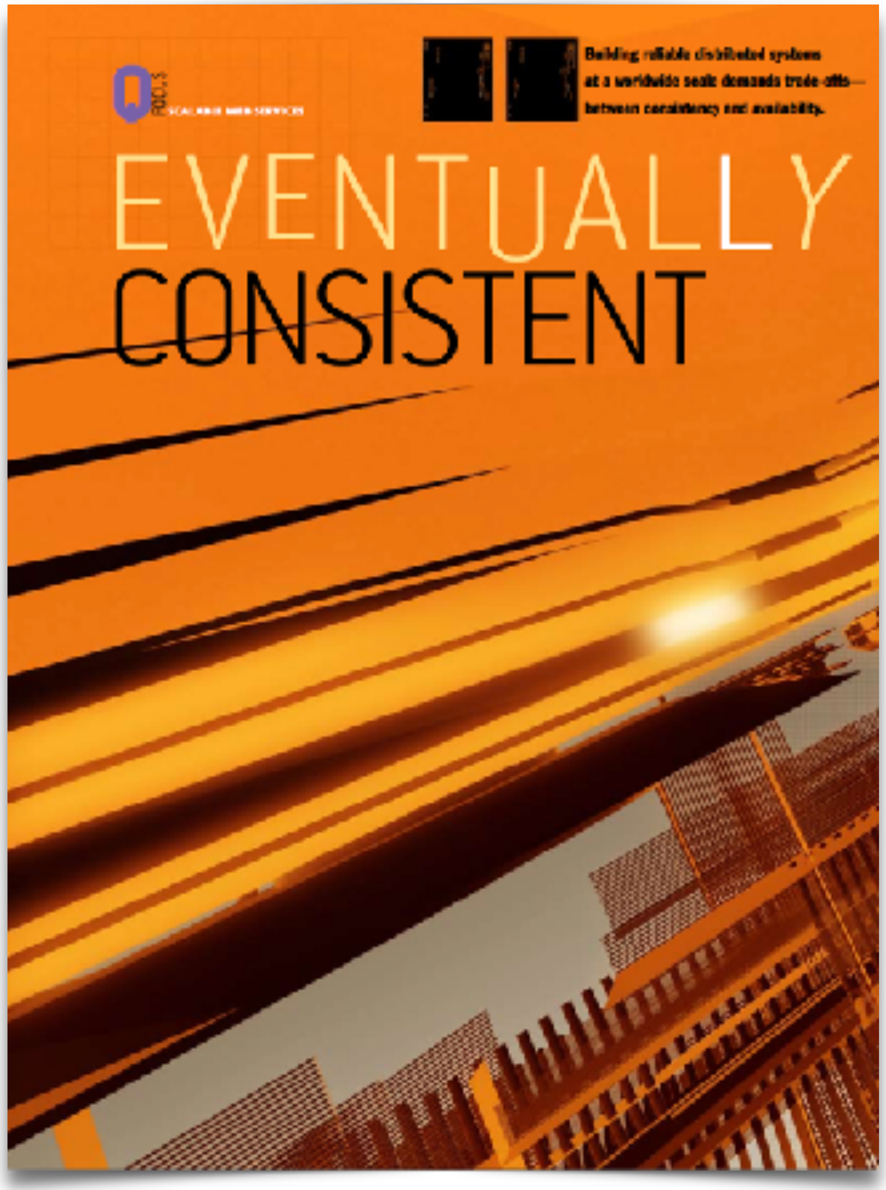
P1:	W(x)a		
P2:	W(x)b		
P3:		R(x)b	R(x)a
P4:		R(x)a	R(x)b

(b)

correct

Eventual Consistency

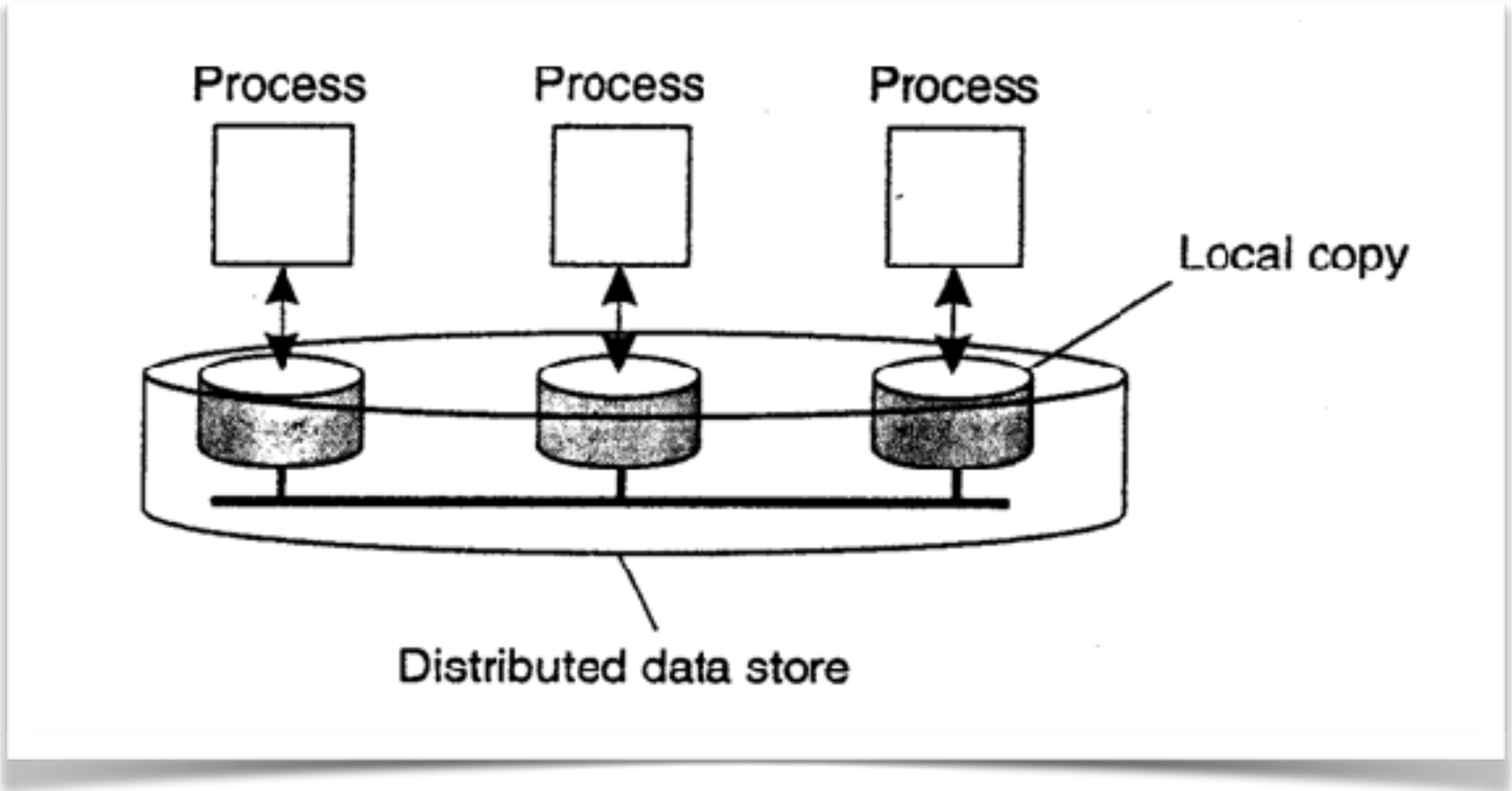
- The only requirement is that all replicas will eventually be the same.
- DNS is a well know example. Updates to a name are distributed according to a configured pattern and in combination with time-controlled caches; eventually, all clients will see the update.



Server Side Consistency

The algorithms that implement the consistency models of choice

- — N = the number of replicas
- — W = the write set
- — R = the read set



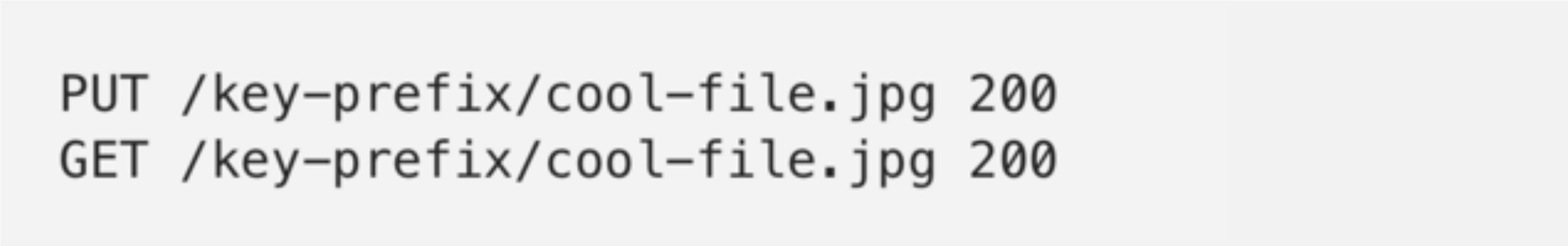
Server Configurations

- W+R > N ⇒ guarantee strong consistency.
- N=2, W=2, and R=1 (primary-copy scenario with synchronous replication).
  - No matter from which replica the client reads, it will always get a consistent answer.

- N=2, W=1, and R=1 (primary-copy scenario with asynchronous replication).
  - In this case R+W=N, and consistency cannot be guaranteed.

- With N=3 and W=3 and only two nodes available, the system will fail to write.

AWS S3

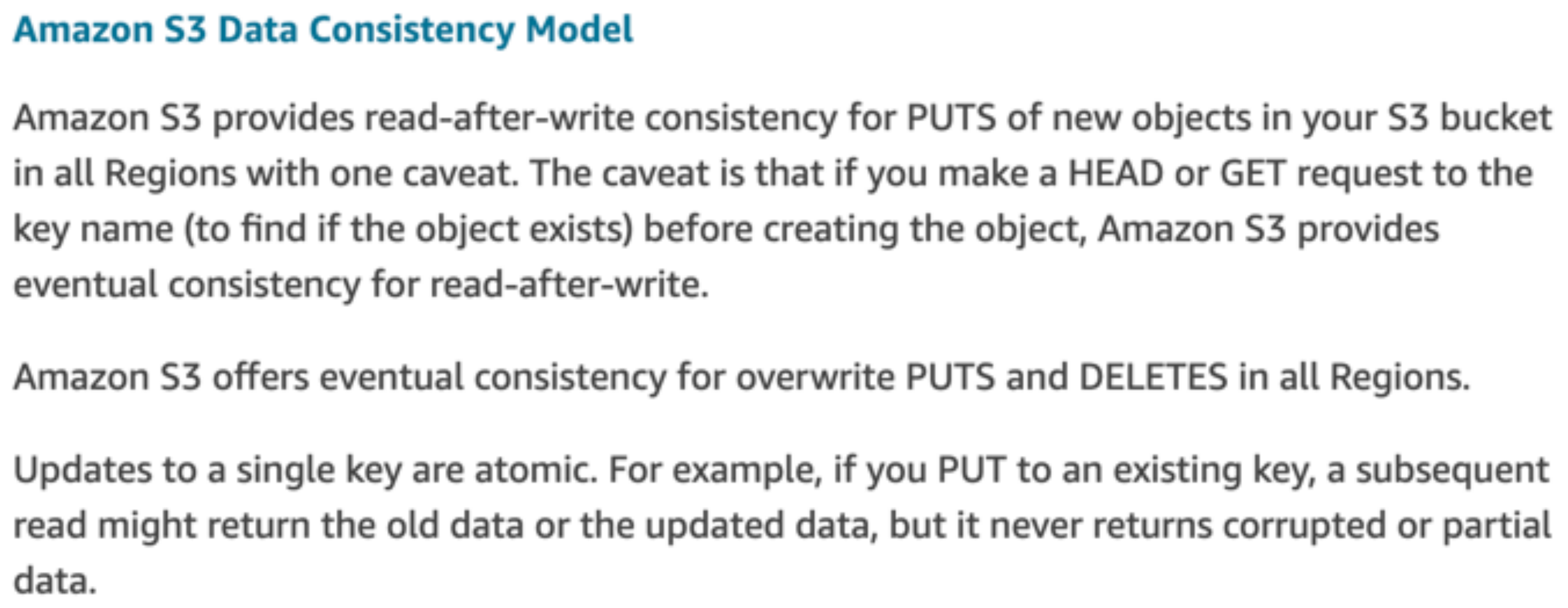


```
PUT /key-prefix/cool-file.jpg 200
GET /key-prefix/cool-file.jpg 200
```

```
PUT /key-prefix/cool-file.jpg 200
PUT /key-prefix/cool-file.jpg 200 (new content)
GET /key-prefix/cool-file.jpg 200
```

```
GET /key-prefix/cool-file.jpg 404
PUT /key-prefix/cool-file.jpg 200
GET /key-prefix/cool-file.jpg 404
```

WTF!?

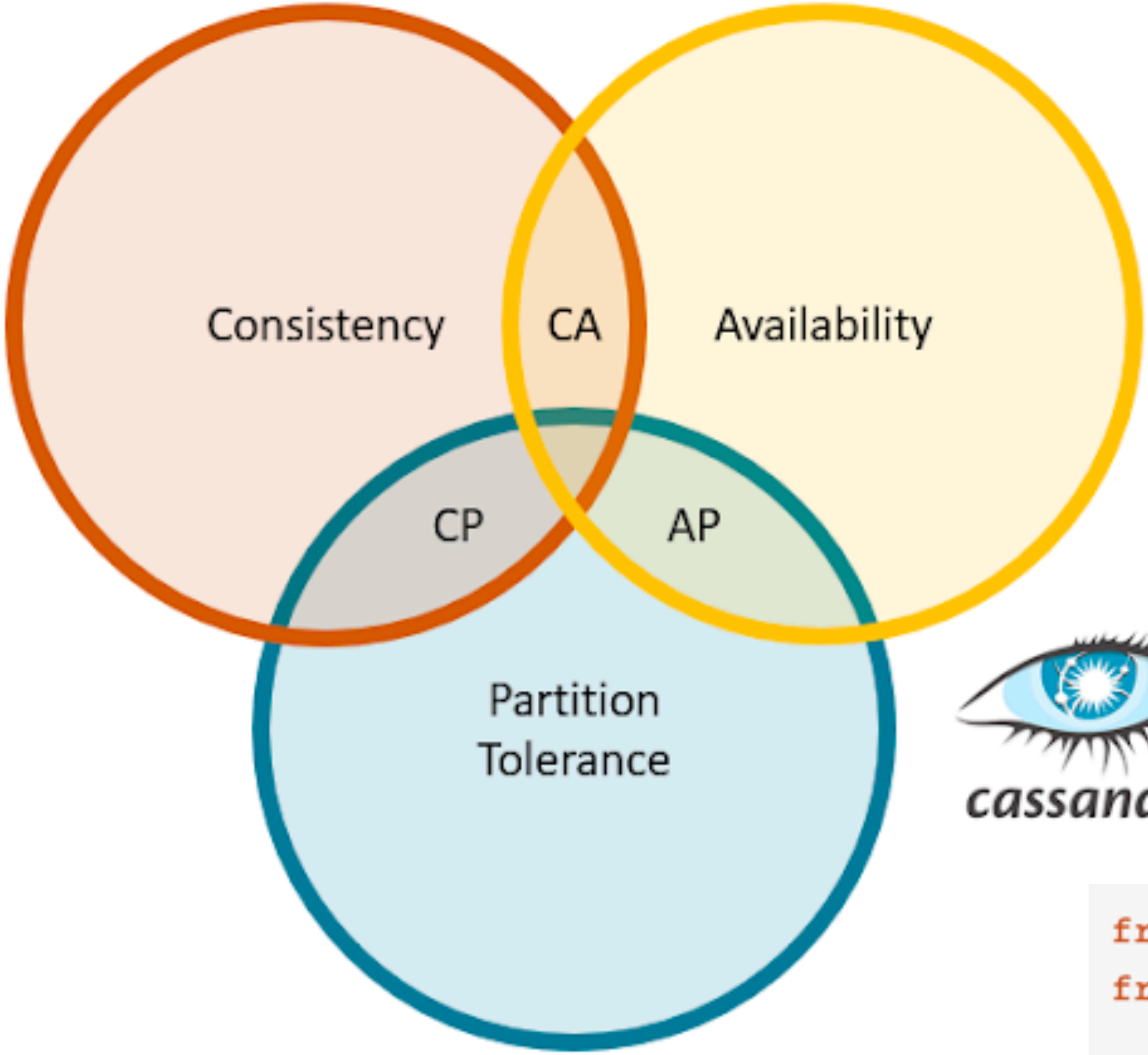


Amazon S3 Data Consistency Model

Amazon S3 provides read-after-write consistency for PUTS of new objects in your S3 bucket in all Regions with one caveat. The caveat is that if you make a HEAD or GET request to the key name (to find if the object exists) before creating the object, Amazon S3 provides eventual consistency for read-after-write.

Amazon S3 offers eventual consistency for overwrite PUTS and DELETES in all Regions.

Updates to a single key are atomic. For example, if you PUT to an existing key, a subsequent read might return the old data or the updated data, but it never returns corrupted or partial data.



```
from cassandra import ConsistencyLevel
from cassandra.query import SimpleStatement

query = SimpleStatement(
    "INSERT INTO users (name, age) VALUES (%s, %s)",
    consistency_level=ConsistencyLevel.QUORUM)
session.execute(query, ('John', 42))
```

Summary

- Overview of the foundations for cloud
- Distributed systems play an crucial role in getting things to work
- Take Jorn Janneck's course on distributed algorithms and learn the details.

Distributed Systems (7.5hp)

To give an introduction to the fundamental concepts of distributed systems, their properties and application in practice.

display basic knowledge of:
different types of distributed systems and their properties,
failure and recovery in distributed systems,
models and abstractions for distributed systems,
distributed models of logical time,
distributed algorithms and protocols,
and distributed state and computing.

be able to reason about properties of distributed systems,
be able to use concepts and abstractions to model distributed systems and to express their behavior,
be able to use fundamental distributed algorithms and protocols in managing resources, sharing state, maintaining distributed state, and coordinate distributed computation,
be able to apply the conceptual knowledge to the implementation of distributed algorithms on a variety of platforms.

be able to judge the suitability of models and platforms for distributed systems for a given problem,
display a basic understanding of the tradeoffs and limits of the concepts and techniques in distributed system design.