Total failure

If all the processes crash, then the processes that were the last to fail need to recover and run the termination protocol.

A last process to fail is one whose failure is not detected by another process.

Total failure (continued)

\[ p_i \text{ fails-before } p_j \equiv p_j \text{ detects } p_i \text{'s failure} \]

- \[ LPF \equiv \{ p_i : \forall p_j : \neg (p_i \text{ fails-before } p_j) \} \]
- rewording, \[ p_i \in LPF \equiv \forall p_j : \neg (p_i \text{ fails-before } p_j) \]

\[ UP_i \text{ is } \{ p_j : \neg (p_j \text{ fails-before } p_i) \} \]

- so, \[ p_i \in LPF \equiv \forall p_j : p_i \in UP_j \] where \( UP_j \) is the value when \( p_j \) crashed.

\[ \text{so, } LPF = \bigcap_{p_i} UP_i \]
Total failure (continued)
Total failure (continued)
Total failure (continued)
Total failure (continued)

\[ LPF = \{ p_1, p_2 \} \]
Total failure (continued)

When $p_i$ detects $p_j$'s failure it:

- Removes $p_j$ from $UP_i$
- Synchronously writes $UP_i$ to stable storage

When $p_i$ recovers:

- Let $R_i$ be the processes $p_i$ knows have recovered.
- When $R_i = \bigcap_{p_j \in R_i} UP_j$

then $p_i$ knows that all of LPF have recovered.
Total failure (continued)
Total failure (continued)
Total failure (continued)

1. $p_1, p_2, p_3$
2. $p_1, p_2, p_3$
3. $p_1, p_2, p_3, p_4$
4. $p_1, p_2, p_3$
5. $p_1, p_2, p_4$
6. $p_2$
7. $p_4$
Total failure (continued)

$p_1$, $p_2$, $p_3$

$p_1$, $p_2$, $p_3$, $p_4$

$p_1$, $p_2$, $p_3$

$p_1$, $p_2$, $p_4$

$p_1$, $p_2$, $p_3$

$p_1$, $p_2$, $p_4$
A weakness in this protocol

- There are cases in which the processes in $LPF$ have recovered but they don’t know it.

  … they collectively think some process $p$ is up when in fact another process detected $p$'s failure.
Total failure (continued)

$p_1$ and $p_2$ didn’t detect all of the failures that were detected. … in particular, $p_1$ and $p_2$ did not detect the failure of $p_3$. 
Total failure (continued)

Consider *fails before* graph.

- Graph is a DAG.
- Sinks are elements of \( LPF \).
- All but upstream neighbors of \( p_i \) are \( UP_i \).
- Let \(|LPF| = u\).
  - If *fails before* is transitive, then graph consists of \( u \) cliques, each one having one sink.
  - Hence, if *fails before* is transitive, then \( LPF = \bigcap_{p_i \in P} UP_i \).
Total failure (continued)

Consider *fails before* graph.

- Graph is a DAG.
- Sinks are elements of $LPF$.
- All but upstream neighbors of $p_i$ are $UP_i$.
- Let $|LPF| = u$.
  - If *fails before* is transitive, then graph consists of $u$ cliques, each one having one sink.
  - Hence, if *fails before* is transitive, then $LPF = \bigcap_{p_i \in P} UP_i$.  

Total failure (continued)

Consider *fails before* graph.

- Graph is a DAG.
- Sinks are elements of \( LPF \).
- All but upstream neighbors of \( p_i \) are \( UP_i \).
- Let \( |LPF| = u \).
  - If *fails before* is transitive, then graph consists of \( u \) cliques, each one having one sink.
  - Hence, if *fails before* is transitive, then \( LPF = \bigcap_{p_i \in P} UP_i \).
Making failure detection transitive I

\[(p_j \text{ detected } p_i \text{'s failure}) \land (p_k \text{ detected } p_j \text{'s failure}) \Rightarrow (p_k \text{ detected } p_i \text{'s failure})\]

This requires \(p_j\) to delay detecting \(p_i\)'s failure until it is sure that any processes that eventually detects \(p_j\)'s failure will first detect \(p_i\)'s failure.
Making failure detection transitive II

To do so, we build a failure detector on top of the existing failure detector.

- when $p_j$ detects failure of $p_i$ by existing failure detector, we say $p_j$ suspects $p_i$
- “detects” will then refer to the transitive failure detector we are implementing.

Transitive Perfect Failure Detector Simulation
Making failure detection transitive III

- $p_i$ maintains:
  - $S_i$: processes $p_i$ suspects but not yet detected their failure.
  - $D_i$: processes whose failures $p_i$ has already detected
    \[ \text{... and so } UP_i \text{ is a synonym for } P - D_i. \]
  - $NeedF_i(p)$: $p_j \in NeedF_i(p)$ means $p_i$ is waiting for $p_j$ to say it suspects $p$.

- Suspicion is accurate: if $p_j$ suspects $p_i$ then $p_i$ has crashed and so can never suspect $p_j$.
- Similarly, if $p_j$ suspects $p_i$ and then sends FAILED $p_i$ to $p_k$, then $p_i$ can never suspects $p_k$.
- In this sense, suspicion is transmitted in messages. Will use this idea in the proof but keep distinctions separate in protocol for clarity.

- Assume channels are FIFO.
- Correctness is with respect to what a process writes to stable storage.
Making failure detection transitive IV

when $p_i$ suspects $p_j$:

if ($p_j \notin S_i \land p_j \notin D_i$)

\[
\forall p \in S_i: \text{Need}F_i(p) = \text{Need}F_i(p) - \{p_j\}
\]

$S_i = S_i \cup \{p_j\}$

send FAILED $p_j$ to all in $P - S_i - D_i - \{p_i\}$

$\text{Need}F_i(p_j) = P - S_i - D_i - \{p_i\}$

when $\exists p$ in $S_i$: $\text{Need}F_i(p) = \emptyset$:

\[
\forall p \in S_i: \text{Need}F_i(p) = \emptyset:
\]

$S_i = S_i - \{p\}$

$D_i = D_i \cup \{p\}$

write $P - D_i$ to stable storage

when $p_i$ receives FAILED $p_j$ from $p_k$:

if ($p_j \notin S_i$)

\[
\forall p \in S_i: \text{Need}F_i(p) = \text{Need}F_i(p) - \{p_j\}
\]

$S_i = S_i \cup \{p_j\}$

send FAILED $p_j$ to all in $P - S_i - D_i - \{p_i\}$

$\text{Need}F_i(p_j) = P - S_i - D_i - \{p_i, p_k\}$

else $\text{Need}F_i(p_j) = \text{Need}F_i(p) - \{p_k\}$
Making failure detection transitive V

Assume that $p_j$ detects $p_i$'s failure and $p_k$ detects $p_j$'s failure without having detected $p_i$'s failure.

Since $p_k$ suspected $p_j$, $p_j$ never suspected $p_k$. Thus, before detecting $p_i$'s failure, $p_j$ received $\text{FAILED } p_i$ from $p_k$. And, when $p_k$ sent $\text{FAILED } p_i$ to $p_j$, $p_j$ was not in $S_k$ while $p_i$ was in $S_k$. That is, $p_k$ suspected $p_i$ before detecting $p_j$.

Since $p_k$ detects $p_j$'s failure without having detected $p_i$'s failure, at some point $p_k$ suspected both $p_i$ and $p_j$: $p_j$ and $p_i$ were both in $S_k$.

Furthermore, there was at least one process $p_x$ in $\text{NeedF}_k(p_i)$ and not in $\text{NeedF}_k(p_j)$, where $p_x \notin \{p_i, p_j, p_k\}$. The following argument applies to all such processes.

Given the two possibilities for $p_x$'s role in each of the two failure detections that did occur, one of four cases happened:

1) $p_j$ received $\text{FAILED } p_i$ from $p_x$ and $p_k$ received $\text{FAILED } p_j$ from $p_x$. $p_k$ suspected $p_j$ by the time it sent $\text{FAILED } p_j$ and so $p_x$ sent $\text{FAILED } p_i$ before $\text{FAILED } p_j$. Given channels are FIFO, $p_x$ was removed from $\text{NeedF}_k(p_j)$ before being removed from $\text{NeedF}_k(p_i)$. ☒

2) $p_j$ suspected $p_x$ and $p_k$ received $\text{FAILED } p_j$ from $p_x$. This requires $p_j$ to suspect $p_x$ and $p_x$ to suspect $p_j$. ☒

3) $p_j$ received $\text{FAILED } p_i$ from $p_x$ and $p_k$ suspected $p_x$. As well as removing $p_x$ from $\text{NeedF}_k(p_j)$, $p_k$'s suspicion also removes $p_x$ from $\text{NeedF}_k(p_i)$. From the protocol, both $p_i$ and $p_i$ would be moved to $D_k$ before $P - D_k$ is written to stable storage. ☒

4) $p_j$ suspected $p_x$ and $p_k$ suspected $p_x$. The argument for the previous case holds for this case as well. ☒
Weakness with this protocol…

☐ It delays detection.

- In some cases, using this protocol could make recovery slower.
- In other cases, using this protocol could make recovery faster.