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:

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.

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 $\langle pl, Send \mid p, [\text{HEARTBEATREQUEST}] \rangle$;
  $alive := \emptyset$;
  $starttimer(\Delta)$;

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

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

Figure 1.4: Perfect Failure Detector implementation

Initialization

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

- $alive := \Pi$ — Initialize the set of all processes
- $detected := \emptyset$ — Initialize the set of detected crashed processes as empty

- Set a periodic timer to trigger every $\Delta + \Phi$ time units

Main Event Handlers

- **Timer Event:** When the timer triggers (upon event $\langle p, \text{timeout} \rangle$):

```
for each process p in Pi do
  if p not in alive and p not in detected then
    detected := detected union {p};
    trigger <fd, Suspect | p>;
  send heartbeat message to p;
alive := empty set;
```

  The process checks for processes that did not respond with a heartbeat and suspects them if they are not already suspected. It then sends heartbeat messages to all processes and resets the alive set.

- **Heartbeat Message Event:** When a heartbeat message is received from process q (upon event $\langle p, \text{Heartbeat} | q \rangle$):

```
alive := alive union {q};
```

  The process that replied, they adds q to the alive set.

es:

Figure 1.5: Example of Perfect Failure Detector execution

correctness

- **Strong Completeness:** Eventually, every crashed process is permanently suspected by every correct process. This is ensured because crashed processes will not send heartbeat messages, leading to their suspicion.
- **Strong Accuracy:** No correct process is ever suspected. This is ensured because correct processes will always respond with heartbeat messages within the expected time frame.

if the timeout is very short, correct processes may be suspected due to network delays or processing time variations.
if the timeout is very long, crashed processes may not be suspected promptly, leading to delays in failure detection.
Thus, the choice of timeout value is crucial for the effectiveness of the Perfect Failure Detector.

1.4 Eventually Perfect Failure Detector ($\Diamond P$)

System model

We consider a distributed system with the following assumptions.

- **Partially Synchronous System:** The system is asynchronous but eventually becomes synchronous.
- **Crash Failures:** Processes can fail by *crash*: a crashed process stops executing its algorithm forever, but they can't behave maliciously. (Byzantine failures are not considered here)
- **Perfect point to point links**

Crashed processes can be detected only after an unknown time t; before time t there could be false suspicions (it behaves like an asynchronous system).

Basic Construction Rules

An Eventually Perfect Failure Detector uses timeouts to suspect processes that did not send expected messages:

- **Suspicion mechanism:** A suspect may be wrong. A process p_i may suspect another one p_j if the current timeout is too short.
- **Ready to change judgment:** $\Diamond P$ is ready to change its judgment as soon as it receives a message from p_j. In this case, the timeout value is updated to accommodate for longer delays.

- **Permanent suspicion:** If p_j has actually crashed, p_i does not change its judgment anymore.

The key difference from Perfect FD is that Eventually Perfect FD can make mistakes initially (false suspicions), but eventually stabilizes and behaves correctly once the system becomes synchronous and timeouts are adjusted appropriately.

specification of Eventually Perfect Failure Detector

Module 2.8: Interface and properties of the eventually perfect failure detector

Module:

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

Events:

Indication: $\langle \diamond\mathcal{P}, \mathit{Suspect} \mid p \rangle$: Notifies that process p is suspected to have crashed.

Indication: $\langle \diamond\mathcal{P}, \mathit{Restore} \mid p \rangle$: Notifies that process p is not suspected anymore.

Properties:

EPFD1: *Strong completeness:* Eventually, every process that crashes is permanently suspected by every correct process.

EPFD2: *Eventual strong accuracy:* Eventually, no correct process is suspected by any correct process.

Figure 1.6: Eventually Perfect Failure Detector properties

implementation of Eventually Perfect Failure Detector

Algorithm 2.7: Increasing Timeout

Implements:
EventuallyPerfectFailureDetector, **instance** $\diamond\mathcal{P}$.

Uses:
PerfectPointToPointLinks, **instance** *pl*.

upon event $\langle \diamond\mathcal{P}, \mathit{Init} \rangle$ **do**
  $\mathit{alive} := \Pi$;
  $\mathit{suspected} := \emptyset$;
  $\mathit{delay} := \Delta$;
  $\mathit{starttimer}(\mathit{delay})$;

upon event $\langle \mathit{Timeout} \rangle$ **do**
  if $\mathit{alive} \cap \mathit{suspected} \neq \emptyset$ **then**
    $\mathit{delay} := \mathit{delay} + \Delta$;
  forall $p \in \Pi$ **do**
    if $(p \notin \mathit{alive}) \wedge (p \notin \mathit{suspected})$ **then**
      $\mathit{suspected} := \mathit{suspected} \cup \{p\}$;
      trigger $\langle \diamond\mathcal{P}, \mathit{Suspect} \mid p \rangle$;
    else if $(p \in \mathit{alive}) \wedge (p \in \mathit{suspected})$ **then**
      $\mathit{suspected} := \mathit{suspected} \setminus \{p\}$;
      trigger $\langle \diamond\mathcal{P}, \mathit{Restore} \mid p \rangle$;
    trigger $\langle pl, \mathit{Send} \mid p, [\text{HEARTBEATREQUEST}] \rangle$;
  $\mathit{alive} := \emptyset$;
  $\mathit{starttimer}(\mathit{delay})$;

upon event $\langle pl, \mathit{Deliver} \mid q, [\text{HEARTBEATREQUEST}] \rangle$ **do**
  trigger $\langle pl, \mathit{Send} \mid q, [\text{HEARTBEATREPLY}] \rangle$;

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

Figure 1.7: Eventually Perfect Failure Detector implementation

Main Event Handlers

- **Timer Event:** When the timer triggers (upon event $\langle p, \text{timeout} \rangle$):

```
If there exists a process q in alive and q in suspected then
```

```
    delay= delay + d;
for each process p in Pi do
  if p not in alive and p not in suspected then
    suspected := suspected union {p};
    trigger <fd, Suspect | p>;
    send heartbeat message to p;
  if p in alive and p in suspected then
    suspected := suspected \ {p};
    trigger <fd, Restore | p>;
    send heartbeat message to p;
alive := empty set;
start timer with delay;
```

The process checks for processes that did not respond with a heartbeat and suspects them if they are not already suspected. If a previously suspected process responds, it is restored. The timeout delay is adjusted based on the suspicion status.

example

Figure 1.8: Example of Eventually Perfect Failure Detector execution

1.5 correctness

- **Strong Completeness:** If a process crashes, it will eventually be suspected by all correct processes. This is ensured because crashed processes will not send heartbeat messages, leading to their suspicion.
- **Eventual Strong Accuracy:** After some unknown time, no correct process is suspected. This is ensured because once the system becomes synchronous and timeouts are adjusted, correct processes will always respond with heartbeat messages within the expected time frame.

Chapter 2

Leadership Election (7-b)

Leadership Election is a fundamental problem in distributed systems where processes need to agree on a single process to act as the leader or coordinator for certain tasks. The leader is responsible for making decisions, coordinating actions, and managing resources among the processes. in the asynchronous system with crash failures,the system stabilizes after some unknown time t, and after t there is a correct process that is never suspected. thus there are two opstions:

- put the assumption on the system (Including links and processes)
- create a separate abstraction that encapsulates the assumption

incapsulating the assumption in a separate abstraction is better because it allows to separate concerns, making the system more modular and easier to understand. While do all in a unique algorithm can lead to complex and hard to proof. thus, we can use a single correct process that monitoring features of the system to elect a leader. we can use several oracle(called leader election module) that provide information about the processes.

2.1 Leader Election

specification of Leader Election

implementation of Leader Election using Eventually Perfect Failure Detector

Main Event Handlers

- **Initialization Event:** When the system starts (upon event $\langle p, \text{Init} \rangle$):

```
leader != maxrank(Pi / suspected) - the process tha have the highest ran
leader := max(Pi/ suspected);
trigger <le, NewLeader | leader>;
```

Module 2.7: Interface and properties of leader election

Module:

Name: LeaderElection, **instance** *le*.

Events:

Indication: $\langle$ *le*, *Leader* | *p* $\rangle$: Indicates that process *p* is elected as leader.

Properties:

LE1: *Eventual detection:* Either there is no correct process, or some correct process is eventually elected as the leader.

LE2: *Accuracy:* If a process is leader, then all previously elected leaders have crashed.

Figure 2.1: Leader Election properties

Algorithm 2.6: Monarchical Leader Election

Implements:
LeaderElection, **instance** *le*.

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

upon event $\langle$ *le*, *Init* $\rangle$ **do**
suspected := $\emptyset$;
leader := $\perp$;

upon event $\langle$ $\mathcal{P}$, *Crash* | *p* $\rangle$ **do**
suspected := *suspected* $\cup$ $\{p\}$;

upon *leader* $\neq$ maxrank(Π $\setminus$ *suspected*) **do**
leader := maxrank(Π $\setminus$ *suspected*);
trigger $\langle$ *le*, *Leader* | *leader* $\rangle$;

Figure 2.2: Leader Election implementation using Eventually Perfect Failure Detector

The process selects the process with the highest rank among those not suspected as the leader and triggers a NewLeader event.

example

Figure 2.3: Example of Leader Election execution

the correctness is guaranteed only if there is perfect failure detector. otherwise, there are two possibilities:

- best scenario: suspect wrongly a crashed process, but the processes that continue to work are correct.
- worst scenario: FALSE POSITIVE, 2 processes could belive to have 2 different leaders.

2.2 Eventual Leader Election

specification of Eventual Leader Election

Module 2.9: Interface and properties of the eventual leader detector

Module:

Name: EventualLeaderDetector, **instance** Ω.

Events:

Indication: $\langle\ \Omega,\ Trust \mid p\ \rangle$: Indicates that process p is trusted to be leader.

Properties:

ELD1: *Eventual accuracy:* There is a time after which every correct process trusts some correct process.

ELD2: *Eventual agreement:* There is a time after which no two correct processes trust different correct processes.

Figure 2.4: Eventual Leader Election properties

Key Properties and Behavior

At first, every process may believe in a different leader, but after some unknown time t, all correct processes agree on the same correct leader.

How it works:

- Every process decides a leader based on the information provided by the Eventually Perfect Failure Detector ($\Diamond P$).
- After time t (when the system stabilizes), all correct processes agree on the same correct leader.
- We don't know when this time t is, and during this initial period, crashed processes can be elected as leaders (false positives).
- Once the system stabilizes and elects a leader, the leader is not changed until it crashes. This stable leader is called **STABILIZED**.

Important assumption: There always exists at least one correct process.

implementation of Eventual Leader Election using Eventually Perfect Failure Detector

Algorithm 2.8: Monarchical Eventual Leader Detection

Implements:
EventualLeaderDetector, **instance** Ω.

Uses:
EventuallyPerfectFailureDetector, **instance** $\diamond\mathcal{P}$.

upon event $\langle\ \Omega, Init\ \rangle$ **do**
$suspected := \emptyset$;
$leader := \perp$;

upon event $\langle\ \diamond\mathcal{P}, Suspect \mid p\ \rangle$ **do**
$suspected := suspected \cup \{p\}$;

upon event $\langle\ \diamond\mathcal{P}, Restore \mid p\ \rangle$ **do**
$suspected := suspected \setminus \{p\}$;

upon $leader \neq maxrank(\Pi \setminus suspected)$ **do**
$leader := maxrank(\Pi \setminus suspected)$;
trigger $\langle\ \Omega, Trust \mid leader\ \rangle$;

Figure 2.5: Eventual Leader Election implementation using Eventually Perfect Failure Detector

Example:

Figure 2.6: Example of Eventual Leader Election execution

2.3 eventual leader election with Crash-Recovery

chrash-recovery model: processes can crash and later recover, resuming their operations from the state they had before crashing. in this system the correct processes are p that never crash or that crash, recover and never crash again.

implementation of Eventual Leader Election in Crash-Recovery model

Figure 2.7: Eventual Leader Election implementation in Crash-Recovery model

Initialization

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

```
epoch := 0; -- Initialize epoch counter(how many times the process has crashed an
store(epoch); -- Store the epoch counter in stable storage
candidates := empty set; -- Initialize the set of candidate leaders as empty
trigger < \Omega , Recovery | >; -- Trigger a Recovery event
```

trigger Recovery

When a process recovers (upon event $\langle p, \text{Recovery} \rangle$):

```
leader := maxrank(Pi) -- Select the process with the highest rank as the leader
trigger <le, trust | leader>; -- Trigger a trust event with the selected leader
delay := deltha; -- Initialize the delay for heartbeat messages
retrieve(epoch); -- Retrieve the epoch counter from stable storage
epoch := epoch + 1; -- Increment the epoch counter
store(epoch); -- Store the updated epoch counter in stable storage
forall q in Pi do
    send [HEARTBEAT, epoch] to q; -- Send heartbeat messages to all processes wit
candidates := empty set; -- Reset the set of candidate leaders
start timer with delay; -- Start the timer with the specified delay
```

Figure 2.8: Eventual Leader Election implementation in Crash-Recovery model

Timeout Event

When the timer triggers (upon event $\langle p, \text{timeout} \rangle$):

```
elect new leader from candidates;
if the new leader != leader then. -- If a new leader is elected
    leader := new leader;
    trigger <le, trust | leader>;
for all q in Pi do
    send [HEARTBEAT, epoch] to q; -- Send heartbeat messages to all processes wit
candidates := empty set; -- Reset the set of candidate leaders
start timer with delay; -- Start the timer with the specified delay
```

example

Figure 2.9: Example of Eventual Leader Election execution in Crash-Recovery model

the timer is incremented when a process receive a heartbeat from a process with higher epoch.(p1)

Figure 2.10: Example of Eventual Leader Election execution in Crash-Recovery model - part 2

after the restore of p1, the epoch is incremented and p2 become the leader.

Chapter 3

Broadcast Communication(9)

Up to now, the comunication model used is between two processes (like clien-server enviroment). In distributed systems, it is often necessary to communicate with multiple processes simultaneously. Broadcast communication is a method that allows a process to send a message to all other processes in the system.

3.1 Best Effort Broadcast (BEB)

Best Effort Broadcast (BEB) is a communication primitive that allows a process to send a message to all other processes in a distributed system with the following properties:

- **Validity:** If a correct process broadcasts a message, then all correct processes eventually deliver that message.
- **No Duplication:** No message is delivered more than once to any process.
- **No Creation:** No message is delivered unless it was previously broadcast by some process.

specification of Best Effort Broadcast

Module 3.1: Interface and properties of best-effort broadcast

Module:

Name: BestEffortBroadcast, **instance** *beb*.

Events:

Request: $\langle$ *beb*, *Broadcast* $\mid m \rangle$: Broadcasts a message m to all processes.

Indication: $\langle$ *beb*, *Deliver* $\mid p, m \rangle$: Delivers a message m broadcast by process p.

Properties:

BEB1: *Validity:* If a correct process broadcasts a message m, then every correct process eventually delivers m.

BEB2: *No duplication:* No message is delivered more than once.

BEB3: *No creation:* If a process delivers a message m with sender s, then m was previously broadcast by process s.

Figure 3.1: Best Effort Broadcast properties

implementation of Best Effort Broadcast

Algorithm 3.1: Basic Broadcast

Implements:
BestEffortBroadcast, **instance** *beb*.

Uses:
PerfectPointToPointLinks, **instance** *pl*.

upon event $\langle$ *beb*, *Broadcast* $\mid m \rangle$ **do**
  forall $q \in \Pi$ **do**
    trigger $\langle$ *pl*, *Send* $\mid q, m \rangle$;

upon event $\langle$ *pl*, *Deliver* $\mid p, m \rangle$ **do**
  trigger $\langle$ *beb*, *Deliver* $\mid p, m \rangle$;

System model

- Asynchronous system
- perfect links
- crash failures

Figure 3.2: Best Effort Broadcast implementation

correctness

- **Validity:** If a correct process broadcasts a message, it sends the message to all processes. In our model there are perfect point-to-point links, so all correct processes will eventually receive and deliver the message.
- **No Duplication:** Each process maintains a set of delivered messages. A message is only delivered if it has not been delivered before, ensuring no duplication.

- **No Creation:** perfect point-to-point links ensure that messages are only delivered if they were sent by some process. Since processes only deliver messages they receive, no message can be created out of thin air.

3.2 (Regular) Reliable Broadcast(RB)

Reliable Broadcast (RB) is a communication primitive that extends Best Effort Broadcast by providing stronger guarantees in the presence of process failures. It ensures that messages are delivered reliably even if some processes crash. the specification of RB are the same of BEB plus:

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

difference:

Figure 3.3: Difference between Best Effort Broadcast and Reliable Broadcast

implementation of Reliable Broadcast IN Synchronous System

Figure 3.4: Reliable Broadcast implementation in Synchronous System

initialization

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

```
correct := Pi; -- Initialize the set of correct processes (all processes)
from [p] := empty set[n]; -- Array to track messages received from each process
```

deliver event

When a message is delivered (upon event $\langle beb, \text{Deliver}|q, [DATA, s, m] \rangle$):

```
if m not in from[q] then -- If the message has not been received from process q b
    from[q] := from[q] union {m}; -- Add the message to the set of messages recei
    trigger <rb, Deliver | s, m>; -- Trigger the delivery event for the reliable
    if s is not in correct then -- If the sender process s is not in the set of c
        trigger <beb, Broadcast | [DATA, s, m]>; -- Trigger the delivery event fo
```

example

Figure 3.5: Example of Reliable Broadcast execution in Asynchronous System

Figure 3.6: Example of Reliable Broadcast execution in Asynchronous System

Lazy Reliable Broadcast

in the best scenario, the RB implementation can generate 1 BEB message per RB message .
in the worst scenario, the RB implementation can generate N-1 BEB messages per RB message.

if the failure detector is not perfect, the algorithm is not correct, because it may trigger unnecessary retransmissions or fail to detect crashed processes accurately.

3.3 Reliable Broadcast (RB) in Asinchronous System

implementation of Reliable Broadcast IN Asynchronous System

Figure 3.7: Reliable Broadcast implementation in Asynchronous System

in this case the best scenario is equal to worst scenario = N BEB messages per RB message, because every message is retransmitted by every process.

3.4 Uniform Reliable Broadcast (URB)

Uniform Reliable Broadcast (URB) is a communication primitive that extends Reliable Broadcast by providing even stronger guarantees, particularly in the presence of process failures. It ensures that messages are delivered uniformly across all processes, meaning that if any process delivers a message, then all correct processes will eventually deliver that message. thus, the specification of URB are the same of RB plus:

- **Uniform Agreement:** If any process (correct or faulty) delivers a message, then all correct processes eventually deliver that message.

difference:

Figure 3.8: Difference between Reliable Broadcast and Uniform Reliable Broadcast

implementation of Uniform Reliable Broadcast IN Asynchronous System

Algorithm 3.4: All-Ack Uniform Reliable Broadcast

Implements:
UniformReliableBroadcast, **instance** *urb*.

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

upon event $\langle$ *urb*, *Init* $\rangle$ **do**
delivered := $\emptyset$;
pending := $\emptyset$;
correct := Π;
forall m **do** *ack*$[m]$:= $\emptyset$;

upon event $\langle$ *urb*, *Broadcast* | m $\rangle$ **do**
pending := *pending* $\cup$ $\{(self, m)\}$;
trigger $\langle$ *beb*, *Broadcast* | [DATA, *self*, m] $\rangle$;

upon event $\langle$ *beb*, *Deliver* | p, [DATA, s, m] $\rangle$ **do**
ack$[m]$:= *ack*$[m]$ $\cup$ $\{p\}$;
if $(s, m) \notin$ *pending* **then**
pending := *pending* $\cup$ $\{(s, m)\}$;
trigger $\langle$ *beb*, *Broadcast* | [DATA, s, m] $\rangle$;

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

function *candeliver*(m) **returns** Boolean **is**
return (*correct* $\subseteq$ *ack*$[m]$);

upon exists $(s, m) \in$ *pending* such that *candeliver*(m) $\wedge$ $m \notin$ *delivered* **do**
delivered := *delivered* $\cup$ $\{m\}$;
trigger $\langle$ *urb*, *Deliver* | s, m $\rangle$;

Figure 3.9: Uniform Reliable Broadcast implementation in Asynchronous System

Chapter 4

Consensus (11)

4.1 Consensus Problem

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

4.2 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 4.1: 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.

4.3 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 4.2: Flooding-based Consensus Algorithm

4.4 Algorithm Description

4.4.1 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* $:= \perp$ — The decision is still empty ($\perp$ 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

4.4.2 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.

4.4.3 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

4.4.4 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.

4.4.5 Practical Example

4.5 Uniform Consensus Specification

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

Figure 4.3: Example of Consensus Algorithm Execution

Figure 4.4: P3 decide the min value after the decision of p1

Figure 4.5: 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 4.6: Uniform Consensus properties

Figure 4.7: at the round 3 there is a decision, that is 5 becaus when start a new round the receivedFrom set is = 0

Chapter 5

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).

5.1 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 6

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.

6.1 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.

6.2 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 6.1: Total Order Broadcast properties

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

6.3 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 6.2: 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 6.3: 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 6.4:

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

6.4 Examples of Total Order Broadcast Variants

Figure 6.5: *
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 6.6: *
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 6.7: *
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.