Consensus and Agreement Algorithms

Lesson 9

Consensus and Agreement Algorithms

The Need for Agreement in Distributed Systems

- Many applications depend on different components acting in a coordinated manner.
- The idea is that separate processes must not proceed independently when a shared outcome is required. Instead, a common understanding must be reached before any application-specific action is taken.
- Coordination usually requires the processes to exchange information. This exchange allows each participant to negotiate and reason about the state of others until a shared viewpoint is developed.
- A classical illustration is the commit decision in database systems. When multiple processes participate in a transaction, the system must collectively decide whether the transaction should be committed or aborted.
- The decision must be common to all participants, as a partial commit or partial abort would lead to inconsistent database states.

Byzantine Behavior: The Problem of Agreement

The Scenario: Attacking the Fort of Byzantium

- The difficulty of achieving **agreement** is illustrated by an example inspired by the long wars of the Byzantine Empire.
- The setup involves an attacking army with **four camps**, each commanded by a **general**, surrounding the fort of Byzantium.
- **Success** is only possible if all generals execute the attack **simultaneously**. Therefore, a crucial task is to agree on the precise **time of attack**.
- The sole means of communication among the generals is by sending **messengers** between the camps.

Modeling the Communication and Traitors

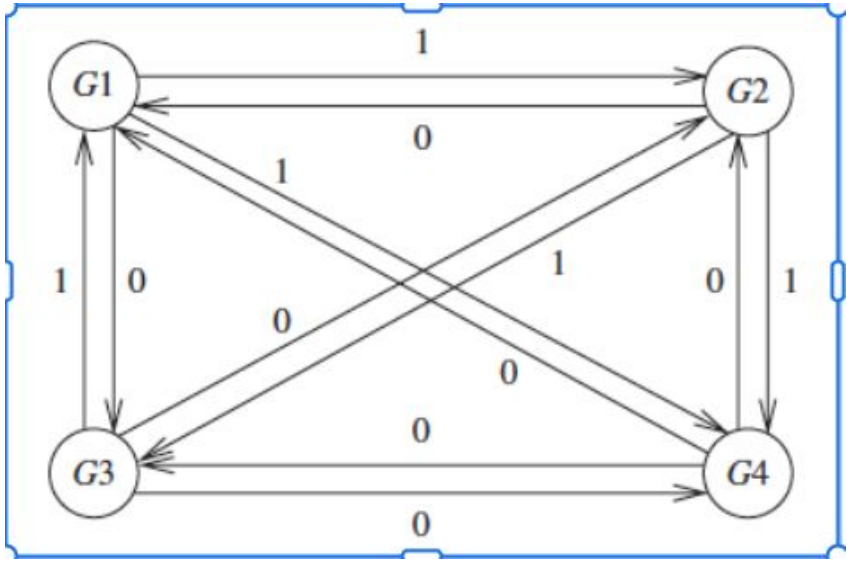
The components of this historical analogy are modeled in a technical system as follows:

- **Messages** are represented by the **messengers** traveling between the camps.
- An **asynchronous system** (where message delay is unpredictable) is modeled by messengers taking an **unbounded time** to travel.
- A **lost message** is modeled by a messenger being **captured by the enemy**.
- A **Byzantine process** (a faulty or malicious node) is modeled by a **general being a traitor**.

The traitor general's goal is to **subvert the agreement-reaching mechanism** by providing **misleading information** to the other loyal generals.

Byzantine Example

- The diagram represents the four generals, labelled **G1**, **G2**, **G3**, and **G4**, exchanging messages with one another.
- Each directed arrow shows a message sent from one general to another.
- The value written on the arrow is the value of the decision variable being communicated, typically represented as either **0** or **1**.
- Each general is expected to send the *same* value to all other generals.
- In a correct, non-Byzantine system, a general must forward exactly what it believes the decision value to be. All outgoing messages from that general should therefore be identical.

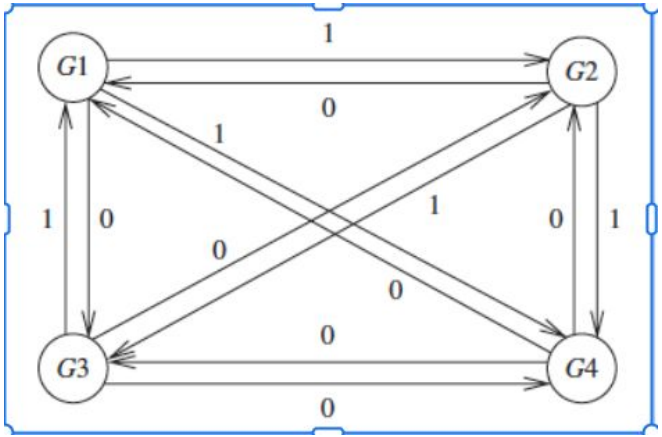


G1 sends the value **1** to G2, 0 to G3 and **0** to G4. Since a correct general must send the same value to all others, this variation shows that G1 is behaving in a Byzantine manner.

G2 sends **0** to G1 and **1** to both G3 and G4. This inconsistency indicates Byzantine behaviour, because the messages do not match across recipients.

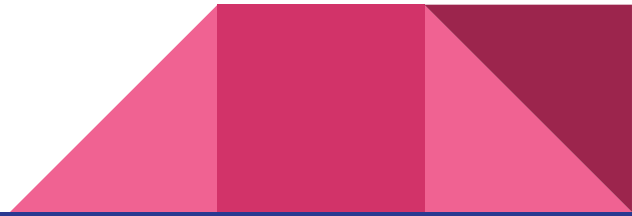
G3 sends **1** to G1 but sends **0** to both G2 and G4. The difference between the message sent to G1 and those sent to the others shows that G3 is also acting in a Byzantine manner.

G4 sends **0** to G1, G2, and G3. Since all outgoing messages are identical, G4 is behaving correctly and not in a Byzantine manner.



Consensus in Failure Free System

- In a system with no failures, agreement is straightforward because every process can broadcast its value and receive all others without loss.
- Once every process has the same set of values, each one applies the **same decision function** (such as majority, max, or min).
- Because inputs and the function are identical, all processes reach the same decision.

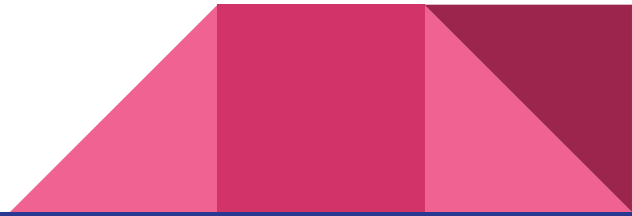


Consensus Algorithm for Crash Failures (Synchronous System)

Overview of the Algorithm

In a **synchronous system**, consensus can be achieved among **n processes** even if up to **f processes** exhibit **crash failures** (also known as the **fail-stop model**).

- The algorithm operates on a **consensus variable x**, which is **integer-valued**.
- Each process starts with an **initial value xi**.
- To tolerate f failures, the algorithm is structured to complete in exactly **f+1 rounds**.



Operation in Each Round

The core mechanism revolves around the exchange and processing of values in each of the f+1 rounds:

- **Sending Phase:** In a given round, a process i **broadcasts** the current value of its variable **xi** to all other processes, but **only if** that specific value **has not been sent before**.
- **Receiving and Update Phase:** Upon receiving messages for the round, each process considers:
  - All the **values received** during the current round.
  - Its **own value xi** at the start of the round.
  - The process then computes the **minimum** of this entire set of values.
  - Its local variable **xi is subsequently updated** to this newly calculated minimum value.

Termination and Guarantee

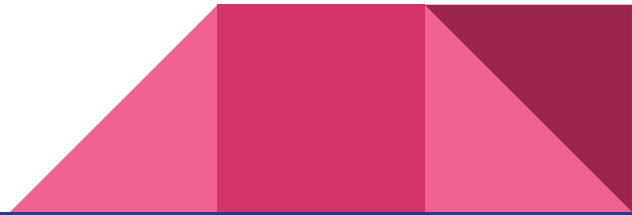
After the completion of all **f+1 rounds**, the final **local value xi** held by any non-faulty process is **guaranteed to be the consensus value**.

- This algorithm ensures **agreement** and **validity** (the consensus value must be one of the initial values) despite the crash failures.

Lower Bound on Number of Rounds

The requirement for **at least f+1 rounds** is a **lower bound** for achieving consensus with f crash failures in a synchronous system, where f < n.

- **Intuition for the Bound:** The necessity for f+1 rounds stems from the worst-case scenario where **one process may fail in each of the first f rounds**.
- **Guaranteed Consensus:** By executing **f+1 rounds**, it is guaranteed that at least **one round** must be **failure-free**.
- **The Critical Failure-Free Round:**
    - In this guaranteed failure-free round, **all messages broadcast can be delivered reliably** to all non-faulty processes.
    - Consequently, all processes that have not crashed can **compute the common function** (in this case, the **minimum**) of the received values and **reach an agreement value**.

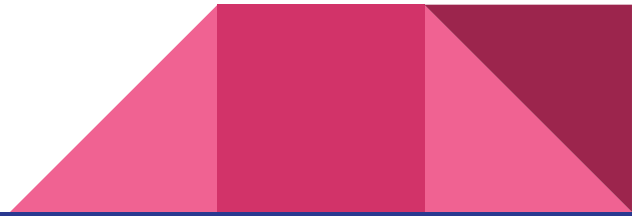


Upper Bound on Byzantine Processes

In a system with **n** processes, Byzantine agreement can be solved in a synchronous system only if the number of Byzantine processes **f** satisfies:

```
f <= floor((n - 1) / 3)
```

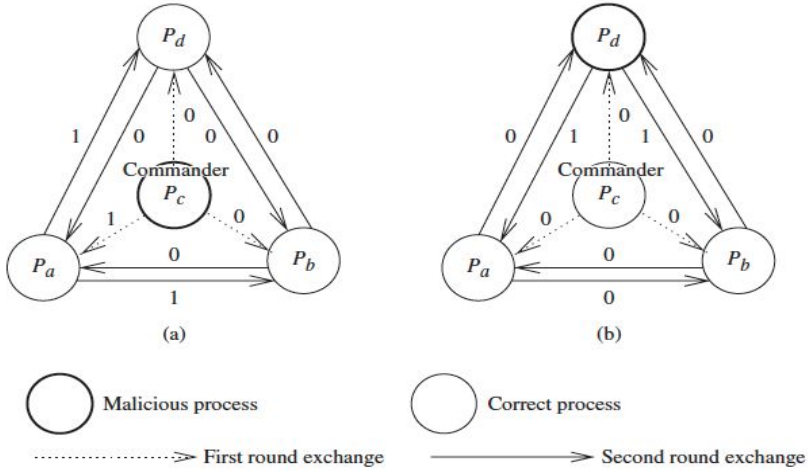
This means the system can tolerate at most one third of the processes (rounded down) behaving in a misleading or arbitrary way. If more than this fraction are Byzantine, the correct (non-Byzantine) processes cannot reliably reach agreement.



Byzantine Agreement Tree Algorithm

Principle of the Algorithm

- Agreement is achieved by exchanging the commander's value through **two rounds of communication**.
- In the first round, each lieutenant receives the commander's value directly.
- In the second round, lieutenants forward the value they received to the other lieutenants.
- Each lieutenant then holds **three versions** of the commander's value (considering there are only four nodes as shown in the diagram).
- The lieutenant selects the **majority value**, which filters out any incorrect value sent by a faulty process.
- This majority rule ensures all correct lieutenants reach the **same decision**, even if one process behaves maliciously.



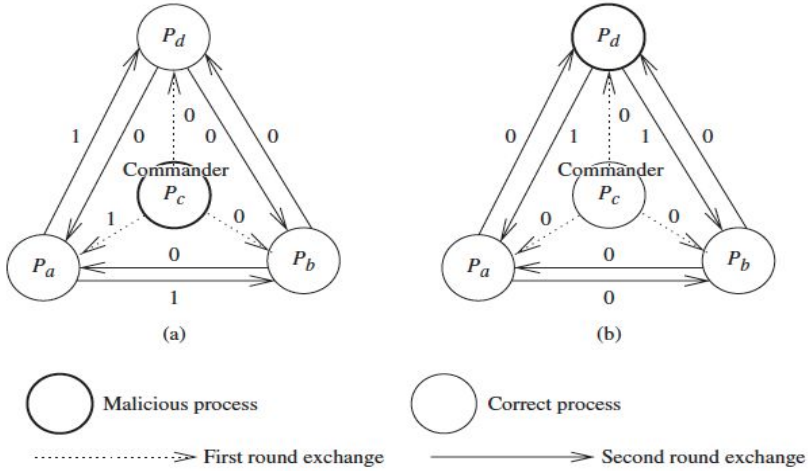
Interpreting the Byzantine Agreement Tree Example

The diagram illustrates how agreement is reached among four processes: a commander (Pc) and three lieutenants (Pa, Pb, Pd). One of these processes may act maliciously. The algorithm ensures that all non-faulty lieutenants arrive at the same final value, even when one process behaves incorrectly.

The number of malicious (faulty) processes allied is determined by the formulae:

```
f <= floor((n - 1) / 3)
```

Where f is the number of maximum faulty process, n is the total number of participants (commander and lieutenants). Now, let's have a look how it works in the next slide.



Interpreting the Byzantine Agreement Tree Example

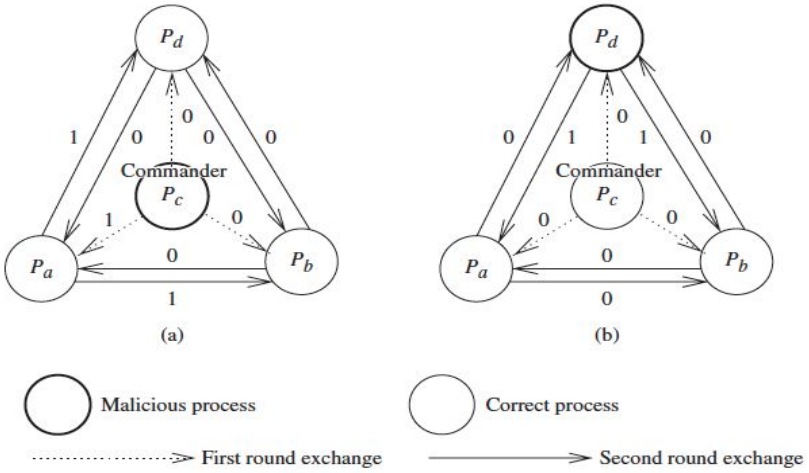
First Round: Commander Sends Its Value

In the first round, Pc (the malicious commander) sends its value to Pa, Pb, and Pd.

The dotted arrows represent these transmissions.

If Pc is faulty, it may send *different* values to different lieutenants.
This behaviour appears in the diagram where Pc sends values such as 1 to lieutenant *Pa*, 0 to *Pb*, and 0 to *Pd*.

The table shows the corresponding values the lieutenants have received from the commander Pc after the first round.



Lieutenants	Values (1st round)
Pa	1
Pb	0
Pd	0

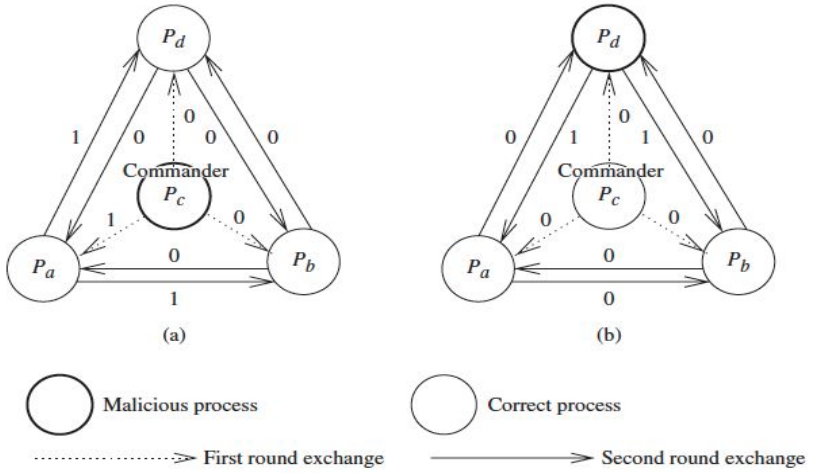
Second Round: Lieutenants Relay What They Heard

Each lieutenant forwards to the other two lieutenants the value it received from Pc in the first round.

The solid arrows between Pa, Pb, and Pd represent this second-round exchange.

At the end of this round, **each lieutenant holds three independent reports** of the commander's value:

1. The value received directly from Pc.
2. The value forwarded by one other lieutenant.
3. The value forwarded by the remaining lieutenant.



Lieutenants	First Round	Second Round
Pa	1	1, 0, 0
Pb	0	0, 1, 0
Pd	0	0, 1, 0

Majority Decision at Each Lieutenant

After collecting these three values, each lieutenant applies a simple majority rule.

This majority serves to filter out any incorrect value introduced by a faulty process.

For instance, if Pc sends (1, 0, 0) to the three lieutenants:

- Each lieutenant eventually observes the set (1, 0, 0) in some order.
- The majority value is 0.
- All correct lieutenants decide on 0.

Thus, agreement is reached even when Pc behaves maliciously.

Q.3 In a distributed network with eight nodes participating need to come to a common agreement with some expected minimum failures. What kind of algorithm can be implemented to suit the above scenario and how many failures are acceptable. Give justification for the number of acceptable failures and explain appropriate algorithm.

*In this distributed system, eight nodes must reach a **common agreement** even when some nodes may fail or send misleading information. The agreement problem under such failures is what the **Byzantine Agreement** model addresses. Byzantine failures include behaviours where nodes may send different values to different nodes, omit messages, or behave unpredictably, making them significantly more complex than simple crash failures. Since the question explicitly requires agreement in the presence of potential incorrect behaviour, an algorithm that handles only crash failures would be insufficient. Therefore, the class of algorithms relevant to this scenario is **Byzantine Agreement algorithms**, which are designed precisely to guarantee consistent decisions even when a subset of nodes behaves incorrectly.*

*Within this class, the **Byzantine Agreement Tree Algorithm** is an appropriate algorithm to implement because it is specifically designed for **synchronous message-passing systems**. The algorithm ensures that all correct nodes reach the same decision by organising communication in multiple rounds.*

First, the initiating node sends its proposed value to all others. Then, in subsequent rounds, each node forwards the value it received so that every correct node eventually obtains a structured set of values, forming a "message tree."

*Each node then applies a **majority rule** over these collected values. The key idea is that, even if faulty nodes send conflicting or deceptive values, the repeated relaying and majority voting filter out false information.*

*This process ensures **agreement**, **validity**, and **termination**, which are the standard correctness requirements for consensus algorithms.*

To determine how many failures the system can tolerate, we use the fundamental bound for Byzantine Agreement in synchronous systems:

$$n \geq 3f + 1$$

where

- *n is the total number of nodes, and*
- *f is the maximum number of Byzantine faulty nodes that can be tolerated.*

With n=8, this gives:

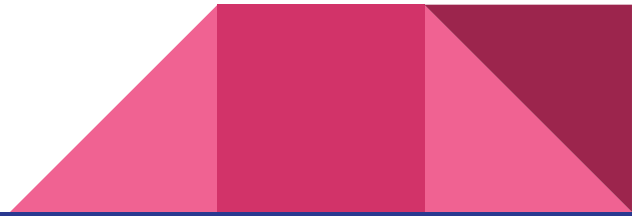
$$8 \geq 3f + 1 \Rightarrow 7 \geq 3f \Rightarrow f \leq 2$$

*Thus, **at most two nodes may fail arbitrarily** while still allowing all correct nodes to reach agreement. The Byzantine Agreement Tree Algorithm depends on correct nodes forming a **majority at each step** when evaluating the values they have received. If faulty nodes were allowed to exceed this limit (for example, three or more), they could coordinate to distort the votes at multiple levels of the message tree, making it impossible for correct nodes to distinguish truthful values from falsified ones. In that case, different correct nodes might construct different majority outcomes, violating the requirement that all correct nodes must agree. Therefore, the limit of **two acceptable failures** is not a design choice—it is enforced by the mathematics of consensus under Byzantine behaviour.*

The Impossibility of Consensus

The classical result by Fischer, Lynch, and Paterson established that agreement cannot be guaranteed in a fully asynchronous message-passing system when even one process may crash. This impossibility refers specifically to the consensus task, where processes must reach a common decision despite uncertain message delays and the possibility that a participant may stop responding.

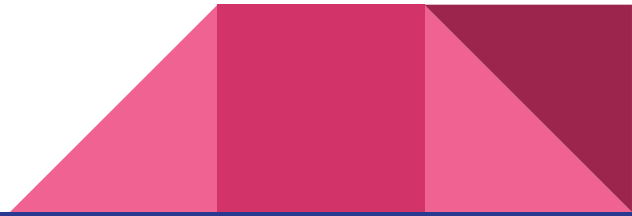
This result is foundational because it shows the inherent limits of designing distributed algorithms that operate in failure-susceptible asynchronous systems. It also introduced the idea of **valence**, which helps explain how global states evolve when processes attempt to decide.



The Terminating Reliable Broadcast Problem

The terminating reliable broadcast problem is centred on a single sender that broadcasts one message. The requirement is that every correct process eventually obtains some message even if the sender crashes during transmission. **When the sender fails mid-broadcast, the delivered message may be a null message, but it must still be delivered by all correct processes.**

An additional requirement is termination: every correct process must eventually deliver some message. This places a strong guarantee on progress, because waiting indefinitely for the sender to finish is not permitted.



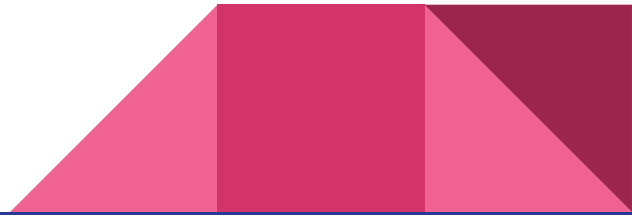
Terminating reliable broadcast guarantees four things:

Validity means that if the sender is correct, then all correct processes will eventually deliver the message it broadcast.

Agreement means that if one correct process delivers the message, then all correct processes will eventually deliver the same message.

Integrity means that each correct process delivers at most one message and that this message must come from an actual broadcast by the sender.

Termination means that every correct process eventually delivers some message, even if the sender crashes. Together, these properties ensure that all correct processes deliver the same value and that delivery must eventually complete.



Explanation of Terminating Reliable Broadcast

Terminating reliable broadcast is defined by four properties: validity, agreement, integrity, and termination. These together force all correct processes to eventually deliver a value, and to deliver the same value. On the next slide, you will see why this abstraction is extremely strong: if each process broadcasts its bit using terminating reliable broadcast, then every other process will either deliver that bit or, if the sender crashes, deliver the null value. Because TRB guarantees both agreement and termination, each process can then *decide* on a bit in exactly the way required by consensus. In other words, if TRB existed, then consensus would be solvable. But consensus is provably impossible under crash failures in an asynchronous system, so the conclusion is that terminating reliable broadcast itself cannot exist.

Distributed Transaction Commit

Purpose of Distributed Commit

A distributed transaction may update data stored across multiple machines.

To preserve ACID properties (atomicity, consistency, integrity, durability), all participants must **either commit together or roll back together**.

The commit step ensures that the entire distributed transaction behaves as a single atomic action.

Commit Decision Process

Before committing, the coordinator polls every participant to check whether each of them is ready to commit.
Each participant replies with either:

- a **vote to commit**, meaning it can safely commit, or
- a **vote to roll back**, meaning it cannot commit.

Atomicity requires unanimity.
A single rollback vote forces the entire transaction to abort.
Once a final decision (commit or rollback) is reached, it must be communicated to all participants.

Theoretical Impossibility vs Practical Protocols

Distributed commit resembles a consensus problem, and consensus is impossible to guarantee in an asynchronous system with crash failures. Yet practical systems still successfully use commit protocols because they assume a slightly different operational model.

Two-Phase Commit (2PC)

The two-phase commit protocol is correct only if certain assumptions hold. It waits indefinitely for missing replies on the belief that a crashed participant will eventually recover and respond.

Since it does not resolve coordinator crashes cleanly, 2PC is a blocking protocol: participants can become stuck waiting forever.

Three-Phase Commit (3PC)

The three-phase commit protocol avoids indefinite waiting by using timeouts. If the coordinator receives no response within a specified time, it defaults to an abort decision.

3PC is non-blocking under certain failure models, but its correctness is not guaranteed under all asynchronous conditions.

Optimisations and Their Limitations

Presumed Abort

A pessimistic optimisation assuming most transactions will abort.
Participants can discard some state early and reduce logging overhead, but correctness still depends on specific failure assumptions.

Presumed Commit

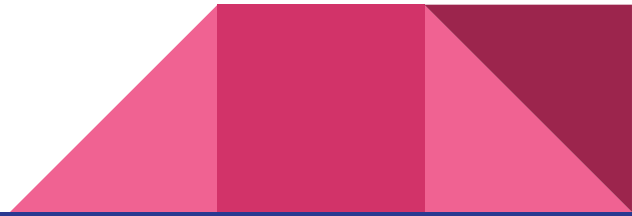
An optimistic optimisation assuming success is more common than failure.
Again, this saves logging work but does not guarantee correctness under all possible crash scenarios.

Timeout-Based Defaults in 3PC

3PC uses timeouts to avoid indefinite blocking.
If participants or the coordinator fail to respond within the timeout, the protocol switches to an abort decision to maintain safety.

Summary

- Distributed commit ensures ACID atomicity across multiple machines.
- All participants vote; one rollback vote aborts the entire transaction.
- Consensus impossibility applies, yet commit protocols work in practice by adopting assumptions not allowed in the theoretical model.
- **2PC** is **blocking**: waits forever for a crashed node to recover.
- **3PC** is **non-blocking** but relies on timeouts and assumptions that do not always hold in fully asynchronous systems.
- **Presumed abort** and **presumed commit** are performance optimisations, not universally correct solutions.



k-Set Consensus

Regular consensus requires all non-faulty processes to decide on **one single value**.
In an asynchronous system with even one crash failure, this is impossible to guarantee.

A weaker but still meaningful alternative is **k-set consensus**.

What k-Set Consensus Allows

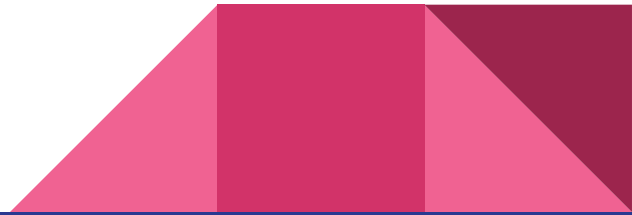
Instead of requiring agreement on **one** value, k-set consensus allows up to **k different decision values**.

The solvability condition is:

- If the number of crash failures is **f < k**, then k-set consensus *is* solvable.
- If **f ≥ k**, k-set consensus becomes impossible.

This relaxes the agreement condition:

- Different non-faulty processes may decide on different values,
- but the **total number of distinct decision values** is at most **k**.



Meaning of the Parameter k

The parameter **k** limits the size of the *set of values* chosen by all non-faulty processes.
Agreement is weakened from “everyone agrees on one value” to “everyone chooses from a set of size at most k”.

Validity and termination requirements remain similar to regular consensus, except that validity is adjusted so the outputs must be drawn from proposed inputs within the bound of k possible results.

Relation to Regular Consensus

k = 1 gives standard consensus, which is impossible with crash failures.

k > 1 softens the agreement requirement enough that the task becomes solvable as long as f < k.

Thus, k-set consensus offers a controlled relaxation: more failures tolerated, but at the cost of allowing multiple decision values.

Numerical Illustration

Let:

- **n = 10** processes
- **f = 2** crash failures
- **k = 3**

Suppose each process proposes a unique value from the set **{1, 2, …, 10}**.

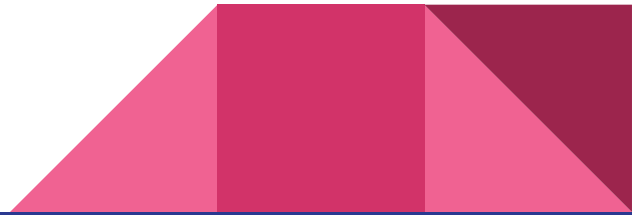
Since **f < k**, k-set consensus is solvable.

A valid output is any set of at most three decision values.
For example, the resulting 3-set might be:

{8, 9, 10}

Any three values would be acceptable, provided:

- they are from the proposed values,
- all non-faulty processes decide within these three values,
- and the size of the decision set does not exceed **k = 3**.



Summary

- k-set consensus is a weaker generalisation of consensus.
- It is solvable in asynchronous systems with crash failures if and only if **f < k**.
- At most **k** different decision values may appear among the non-faulty processes.
- Validity and termination remain similar to consensus, but agreement is weakened.
- Example: with **f = 2** and **k = 3**, any decision set of size at most 3 is valid.