Distributed Systems
Instructor: Ajay Kshemkalyani

Failure Detectors

Presented by,
Archana
Bharath
Lakshmi
Failure Detector

• **Failure detector** is an application that is responsible for detection of node failures or crashes in a distributed system.

• A **failure detector** is a distributed oracle that provides hints about the operational status of other processes.
Why Failure Detectors

• The design and verification of \textit{fault-tolerant} distributed system is a difficult problem.

• The \textit{detection of process failures} is a crucial problem, system designers have to cope with in order to build fault tolerant distributed platforms
Synchronous Vs Asynchronous

• A distributed system is **synchronous** if:
  – there is a **known upper bound** on the transmission delay of messages
  – there is a **known upper bound** on the processing time of a piece of code

• A distributed system is **asynchronous** if:
  – there is **no bound** on the transmission delay of messages
  – there is **no bound** on the processing time of a piece of code
Why Failure Detectors cont...

• To stop waiting or not to stop waiting?
• Unfortunately, it is impossible to distinguish with certainty a crashed process from a very slow process in a purely asynchronous distributed system.
• Look at two major problems
  • Consensus
  • Atomic Broadcast
Liveness & Safety

• The problem can be defined with a safety and a liveness property.
• The safety property stipulates that “nothing bad ever happens”
• The liveness property stipulates that “something good eventually happens”
‘q’ not crashed

• The message from \textit{q to p} is only very slow.
• Assuming that ‘q’ has crashed will violate the \textbf{safety} property
‘q’ has crashed

• To prevent the bad previous scenario from occurring, p must wait until it gets q’s message.
• It is easy to see that p will wait forever, and the liveness property of the application will never be satisfied.
Characterizing Failure Detectors

- Completeness
  - Suspect every process that actually crashes

- Accuracy
  - Limit the number of correct processes that are suspected
Completeness

• **Strong Completeness**
  – Eventually, every crashed process is permanently suspected by *every* correct process

• **Weak Completeness**
  – Eventually, every crashed process is permanently suspected by *some* correct process
Strong Completeness
Weak Completeness

Suspects_{p_0}\{p_1\}

Suspects_{p_1}\{}

Suspects_{p_2}\{\}

Suspects_{p_3}\{p_1,p_4\}

Suspects_{p_4}\{}

Suspects_{p_5}\{p_4\}
Accuracy

• Strong Accuracy
  – A process is *never* suspected before it crashes by any correct process

• Weak Accuracy
  – Some correct process *never* suspected by any correct process

**Perpetual Accuracy!**
As these properties hold all the times
Eventual Accuracy

• Eventual Strong Accuracy
  – After a time, correct processes do not suspect correct processes

• Eventual Weak Accuracy
  – After a time, some correct process is not suspected by any correct process
# Failure Detector Classes

<table>
<thead>
<tr>
<th>Completeness</th>
<th>Accuracy</th>
<th>Strong</th>
<th>Weak</th>
<th>Eventually Strong</th>
<th>Eventually Weak</th>
</tr>
</thead>
<tbody>
<tr>
<td>Strong</td>
<td>Strong</td>
<td>Perfect</td>
<td>Strong</td>
<td>Eventually Perfect</td>
<td>Eventually Strong</td>
</tr>
<tr>
<td></td>
<td>Weak</td>
<td>( P )</td>
<td>( S )</td>
<td>( P )</td>
<td>( S )</td>
</tr>
<tr>
<td>Weak</td>
<td>( v )</td>
<td>( v )</td>
<td>( W )</td>
<td>( W )</td>
<td></td>
</tr>
</tbody>
</table>
Reducibility

• A Failure detector D is reducible to another failure detector D’ if there exist a reduction algorithm $T_D \rightarrow D'$ that transforms D to D’.

• Then
  – D’ is Weaker than D (i.e) $D \sqsubseteq D'$

• If $D \sqsubseteq D'$ and $D' \sqsubseteq D$ then D and D’ are equivalent (i.e) $D \equiv D'$

• Suppose a given algorithm ‘A’ requires failure detector D’, but only D is available.
Example

\[ D \]

\[ T_{D \rightarrow D'} \]

\[ D' \text{ emulated} \]

Algorithm A uses \( D' \)
Reducibility of FD

• $\mathcal{P} \subseteq \mathcal{V} \ ; \ s \subseteq \mathcal{W} \ ; \ \Diamond \mathcal{P} \subseteq \Diamond \mathcal{V} \ ; \ \Diamond s \subseteq \Diamond \mathcal{W}$

• $\mathcal{V} \subseteq \mathcal{P} \ ; \ \mathcal{W} \subseteq s \ ; \ \Diamond \mathcal{V} \subseteq \Diamond \mathcal{P} \ ; \ \Diamond \mathcal{W} \subseteq \Diamond s$

• $\mathcal{P} \equiv \mathcal{V} \ ; \ s \equiv \mathcal{W} \ ; \ \Diamond \mathcal{P} \equiv \Diamond \mathcal{V} \ ; \ \Diamond s \equiv \Diamond \mathcal{W}$

• Hence if we solve a problem for four failure detectors with strong completeness, the problem is automatically solved for the remaining four failure detectors.
Comparing Failure detectors by Reducibility

\[ C \rightarrow C' : C' \text{ is strictly weaker than } C \]
\[ C' \quad C' : C \text{ is equivalent to } C' \]
Failure Detectors : Reducibility

• Two failure detectors are equivalent if they are reducible to each other.
• Failure detector with weak completeness is equivalent to corresponding failure detector with strong completeness.
• $P \equiv \nu ; \Diamond P \equiv \Diamond \nu ; S \equiv W ; \Diamond S \equiv \Diamond W$
• Solving a problem for the four failure detectors with strong completeness, automatically solves for the remaining four failure detectors.
Weak to Strong Completeness

- Every process $p$ executes the following:
- Output $p \leftarrow \text{Null}$
- cobegin
  - //Task 1: repeat forever
    - suspects $p \leftarrow D_p \{ p \text{ queries its local failure detector module } D_p \}$
    - send($p, \text{suspects}_p$) to all other processes.
  - //Task 2: when receive $(q, \text{suspects}_q)$ for a process $q$
    - output $p \leftarrow \text{output}_p \cup \text{suspects}_q - \{q\} \{\text{output}_p \text{ emulates } E_p\}$
- coend
Weak to Strong Completeness
Weak to Strong Completeness
The consensus problem

- **Termination**: Every correct process eventually decides some value.
- **Uniform integrity**: Every process decides at most once.
- **Agreement**: No two correct processes decide differently.
- **Uniform validity**: If a process decides a value \( v \), then some process proposed \( v \).
- It is widely known that the consensus cannot be solved in *asynchronous systems in the presence of even a single crash failure*
Solutions to the consensus problem

• $P \equiv v ; \Diamond P \equiv \Diamond v ; S \equiv W ; \Diamond S \equiv \Diamond W$

• Solving a problem for the four failure detectors with strong completeness, automatically solves for the remaining four failure detectors

• Since $P$ is reducible to $S$ and $\Diamond P$ is reducible to $\Diamond S$.

• The algorithm for solving consensus using $S$ also solve consensus using $P$.

• The algorithm for solving consensus using $\Diamond S$ also solve consensus using $\Diamond P$. 
Consensus using $S$

Every process $p$ executes the following:

**procedure** propose($v_p$)

$V_p \leftarrow \langle \bot, \bot, \ldots, \bot \rangle$

$p$'s estimate of the proposed values

$V_p[p] \leftarrow v_p$

$\Delta_p \leftarrow V_p$

**Phase 1:** {asynchronous rounds $r_p$, $1 \leq r_p \leq n - 1$}

for $r_p \leftarrow 1$ to $n - 1$

send $(r_p, \Delta_p, p)$ to all

**wait until** $[\forall q : \text{received } (r_p, \Delta_q, q) \text{ or } q \in D_p]$

query the failure detector

$\text{msgs}_p[r_p] \leftarrow \{(r_p, \Delta_q, q) \mid \text{received } (r_p, \Delta_q, q)\}$

$\Delta_p \leftarrow \langle \bot, \bot, \ldots, \bot \rangle$
for $k \leftarrow 1$ to $n$

if $V_p[k] = \bot$ and $\exists (r_p, \Delta_q, q) \in \text{msgs}_p[r_p]$ with $\Delta_q[k] \neq \bot$ then

$V_p[k] \leftarrow \Delta_q[k]$  
$\Delta_p[k] \leftarrow \Delta_q[k]$

Phase 2: send $V_p$ to all

wait until $[\forall q : \text{received } V_q \text{ or } q \in \mathcal{D}_p]$ \{query the failure detector\}

$\text{lastmsgs}_p \leftarrow \{V_q \mid \text{received } V_q\}$

for $k \leftarrow 1$ to $n$

if $\exists V_q \in \text{lastmsgs}_p$ with $V_q[k] = \bot$ then $V_p[k] \leftarrow \bot$

Phase 3: $\text{decide}(\text{first non-}\bot\text{ component of } V_p)$
Solving Consensus using $s$:

Rotating Coordinator Algorithms

Work for up to $f < n/2$ crashes

• Processes are numbered 1, 2, ..., n
• They execute asynchronous *rounds*

• In round $r$, the *coordinator* is
  process $(r \mod n) + 1$

• In round $r$, the coordinator:
  - tries to impose its estimate as the consensus value
  - succeeds if it does not crash and it is not suspected by $S$
Consensus using $\diamond S$

• The algorithm goes through
  – three Asynchronous stages
    • Each stage has several asynchronous rounds
      – Each round has 2 tasks
        » Task 1
          • Four asynchronous phases
        » Task 2

• In the first stage, several decision values are proposed

• In second stage, a value gets locked: no other decision value is possible

• In the third and final stage, the processes decide on the locked value and consensus is reached.
Consensus using $\diamondsuit$ $S$

- **Task 1**
  - **Phase 1**
    - Every process ‘p’ sends
      - Current estimate to coordinator $C_p$
      - Round number $t_{s_p}$
  - **Phase 2**
    - $C_p$ gathers $\lceil(n+1)/2\rceil$ estimates
    - Selects one with largest time stamp estimate $e_{\text{ts,estimate}_p}$
    - Send the new estimate to all processes
  - **Phase 3**
    - Each process ‘p’
      - May receive estimate $e_{\text{estimate}_p}$
        » Send an ack to $C_p$
      - May not receive estimate $e_{\text{estimate}_p}$
        » Send an nack to $C_p$ (suspecting $C_p$ has crashed)
  - **Phase 4**
    - Waits for $\lceil(n+1)/2\rceil$ (acks or nacks)
      - If all are acks then estimate $e_{\text{estimate}_p}$ is locked
      - $C_p$ broadcasts the decided value $e_{\text{estimate}_p}$

- **Task 2**
  - If a process ‘p’ receives a broadcast on decided value and has not already decided
    - Accepts the value
Consensus using $\diamondsuit S$

Let $ts2 < ts1 < ts3$
Consensus using $\diamondsuit S$

![Diagram showing nodes and edges with Est_p = 3 connections between nodes 1, 2, and 3]
Consensus using $\diamond S$
Consensus using $\diamond S$

Locks 3 and broad casts
Consensus using $\diamond S$

Locks 3 and broad casts
Consensus using $\diamondsuit S$

Every process $p$ executes the following:

\textbf{procedure} propose($v_p$)
\begin{align*}
estimate_p &\leftarrow v_p & \{\text{estimate}_p \text{ is } p\text{'s estimate of the decision value}\} \\
\text{state}_p &\leftarrow \text{undecided} \\
r_p &\leftarrow 0 & \{r_p \text{ is } p\text{'s current round number}\} \\
t_{sp} &\leftarrow 0 & \{t_{sp} \text{ is the last round in which } p \text{ updated } \text{estimate}_p, \text{ initially } 0\}
\end{align*}

{\text{Rotate through coordinators until decision is reached}}

\textbf{while} state$_p = \text{undecided}$
\begin{align*}
r_p &\leftarrow r_p + 1 \\
c_p &\leftarrow (r_p \mod n) + 1 & \{c_p \text{ is the current coordinator}\}
\end{align*}

\textbf{Phase 1:} \{\text{All processes } p \text{ send } \text{estimate}_p \text{ to the current coordinator}\}

send $(p, r_p, \text{estimate}_p, t_{sp})$ to $c_p$
Consensus using $\Diamond S$ cont...

Phase 2: \{The current coordinator gathers $\lceil \frac{(n+1)}{2} \rceil$ estimates and proposes a new estimate\}

if $p = c_p$ then

wait until \{for $\lceil \frac{(n+1)}{2} \rceil$ processes $q :$ received $(q, r_p, estimate_q, ts_q)$ from $q$\}

$msgs_p[r_p] \leftarrow \{(q, r_p, estimate_q, ts_q) \mid p$ received $(q, r_p, estimate_q, ts_q)$ from $q\}$

t $\leftarrow$ largest $ts_q$ such that $(q, r_p, estimate_q, ts_q) \in msgs_p[r_p]$

$estimate_p \leftarrow$ select one $estimate_q$ such that $(q, r_p, estimate_q, t) \in msgs_p[r_p]$

send $(p, r_p, estimate_p)$ to all

Phase 3: \{All processes wait for the new estimate proposed by the current coordinator\}

wait until \{received $(c_p, r_p, estimate_{c_p})$ from $c_p$ or $c_p \in D_p$\}\{Query the failure detector\}

if \{received $(c_p, r_p, estimate_{c_p})$ from $c_p$\} then $p$ \{received $estimate_{c_p}$ from $c_p$\}

$estimate_p \leftarrow estimate_{c_p}$

ts_p $\leftarrow r_p$

send $(p, r_p, ack)$ to $c_p$

else send $(p, r_p, nack)$ to $c_p$ $\{p$ suspects that $c_p$ crashed$\}$
Consensus using $\diamond S$ cont...

Phase 4: \[
\left\{ \begin{array}{l}
\text{The current coordinator waits for } \left\lceil \frac{n+1}{2} \right\rceil \text{ replies. If they indicate that } \left\lceil \frac{n+1}{2} \right\rceil \\
\text{processes adopted its estimate, the coordinator R-broadcasts a decide message}
\end{array} \right. \\
\text{if } p = c_p \text{ then}
\begin{align*}
&\text{wait until } \left\lceil \frac{n+1}{2} \right\rceil \text{ processes } q : \text{received } (q, r_p, \text{ack}) \text{ or } (q, r_p, \text{nack}) \\
&\text{if } \left\lceil \frac{n+1}{2} \right\rceil \text{ processes } q : \text{received } (q, r_p, \text{ack}) \text{ then}
\end{align*}
\begin{align*}
&R\text{-broadcast}(p, r_p, \text{estimate}_p, \text{decide})
\end{align*}
\]

\{If p R-delivers a decide message, p decides accordingly\}

\text{when } R\text{-deliver}(q, r_q, \text{estimate}_q, \text{decide})
\begin{align*}
&\text{if } \text{state}_p = \text{undecided} \text{ then}
\end{align*}
\begin{align*}
&\text{decide(estimate}_q) \\
&\text{state}_p \leftarrow \text{decided}
\end{align*}
Atomic Broadcast

• Informally, atomic broadcast requires that all correct processes deliver the same set of messages in the same order (i.e., deliver the same sequence of messages).

• Formally atomic broadcast can be defined as a reliable broadcast with the total order property.

• Chandra and Toueg showed that the result of consensus can be used to solve the problem of atomic broadcast.
• Reliable Broadcast
  – Validity: If the sender of a broadcast message $m$ is non-faulty, then all correct processes eventually deliver $m$.
  – Agreement: If a correct process delivers a message $m$, then all correct processes deliver $m$.
  – Integrity: Each correct process delivers a message at most once.

• Total Order
  – If two correct processes $p$ and $q$ deliver two messages $m$ and $m'$, then $p$ delivers $m$ before $m'$ if and only if $q$ delivers $m$ before $m'$. 
Reliable Broadcast

Every process $p$ executes the following:

To execute $R$-broadcast$(m)$:
    send $m$ to all (including $p$)

$R$-deliver$(m)$ occurs as follows:
    when receive $m$ for the first time
        if $sender(m) \neq p$ then send $m$ to all
            $R$-deliver$(m)$
Atomic Broadcast

• The algorithm consists of three tasks:

• **Task 1:**
  – when a process $p$ wants to A-broadcast a message $m$, it \( R_{broadcasts} m \).

• **Task 2:**
  – a message $m$ is added to set \( R_{delivered} \) when process $p$ \( R_{delivers} \) it.

• **Task 3:**
  – when a process $p$ \( A_{delivers} a \ message \ m, \ it \ adds \ m \ to \ set \ A_{delivered} \).
  – Process $p$ periodically checks whether \( A_{undelivered} \) contains messages. If it contains messages, $p$ enters its next execution of consensus, say the kth one, and proposes \( A_{undelivered} \) as the next batch of messages to be \( A_{delivered} \).
Atomic Broadcast

Every process $p$ executes the following:

Initialisation:

\[
R_{\text{delivered}} \leftarrow \emptyset \\
A_{\text{delivered}} \leftarrow \emptyset \\
k \leftarrow 0
\]

To execute $A$-broadcast($m$):

\[
\{ \text{ Task 1 } \}
\]

$R$-broadcast($m$)

$A$-deliver($-$) occurs as follows:

\[
\text{ when } R\text{-deliver}(m)
\quad R_{\text{delivered}} \leftarrow R_{\text{delivered}} \cup \{m\}
\]

\[
\text{ when } R_{\text{delivered}} - A_{\text{delivered}} \neq \emptyset
\quad k \leftarrow k + 1 \\
A_{\text{undelivered}} \leftarrow R_{\text{delivered}} - A_{\text{delivered}} \\
\text{ propose}(k, A_{\text{undelivered}}) \\
\text{ wait until decide}(k, \text{msgSet}^k) \\
A_{\text{deliver}}^k \leftarrow \text{msgSet}^k - A_{\text{delivered}}
\]

atomically deliver all messages in $A_{\text{deliver}}^k$ in some deterministic order

\[
A_{\text{delivered}} \leftarrow A_{\text{delivered}} \cup A_{\text{deliver}}^k
\]

\[
\{ \text{ Task 2 } \}
\]

\[
\{ \text{ Task 3 } \}
\]
Implementation of failure detector

• **Task 1**: Each process $p$ periodically sends a “$p$-is-alive” message to all other processes. This is like a heart-beat message that informs other processes that process $p$ is alive.

• **Task 2**: If a process $p$ does not receive a “$q$-is-alive” message from a process $q$ within $p(q)$ time units on its clock, then $p$ adds $q$ to its set of suspects if $q$ is not already in the suspect list of $p$.

• **Task 3**: When a process delivers a message from a suspected process, it corrects its error about the suspected process and increases its timeout for that process.
  – If process $p$ receives “$q$-is-alive” message from a process $q$ that it currently suspects, $p$ knows that its previous timeout on $q$ was premature – $p$ removes $q$ from its set of suspects and increases its timeout period for process $q$, $p(q)$. 
Implementation of failure detector

Every process \( p \) executes the following:

\[
\text{output}_p \leftarrow \emptyset \\
\text{for all } q \in \Pi \\
\quad \Delta_p(q) \leftarrow \text{default time-out interval}
\]

\[
\text{cobegin} \\
\quad || \text{ Task 1: repeat periodically} \\
\quad \quad \text{send “p-is-alive” to all}
\]

\[
\quad || \text{ Task 2: repeat periodically} \\
\quad \quad \text{for all } q \in \Pi \\
\quad \quad \quad \text{if } q \notin \text{output}_p \text{ and} \\
\quad \quad \quad \quad \text{p did not receive “q-is-alive” during the last } \Delta_p(q) \text{ ticks of } p\text{’s clock} \\
\quad \quad \quad \quad \quad \text{output}_p \leftarrow \text{output}_p \cup \{q\} \\
\quad \quad \quad \quad \quad \text{p times-out on } q: \text{ it now suspects } q \text{ has crashed}
\]

\[
\quad || \text{ Task 3: when receive “q-is-alive” for some } q \\
\quad \quad \text{if } q \in \text{output}_p \\
\quad \quad \quad \text{output}_p \leftarrow \text{output}_p \setminus \{q\} \\
\quad \quad \quad \Delta_p(q) \leftarrow \Delta_p(q) + 1 \\
\quad \quad \quad \{p \text{ knows that it prematurely timed-out on } q\} \\
\quad \quad \quad \{1. \ p \text{ repents on } q, \text{ and}\} \\
\quad \quad \quad \{2. \ p \text{ increases its time-out period for } q\}
\]

\text{coend}

Fig. 10. A time-out based implementation of \( D \in \Diamond \mathcal{P} \) in models of partial synchrony.
Lazy failure detection protocol

• A relatively simple protocol that allows a process to “monitor” another process, and consequently to detect its crash.
• This protocol enjoys the nice property to rely as much as possible on application messages to do this monitoring.
• The cost associated with the implementation of a failure detector incurs only when the failure detector is used (hence, it is called a lazy failure detector).
• Each process pi has a local hardware clock hc_i that strictly monotonically increases.
• The local clocks are not required to be synchronized.
• Every pair of processes is connected by a channel and they communicate by sending and receiving messages through channels.
• Channels are not required to be FIFO
Lazy failure detection protocol

(1) when SEND $M$ to $p_j$ is invoked:
   (2) $m.content \leftarrow M$; $m.st \leftarrow hc_i$;
   (3) $pending\_msg\_st_i[j] \leftarrow pending\_msg\_st_i[j] \cup \{m.st\}$
   (4) send appl($m$) to $p_i$

(5) when type($m$) is received from $p_j$:
   (6) case type=appl then transmit $M = m.content$ to the upper layer; % RECEIVE M %
       send ack($m$) to $p_j$ % $m.st$ keeps its value %
   (7) case type=ack then $rt \leftarrow hc_i$;
       $max\_rtd_i[j] \leftarrow \max(\max\_rtd_i[j], rt - m.st)$;
   (8) case type=ping then send ack($m$) to $p_j$ % $m.st$ keeps its value %
       $pending\_msg\_st_i[j] \leftarrow pending\_msg\_st_i[j] - \{m.st\}$
       endif

(13) when QUERY($j$) is invoked:
(14) if $pending\_msg\_st_i[j] = \emptyset$ then create a control message $m$;
   (15) $m.content \leftarrow \text{null}$; $m.st \leftarrow hc_i$;
   (16) send ping($m$) to $p_j$;
   (17) $pending\_msg\_st_i[j] \leftarrow \{m.st\}$;
   (18) return (no_suspect)
   (19) else $rt \leftarrow hc_i$;
   (20) if $rt - \min(pending\_msg\_st_i[j]) > max\_rtd_i[j]$
       then return (suspect)
   (21) else return (no_suspect)
   (22) endif
A short introduction to failure detectors for asynchronous Distributed Systems
Failure Detectors-Definition

Why use FD?

- Based on well defined set of Abstract concepts
- Not dependant on any particular implementation
- Layered approach favors design, proof and portability of protocol
- Helps to solve impossible time-free asynchronous distributed system problems like the Consensus problem.
- Eventually accurate failure detectors helps in designing indulgent algorithms.
Asynchronous System Models

Process model

• A process can fail by premature halting (crashing).
• A process is correct if it does not crash else it is faulty

Computation models

• **FLP**  Crash-prone processes and reliable links
• **FLL**  Crash-prone processes and fair lossy links
Asynchronous System Models

Communication model

Processes communicate and synchronize by exchanging messages through links.

Reliable

• Does not create or duplicate messages
• Every message sent by Pi to Pj is eventually received by Pj

Fair lossy

• Does not create or duplicate messages
• Can lose message
• Can send infinite number of messages from one process to another
Consensus
Consensus

• All the processes, propose a initial value and they all have to agree upon some common value proposed

• Solving consensus is key to solving many problems in distributed computing (e.g., total order broadcast, atomic commit, terminating reliable broadcast)
Consensus definition

**C-Validity**: Any value decided is a value proposed

**C-Agreement**: No two correct processes decide differently

**C-Termination**: Every correct process eventually decides

**C-Integrity**: No process decides twice

**C-Uniform Agreement**: No two (correct or not) processes decide differently
Consensus

p1
- propose(0)
- propose(1)
- decide(1)
- decide(0)

p2
- propose(0)
- decide(1)
- crash

p3
- propose(0)
- decide(0)
Eventually accurate failure detectors

• **Strong Completeness**
  Eventually, all processes that crash are suspected by every correct process

• **Eventually Weak Accuracy**
  There is a time after which some correct process is never suspected by the correct processes
### S-based Consensus Protocol

- **FLP model**
- **Indulgent**
  - Never violates consensus safety
  - Terminates when the sets contain correct values during a long enough period
- Requires majority of correct processes \((t < n/2)\)
- Proceeds in asynchronous consecutive rounds
- Each round \(r\) is coordinated by process \(p_c\) such that, \(c = (r \mod n) + 1\)
Initialization

- \( v_i = \text{value initially proposed by } p_i \).
- \( est_i = p_i\text{'s estimate of the decision value.} \)
- In round \( r \), its coordinator \( p_c \) tries to impose its current estimate as the decision value.
- Algorithm runs in two phases.
Phase 1

- $p_c$ sends $est_c$ to all the processes
- process $p_i$ waits until it receives $p_c$’s estimate or suspects it.
- Based on result of waiting, either
  \[ aux_i = v(=est_c) \]
  or
  \[ aux_i = \bot \]
- Due to the completeness property of the underlying failure detector no process can block forever
Phase 2

- All process exchange the values of their \( aux_i \) variables
- Due to the "majority of correct processes" assumption, no process can block forever
- Only two values can be exchanged: \( v = est_c \) or \( \perp \).
- Therefore,

\[
rec_i = \{\{v\}, \{v, \perp\}, \text{ or } \{\perp\}\}
\]

- Impossible for two sets \( rec_i \) and \( rec_j \) to be such that

\[
rec_i = \{v\}
\]
\[
rec_j = \{\perp\}\]
Phase 2

\[
\text{rec}_i = \{v\} \Rightarrow (\forall p_j : (\text{rec}_j = \{v\}) \lor (\text{rec}_j = \{v, \bot\}))
\]

\[
\text{rec}_i = \{\bot\} \Rightarrow (\forall p_j : (\text{rec}_j = \{\bot\}) \lor (\text{rec}_j = \{v, \bot\})).
\]

\[
\text{rec}_i = \{v\}
\]

\[
est_i = v.
\]

To prevent possible deadlock situations, \(p_i\) broadcasts its decision value.

\[
\text{rec}_i = \{v, \bot\}
\]

\[
est_i = v.
\]

proceeds to the next round.

\[
\text{rec}_i = \{\bot\}
\]

\[
p_i\) proceeds to the next round without modifying \(\text{est}_i\).
A Simple $S$-Based Consensus Protocol $(t < n/2)$

Function Consensus($v_i$)

Task $T1$:

1. $r_i \leftarrow 0$; $est_i \leftarrow v_i$
2. while true do
3. $c \leftarrow (r_i \mod n) + 1$; $r_i \leftarrow r_i + 1$; $% 1 \leq r_i < +\infty %$

Phase 1 of round $r$: from pc to all

4. if ($i = c$) then broadcast phase1($r_i$, $est_i$) endif;
5. wait until (phase1($r_i$, $v$) has been received from $p_c$ $\forall c \in \text{suspected}_i$);
6. if (phase1($r_i$, $v$) received from $p_c$) then $aux_i \leftarrow v$ else $aux_i \leftarrow \bot$ endif;

Phase 2 of round $r$: from all to all

7. broadcast phase2($r_i$, $aux_i$);
8. wait until (phase2 ($r_i$, $aux$) msgs have been received from a majority of proc.);
9. let $rec_i$ be the set of values received by $p_i$ at line 8;
% We have $rec_i = \{v\}$, or $rec_i = \{v, \bot\}$, or $rec_i = \{\bot\}$ where $v = \text{est}_c$ %
10. case $rec_i = \{v\}$ then $est_i \leftarrow v$; broadcast decision($est_i$); stop $T1$
11. $rec_i = \{v, \bot\}$ then $est_i \leftarrow v$
12. $rec_i = \{\bot\}$ then skip
13. endcase
14. endwhile

Task $T2$: when decision($est$) is received: broadcast decision($est_i$); return($est$)
Findings

• The strong completeness property is used to show that the protocol never blocks.
• The eventual weak accuracy property is used to ensure termination.
• The majority of correct processes is used to prove consensus agreement.
Interactive consistency

• Harder than consensus problem
• Process has to agree on a vector of values!

Termination

Every correct process eventually decides on a vector

Validity

Any decided vector $D[i] \in \{v_i, \perp\}$, and is $v_i$ if $p_i$ does not crash

Agreement:

No two processes decide differently
Perfect failure detectors

• Requires perfect failure detectors

**Strong Completeness**
• Every process that crashes is eventually permanently suspected

**Strong Accuracy**
• No process is suspected before it crashes
Perfect failure detector

init: \( suspected_i \leftarrow \emptyset \); \( seq_i \leftarrow 0 \)

**task T1:** while true do

\[ seq_i \leftarrow seq_i + 1; \] % IC instance number %

\[ D_i \leftarrow IC \text{ Protocol}(seq_i, v_i); \] % \( v_i = \bot \) %

\[ suspected_i \leftarrow \{ j \mid D_i[j] = \bot \} \]

enddo

**task T2:** when \( p_i \) issues QUERY:

return(\( suspected_i \))
Non-Blocking Atomic Commit Problem (NBAC)

- Yet another agreement problem in the world of distributed computing
- Each process cast their votes (yes or no).
- Non-crashed process decide on single value (commit or abort)
Properties

The problem is defined by following properties:

- **NBAC - Termination**: Every correct process eventually decides.

- **NBAC - Validity**: A decided value is either commit or abort. Moreover:
  - If process decides commit, all process have voted yes.
  - If all process vote yes and there is no crash, then the decision value is commit.

- **NBAC - Obligation**: No two process decide differently.
Continued

- Justification property relates commit decision to yes.

- Obligation property eliminates trivial solution of all process opting abort.
  - “good” run – all process wants to commit and the environment is free of crashes.

- Process crashes are explicit in NBAC compared consensus.
Appropriate Failure Detector

Why appropriate failure detector?

- To solve NBAC in the FLP model

*Timeless failure detectors* – No information (sense of time) when failure occurred.

**Anonymously Perfect Failure Detectors**

P and $\diamond S$ - timeless failure detectors.

To address this problem, class ?P anonymous perfect failure detector introduced.

- **Anonymous completeness**: If a crash occurs, eventually every correct process is permanently informed that some crash occurred.
- **Anonymous accuracy**: No crash is detected unless some process crashed.

Class ?P + $\diamond S$ - weakest class to solve NBAC, assuming a majority of correct process.

The following protocol converts NBAC to consensus and subsequently uses subroutine consensus protocol.
Simple $\mathcal{P} + \diamondsuit$-Based NBAC protocol ($t < n/2$)

**Function** \textbf{Nbac}($vote_i$)

- broadcast MY\_VOTE($vote_i$);
- \textbf{wait until} (MY\_VOTE($vote_i$) has been received from each process $\lor ap\_flag_i$);
- \textbf{if} (a vote yes has been received from each of the $n$ processes)
  - \textbf{then} output$_i$ $\leftarrow$ Consensus(commit)
  - \textbf{else} output$_i$ $\leftarrow$ Consensus(abort)
- \textbf{endif};

return(output$_i$)
Quiescence Problem

• Consider processes $p_i$ and $p_j$ that do not crash connected by fair lossy link, a basic communication problem is to build a reliable link on top of fair lossy link.

• Protocol used (including TCP) are quiescent—no message transfer after some time. (communication ceases)

• What if process $p_j$ crashes?
• How to solve quiescent communication problem?
  – Heartbeat failure detectors
Heartbeat Failure Detector

• Failure detector outputs an array $HB_i[1..n]$ – non decreasing counter at each process which satisfies......
  – **HB-completeness:** If $p_j$ crashes, then $HB_i[j]$ stops increasing.
  – **HB-accuracy:** If $p_j$ is correct, then $HB_i[j]$ never stops increasing.

• Easy implementation but it is not quiescent.
• Allows the non-quiescent part of communication protocol to be isolated.
• Favors design modularity and eases correctness proof.

• “service” can be extended to upper layer applications.
Quiescent Implementation

Sender $p_i$:
when SEND($m$) TO $p_j$ is invoked:

\[ \text{seq}_i \leftarrow \text{seq}_i + 1; \]
\[ \text{fork task } \text{repeat_send}(m, \text{seq}_i) \]

\[
\text{task repeat_send}(m, \text{seq}_i)
\]
\[ \text{prev}_hb \leftarrow 1; \]
\[ \text{repeat periodically } \text{hb} \leftarrow \text{HB}_i[j]; \]
\[ \text{if } (\text{prev}_hb < \text{hb}) \text{ then send } \text{msg}(m, \text{seq}) \text{ to } p_j; \]
\[ \text{prev}_hb \leftarrow \text{hb} \]
\[ \text{endif} \]
\[ \text{until } (\text{ack}(m, \text{seq}) \text{ is received}) \]

Receiver $p_j$:
when $\text{msg}(m, \text{seq})$ is received from $p_i$:
\[ \text{if (first reception of } \text{msg}(m, \text{seq})) \text{ then } m \text{ is RECEIVED endif; } \]
\[ \text{send } \text{ack}(m, \text{seq}) \text{ to } p_i \]
Failure Detectors in Synchronous Systems

Synchronous System Model

• Synchronous systems – characterized by time bound to receive & send message.
• Local computations take no time & transfer delays bounded by D.
  – Message sent at time ‘t’ is not received after t+D (D-timeliness)
  – Links are reliable (no duplication, losses)
  – Process have access to common clock.

Consider $p_i$ sends message to $p_j$ & $p_k$, D-timeliness and no-loss properties gives rise to following scenarios...

  – $P_i$ crashes at time t, no message sent
  – $P_i$ crashes at time t, $p_j$ receives while $p_k$ doesn’t by t + D, vice versa.
  – $P_i$ doesn’t crash, $p_j$ & $p_k$ receives message by t + D
Fast Failure Detectors

• Fast failure detector provides processes with following properties ($d < D$)
  – $d$ – **Timely completeness**: If a process $p_j$ crashes at time $t$, then, by time $t + d$, every alive process suspects it permanently.
  – **Strong accuracy**: No process is suspected before it crashes.

• Implemented with specialized hardware, also attains time complexity lower bounds $\ll$ pure synchronous system.

• Protocol described in the following slide illustrates early deciding property, reducing time complexity to $D + fd$ ($f$ – actual number of process crashes)

• Snapshot of the Synchronous Consensus with Fast Failure Detector implementation is illustrated as follows...
Fast Failure Detector Implementation

\textbf{init} \ est_i \leftarrow v_i; \ max_i \leftarrow 0

\textbf{when} (est,j) \textit{is received:}
\hspace{1em} \textbf{if} \ (j > max_i) \textbf{then} \ est_i \leftarrow est; \ max_i \leftarrow j \textbf{endif}

\textbf{at time} \ (i-1)d \textit{do}
\hspace{1em} \textbf{if} \ \{p_1, p_2, \ldots, p_{i-1}\} \subseteq \text{suspected}_i \textbf{then broadcast} \ (est_i, i) \textbf{endif}

\textbf{at time} \ (j-1)d + D \textit{for every} \ 1 \leq j \leq n \textit{do}
\hspace{1em} \textbf{if} \ ((p_j \notin \text{suspected}_i) \land (p_i \text{ has not yet decided})) \textbf{then return} \ (est_i) \textbf{endif}
Thank You