# Coordination and agreement

*Distributed software systems*
*CdLM Informatica - Università di Bologna*

## Agenda

Consensus and related problems: eg. consistency

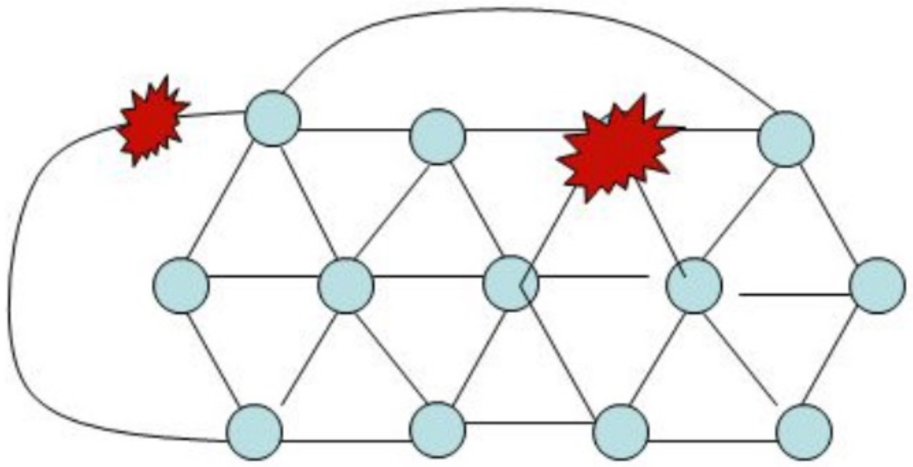
CAP theorem (**C**onsistency, **A**vailability, **P**artition tolerance)

Distributed mutual exclusion

Elections

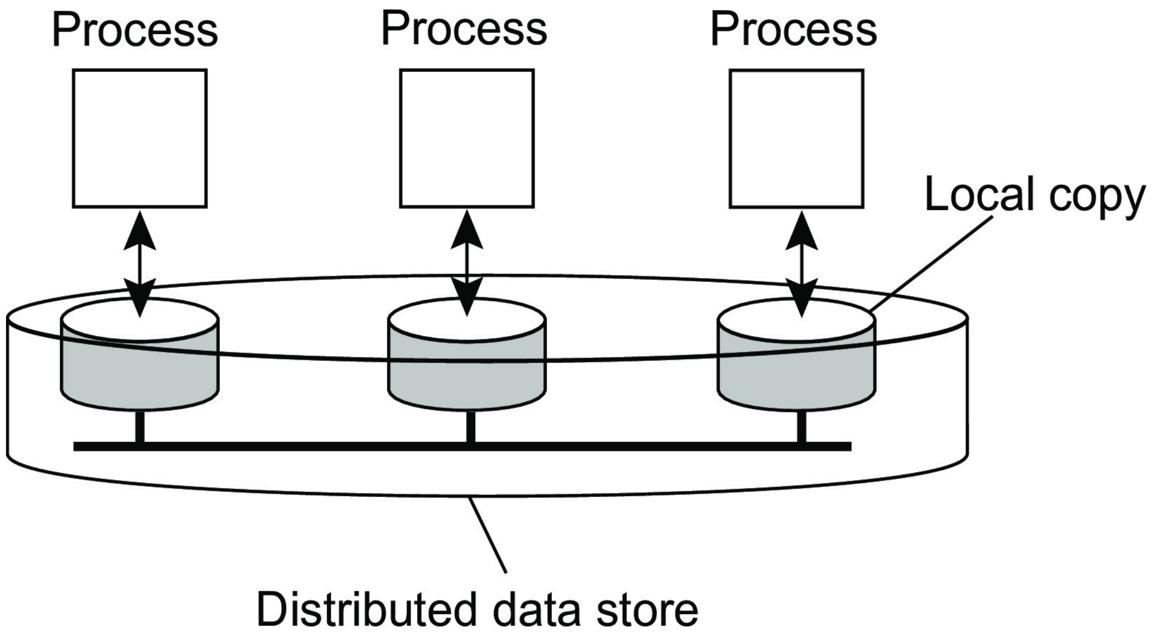
## Distributed consensus

A critical problem in distributed computing is to achieve overall system reliability (=getting correct results) even if there are some faulty processes or broken links

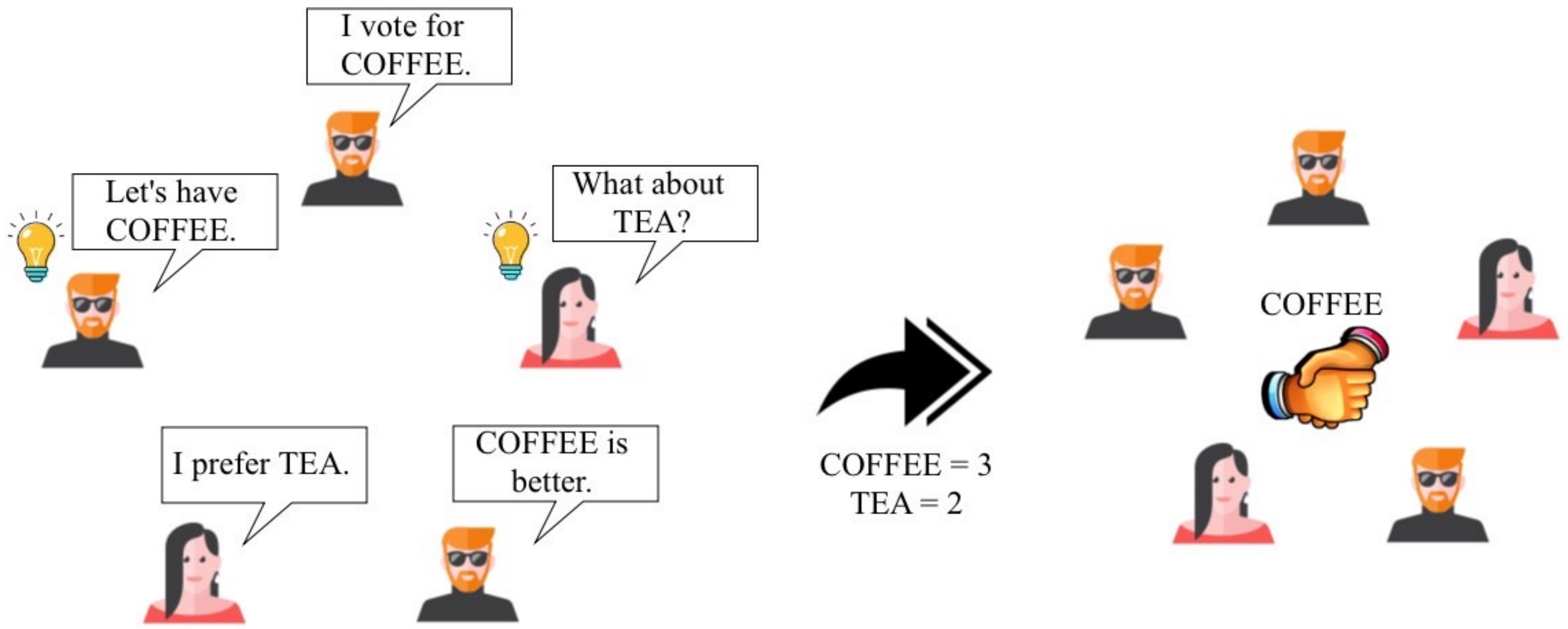


## Distributed consensus

We discuss the topics of how distributed processes coordinate their actions and agree on shared values - reach consensus - despite failures



## The problem of consensus in distributed systems



1. Electing a leader among distributed nodes
2. Taking a commit decision involving all nodes
3. Replicating data into some nodes

} How do we reach consensus?

## Consensus when some processes are faulty

- A fundamental problem in distributed computing is to achieve overall system reliability in the presence of some faulty processes
- This often requires processes to agree on some data value that is needed during computation: consensus
- Examples of applications of consensus include: to commit a transaction to a database, agreeing on the identity of a leader, state machine replication, and atomic broadcasts
- The real world applications include clock synchronization, PageRank, smart power grids, state estimation, control of drones (and multiple robots/agents in general), load balancing, blockchain, and many others

## PageRank as consensus

The PageRank algorithm can be viewed as a consensus mechanism for ranking nodes in a distributed graph (like the web).

### Distributed Nature of PageRank

- PageRank operates on the graph of the web, where each page (node) can be seen as part of a distributed system. Links (edges) represent connections between these nodes.
- The algorithm computes the rank of each page iteratively, propagating influence across the graph. Each node updates its state (rank) based on the states of its neighbors.

### Iterative Convergence to Agreement

- PageRank repeatedly updates the ranks of all nodes distributing the "importance" of each node across its neighbors.
- After several iterations, the system reaches a steady state where the ranks stabilize. This stable is a form of consensus: all nodes "agree" on their respective ranks.

### Global Agreement on Ranking

- The final ranks produced by the PageRank algorithm are globally consistent across the distributed graph, meaning all nodes have a shared understanding of the relative importance of each node.

### Fault Tolerance and Robustness

- Like consensus algorithms, PageRank is robust to some changes in the graph (e.g., addition or removal of nodes or edges). The iterative process adjusts to these changes and eventually converges again.

Limitations -While PageRank involves a form of consensus in ranking, it is not a strict distributed consensus algorithm:

- It does not deal with classic challenges of distributed systems like network partitions, Byzantine faults, or the need for a single agreed value among nodes.
- Its goal is to compute a global ranking, not to agree on a discrete value or state.

In summary, PageRank exemplifies consensus in a loose sense because it ensures a globally consistent ranking across a distributed network through iterative computation, aligning the states (ranks) of nodes across the system.

## Main questions and an impossibility result

How can processes in different computers coordinate their actions and agree on shared values, despite failures?

How can a group of processes agree on a new coordinator of their activities after the previous coordinator has failed?

CAP theorem: there are some conditions where it is impossible to guarantee that processes on different computers will reach consensus

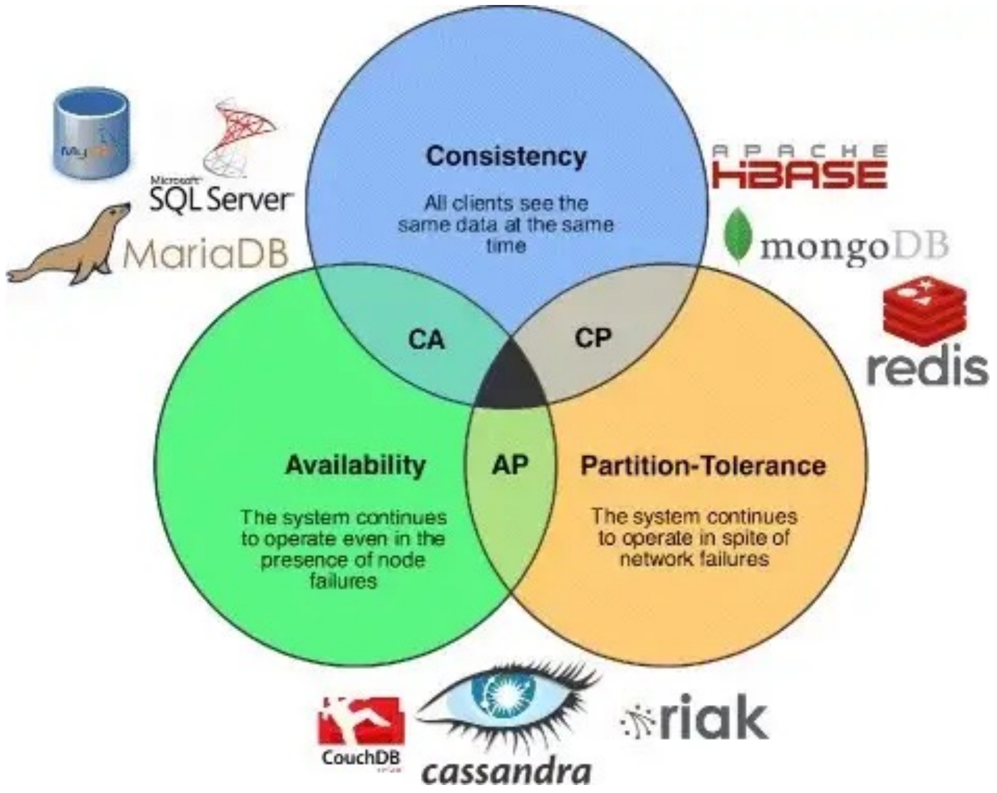
## CAP theorem

**Consistency**: Every read receives the most recent writes or an error.

**Availability**: Every request received by a non-failing node in the system must result in a response. Whether you want to read or write you will get some response back.

**Partition tolerance:** The system continues to operate despite an arbitrary number of messages being dropped(or delayed) by the network between nodes. When a network is partitioned, all messages sent from nodes in one component of the partition to nodes in another component are lost.

**CAP theorem** states that it is impossible for a distributed system to simultaneously provide more than two out of the above three guarantees.



Consistency
All clients see the same data at the same time
CA
CP
Availability
The system continues to operate even in the presence of node failures
AP
Partition-Tolerance
The system continues to operate in spite of network failures
SQL Server
MariaDB
APACHE HBASE
mongoDB
redis
CouchDB
cassandra
riak

## The problem

- The **consensus problem** is a fundamental problem in distributed systems.
- The consensus problem requires agreement among a number of processes for a single data value: the processes must produce their candidate values, communicate, and agree on a single value
- One approach to generating consensus is for all processes to agree on a majority value (where each process is given a vote): however, one or more faulty processes may skew the resultant outcome such that consensus may not be reached, or reached incorrectly
- Protocols that solve consensus problems are designed to deal with limited numbers of faulty processes.

## Agenda

Distributed mutual exclusion

Elections

Consensus and related problems: eg. Eventual consistency

CAP theorem (Consistency, Availability, Partition tolerance)

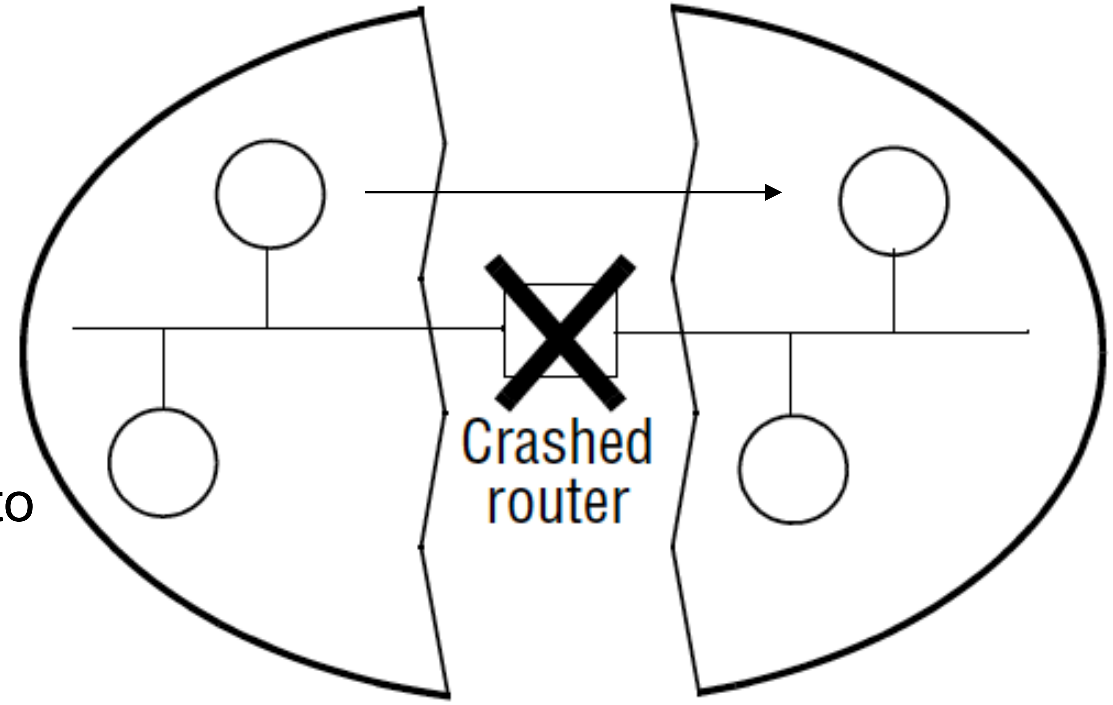
Transactions

## Issues

- How do processes coordinate their actions and agree on shared values in distributed systems, despite failures?
- We need algorithms to achieve **mutual exclusion** among processes, so as to coordinate their accesses to shared resources.
- How an **election** can be implemented in a distributed system? – that is, how a group of processes can agree on a new coordinator after the previous one has failed?
- We deal with **group communication**, consensus, **Byzantine agreement**, and interactive consistency

## A network partition

Complex network topologies and independent routing choices mean that connectivity may be **asymmetric**: communication is possible from process p to process q, but not vice versa.
Connectivity may also be **intransitive**: communication is possible from p to q and from q to r, but p cannot communicate directly with r.



For examples of non transitivity see https://www.usenix.org/legacy/events/worlds05/tech/full_papers/freedman/freedman_html/index.html

## Mutual exclusion

- Mutual exclusion is a property of concurrency control, which is instituted for the purpose of preventing race conditions
- Only one process is allowed to execute a critical section of a code at any given time
- In a distributed system, we cannot exploit shared variables (semaphores) or a local kernel to implement mutual exclusion.
- Message passing is the sole means for implementing distributed mutual exclusion

## Distributed mutual exclusion

Distributed mutual exclusion algorithms must deal with unpredictable message delays and incomplete knowledge of the system state.

Three basic approaches for distributed mutual exclusion:

1. Token based approach
2. Non-token based approach
3. Quorum based approach

## Mutual Exclusion approaches

Token based approach:

- A unique token is shared among the processes.
- A process is allowed to enter its CS if it possesses the token.
- Mutual exclusion is ensured because the token is unique.

Non-token based approach:

- Two or more successive rounds of messages are exchanged among the processes to determine which one will enter the CS next.

Quorum based approach:

- Each process requests permission to execute the CS from a subset of processes (called a *quorum*).
- Any two quorums contain a common process.
- This common process is responsible to make sure that only one request executes the CS at any time.

## Comparing algorithms for mutual exclusion

The performance of distributed algorithms for mutual exclusion is evaluated according to:

- the *bandwidth consumed*, which is proportional to the number of messages sent in each entry and exit operation
- the *client delay* incurred by a process at each entry and exit operation;
- the algorithm's effect upon the *throughput* of the system: this is the rate at which the collection of processes as a whole can access the critical section, given that some communication is necessary between successive processes.
- We use the synchronization delay between one process exiting the critical section and the next process entering it; the throughput is greater when the synchronization delay is shorter

## Mutual exclusion

The application-level protocol for executing a critical section CS:

- enter() // enter critical section – block if necessary
- resourceAccesses() // access shared resources in critical section
- exit() // leave critical section – other processes may now enter

## Requirements for mutual exclusion (ME) algorithms

Our essential requirements for mutual exclusion are:

I. ME1: (safety) At most one process may execute in the critical section (CS) at a time.

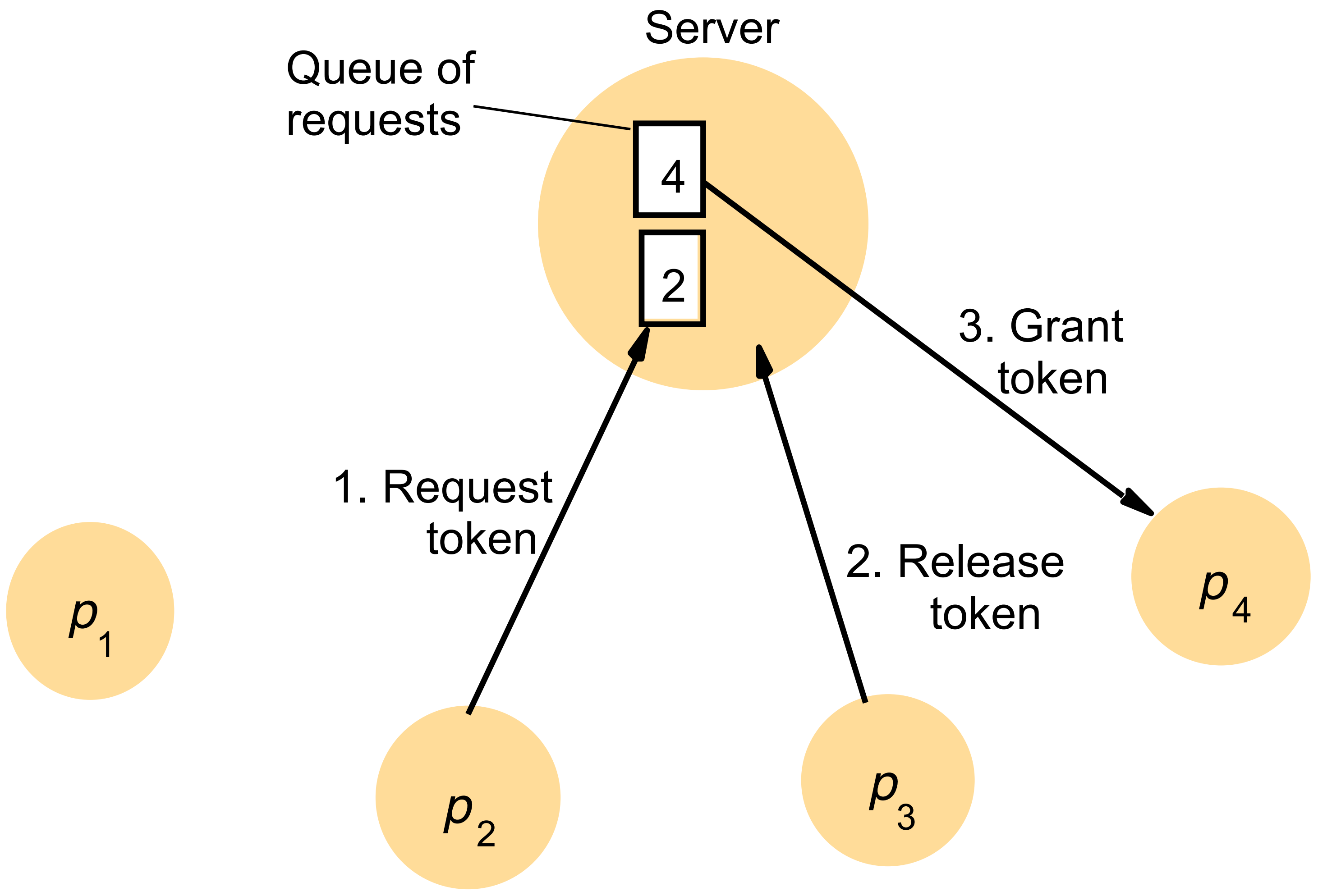
II. ME2: (liveness) Requests to enter and exit the critical section eventually succeed.

III. ME3: (-> ordering) If one request to enter the CS *happened-before* another, then entry to the CS is granted in that order.

The absence of starvation (ME2) is a fairness condition.

Another fairness issue is the order in which processes enter the critical section (ME3). Remark: It is not possible to order the entry into the critical section by the times that the processes requested it, because of the absence of a global clock

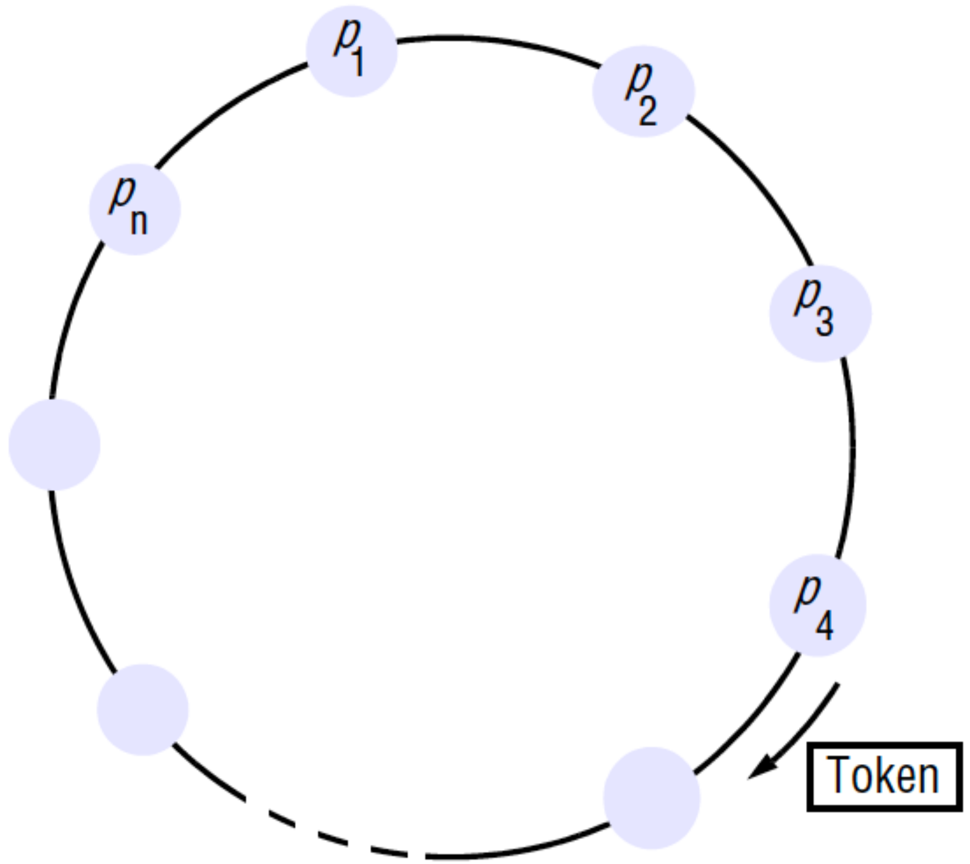
## Server managing a mutual exclusion token for a set of processes



The simplest way to achieve mutual exclusion is to employ a server that grants permission to enter the critical section

## A ring of processes transferring a mutual exclusion token

If a process does not require to enter the critical section when it receives the token, then it immediately forwards the token to its neighbour.
A process that requires the token waits until it receives it, but retains it.
To exit the critical section, the process sends the token on to its neighbour
This algorithm works but continuously consumes network bandwidth



## Discussion of the ring algorithm

- the conditions ME1 and ME2 are met by the ring algorithm, but the token is not necessarily obtained in *happened-before* order (processes may exchange messages independently of the rotation of the token)
- This algorithm continuously consumes network bandwidth (except when a process is inside the critical section): the processes send messages around the ring even when no process requires entry to the critical section.
- The delay experienced by a process requesting entry to the critical section is between 0 messages (when it has just received the token) and N messages (when it has just passed on the token).
- To exit the critical section requires only one message.

## Consensus

- Our system model includes a collection of processes $p_i$ communicating by message passing.
- A requirement that applies in many practical situations is to reach consensus even in the presence of faults: communication is reliable but processes may fail
- We consider **Byzantine** (arbitrary) process failures, as well as crash failures.
- We sometimes specify an assumption that up to some number f of the N processes are *faulty* – that is, they exhibit some specified types of fault; the other processes are correct

## Consensus

To reach consensus, every process $p_i$ begins in the undecided state and proposes a single value $v_i$, drawn from a set D

The processes communicate with one another, exchanging values.

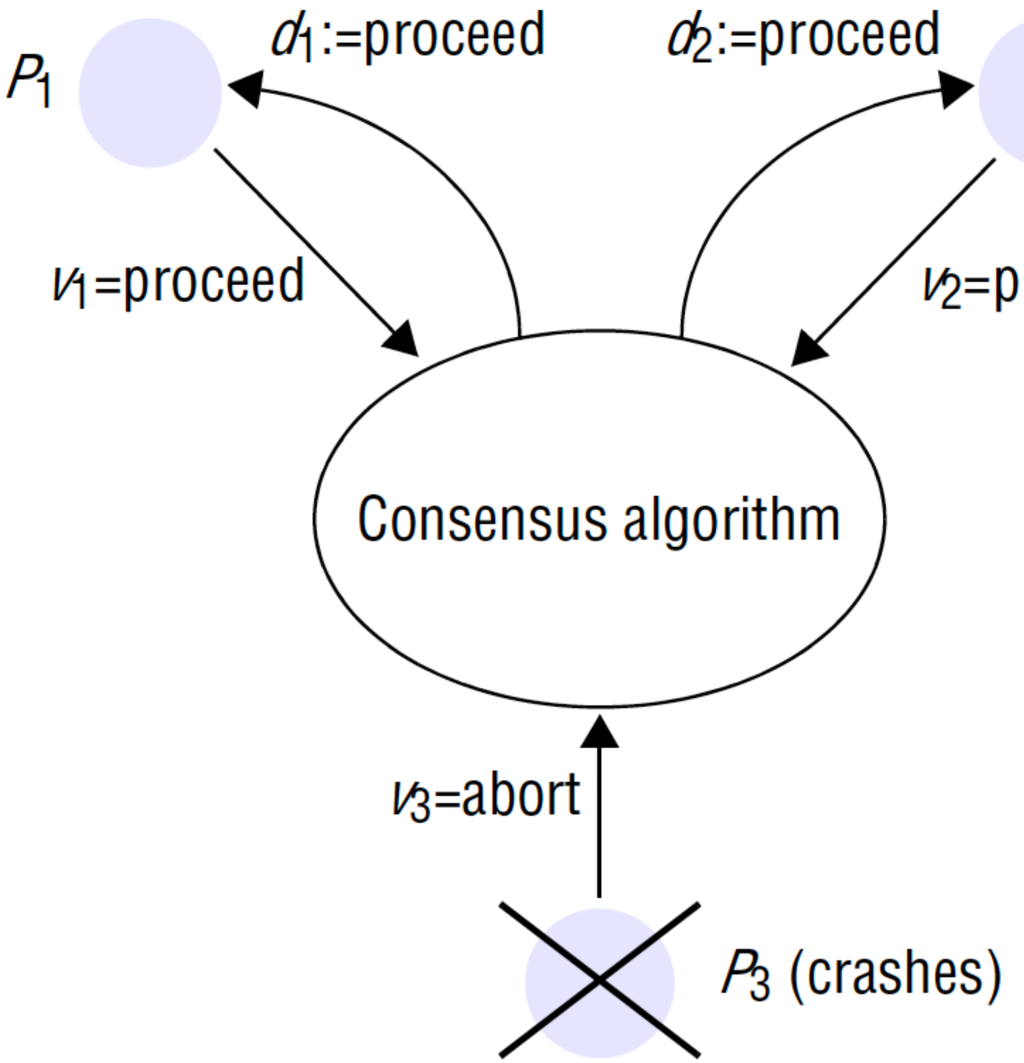
Each process then sets the value of a decision variable, $d_i$ .

In doing so $p_i$ enters the decided state, in which it may no longer change $d_i$

Next slide shows three processes engaged in a consensus algorithm: two processes propose '*proceed*' and a third proposes '*abort*' but then crashes.

The two processes that remain correct each decide '*proceed*'.

## Consensus for three processes



Two processes propose ‘proceed’ and a third proposes ‘abort’ but then crashes. The two processes that remain correct each decide ‘proceed’.

## Requirement for consensus

The requirements of a consensus algorithm are that the following conditions should hold for every execution:

- Termination: Eventually each correct process sets its decision variable.
- Agreement: The decision value of all correct processes is the same: if $p_i$ and $p_j$ are correct and have entered the decided state, then $d_i = d_j$
- Integrity: If the correct processes all proposed the same value, then any correct process in the decided state has chosen that value.

## Byzantine consensus

- Three or more generals are to agree to attack or to retreat. One, the commander, issues the order. The others, lieutenants to the commander, must decide whether to attack or retreat.
- But one or more of the generals may be ‘treacherous’ – that is, faulty. If the commander is treacherous, he proposes attacking to one general and retreating to another. If a lieutenant is treacherous, he tells one of his peers that the commander told him to attack and another that they are to retreat.
- The Byzantine generals problem differs from consensus in that a distinguished process supplies a value that the others are to agree upon, instead of each of them proposing a value.

The requirements are:

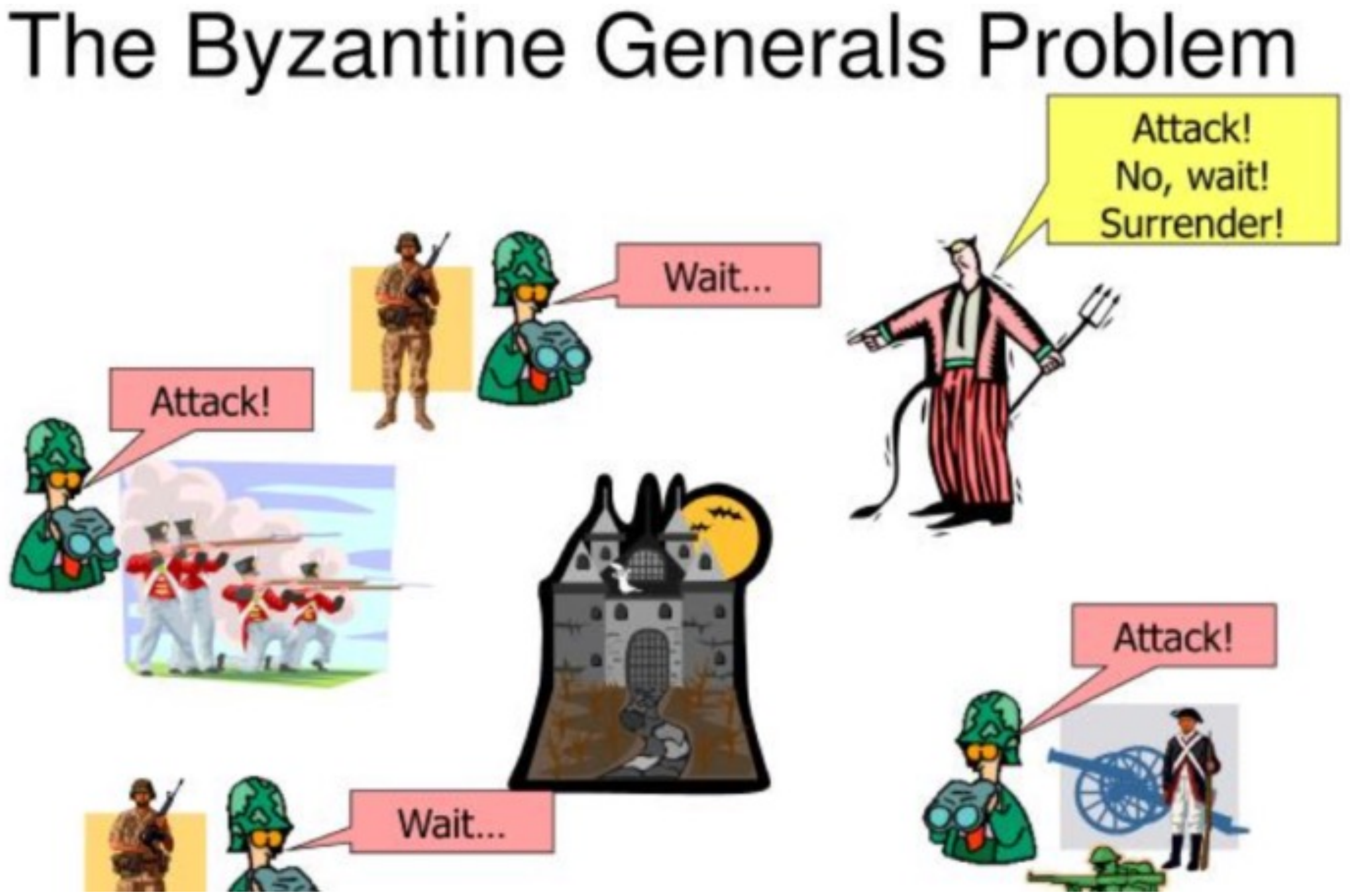
Termination: Eventually each correct process sets its decision variable.

Agreement: The decision value of all correct processes is the same: if pi and pj are correct and have entered the decided state, then di = dj

Integrity: If the commander is correct, then all correct processes decide on the value that the commander proposed.

Note that, for the Byzantine generals problem, integrity implies agreement when the commander is correct; but the commander need not be correct

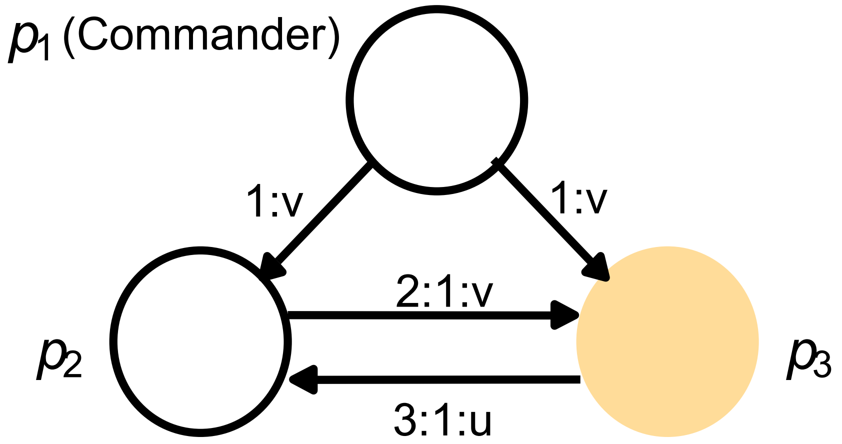
## Byzantine generals

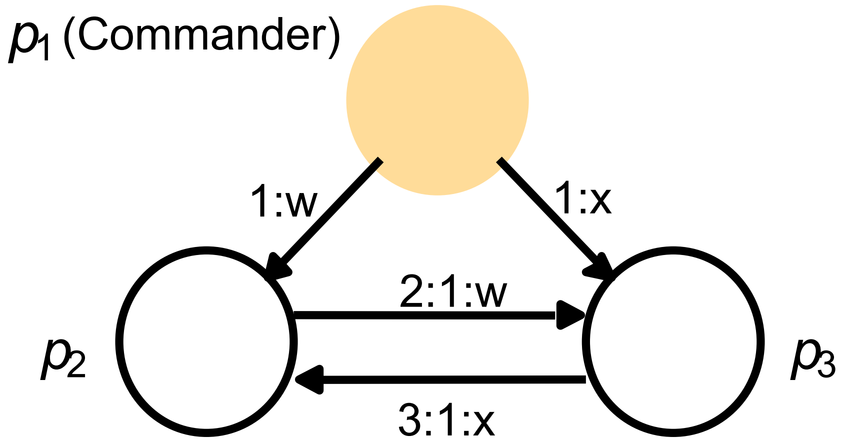


## The Byzantine generals' problem

- A number of Byzantine Generals each have a computer and want to attack the King's wi-fi by brute forcing the password, which they have learned is a certain number of characters in length. Once they stimulate the network to generate a packet, they must crack the password within a limited time to break in and erase the logs, lest they be discovered.
- They only have enough CPU power to crack it fast enough if a majority of them attack at the same time.
- They don't particularly care when the attack will be, just that they agree. It has been decided that anyone who feels like it will announce an attack time, which we'll call the "plan", and whatever plan is heard first will be the official plan.
- The problem is that the network is not instantaneous, and if two generals announce different plans at close to the same time, some may hear one first and others hear the other first.

## Three Byzantine generals

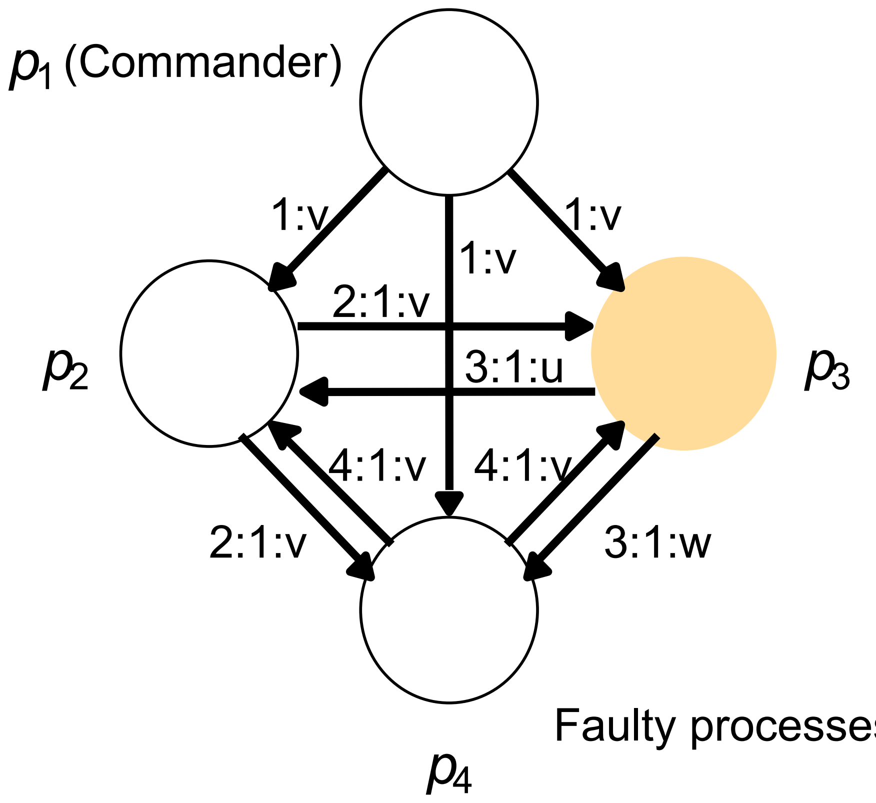


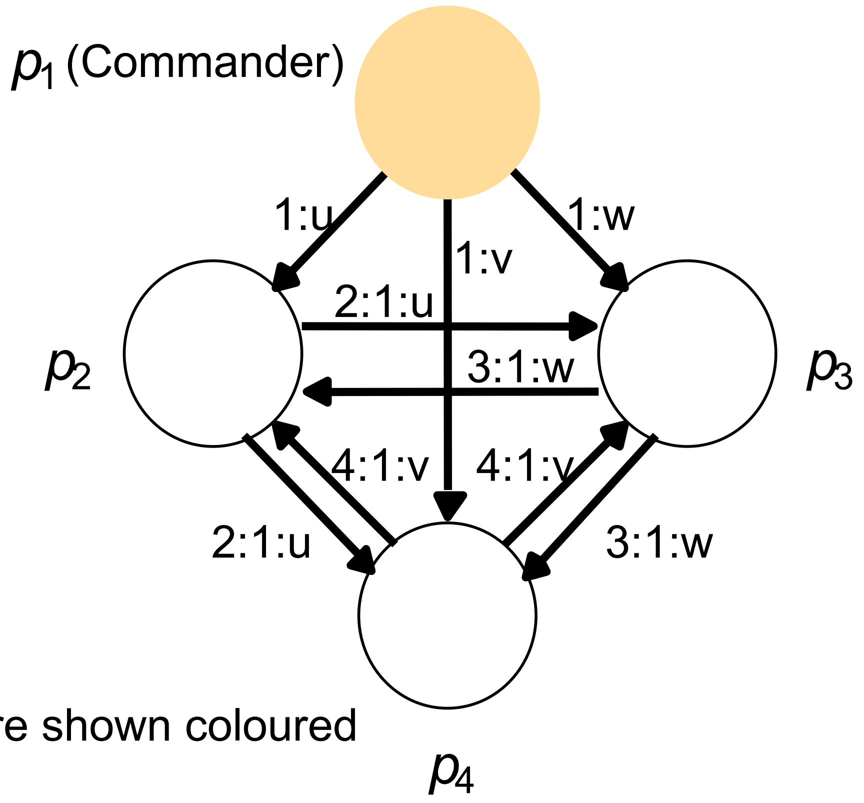


Faulty processes are shown coloured

two scenarios in which just one of three processes is faulty.
In the lefthand configuration one of the lieutenants, p3, is faulty; on the right the commander, p1, is faulty. Each scenario shows two rounds of messages: the values the commander sends, and the values that the lieutenants subsequently send to each other. The numeric prefixes serve to specify the sources of messages and to show the different rounds. Read the ':' symbol in messages as '*says*'; for example, '3:1:u' is the message '3 says 1 says u'.
In the rightmost picture p2 and p3 agree on «*undef*», as no majority of values exists

## Four Byzantine generals





Faulty processes are shown coloured

in the lefthand configuration one of the lieutenants, p3, is faulty; on the right, the commander, p1 is faulty.

In the lefthand case, the two correct lieutenant processes agree, deciding on the commander's value:
p2 decides on majority(v,u,v) = v
p4 decides on majority(v,v,w) = v

In the righthand case the commander is faulty, but the three correct processes agree: p2, p3 and p4 decide on majority(u,v,w) = *undef* (the special value *undef* applies where no majority of values exists).

## Interactive consistency

- The interactive consistency problem is another variant of consensus, in which every process proposes a single value.
- The goal of the algorithm is for the correct processes to agree on a vector of values, one for each process: this is the ‘decision vector’
- For example, the goal could be for each of a set of processes to agree about their respective states.

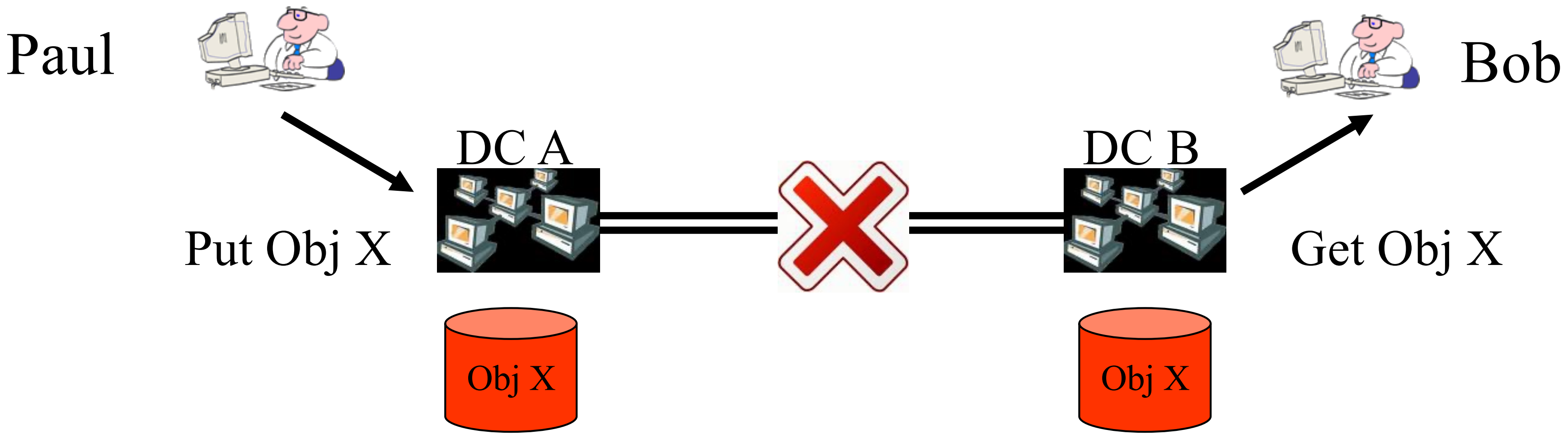
The requirements for interactive consistency are:

Termination: Eventually each correct process sets its decision variable.

Agreement: The decision vector of all correct processes is the same.

Integrity: If $p_i$ is correct, then all correct processes decide on $v_i$ as the ith component of their vector.

## Consistency Models



**Consistency models** are used in distributed systems like: distributed shared memory systems, or distributed data stores (such as a filesystems, databases, optimistic replication systems or web caching).
The system is said to support a given **model** if operations on memory follow specific rules

## CAP Theorem

The **CAP theorem**, also known as **Brewer's theorem**, states that it is impossible for a distributed system to simultaneously provide all three of the following guarantees (max two guarantees are allowed, never three):

1. *Consistency* (all nodes see the same data at the same time)
2. *Availability* (node failures do not prevent survivors from continuing to operate)
3. *Partition Tolerance* (the system continues to operate despite arbitrary msg loss)

According to the theorem, a distributed system can satisfy any two of these guarantees at the same time, but not all three.

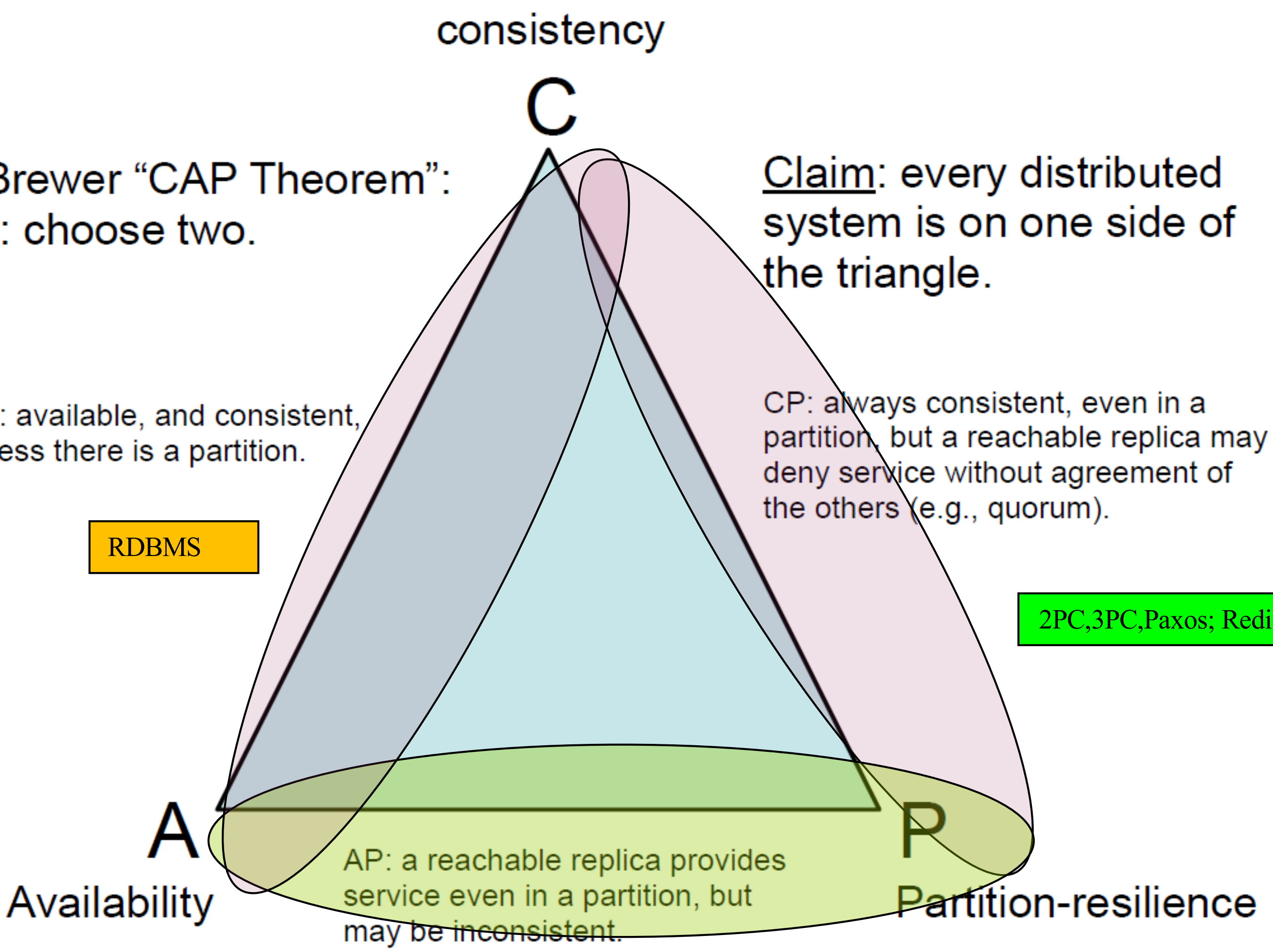
CAP began as a conjecture made by Berkeley computer scientist Eric Brewer at the 2000 Symposium on Principles of Distributed Computing (PODC).

In 2002, Seth Gilbert and Nancy Lynch of MIT published a formal proof of Brewer's conjecture, establishing it as a theorem.

# CAP Theorem and its applications

Fox&Brewer "CAP Theorem": C-A-P: choose two.

<u>Claim</u>: every distributed system is on one side of the triangle.


consistency
C
CA: available, and consistent, unless there is a partition.
CP: always consistent, even in a partition, but a reachable replica may deny service without agreement of the others (e.g., quorum).
A
Availability
AP: a reachable replica provides service even in a partition, but may be inconsistent.
P
Partition-resilience

RDBMS

2PC,3PC,Paxos; Redis, MongoDB

DNS, Cassandra, IceCube, Bayou, eBay, Amazon Dynamo, and many other…

## Consistency Models: Client and Server

There are two ways of looking at consistency:

1. developer/client point of view: how clients observe data updates. There are three approaches: strong, weak, and eventual consistency
2. server side point of view: how updates flow through the system and what guarantees systems can give with respect to updates.

## Client Side Consistency

Strong consistency: after the update completes, any subsequent access will return the updated value.

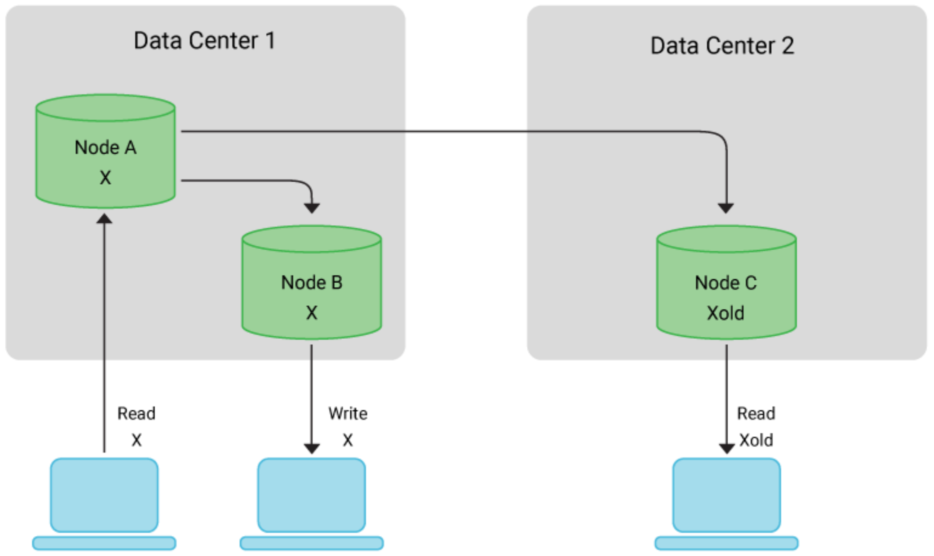
Weak consistency: The system does not guarantee that subsequent accesses will return the updated value. The period between the update and the moment when it is guaranteed that any observer will always see the updated value is dubbed the *inconsistency window*.

Eventual consistency: a specific form of weak consistency; the storage system guarantees that if no new updates are made to the object, eventually all accesses will return the last updated value.

The most popular system that implements eventual consistency is DNS (Domain Name System): 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.

## Eventual consistency

Eventual Consistency is a guarantee that when an update is made in a distributed database, that update will eventually be reflected in all nodes that store the data, resulting in the same response every time the data is queried.



## Weak vs eventual consistency

- **Weak Consistency:** In a weak consistency model, there are minimal guarantees about the state of data across nodes at any given time. Different replicas or nodes might have different views of the data, and the consistency is typically low immediately after an update. However, over time, these inconsistencies will be resolved, and the system will converge to a consistent state. This lack of immediate consistency allows for high availability and performance but can lead to potential conflicts or divergence in views among nodes until reconciliation occurs.
- **Eventual Consistency:** Eventual consistency ensures that, given enough time without further updates, all replicas in the system will converge to a consistent state. It doesn't guarantee that immediately after an update, all replicas will reflect the changes. Instead, it permits temporary inconsistencies but guarantees that if no new updates occur, eventually, all replicas will be consistent. This approach often involves mechanisms like conflict resolution or reconciliation to achieve eventual convergence.

## Server Side Consistency

Definitions:

N = the number of nodes that store replicas of the data

W = the number of replicas that need to acknowledge the receipt of the update before the update completes

R = the number of replicas that are contacted during a read operation

If W+R > N, then the write set and the read set always overlap and one can guarantee strong consistency.

In the primary-backup RDBMS scenario, which implements synchronous replication, **N=2, W=2**, and **R=1**. No matter from which replica the client reads, it will always get a consistent answer.

In asynchronous replication with reading from the backup enabled, **N=2, W=1**, and **R=1**. In this case R+W=N, and consistency cannot be guaranteed.

## Server Side Consistency – Cont.

In distributed-storage systems that need to provide high performance and high availability, the number of replicas is in general higher than two.

Systems that focus solely on fault tolerance often use **N=3, W=2, R=2** configurations.

Systems that need to serve very high read loads often replicate their data beyond what is required for fault tolerance; N can be tens or even hundreds of nodes, with **R=1** such that a single read will return a result.

Systems that are concerned with consistency are set to **W=N** for updates, which may decrease the probability of the write succeeding (when the system cannot write to W nodes because of failures, the write operation has to fail, marking the unavailability of the system).

A common configuration for systems that are concerned about fault tolerance but not consistency is to run with **W=1** to get minimal durability of the update and then rely on a lazy (epidemic) technique to update the other replicas.

## Server Side Consistency – Cont.

Examples:

What do we optimize for in **R=1,N=W**?

What do we optimize for In **W=1,R=N?**

When optimizing for writes, durability is not guaranteed in the presence of failures, and if $W < (N+1)/2$, there is the possibility of conflicting writes when the write sets do not overlap.

Weak/eventual consistency arises when $W+R <= N$, meaning that there is a possibility that the read and write set will not overlap.

The period until all replicas have been updated is the inconsistency window discussed before. If $W+R <= N$, then the system is vulnerable to reading from nodes that have not yet received the updates.

## Conclusions

- We described the problems of consensus and consistency.
- We defined some conditions for their solution.
- Solutions exist in a synchronous system
- Instead, it is theoretically impossible guaranteeing consensus in an asynchronous system (CAP theorem)
- Nonetheless, in practice systems regularly do reach agreement even in asynchronous systems.