



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 switch 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.

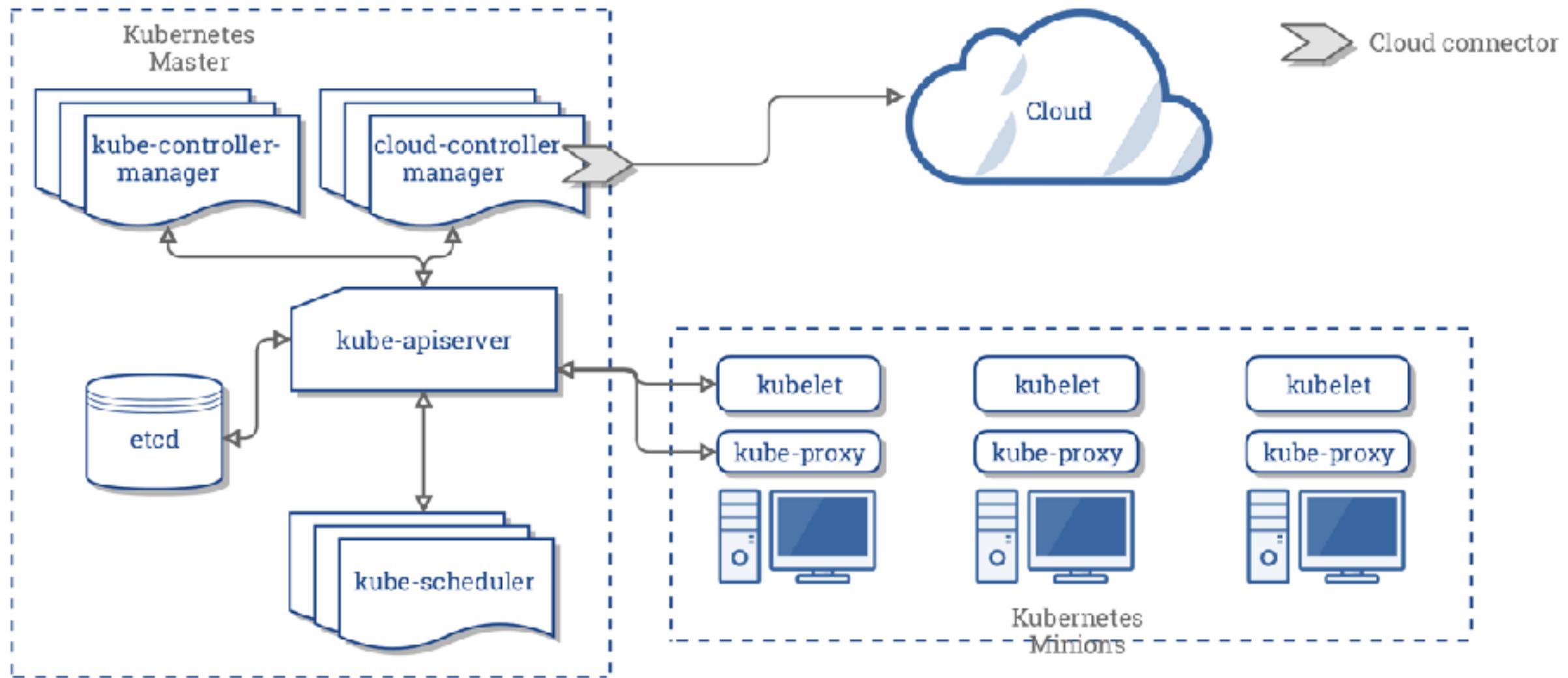
If any other prisoner goes into the room:

If the switch is down and he has not
previously moved it up:

Move it up.

Otherwise leave it.

Last week: k8 architecture





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.

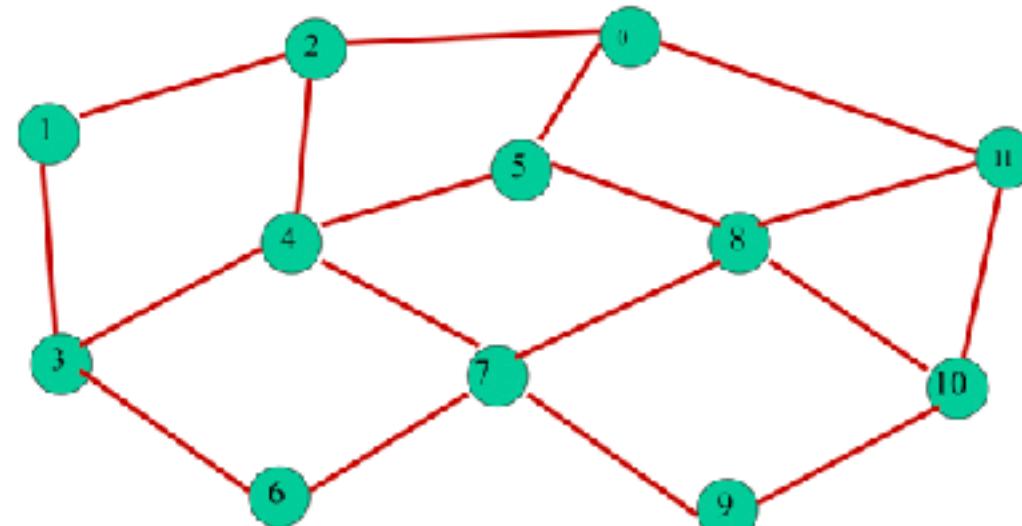
Leslie Lamport
email communication
1987

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

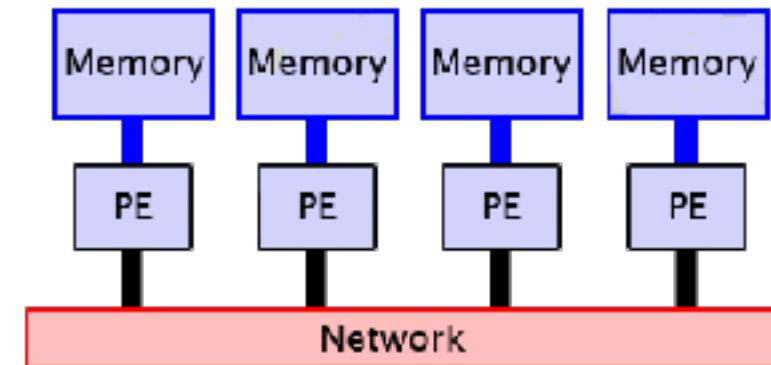
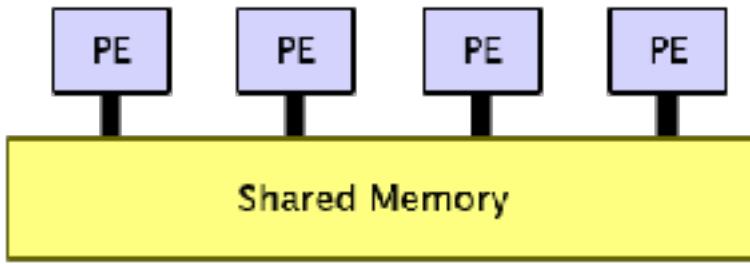
A Distributed System

- 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

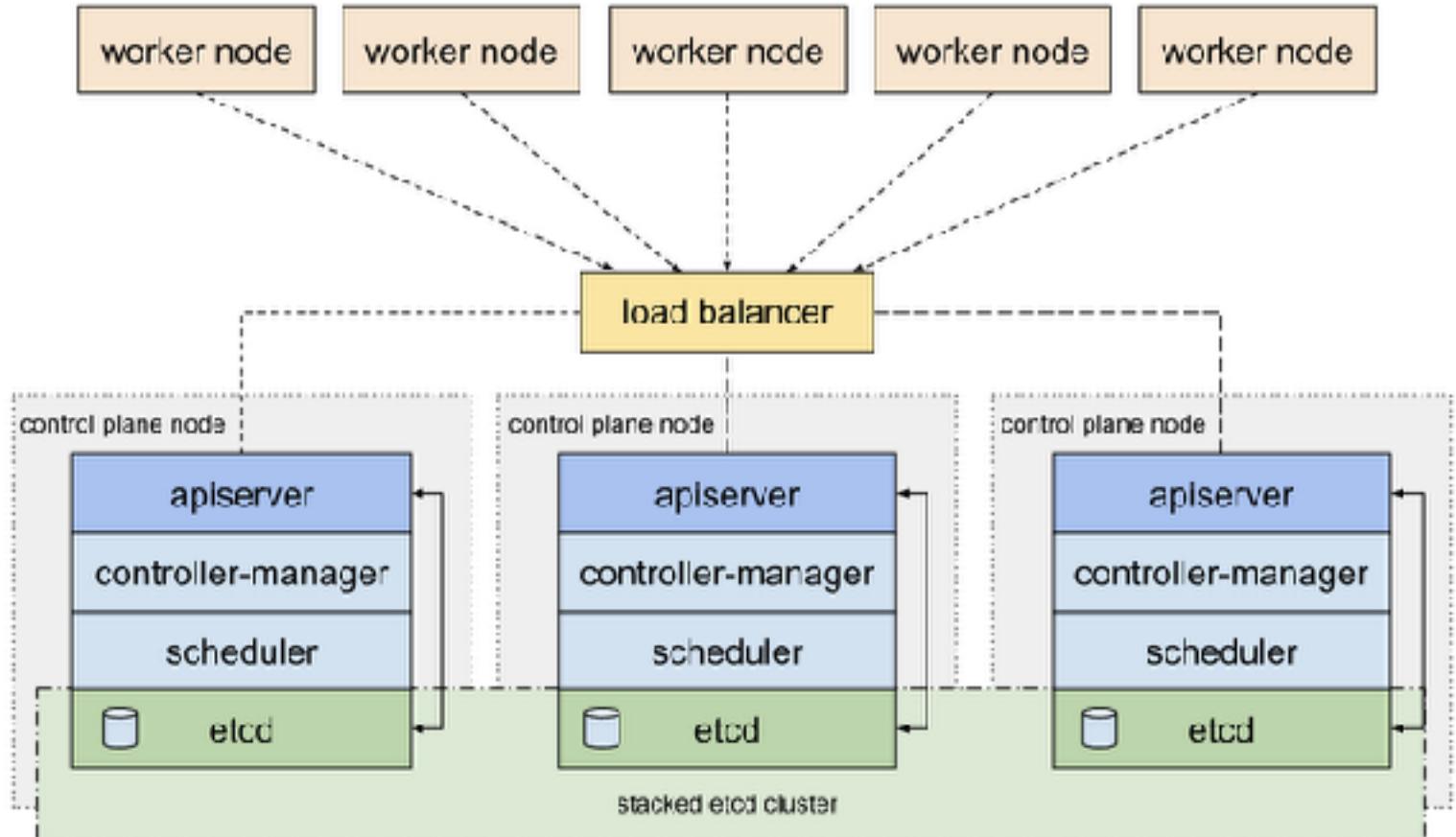
Partially synchronous systems:

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.2.1 [Computer-Communication Networks]: Network Protocols-protocol architecture; C.1.4 [Computer-Communication Networks]: Distributed Systems-distributed applications; distributed database; network operating systems; C.4 [Performance of Systems]: Reliability, Availability, and Serviceability; F.1.2 [Computations 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 N00014-81-K-0134, 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 2138, 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

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

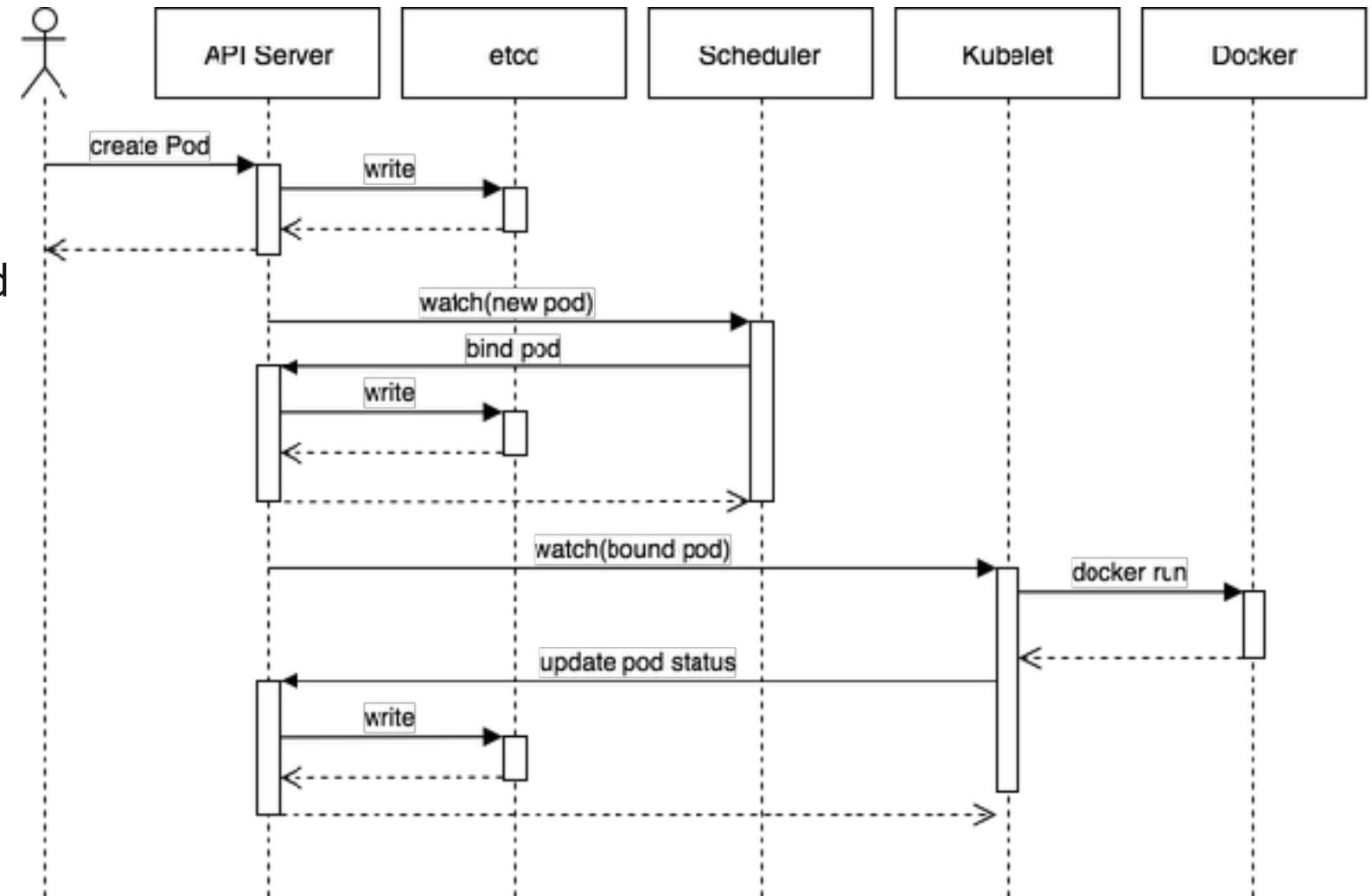
© 1985 ACM 0004-5411/85/0400-0374 \$00.75

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.

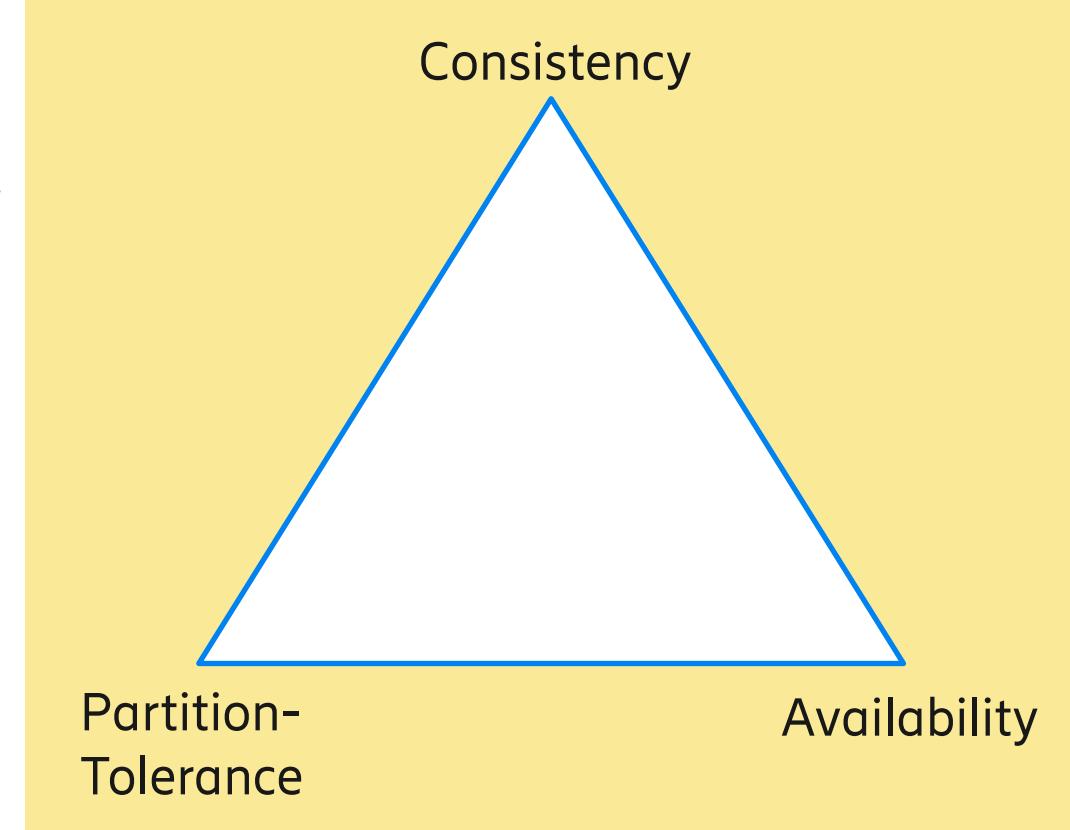
A read must return the latest written value by any client.

Availability

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

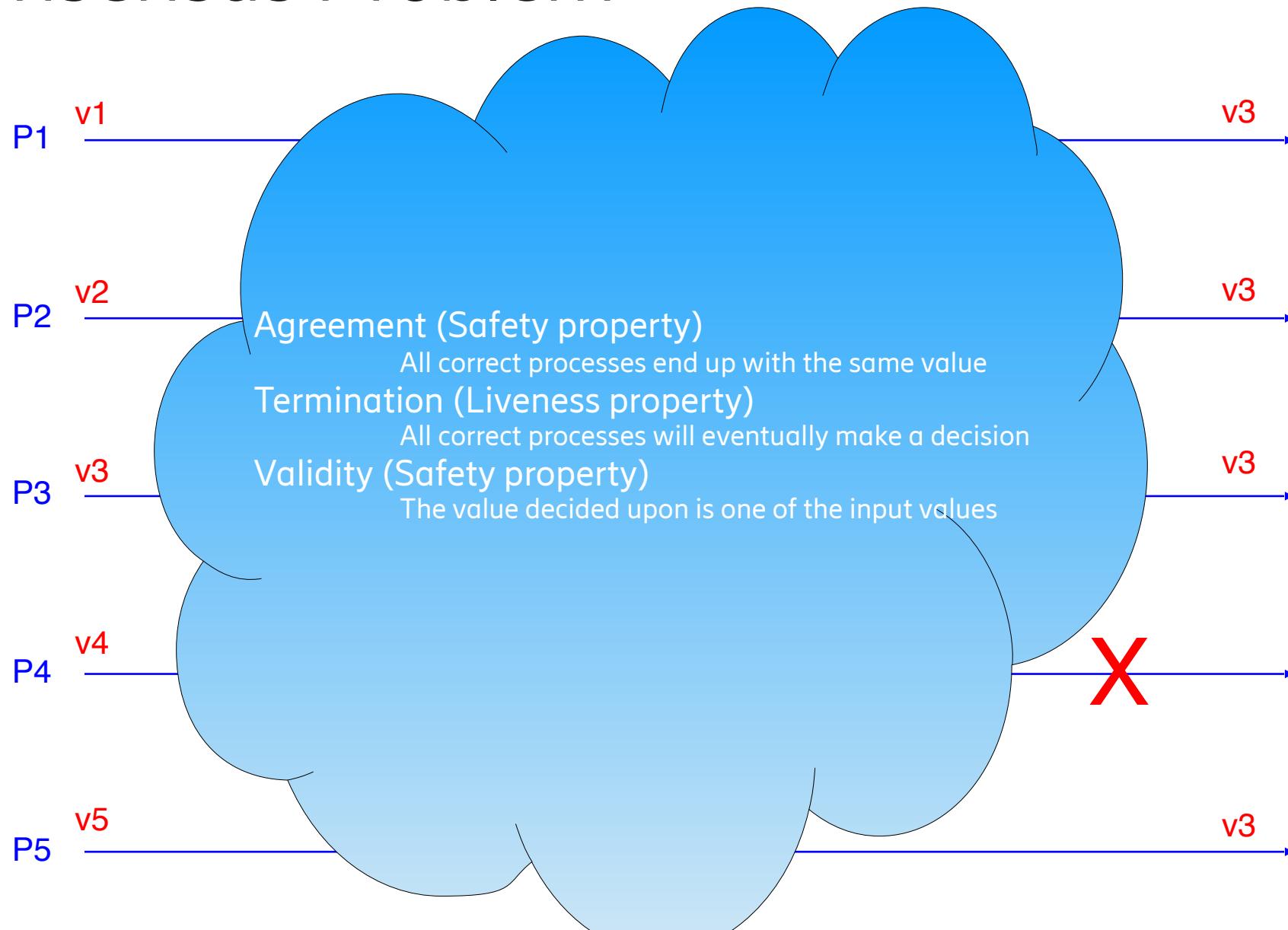
Partition-Tolerance

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





The Consensus Problem





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?

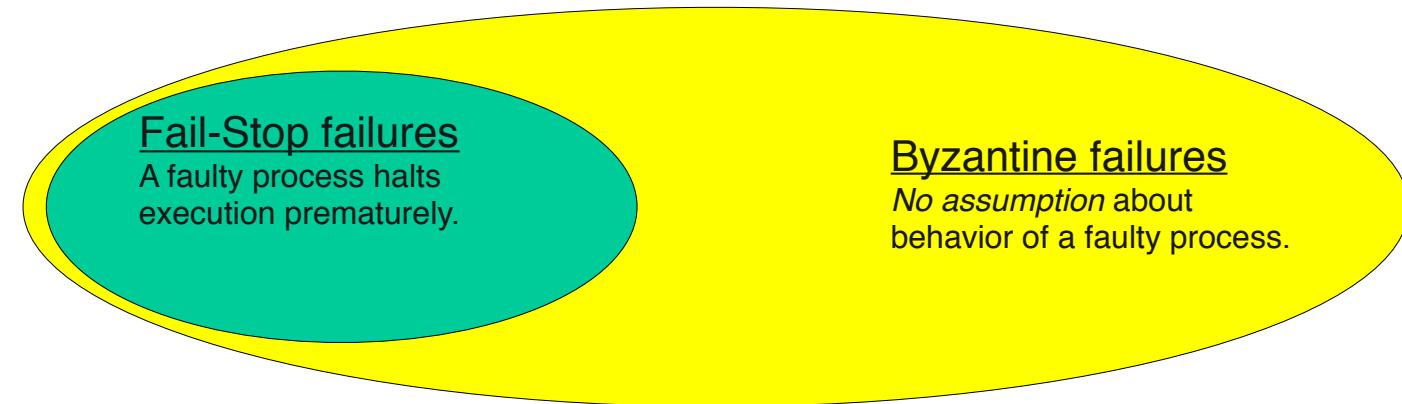


A Naïve Protocol

- 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

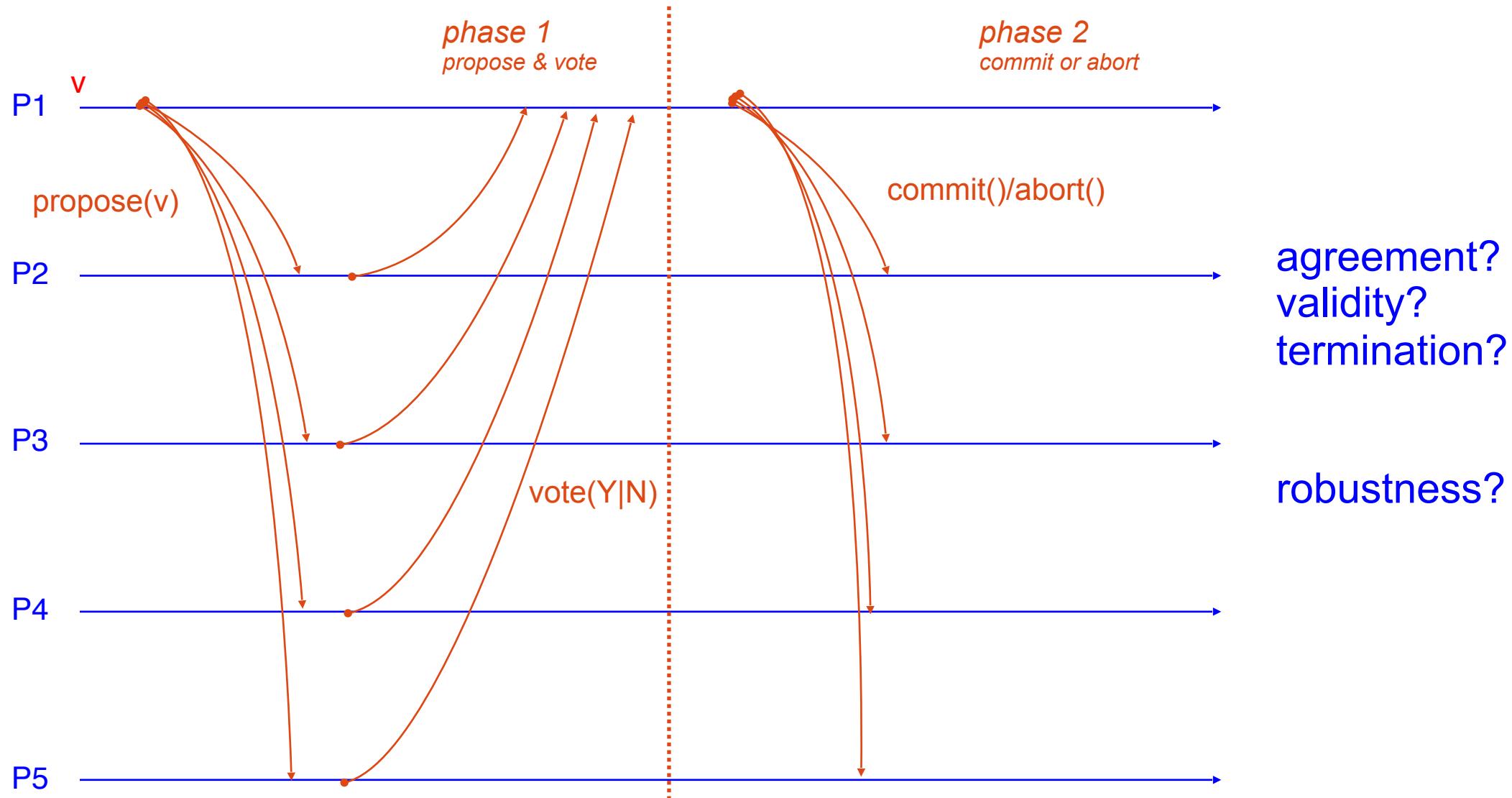


Some failure models (not a complete list)

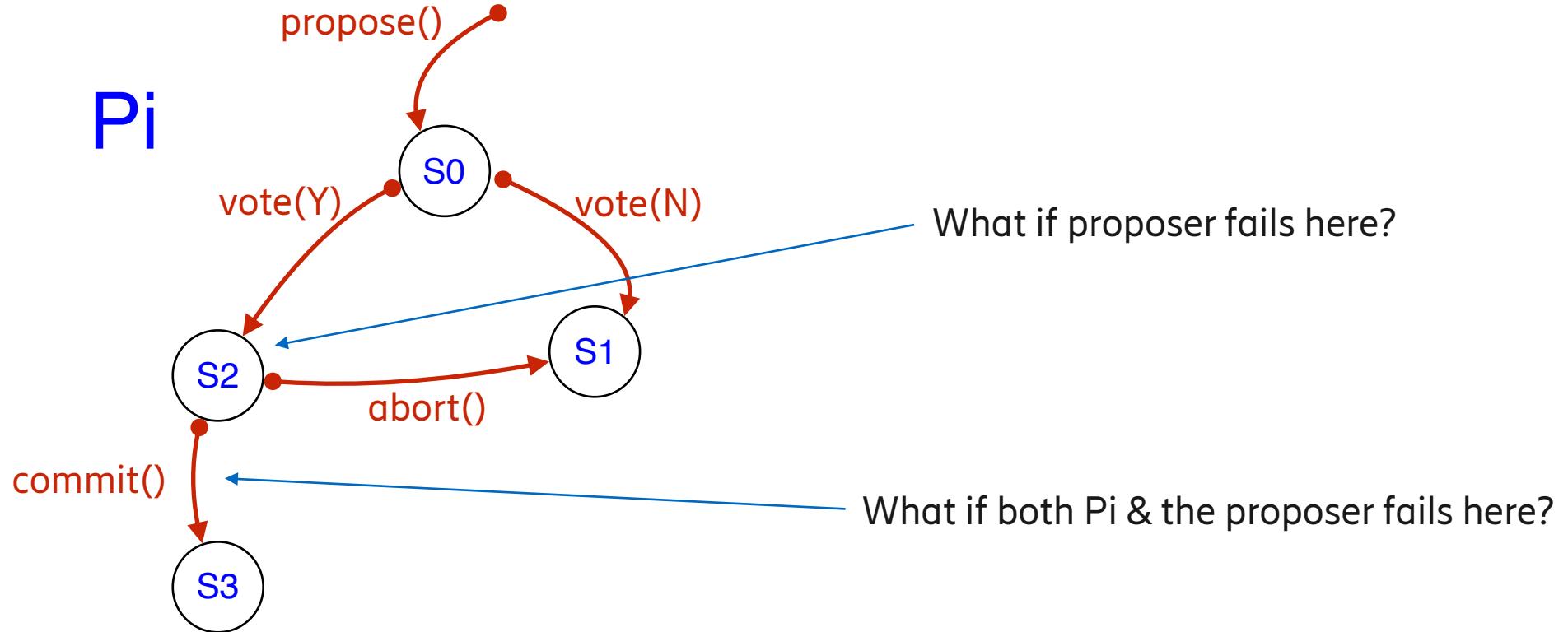
- Fail-Stop (Crash-stop)
 - A processor stops, and never starts again
- Byzantine
 - A processor behaves adversarially, maliciously.
- Crash-recover
 - Well, it crashes and then restarts sometime later
- Omission
 - Doesn't respond to input (or infinitely late)
- Timing
 - Correct response, but outside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



2PC (Two-Phase Commit)



2PC (Two-Phase Commit)



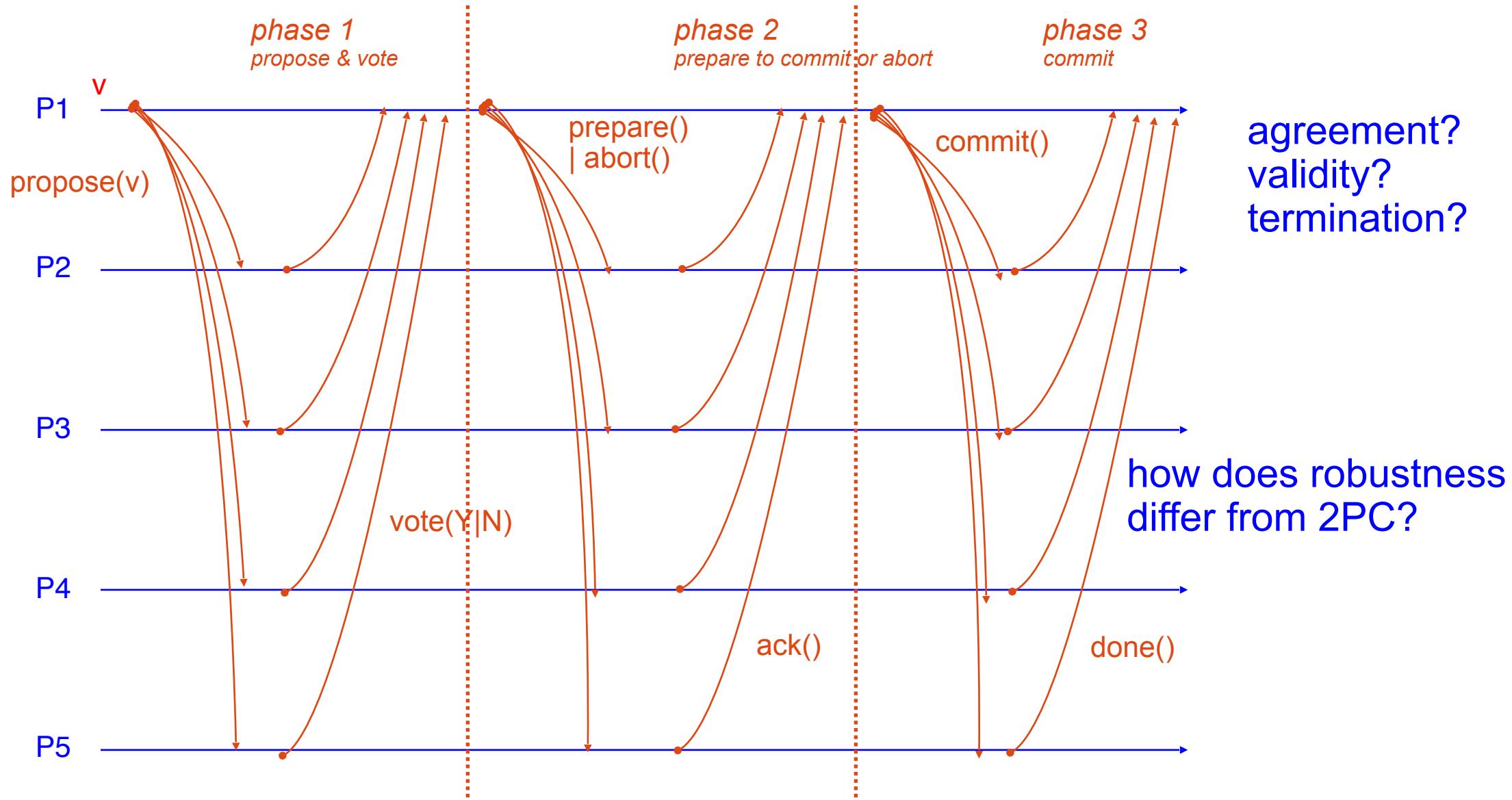
S1 - aborted: Voted N, received abort()

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

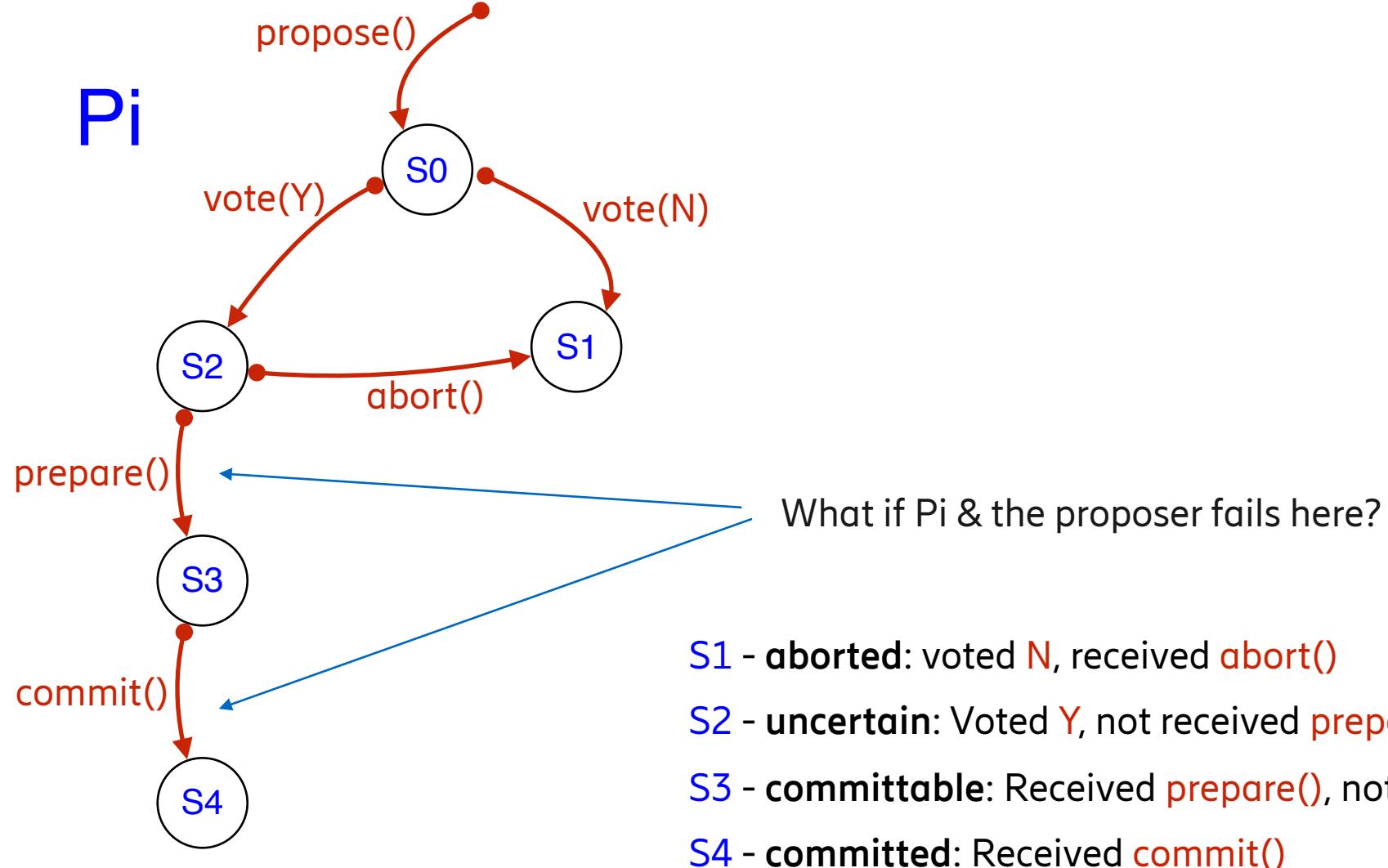
S3 - committed: Received commit()



3PC (Three-Phase Commit)



3PC (Three-Phase 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

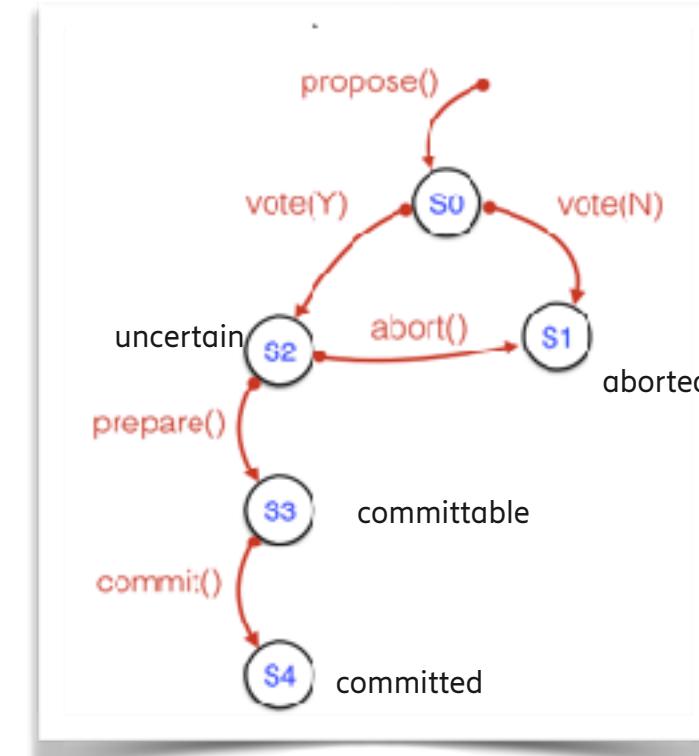
else if all processes in state **uncertain**

 send **abort()** to all

else if some process in state **commitable**

 send **prepare()** to all process in state **uncertain**

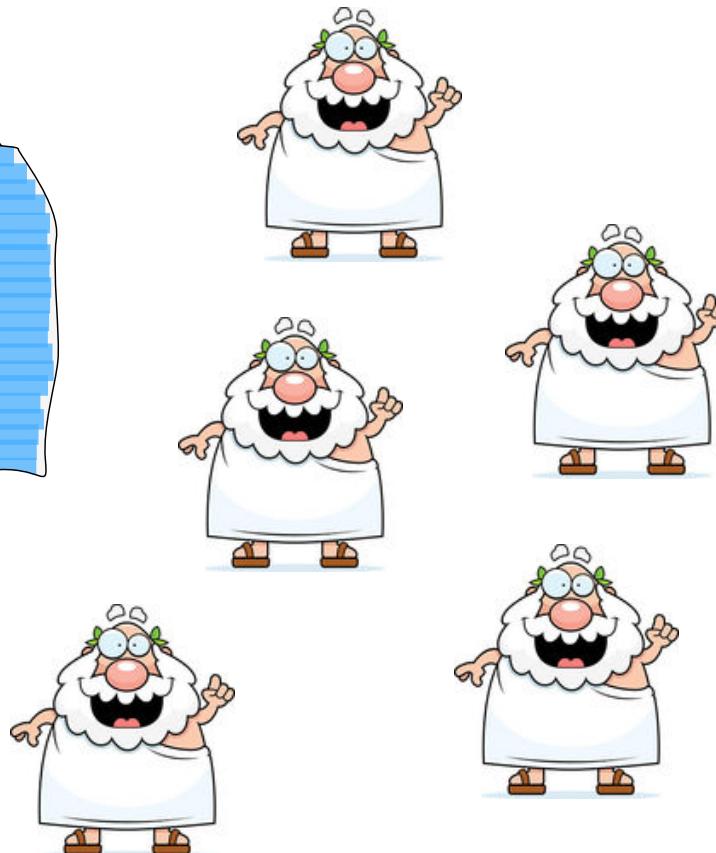
 wait for **ack()** and then send **commit()** to all



≡



Paxos and Chubby



The Part-Time Parliament

LESLIE LAMPORT
Digital Equipment Corporation

Received August 14, 1994; revised October 1, 1995. This paper revises the author's previous work on the design of Paxos and Chubby to extend its scope to the part-time property of its part-time legislature. The legislature maintains constant copies of the partitioned state, despite their frequent changes from the choices and the frequent failure of their members. The Paxos protocol's protocol provides a new way of implementing the state-machine approach to the design of distributed systems.

Keywords and Subject Descriptors: 1.3.4 [Computer Communications Networks]: Network Protocols—Reliable; 1.3.5 [Computer Systems]: 1.3.5 [Operating systems]; 1.3.6 [Distributed Systems]; 1.3.7 [Administrative Data Processing]; Government

General Terms: Design, Reliability

Additional Key Words and Phrases: State machine, 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 cold work in the Greek Isles and cannot be reached. I was asked to prepare it for publication.

The author appears to be an archaeologist with only a passing interest in computer science. This is unfortunate, even though the chosen subject, Paxos civilization, is destined to be the most important example of scientific inquiry in computer science. An excellent model for how to implement a distributed computer system in an asynchronous environment. Indeed, some of the refinements the Paxos made to this protocol appear to be unknown in the systems literature.

The author does give a brief discussion of the Paxos algorithm's relevance to distributed computing in Section 4. Computer scientists will probably want to skip that section but, for those that do, they might want to read the explanation of the algorithm for computer scientists by Lamport [1998]. The algorithm looks very familiarly by De Prisco et al. [1997]. I have added further comments on the relation between the original protocols and more recent work at the end of Section 4.

Krish Namkoong
University of California, San Diego

Author's address: System Research Center, Digital Equipment Corporation, 100 Taylor Street, Palo Alto, CA 94301.

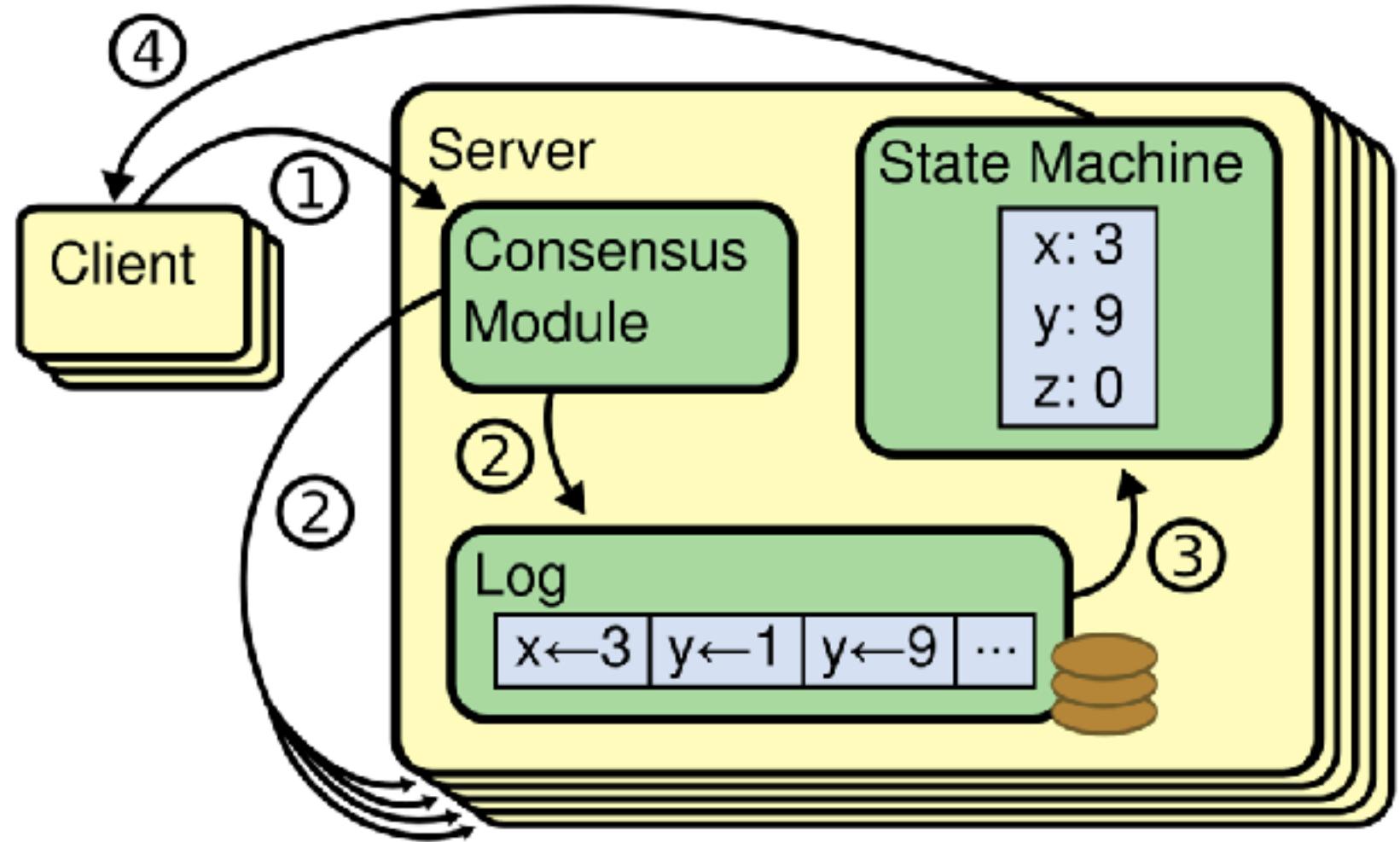
Permission to copy without fee all or part of this material is granted, provided that the copies are not made or distributed for commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise or to republish, requires a fee and/or specific permission.

Took 8 years to review...

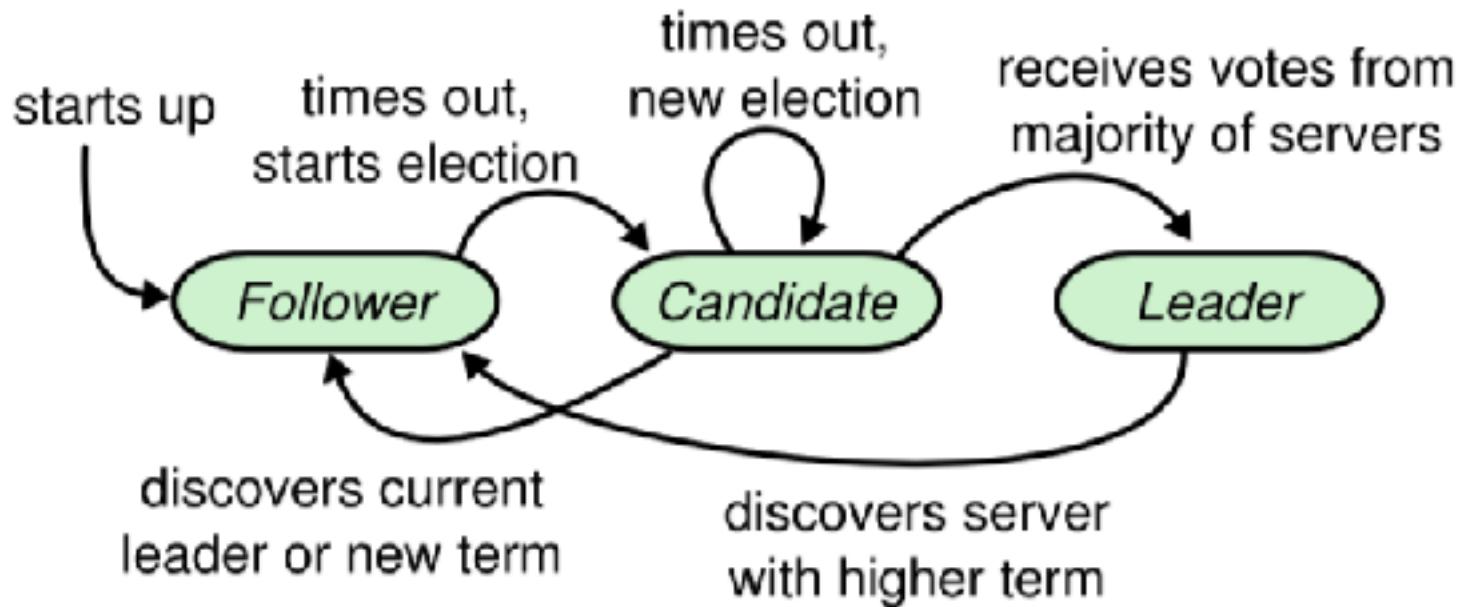


Raft

- Designed to be understood



Raft



Consensus using etcd



```
"""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
```

```
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')

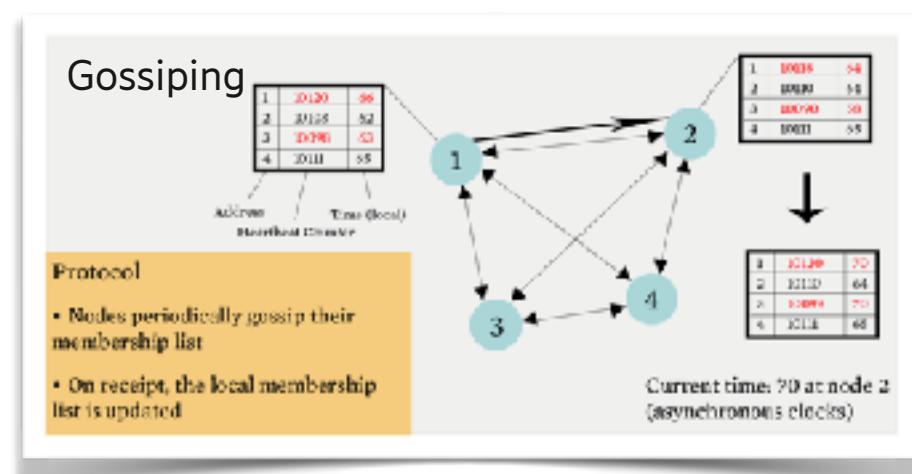
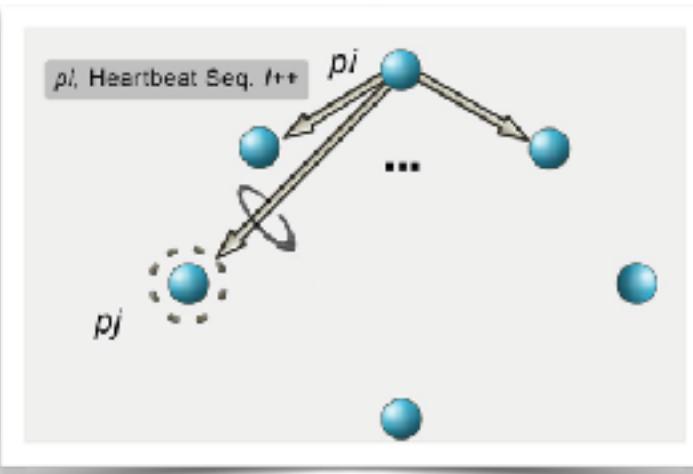
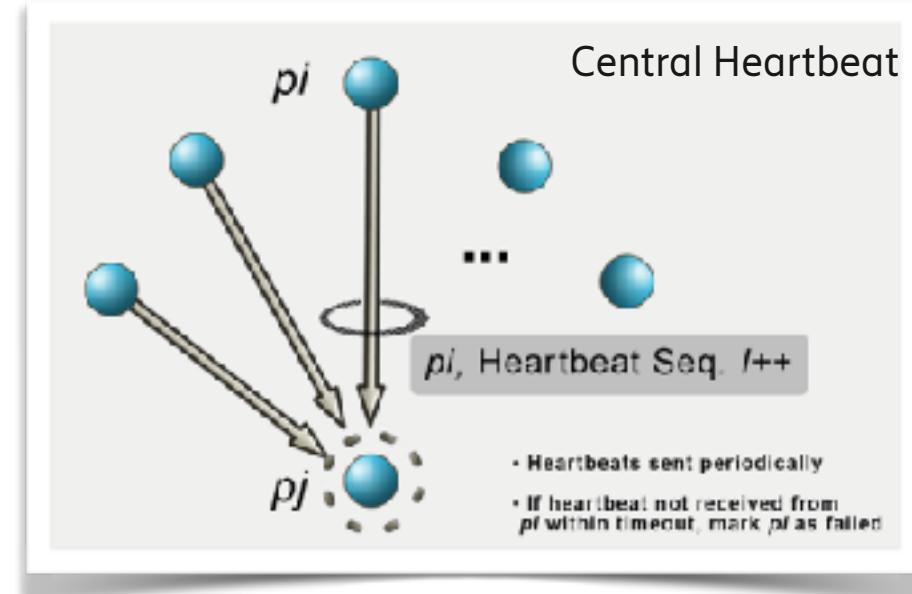
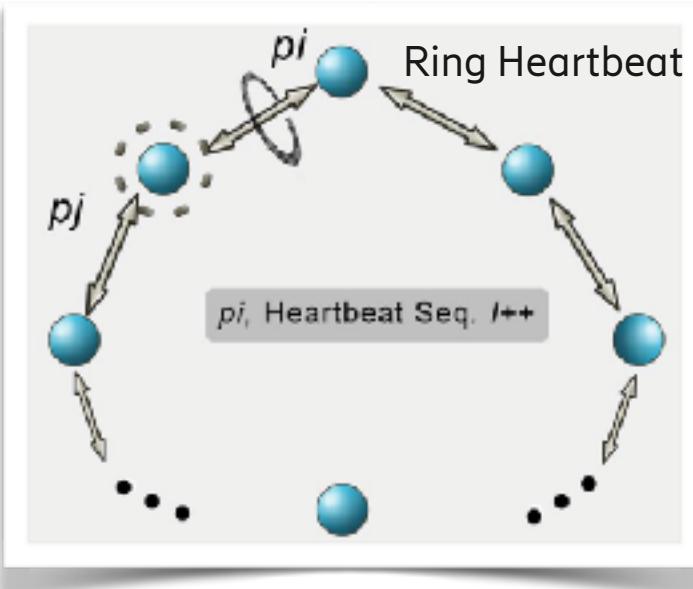
            election_event = Event()
            def watch_cb(event):
                if isinstance(event, etcd3.events.DeleteEvent):
                    election_event.set()
            watch_id = client.add_watch_callback(LEADER_KEY, watch_cb)

            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



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.4 [Computer-Communication Networks]: Distributed Systems—network operating systems; D.4.4 [Operating Systems]: Concurrency Management—synchronization; D.4.4 [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 encamped 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

This research was supported in part by the National Aeronautics and Space Administration under contract NAS(1542) Mod. 2, the Ballistic Missile Defense System Command under contract DASG08-78-C-036, and the Army Research Office under contract DAAG29-79-C-0392.

Authors' address: Computer Science Laboratory, SRI International, 333 Ravenswood Avenue, Menlo Park, CA 94025.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

© 1982 ACM 0164-0925/82/0706-0015\$01.50

ACM Transactions on Programming Languages and Systems, Vol. 4, No. 3, July 1982, Pages 361-381.



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



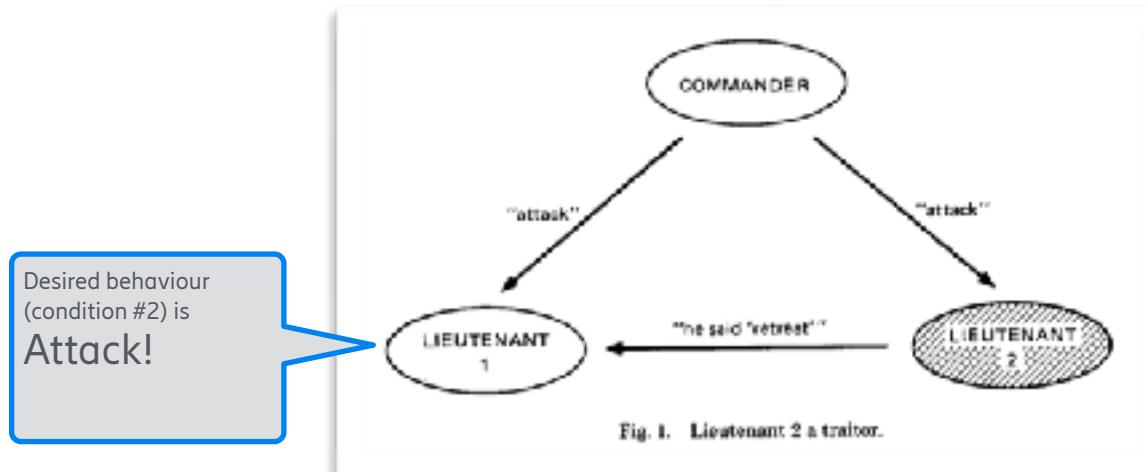
Byzantine General Problems

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

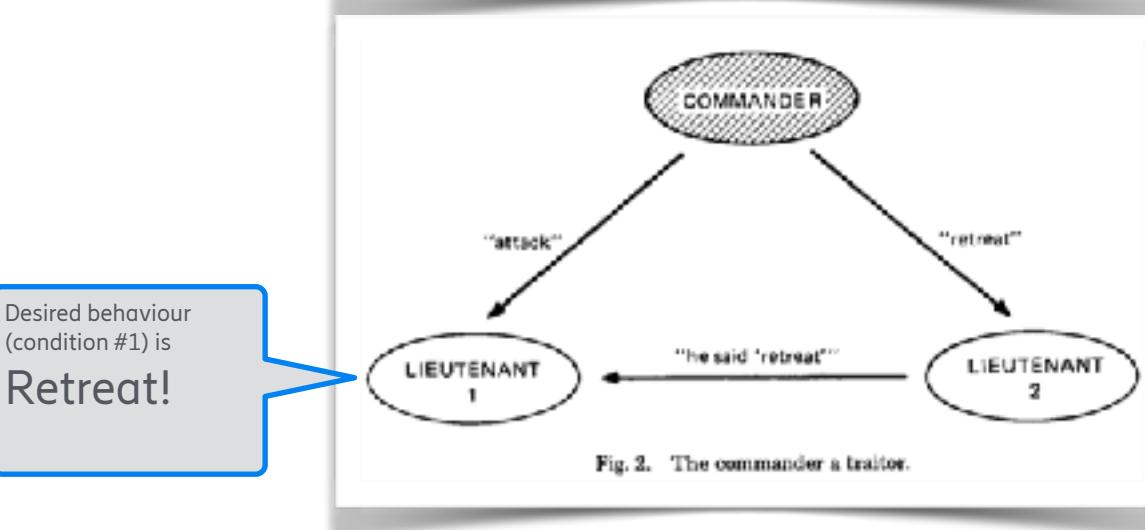
The Byzantine Generals



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! \Rightarrow impossible to fulfil requirements 1 & 2





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

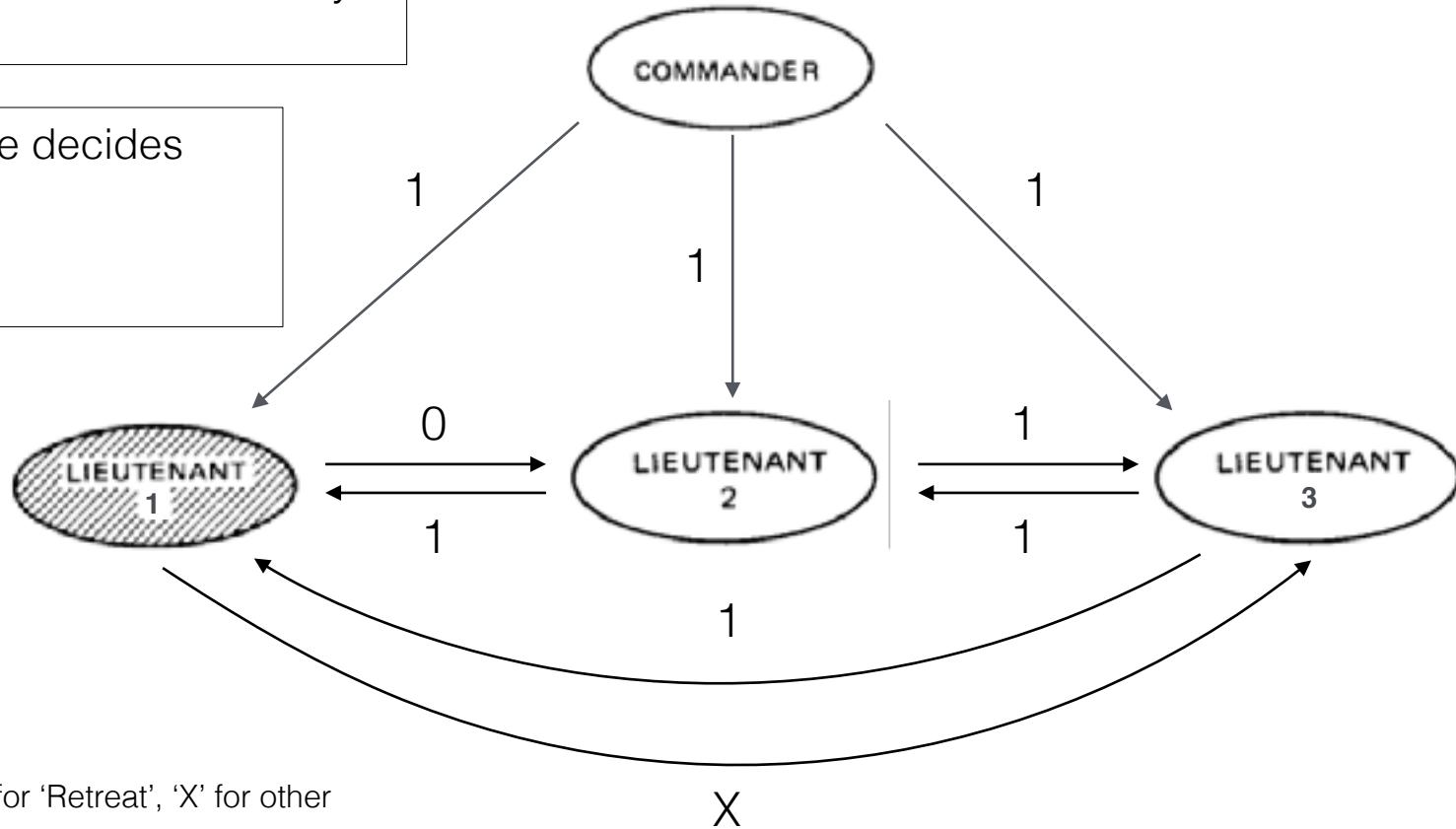
$\Rightarrow n > 3m + 1$ for all algorithms that solve the
Byzantine Generals problem

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.



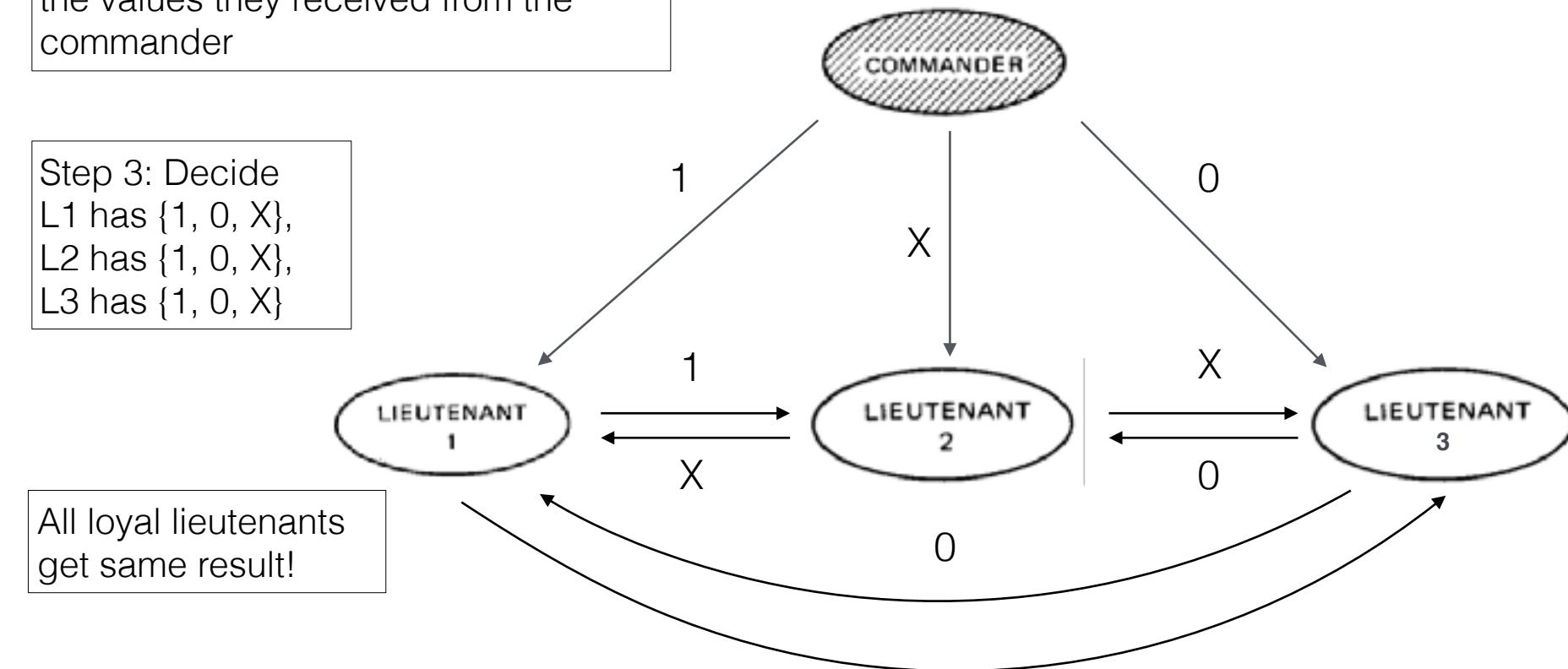
A Traitor Commander with Loyal Lieutenants

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!

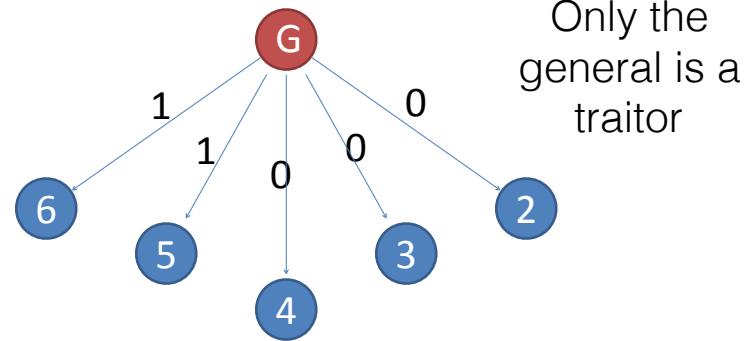


Use '1' for 'Attack', or '0' for 'Retreat', 'X' for other

Let's try this live



The commander sends a message to all lieutenants



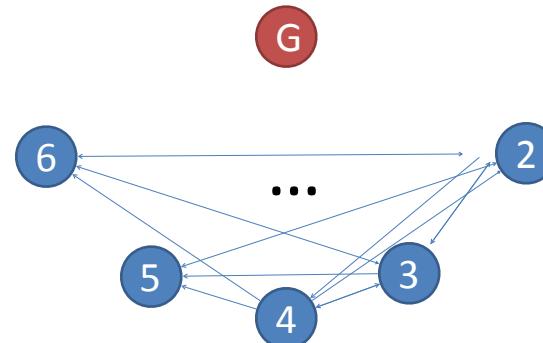
Only the general is a traitor

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}

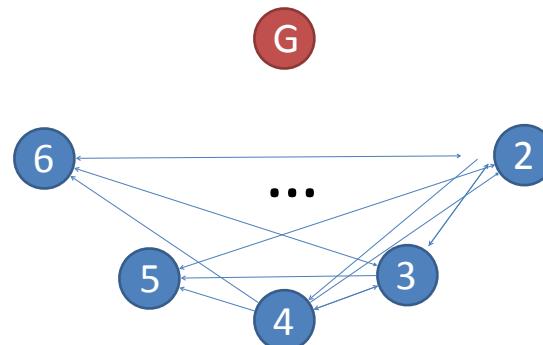
The message {0,12} is sent to all other nodes by node 2, and so on



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} {1,145} {1,165}	{0,126} {0,136} {0,146} {1,156}

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

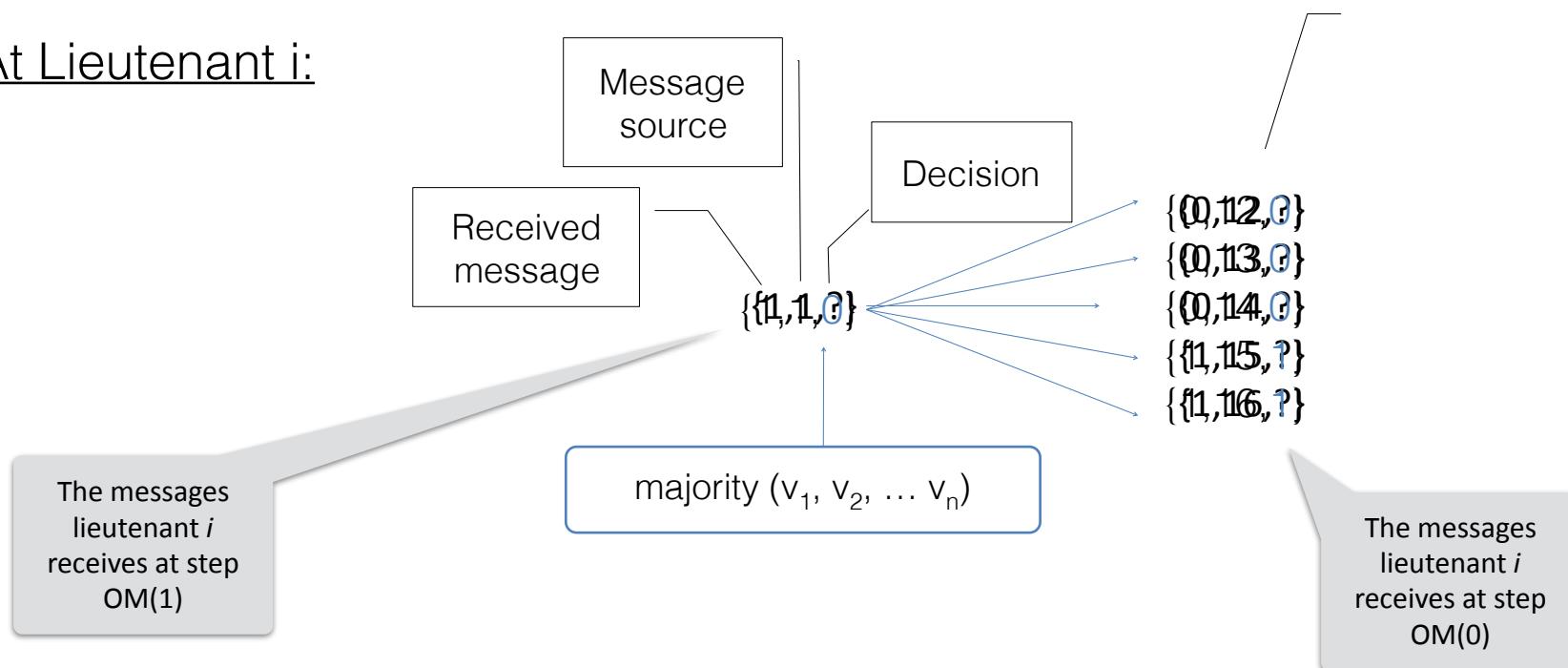


The Decision Making

All received messages are kept and organised in a tree structure

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

At Lieutenant i:



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 Gross,
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

Introduction

The concept of time is fundamental to our way of thinking. It is derived from the more basic concept of the order in which events occur. We say that something happened at 3:15 if it occurred after our clock read 3:13 and before it read 3:16. The concept of the temporal ordering of events provides our thinking about systems. For example, in an air traffic reservation system we specify that a request for a reservation should be granted if it is made before the flight is 3:15. However, we will see that this concept must be carefully reexamined when considering events in a distributed system.

General terminology and notation in writing or reading of all papers in this journal is guided by the rules of a widely used formal language for this purpose that ACM's computer section has adopted. This language is based on the publication of its rules of syntax and the fact that reprinting privileged work granted by permission of the Association for Computing Machinery. To obtain or reprint a figure, table, or other material from any of the trials work means specific permission to do so explicitly. It is not sufficient to cite the reference.

This work was supported by the Advanced Research Projects Agency of the Department of Defense and Rome Air Development Center. It was monitored by Rome Air Development Center under contract number F30602-73-C-0074.

Author's address: Computer Science Laboratory, SRI International, 3241 Malvern Avenue, Suite 100, Menlo Park, CA 94025.
© 1978 ACM 0001-0732/78/0303-0001\$00.75

644

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 multiprocessor 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.

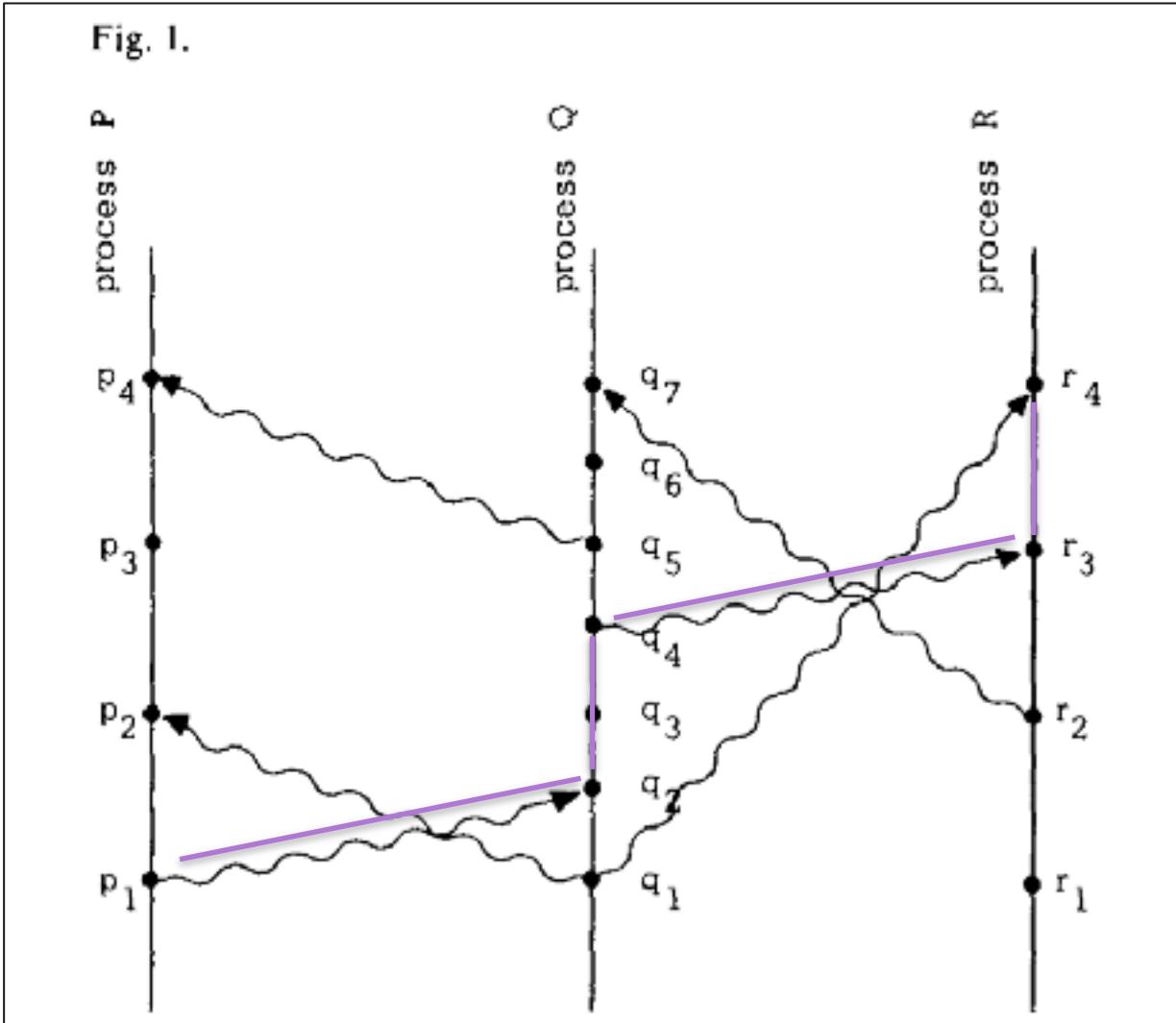
In this paper, we discuss the partial ordering defined by the “happened before” relation, and give a distributed algorithm for extending it to a consistent total ordering of all the events. This algorithm can provide a useful mechanism for implementing a distributed system. We illustrate its use with 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 avoided by introducing real, physical clocks. We describe a simple method for synchronizing these clocks, and derive an upper bound on how far out of synchrony they can drift.

The Partial Ordering

Most people would probably say that an event *a* happened before an event *b* if *a* happened at an earlier time than *b*. They might justify this definition in terms of physical theories of time. However, if a system is to meet a specification correctly, then that specification must be given in terms of events observable within the system. If the specification is in terms of physical time, then the system must contain real clocks. Even if it does contain real clocks, there is still the problem that such clocks are not perfectly accurate and do not keep precise physical time. We will therefore define the “happened before” relation without using physical clocks.

We begin by defining our system more precisely. We assume that the system is composed of a collection of processes. Each process consists of a sequence of events. Depending upon the application, the execution of a subroutine in a computer could be one event, or the execution of a single machine instruction could be one

The 'Happened-before' Relation



For example:

$p_1 \rightarrow r_4$

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 $C_i(a)$ assigns a number to an event a in process P_i
 - Could be a number or actual time
 - The Clock Condition: $a \rightarrow b \Rightarrow C(a) < C(b)$ (the opposite is not true, why?)

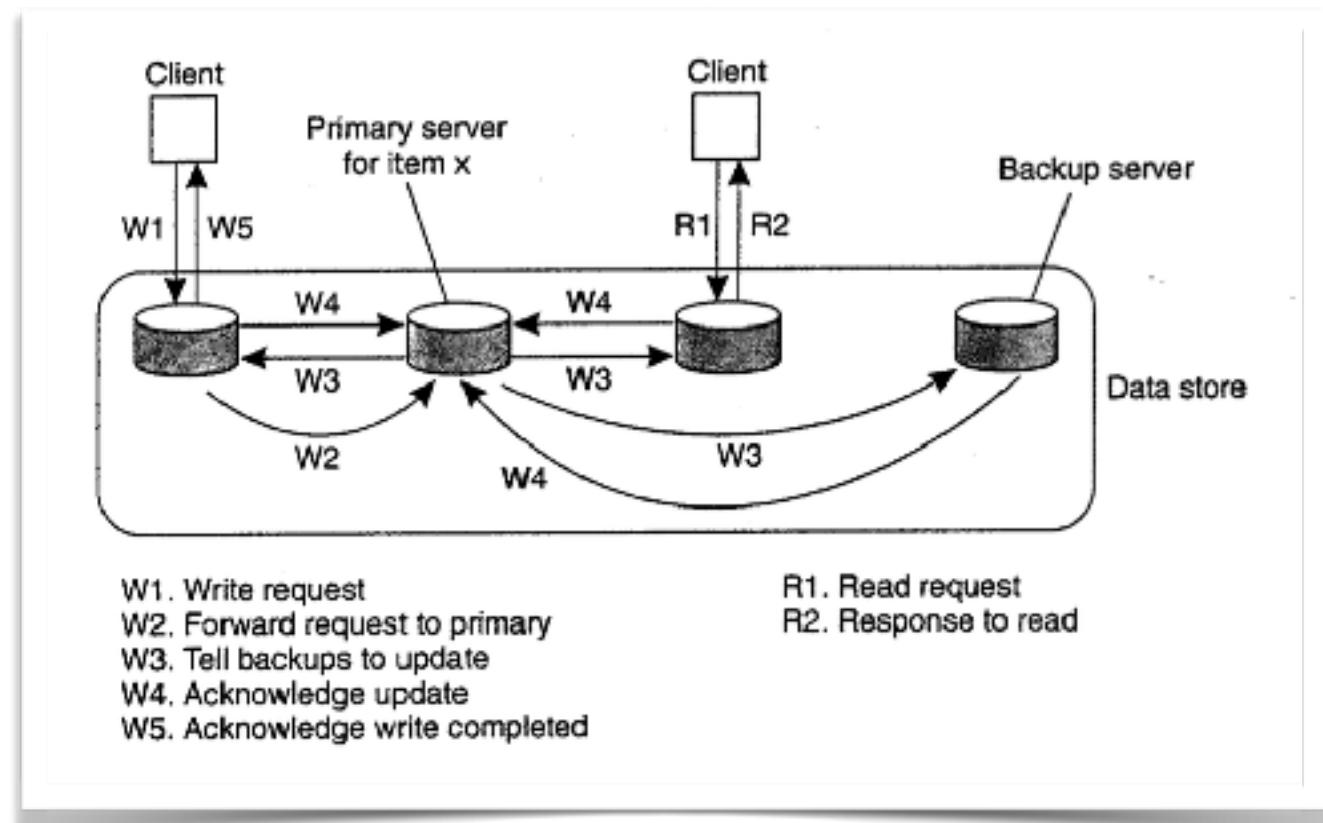


Total Ordering

- Use system of clocks satisfying the Clock Condition to place a total ordering of all events denoted " \Rightarrow "
 - Simply order events by their time $C(a)$ 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 $C_i(a) < C_j(b)$ or $C_i(a) = C_j(b)$ 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





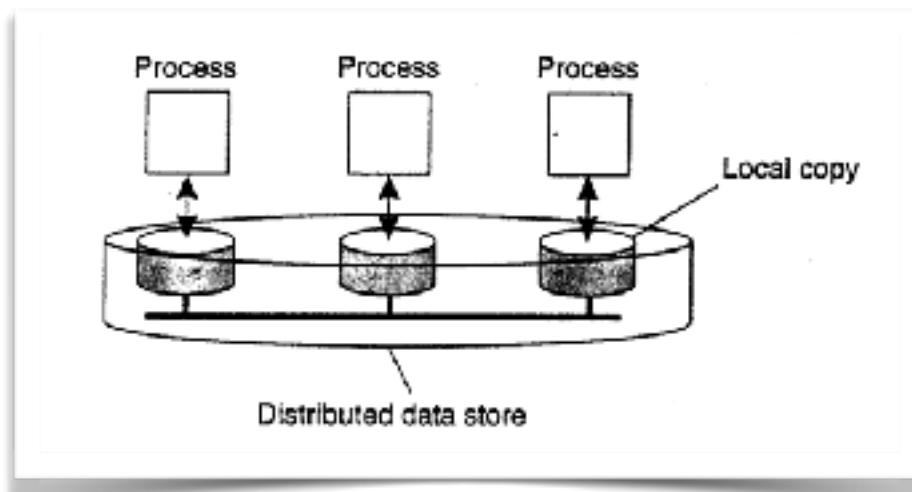
Example: Facebook

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



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!

P1: $\mathcal{W}(x)a$
P2: $\mathcal{R}(x)a$

(a)

P1: $\mathcal{W}(x)a$
P2: $\mathcal{R}(x)\text{NIL}$ $\mathcal{R}(x)a$

(b)

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.

P1: $W(x)a$

P2: $W(x)b$

P3: $R(x)b$ $R(x)a$

P4: $R(x)b$ $R(x)a$

(a)

correct

P1: $W(x)a$

P2: $W(x)b$

P3: $R(x)b$ $R(x)a$

P4: $R(x)a$ $R(x)b$

(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:	$R(x)a$
P3:	$W(x)b$
P4:	$R(x)b$

(a)

incorrect

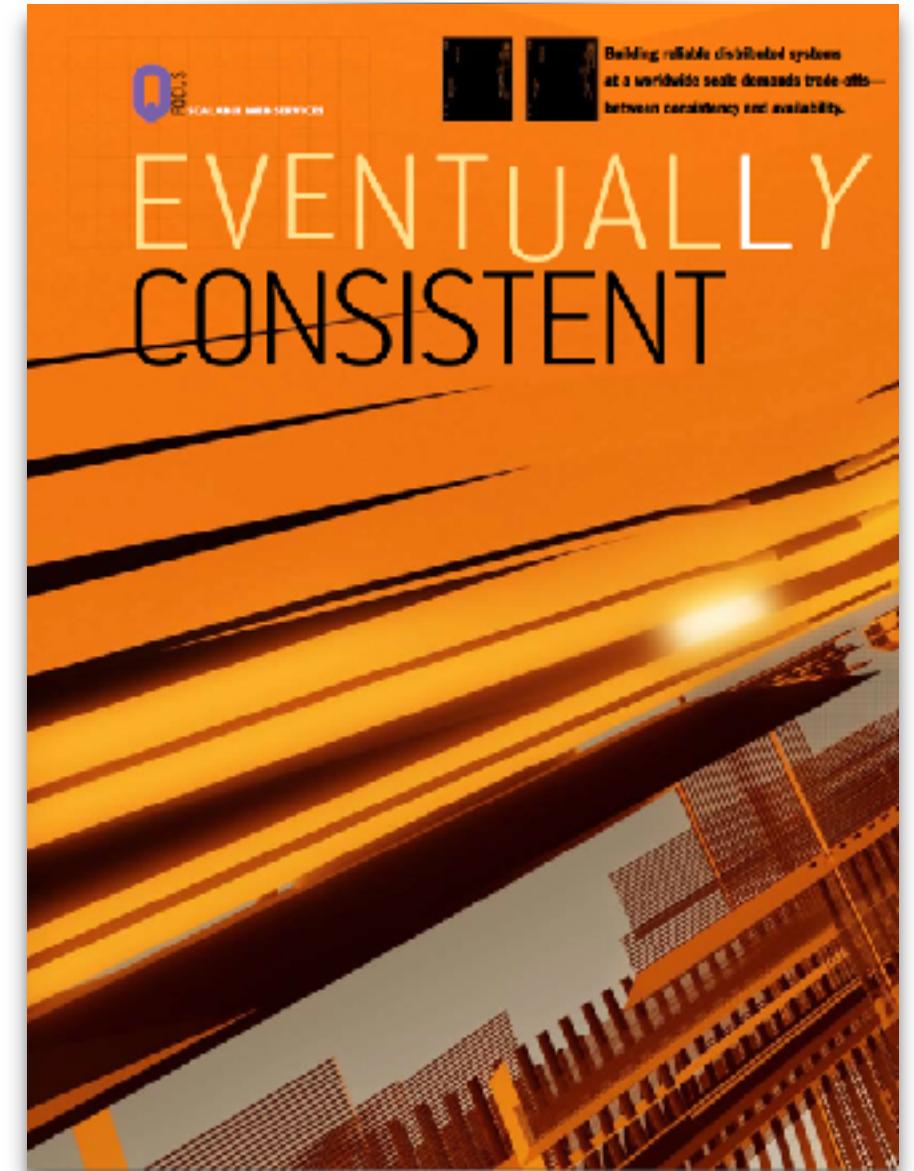
P1:	$W(x)a$
P2:	$W(x)b$
P3:	$R(x)b$
P4:	$R(x)a$

(b)

correct

Eventual Consistency

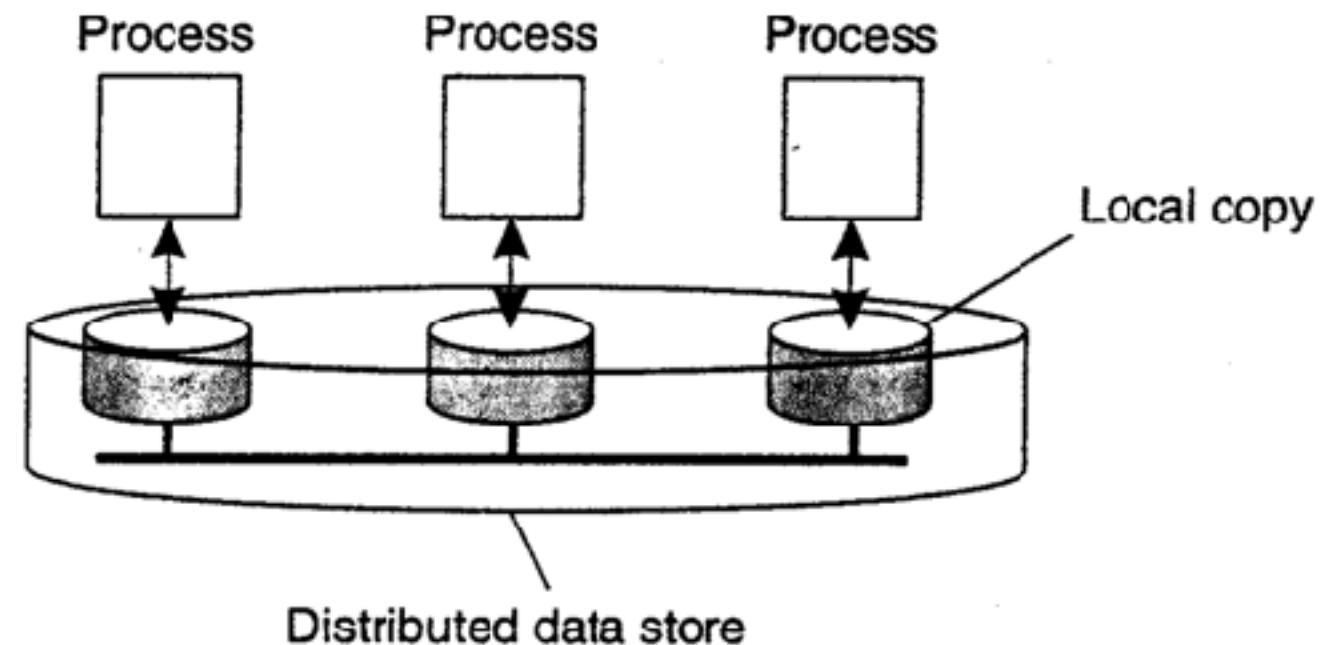
- The only requirement is that all replicas will eventually be the same.
- DNS is a well known example. Updates to a name are distributed according to a configured pattern and in combination with time-controlled caches; eventually, all clients will see the update.



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

- $W+R > N \Rightarrow$ 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!?

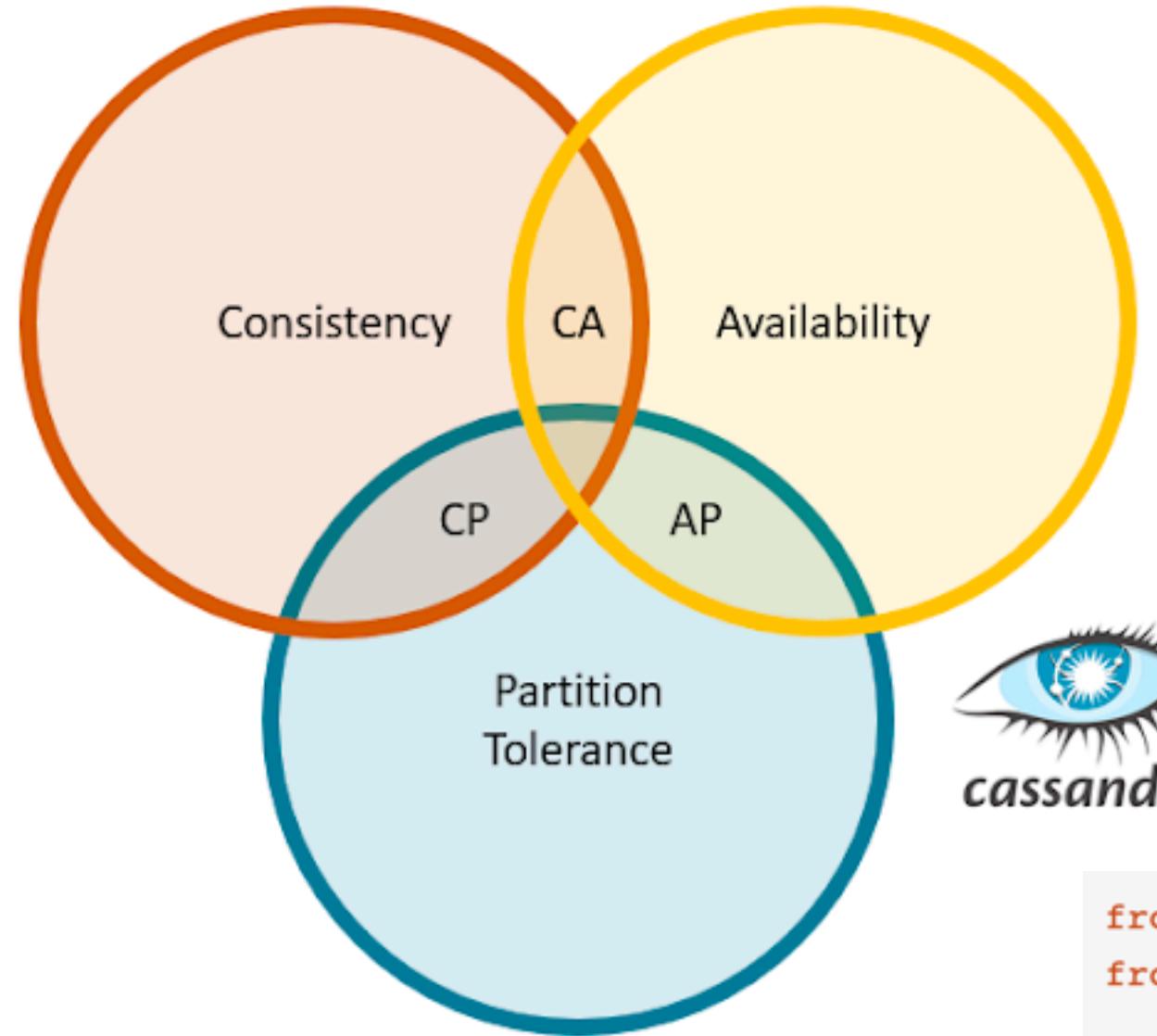


Amazon S3 Data Consistency Model

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.