Dependable Distributed Systems

Ida Francesca Benvenuto

novembre 2025

Chapter 1

Failure Detection (7)

Failure Detectors (FD) are mechanisms used in distributed systems to provide information about the status of processes, specifically whether they are operational (correct) or have failed (crashed). They help to overcome the challenges posed by asynchrony and failures in distributed systems.

1.1 Timing assumptions

Failure Detectors can be classified based on the timing assumptions they make about the system:

- **Synchronous FD:** Assume known upper bounds on message transmission times and processing times. They can provide accurate information about process failures.
- **Asynchronous FD:** Do not assume any timing bounds. They may provide inaccurate information, leading to false suspicions of process failures.
- **Partially Synchronous FD:** Assume that the system is asynchronous but eventually becomes synchronous. They can provide more reliable information over time.

manipulating the timing assumptions allows to design failure detectors that can provide different levels of accuracy and reliability in detecting process failures. the time managment can be :

- **Explicit:** Using timeouts to detect failures. If a process does not respond within a specified timeout period, it is suspected to have failed.
- **Implicit:** Relying on the absence of expected messages to infer failures. If a process does not send expected messages, it is suspected to have failed.

1.2 Failure Detector Abstraction

A Failure Detector (FD) is an abstraction that provides processes with information about the status of other processes in the system. It can be modeled as a module that generates two types of events:

- **Suspect Event:** Indicates that a process is suspected to have failed.
- **Restore Event:** Indicates that a previously suspected process is now considered operational again.

it generally is defined by two properties:

- **Completeness:** Ensures that all crashed processes are eventually suspected by all correct processes.
- **Accuracy:** Ensures that no correct process is ever suspected.

Based on these properties, failure detectors can be classified into different types:

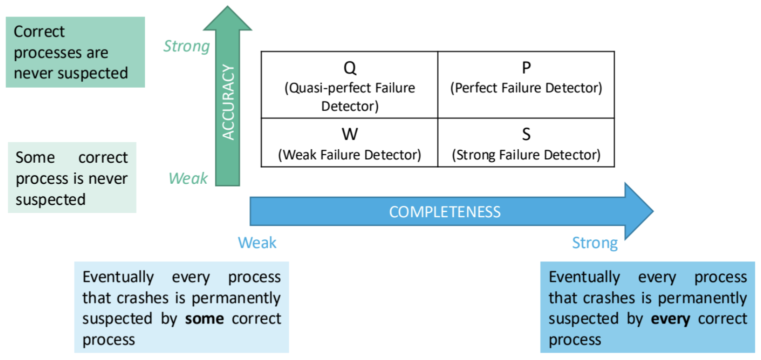

Correct processes are never suspected
Strong
ACCURACY
Weak
Some correct process is never suspected
Q (Quasi-perfect Failure Detector)
P (Perfect Failure Detector)
W (Weak Failure Detector)
S (Strong Failure Detector)
COMPLETENESS
Weak
Strong
Eventually every process that crashes is permanently suspected by some correct process
Eventually every process that crashes is permanently suspected by every correct process

Figure 1.1: Failure Detector types based on Completeness and Accuracy

Observations:

- Strong completeness means that eventually every crashed process is permanently suspected by every correct process.
- Weak completeness means that eventually every crashed process is permanently suspected by at least one correct process.
- Strong accuracy means that no correct process is ever suspected.
- Weak accuracy means that there is at least one correct process that is never suspected.

- Eventual strong accuracy means that after some time, no correct process is suspected.
- Eventual weak accuracy means that after some time, there is at least one correct process that is never suspected.

weakness guarantees that there is at least one correct process that is never suspected, while strong guarantees that no correct process is ever suspected. Partially speacking, this could be difficult to achieve and thus it is worth to consider weaker forms of accuracy.In particular, eventual accuracy properties are often more practical in real-world distributed systems, where temporary network issues or delays can lead to false suspicions of process failures.

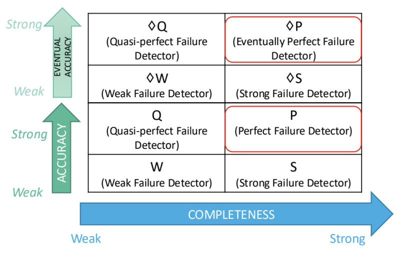

Strong
EVENTUAL ACCURACY
Weak
Strong
ACCURACY
Weak
◊Q
(Quasi-perfect Failure Detector)
◊P
(Eventually Perfect Failure Detector)
◊W
(Weak Failure Detector)
◊S
(Strong Failure Detector)
Q
(Quasi-perfect Failure Detector)
P
(Perfect Failure Detector)
W
(Weak Failure Detector)
S
(Strong Failure Detector)
COMPLETENESS
Weak
Strong

Figure 1.2: Failure Detector properties

1.3 Perfect Failure Detector (P)

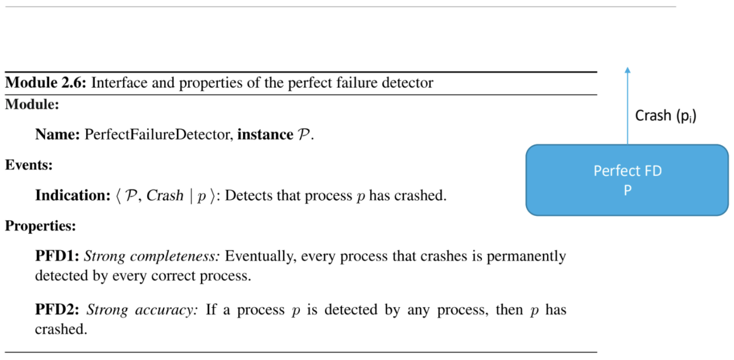
System model

We consider a distributed system with the following assumptions.

- **Synchronous System**
- **Crash Failures:** 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)

using its own clock and the bounds of synchronous model, a process can infer if another process is crashed or just slow.

specification of Perfect Failure Detector



Module 2.6: Interface and properties of the perfect failure detector

Module:

Name: PerfectFailureDetector, **instance** $\mathcal{P}$.

Events:

Indication: $\langle \mathcal{P}, Crash \mid p \rangle$: Detects that process p has crashed.

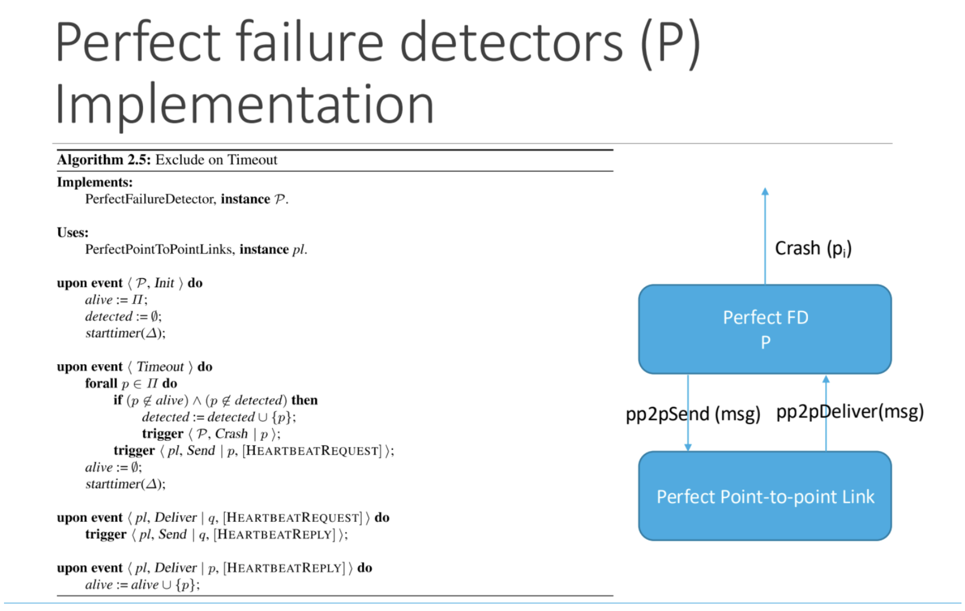
Properties:

PFD1: *Strong completeness:* Eventually, every process that crashes is permanently detected by every correct process.

PFD2: *Strong accuracy:* If a process p is detected by any process, then p has crashed.

Figure 1.3: Perfect Failure Detector properties

implementation of Perfect Failure Detector



Perfect failure detectors (P) Implementation

Algorithm 2.5: Exclude on Timeout

```
Implements:
    PerfectFailureDetector, instance $\mathcal{P}$.

Uses:
    PerfectPointToPointLinks, instance $pl$.

upon event $\langle \mathcal{P}, Init \rangle$ do
    $alive := \Pi$;
    $detected := \emptyset$;
    starttimer($\Delta$);

upon event $\langle Timeout \rangle$ do
    forall $p \in \Pi$ do
        if $(p \notin alive) \wedge (p \notin detected)$ then
            $detected := detected \cup \{p\}$;
            trigger $\langle \mathcal{P}, Crash \mid p \rangle$;
        trigger $\langle pl, Send \mid p$, [HEARTBEATREQUEST] $\rangle$;
    $alive := \emptyset$;
    starttimer($\Delta$);

upon event $\langle pl, Deliver \mid q$, [HEARTBEATREQUEST] $\rangle$ do
    trigger $\langle pl, Send \mid q$, [HEARTBEATREPLY] $\rangle$;

upon event $\langle pl, Deliver \mid p$, [HEARTBEATREPLY] $\rangle$ do
    $alive := alive \cup \{p\}$;
```

Figure 1.4: Perfect Failure Detector implementation

Chapter 2

Consensus(11)

2.1 Consensus Problem

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

2.2 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.

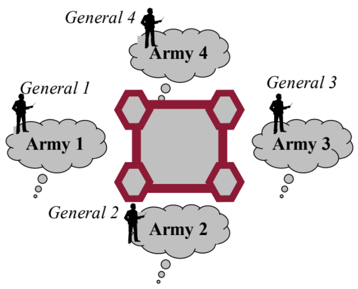


Figure 2.1: Consensus Problem

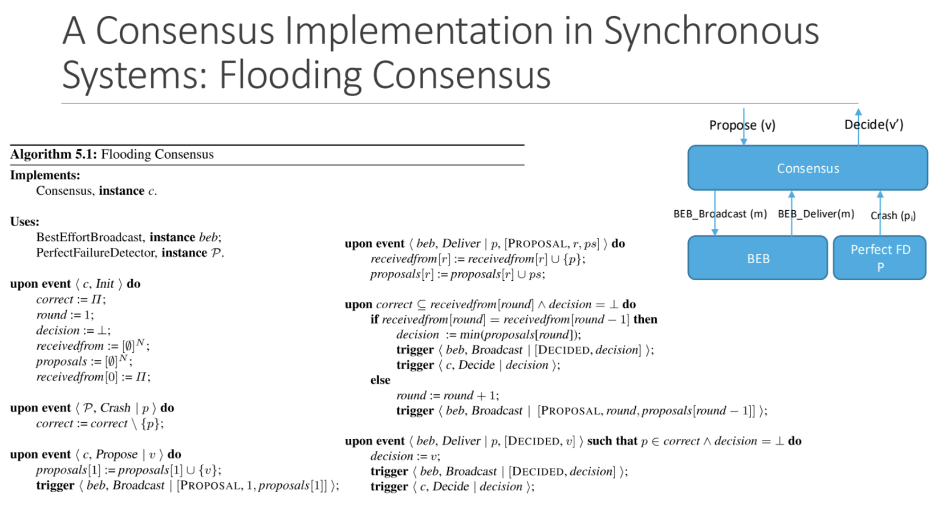
Impossibility of Consensus in Asynchronous Systems with Failures

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.

2.3 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)



A Consensus Implementation in Synchronous Systems: Flooding Consensus

```
Algorithm 5.1: Flooding Consensus

Implements:
    Consensus, instance $c$.

Uses:
    BestEffortBroadcast, instance $beb$;
    PerfectFailureDetector, instance $\mathcal{P}$.

upon event $\langle c, Init \rangle$ do
    $correct := \Pi$;
    $round := 1$;
    $decision := \bot$;
    $receivedfrom := [\emptyset]^N$;
    $proposals := [\emptyset]^N$;
    $receivedfrom[0] := \Pi$;

upon event $\langle \mathcal{P}, Crash \mid p \rangle$ do
    $correct := correct \setminus \{p\}$;

upon event $\langle c, Propose \mid v \rangle$ do
    $proposals[1] := proposals[1] \cup \{v\}$;
    trigger $\langle beb, Broadcast \mid [\textsc{Proposal}, 1, proposals[1]] \rangle$;

upon event $\langle beb, Deliver \mid p, [\textsc{Proposal}, r, ps] \rangle$ do
    $receivedfrom[r] := receivedfrom[r] \cup \{p\}$;
    $proposals[r] := proposals[r] \cup ps$;

upon $correct \subseteq receivedfrom[round] \wedge decision = \bot$ do
    if $receivedfrom[round] = receivedfrom[round-1]$ then
        $decision := \min(proposals[round])$;
        trigger $\langle beb, Broadcast \mid [\textsc{Decided}, decision] \rangle$;
        trigger $\langle c, Decide \mid decision \rangle$;
    else
        $round := round + 1$;
        trigger $\langle beb, Broadcast \mid [\textsc{Proposal}, round, proposals[round-1]] \rangle$;

upon event $\langle beb, Deliver \mid p, [\textsc{Decided}, v] \rangle$ such that $p \in correct \wedge decision = \bot$ do
    $decision := v$;
    trigger $\langle beb, Broadcast \mid [\textsc{Decided}, decision] \rangle$;
    trigger $\langle c, Decide \mid decision \rangle$;
```

Figure 2.2: Flooding-based Consensus Algorithm

2.4 Algorithm Description

2.4.1 Initialization

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

- $\textit{correct} := \Pi$ — Initialize the set of correct processes (all processes)
- $\textit{round} := 1$ — Start from round 1
- $\textit{decision} := \perp$ — The decision is still empty ($\perp$ means "not decided")
- $\textit{receivedFrom}[r] := [\emptyset]^N$ — Array tracking from which processes we have received messages for each round
- $\textit{proposals}[r] := [\emptyset]^N$ — Array tracking the proposals received for each round

2.4.2 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.

2.4.3 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

2.4.4 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.

2.4.5 Practical Example

2.5 Uniform Consensus Specification

The difference between uniform consensus and (non-uniform) consensus is that uniform consensus requires **Uniform Agreement**: no two processes

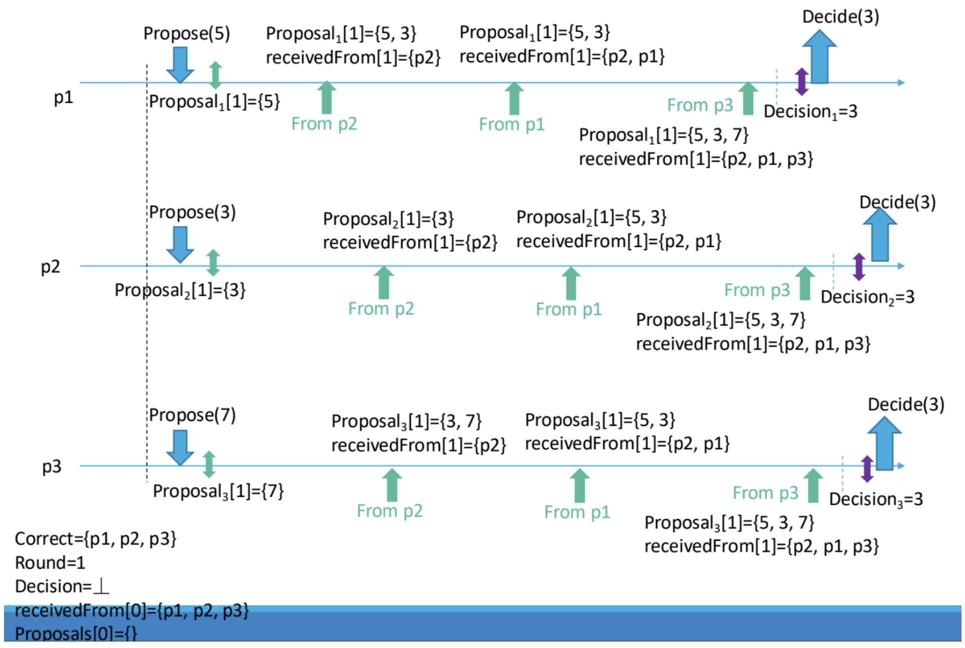


Figure 2.3: Example of Consensus Algorithm Execution

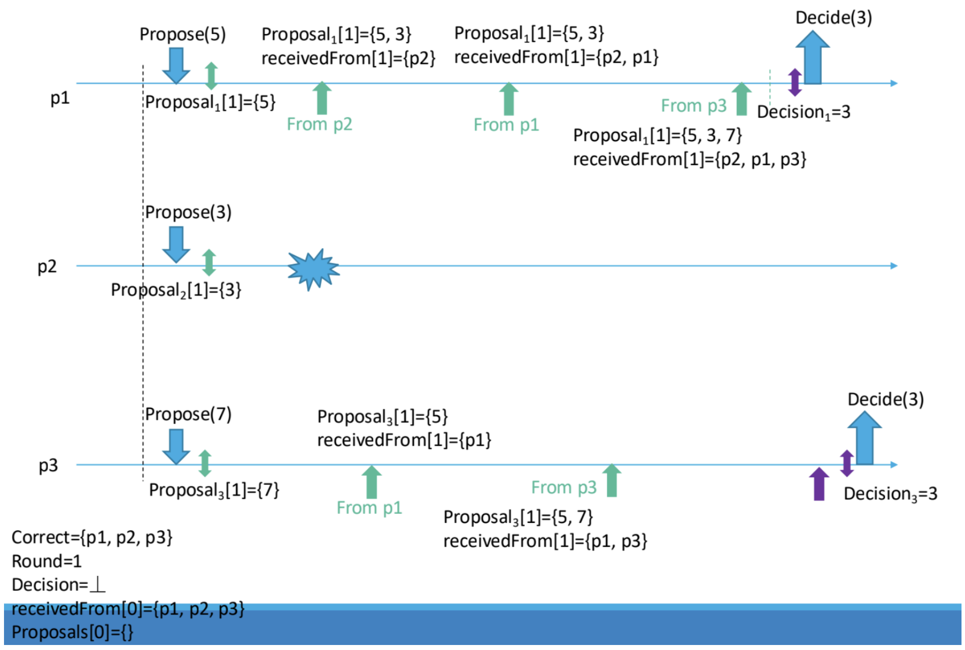


Figure 2.4: P3 decide the min value after the decision of p1

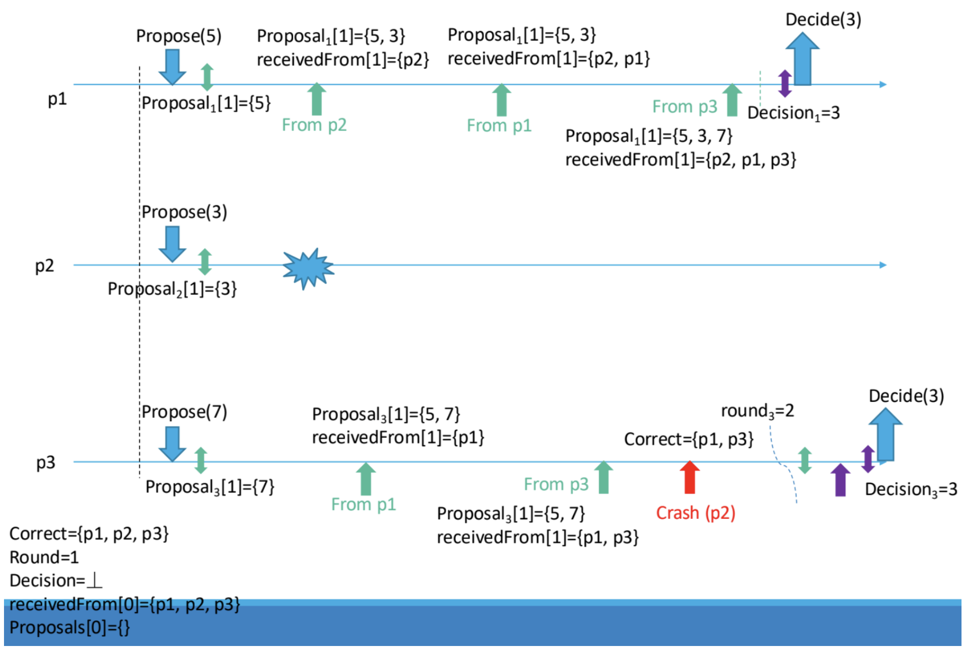
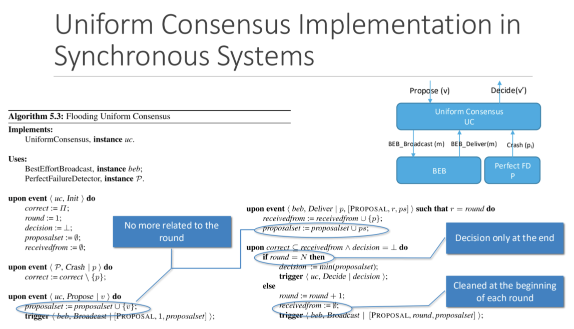


Figure 2.5: p2 is the only one thet can decide but crash before the broadcast of the value, so P1 and P3 go to the next round and decide an other value

(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



Uniform Consensus Implementation in Synchronous Systems

Algorithm 5.3: Flooding Uniform Consensus

```
Implements:
    UniformConsensus, instance uc.

Uses:
    BestEffortBroadcast, instance beb;
    PerfectFailureDetector, instance $\mathcal{P}$.

upon event ⟨ uc, Init ⟩ do
    correct := $\Pi$;
    round := 1;
    decision := $\perp$;
    proposalset := $\emptyset$;
    receivedfrom := $\emptyset$;

upon event ⟨ $\mathcal{P}$, Crash | p ⟩ do
    correct := correct \ {p};

upon event ⟨ uc, Propose | v ⟩ do
    proposalset := proposalset ∪ {v};
    trigger ⟨ beb, Broadcast | [PROPOSAL, 1, proposalset] ⟩;

upon event ⟨ beb, Deliver | p, [PROPOSAL, r, ps] ⟩ such that r = round do
    receivedfrom := receivedfrom ∪ {p};
    proposalset := proposalset ∪ ps;

upon correct ⊆ receivedfrom ∧ decision = $\perp$ do
    if round = N then
        decision := min(proposalset);
        trigger ⟨ uc, Decide | decision ⟩;
    else
        round := round + 1;
        receivedfrom := $\emptyset$;
        trigger ⟨ beb, Broadcast | [PROPOSAL, round, proposalset] ⟩;
```

Figure 2.6: Uniform Consensus properties

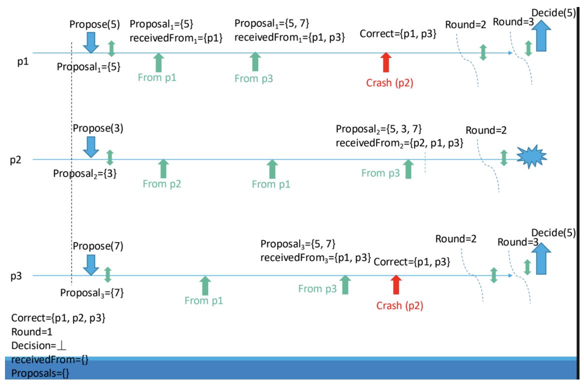


Figure 2.7: at the round 3 there is a decision, that is 5 becaus when start a new round the receivedFrom set is = 0

Chapter 3

Paxos (12)

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).

3.1 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 4

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.

4.1 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.

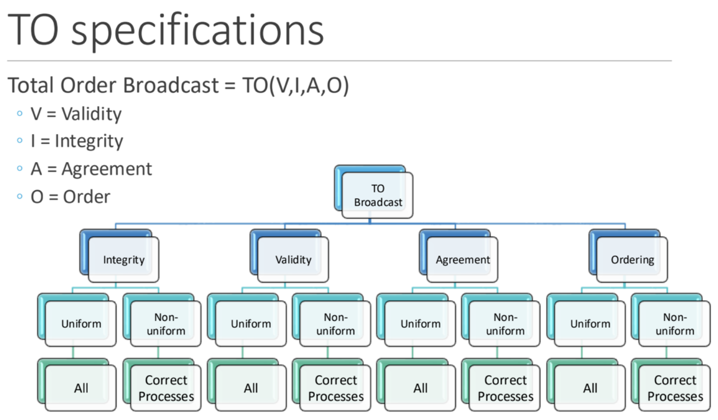
4.2 Total Order specification

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.



TO specifications

Total Order Broadcast = TO(V,I,A,O)
- V = Validity
- I = Integrity
- A = Agreement
- O = Order

Figure 4.1: Total Order Broadcast properties

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

4.3 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.

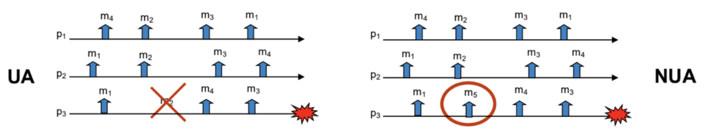


Figure 4.2: Non-Uniform Agreement vs Uniform Agreement

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.

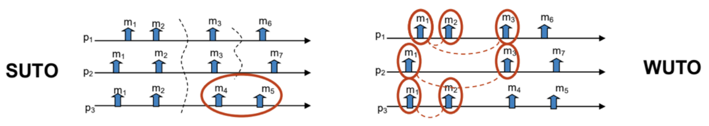


Figure 4.3: Strong Uniform Total Order vs Weak Uniform Total Order

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'.

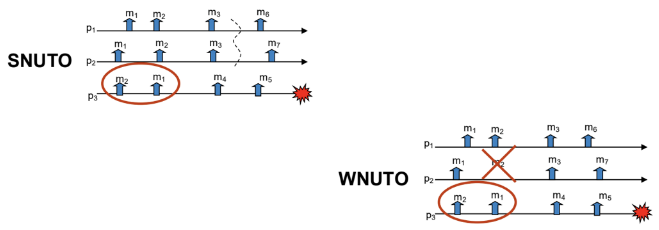


Figure 4.4:

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

4.4 Examples of Total Order Broadcast Variants

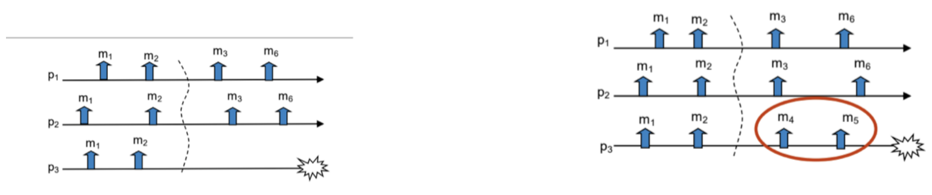


Figure 4.5: *
UA + SUTO: All processes deliver the same set of messages in the same order.
NUA + SUTO: Only correct processes deliver the same set/order; faulty process may miss messages (m_4, m_5 missing).

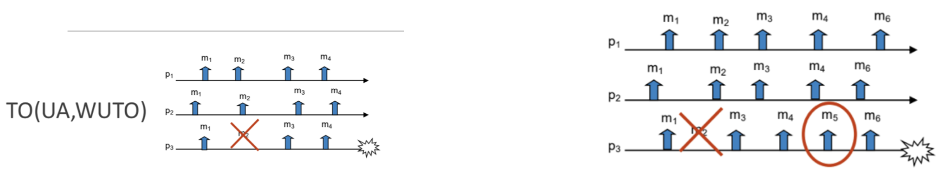


Figure 4.6: *
UA + WUTO: All processes deliver the same set with consistent relative order.
NUA + WUTO: Only correct processes deliver the same set with consistent relative order.

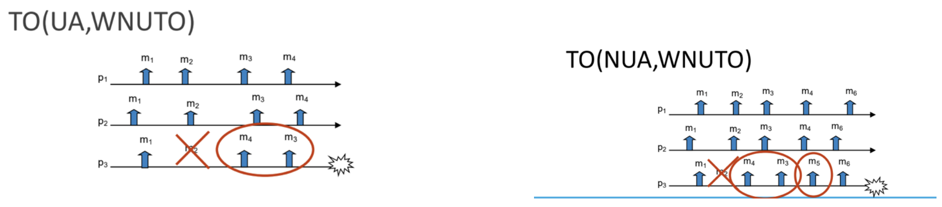


Figure 4.7: *
UA + WNUTO: Correct processes deliver the same set with consistent relative order; faulty process behavior is undefined.
NUA + WNUTO: Correct processes deliver the same set with consistent relative order; m_5 missing from some processes.