Dependable Distributed Systems

Ida Francesca Benvenuto

novembre 2025

Chapter 1

Consensus

1.1 Consensus Problem

The consensus problem in distributed systems involves a set of processes that must agree on a single value.

1.2 consensus specification

The consensus problem is defined by the following properties:

- **Termination:** Every correct process eventually decides on a value.

- **Agreement:** No two correct processes decide on different values.

- **Validity:** If all correct processes propose the same value v, then any correct process that decides must decide v.

- **Integrity:** No correct process decides more than once.
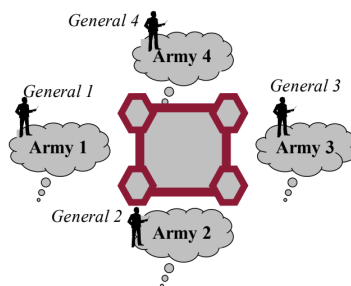

Figure 1.1: Consensus Problem

Impossibility of Consensus in Asynchronous Systems with Failures

In an asynchronous distributed system where processes can fail by crashing, it is impossible to design a deterministic consensus algorithm that satisfies all the properties of termination, agreement, validity, and integrity.

This result is known as the FLP impossibility theorem, named after Fischer, Lynch, and Paterson.The key intuition behind the FLP impossibility result is that in an asynchronous system, there is no upper bound on message delivery times or process execution speeds.

1.3 Consensus Implementation in Synchronous Systems

Flooding-based Consensus Algorithm

In a synchronous system, we can implement a consensus algorithm using a flooding-based approach.All processes exchange their proposed values in rounds, and after a fixed number of rounds, they decide on a value based on the received proposals. the rounds are fundamental because they provide a structured way for processes to communicate and ensure that all correct processes have the opportunity to share their proposed values before making a decision.(in this way we are sure that there aren't lost values)
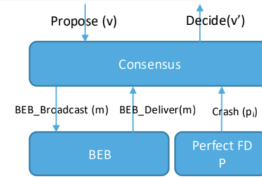

Figure 1.2: Flooding-based Consensus Algorithm

1.4 Algorithm Description

1.4.1 Initialization

When the system starts (upon event $\langle c, \text{Init} \rangle$):

- $correct := \Pi$ — Initialize the set of correct processes (all processes)

- $round := 1$ — Start from round 1

- $decision := \bot$ — The decision is still empty ($\bot$ means "not decided")

- $receivedFrom[r] := [\emptyset]^N$ — Array tracking from which processes we have received messages for each round

- $proposals[r] := [\emptyset]^N$ — Array tracking the proposals received for each round

1.4.2 Main Event Handlers

Propose Event

When a process proposes a value:

```
upon event <c, Propose | v> do
  proposals[1] := proposals[1] union {v};
  trigger <beb, Broadcast | [PROPOSAL, 1, proposals[1]]>;
```

The process adds its initial value to the proposals of round 1 and broadcasts it to all processes.

Crash Event

When the Failure Detector detects a crash:

```
upon event <P, Crash | p> do
  correct := correct \ {p};
```

Removes the process from the set of correct processes.

Deliver Event

When a PROPOSAL is received:

```
upon event <beb, Deliver | p, [PROPOSAL, r, ps]> do
  receivedFrom[r] := receivedFrom[r] union {p};
  proposals[r] := proposals[r] union ps;
```

Updates the list of processes from which messages have been received and merges the received proposals.

1.4.3   Decision Logic

The crucial logic for deciding:

```
upon correct subseteq receivedFrom[round] AND decision = bottom do
  if receivedFrom[round] = receivedFrom[round-1] then
    decision := min(proposals[round]);
    trigger <beb, Broadcast | [DECIDED, decision]>;
    trigger <c, Decide | decision>;
  else
    round := round + 1;
    trigger <beb, Broadcast | [PROPOSAL, round, proposals[round-1]]>;
```

This is the most important part:

- **Waiting condition:** The process waits until it has received messages from all correct processes ($correct \subseteq receivedFrom[round]$)

- **Stability:** If the set of processes from which messages were received in this round is the same as in the previous round (no new crashes), we can trust the data

- **Decision:** Chooses the minimum value among the received proposals (deterministic function common to all processes, can be replaced with other functions)

- **Decision broadcast:** Communicates the decided value to all other processes

- **Else:** If the set changes (new crash detected), increment the round and continue

1.4.4   Receiving DECIDED

```
upon event <beb, Deliver | p, [DECIDED, v]>
  such that p in correct AND decision = bottom do
  decision := v;
  trigger <beb, Broadcast | [DECIDED, decision]>;
  trigger <c, Decide | decision>;
```

If a correct process communicates the decided value and the receiving process has not yet decided, it decides this value.

1.4.5   Practical Example

1.5   Uniform Consensus Specification

The difference between uniform consensus and (non-uniform) consensus is that uniform consensus requires **Uniform Agreement**: no two processes
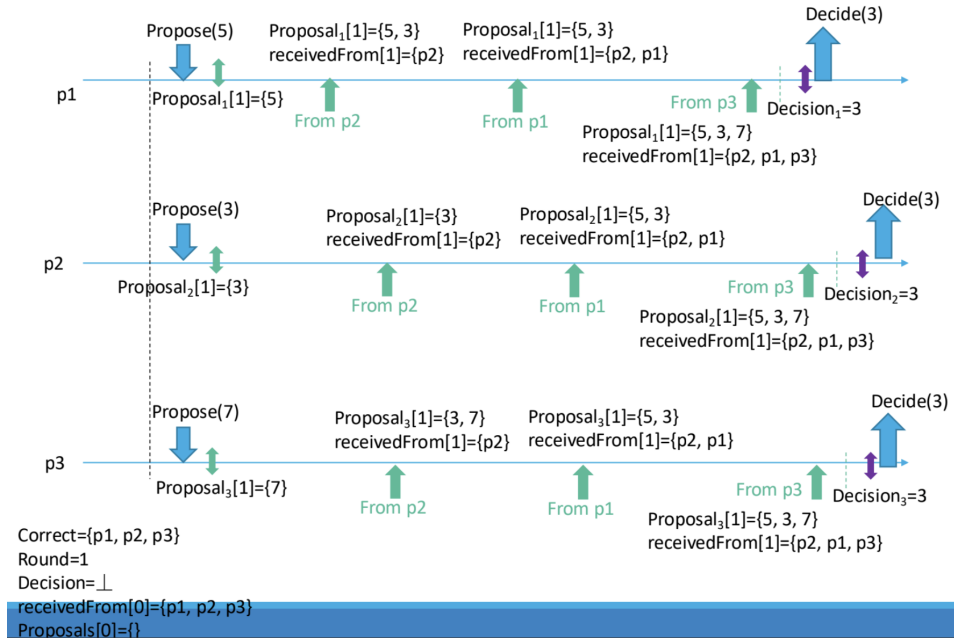
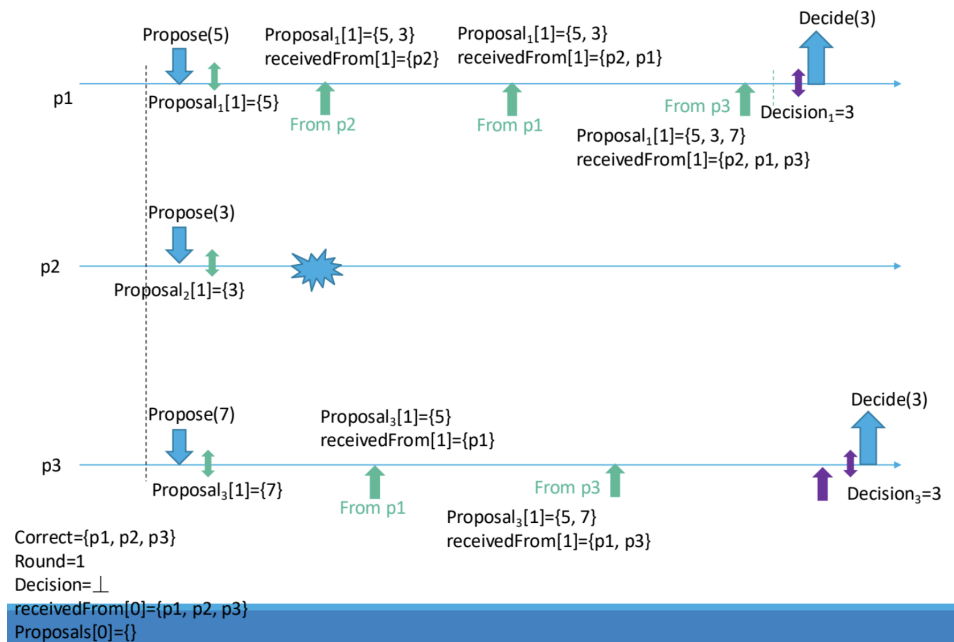Figure 1.3: Example of Consensus Algorithm Execution

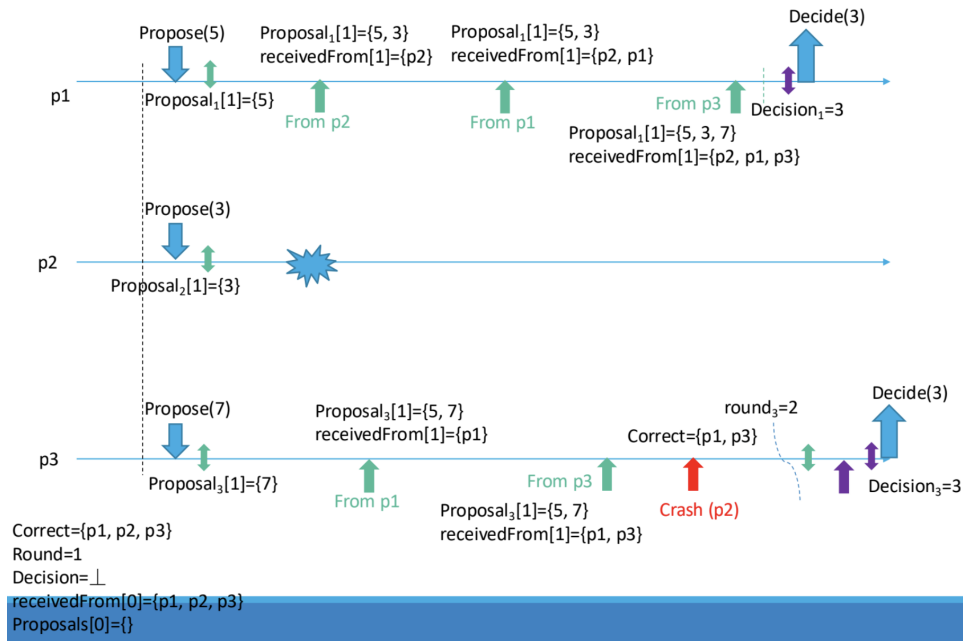Figure 1.4: P3 decide the min value after the decision of p1

Figure 1.5: p2 is the only one thet can decide but crash before the broadcast of the value, so P1 and P3 go to the next round and decide an other value

(correct or faulty) can decide different values. In standard consensus, only correct processes are required to agree, meaning a faulty process may decide a different value before crashing. in the algorithm the decision is choseen at the round N, (at the end) and the set of proposals is the same for all processes, so also faulty processes decide the same value of correct processes.
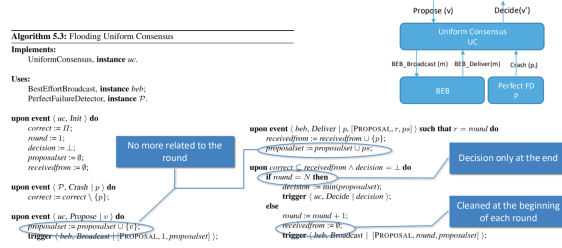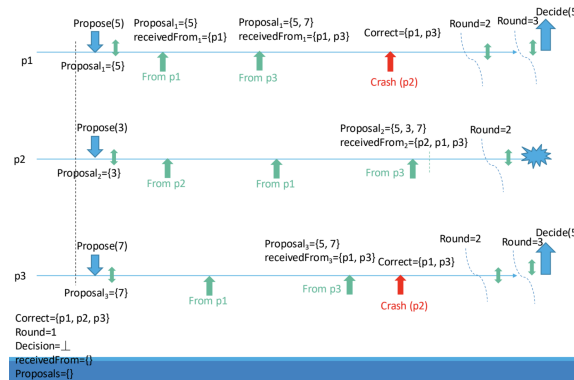
Example

Figure 1.6: Uniform Consensus properties

Figure 1.7: at the round 3 there is a decision, that is 5 becaus when start a new round the receivedFrom set is = 0

Chapter 2

Paxos

Paxos is a consensus algorithm designed for asynchronous distributed systems with crash failures. The safety is always guaranteed, while the liveness is guaranteed only under some conditions (like eventual message delivery and a majority of correct processes).

2.1 Assumptions

- **Agents:** Operate at arbitrary speed, may fail by stopping (and may restart).

  - **Observation:** Since all agents may fail after a value is chosen and then restart, a solution is impossible unless some information can be remembered by an agent that has failed and restarted.

- **Messages:** Can take arbitrarily long to be delivered, can be duplicated, and can be lost, but they are not corrupted.

Chapter 3

Total Order Broadcast

Total Order Broadcast (TOB) is a fundamental communication primitive in distributed systems that ensures all processes deliver messages in the same order.

3.1   System model

We consider a distributed system with the following assumptions.

- **Processes:**

  - There is a static set of processes $p_1, \ldots, p_n$.
  - Processes can fail by *crash*: a crashed process stops executing its algorithm forever, but they can't behave maliciously.(bizantine failures are not considered here)
  - A process that never crashes is called a *correct* process.

- **Communication:**Channels between correct processes are *perfect*: messages are not lost, not duplicated, and no fake messages are created.

- **Asynchronous:** there is no known upper bound on message transmission time or processing time.

3.2   Total Order specification

TO (Total Order) is composed by 4 specifications:

- **Validity:** If a correct process broadcasts a message m, then it eventually delivers m.

- **Uniform Agreement:** If a process delivers a message m, then all correct processes eventually deliver m.

- **Uniform Integrity:** For any message m, every process delivers m at most once, and only if m was previously broadcast by its sender.

- **Uniform Total Order:** If two processes p_i and p_j deliver two messages m and m', then p_i delivers m before m' if and only if p_j delivers m before m'.

Validity parla di consegna assicurata per messaggi validi.
Uniform Agreement parla di accordo sull'insieme dei messaggi consegnati.
Uniform Integrity parla di evitare duplicati e messaggi inventati.
Uniform Total Order parla di accordo sull'ordine esatto di consegna.
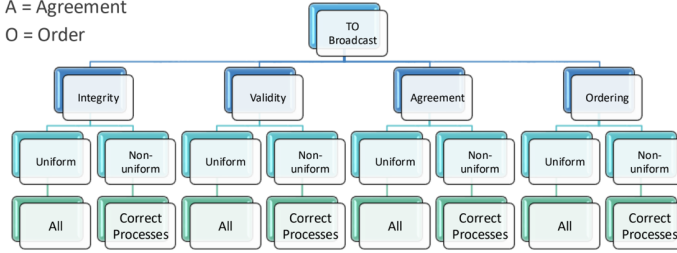

Figure 3.1: Total Order Broadcast properties

A Uniform properties is stronger than a non-uniform one because it applies to all processes, not just correct ones.

3.3 Non-Uniform Validity-Uniform Integrity

Such that our system have perfect channel and chra failures, the properties satisfied are:
NUV (Non-Uniform Validity): If a correct process broadcasts a message m, then it eventually delivers m.
UI (Uniform Integrity): For any message m, every process delivers m at most once, and only if m was previously broadcast by its sender.
the properties that are not satisfied are: Agreement and Total Order.

Agreement property

- **Non-Uniform Agreement:** If a CORRECT process delivers a message m, then all correct processes eventually deliver m.

- **Uniform Agreement:** If a process delivers a message m, then all correct processes eventually deliver m.

  The difference is that in Uniform Agreement, the process delivering m can be faulty.The correct processes deliver the same set of messages, while faulty processes may deliver a different set of messages or none at all.
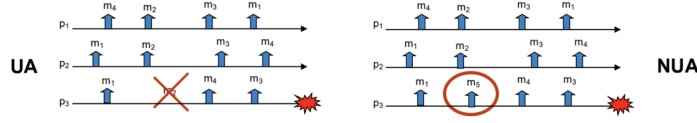

Figure 3.2: Non-Uniform Agreement vs Uniform Agreement

  UA: If P3 (faulty process) delivers m2, then P1 and P2 (correct processes) must also deliver m2.
  NUA: If P3 (faulty process) delivers m5, it does not imply that P1 and P2 (correct processes) must also deliver m5.

Total Order property

- **Strong Uniform Total Order:** if some process deliver message m before m', then a process deliver m' only after m.

- **Weak Uniform Total Order:** If two processes p_i and p_j deliver two messages m and m', then p_i delivers m before m' if and only if p_j delivers m before m'.

The difference is that in SUTO, all processes must deliver the same set of messages in the same order, while in WUTO, processes may deliver different sets of messages, but the relative order of any two messages is consistent across processes that deliver both messages.
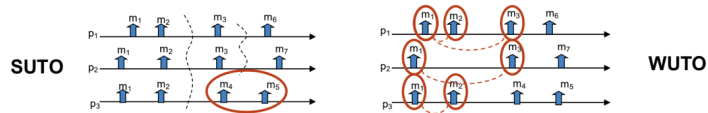

Figure 3.3: Strong Uniform Total Order vs Weak Uniform Total Order

  SUTO: if process p3,delivers message m4 before m5, then process p4 must also deliver m4 before m5.
WUTO: Each process may deliver a different set of messages, but the relative

order between any two messages is always the same for processes that deliver both messages.

- **Strong non-Uniform Total Order:** if some CORRECT process deliver message m before m', then a CORRECT process deliver m' only after m.

- **Weak non-Uniform Total Order:** If two CORRECT processes p_i and p_j deliver two messages m and m', then p_i delivers m before m' if and only if p_j delivers m before m'.
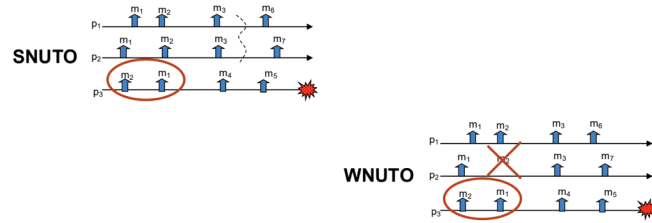

Figure 3.4:

Strong assumptions is more restrictive than weak assumptions (SUTO=>WUTO-SNUTO=>WNUTO).

3.4 Examples of Total Order Broadcast Variants

Figure 3.5: *

UA + SUTO: All processes deliver the same set of messages in the same order.

NUA + SUTO: Only correct processes deliver the same set/order; faulty process may miss messages (m_4, m_5 missing).
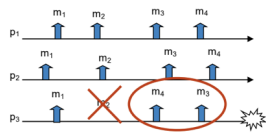

Figure 3.6: *

UA + WUTO: All processes deliver the same set with consistent relative order.

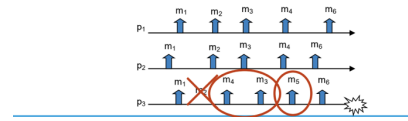NUA + WUTO: Only correct processes deliver the same set with consistent relative order.

Figure 3.7: *

UA + WNUTO: Correct processes deliver the same set with consistent relative order; faulty process behavior is undefined.

NUA + WNUTO: Correct processes deliver the same set with consistent relative order; m_5 missing from some processes.