Contents

1. Previous Lecture Time - Clocks & Clock Synchronisation

2. Previous Lecture Mutual Exclusion, Leader Election, Distributed Consensus

3. Previous Lecture Protocols to co-ordinate processes in distributed applications to accomplish goals
4. This Lecture Failure Detection
Synchronous and Asynchronous Systems

- With an asynch. system, we make no assumptions about process execution speeds, message delivery delays, or clock drift rates.

- With a synch. system, we make assumptions and have to bound (a) relative speeds of processes, (b) message transmission delays, (c) (local) clock drift rate with an external reference clock. Need known bounds.
Distributed Systems - Issues To Address

- Independent Failure
- Unreliable Communication
- Insecure Communication
- Costly Communication
Failure Models (a.k.a Failure Patterns)

The design of fault-tolerant systems will be simple (not obviated) if faulty processes can be reliably detected.

- **Omission** failures: process/channel fails to perform intended actions

- **Arbitrary (Byzantine)** failures: wrong process/channel behaviour

- **Timing** failures (in synchronous systems)
Defining Consensus

A dist. system contains $n$ processes $\{0, 1, 2, \ldots, n - 1\}$. Every process has an initial value from a mutually agreed domain of values. Devise an algorithm/protocol s.t. despite the occurrence of failures, processes eventually agree on an irrevocable final decision value that satisfies the ffg. 3 conditions:

- **Termination**: Every non-faulty process must eventually decide

- **Agreement**: The final decision of every non-faulty process must be identical
• **Validity/Integrity**: If every non-faulty process begins with the same initial value \( v \), then their final decision must be \( v \)
  - validity adds a dose of sanity check - it is silly to reach agreement when the agreed value reflects nobody’s initial choice
  - variations of integrity requirement depending on the application, e.g. agreed value was proposed by some process
FLP Impossibility & Failure Detection

- FLP Impossibility: No deterministic algorithm solves Consensus in an asynchronous system and does not tolerate even a single crash failure.

- Holds even if the network is completely connected & reliable

- In an asynch. system, a crashed process cannot be distinguished from an extremely slow process.

- This inability to accurately determine really faulty processes is at the core of FLP Impossibility
• This observation led to extending the asynch. model with **failure detectors**
Failure Detection - Some Motivations
[Coulouris]

Agreement in Pepperland: 2 divisions of the Pepperland army, ‘Apple’ and ‘Orange’, are encamped at the top of two nearby hills. Further along the valley below are the invading Blue Meanies. The Pepperland divisions are safe as long as they remain in their encampments, and they can send out messengers reliably through the valley to communicate. The Pepperland divisions need to agree on which of them will lead the charge against the Blue Meanies and when the charge will take place. Even in an asynchronous Pepperland, it is possible to agree on who will lead the charge. For example, each division can send the number of its remaining members,
and the one with most will lead (if a tie, division Apple wins over Orange). But when should they charge? Unfortunately, in **asynch.** Pepperland, the messengers are very variable in their speed. If, say, Apple sends a messenger with the message ‘Charge!’, Orange might not receive the message for, say, 3 hrs or … 5 mins. In a **synchronous** Pepperland, there is still a coordination problem, but the divisions know some useful constraints: every message takes at least $min$ minutes and at most $max$ minutes to arrive. If the division that will lead the charge sends a message ‘Charge!’; it waits for $min$ minutes; then it charges. The other division waits for 1 min after receipt of the message, then charges. Its charge is guaranteed to be after the leading division’s, but no more than $(max - min + 1)$ mins. after it.
Failure Detection - Some Motivations
The Byzantine generals problem
3 Scenarios highlighting Importance of Failure Detection

• In a sensor network, a base station delegates the task of monitoring an environment to a set of geographically dispersed sensor nodes. These sensors send the monitored values back to the base station. If a sensor node crashes, and the crash is reliably detected, then its task can be assigned to another sensor.

• In group-oriented activities, sometimes a task is divided among the members of a group. If a member crashes, or a member voluntarily leaves the group,
then the other members can take over the task of the failed member only if they can detect the failure.

- Distributed consensus has trivially simple solution if there is a reliable failure detection service. In the byzantine generals problem, if the traitors could be reliably identified, then consensus could be reached by ignoring the inputs from the traitors.
What Is A Failure Detector

- A service that generates a list of processes that are suspected to have failed. Each process has a local detector that coordinates with its counterparts in other processes to provide the service.

- For asynch. systems, detection mechanisms are unreliable and error prone. Processes are suspected based on local observations or indirect information, and different processes may have different list of suspects.

- One way to compile a suspect list is to send a probe to all other processes with a request to respond. If a
process does not respond within a specified period, then it is suspected to have crashed

- List of suspects is influenced by how long a process waits for the response, and the failure scenario. If the waiting time is too short, then every process might appear to be a suspect

- If a process \( i \) receives two probes from \( j \) and \( k \), responds to \( j \), and then crashes, \( j \) will treat \( i \) to be a correct process, but \( k \) will suspect \( i \) as a crashed process

- Even the detector itself might be a suspect (it is a software process, after all)
The Underlying System Model

Processes The system is made up of a set of $n$ sequential processes: $\Pi = \{p_1, \ldots, p_n\}$ ($\Pi$ is sometimes used to denote the set of identities $\{1, \ldots, n\}$). Each process, $p_i$, has an initial state and executes a sequence of steps defined by the process's transition function, called the local algorithm. A step is atomic and corresponds to execution of a statement. The set of these local algorithms is called a distributed algorithm.

Communication Medium Each pair of processes is connected by a bidirectional link, and a process can also send a message to itself.
The Underlying System Model - Contd.

**Failure Model** A process can crash. After crashing, it executes no more steps. There is no recovery. A process that crashes in a run is *faulty* in that run; otherwise, it is *non-faulty*.

**Timing Model** The processes are asynchronous, i.e. each process proceeds at its own speed. No assumption on relative speeds of processes. The only assumption is that, until it crashes, if it crashes, a process’ speed is always +ve, i.e. a non-faulty process eventually executes the next step of its algorithm.
Failures Pattern - Definition

**Failure Pattern** A *failure pattern* defines a possible set of failures that can occur during an execution. Formally, a failure pattern is a function: $F : \mathbb{N} \rightarrow 2^\Pi$, where $\mathbb{N}$ is the set of natural numbers from the time domain, $\Pi$ is the set of process (identifiers), and $2^\Pi$ is the power-set of $\Pi$. $F(t)$ is the set of processes that have crashed up to time $t$. Since, a crashed process does not recover, $F(t) \subseteq F(t + 1)$. Let $Faulty(F)$ be set of processes that crash in an execution with failure pattern $F$. Let $t_{max}$ denote the end of that execution. We have $Faulty(F) = F(t_{max})$. 

13
**Failure Detectors - Properties**

<table>
<thead>
<tr>
<th>Completeness</th>
<th>Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Strong Completeness</td>
<td>Strong Accuracy</td>
</tr>
<tr>
<td>Weak Completeness</td>
<td>Weak Accuracy</td>
</tr>
</tbody>
</table>

Completeness is a liveness property. Accuracy is a safety property.
Failure Detectors - Properties

- *Completeness*: Every crashed process is suspected

  1. *Strong Completeness*: Every crashed process is eventually suspected by every process and remains a suspect thereafter

  2. *Weak Completeness*: Every crashed process is eventually suspected by at least one correct process and remains a suspect thereafter

- *Accuracy*: No correct process is suspected

  1. *Strong Accuracy*: No correct process is ever suspected
2. *Weak Accuracy*: There is at least one correct process that is never suspected

Accuracy can be weakened further by making them ‘eventual’ (◊).

- *Eventually strongly accurate*: \( \exists t_x : T \) after which no correct process is suspected. Before \( t_x \), a correct process may be added to or removed from the suspect list any number of times.

- *Eventually weakly accurate*: \( \exists t_x : T \) after which at least one correct process is no longer suspected. Before \( t_x \), every correct process could be a suspect
Classes of Failure Detectors

By combining the completeness and accuracy types, we can define the five classes of failure detectors:

- **Perfect P**: (Strongly) Complete & strongly accurate
- **Strong S**: (Strongly) Complete & weakly accurate
- **Eventually perfect ♦ P**: (Strongly) Complete & eventually strongly accurate
- **Eventually strong ♦ S**: (Strongly) Complete & eventually weakly accurate
• Eventually weak failure detector $\diamond W$: Weakly complete & weakly accurate
Some Implementations of Failure Detectors

The class $\Omega$ of eventual leader failure detectors. Provides each process $p_i$ with a local variable $leader_i$ s.t. $leader_i \forall 1 \leq i \leq n$ collectively satisfy the ffg. properties:

- Let $leader_i^t$ be the value of $leader_i$ at time $t$. If $F$ denotes a failure pattern, $F(t)$ is the set of processes failed at $t$, $Faulty(F)$ the set of processes that crash with failure pattern $F$, and $Correct(F)$ the set of processes that are non-faulty in the failure pattern $F$

- Validity: $\forall i \land \forall t. leader_i^t$ contains a process identity
• Eventual Leadership: \( \exists l \in Correct(F) \land \exists t \text{ s.t. } \forall t' \geq t : \forall i \in Correct(F) . \text{leader}^t_i = l \)

The properties above mean that a unique leader is eventually elected; this leader is not a faulty process, but there is no knowledge of when this leader is elected. Several leaders can co-exist during an arbitrary long (but finite) period of time, and there’s no way for a process to know when this turbulent period will be over. During this turbulence, it is possible that crashed processes are considered as leaders by non faulty processes and different processes may have different leaders.
Some Implementations of Failure Detectors - Contd.

Consensus using Perfect Failure Detector $P$

Underlying mechanism behind reaching consensus for Perfect failure detector is *reliable multicast*. A perfect failure detector lets each correct process receive inputs from every other correct process. For every process $p$, let $V_p$ be a vector of size $n$, the $i^{th}$ component of which represents the input from process $i$. If each process multicasts its input value to others, then everyone will receive an identical set of values, apply a choice function, and reach a consensus.
Algorithm runs in 2 phases, and computation progresses in rounds. The rounds are asynchronous, with each process $p$ keeping track of its own round number $r_p$.

In Phase 1, for each round, every process $p$ exchanges its $V_p$ with other processes. If $p$ receives no message from $q \neq p$, and no other process reports to $p$ about receiving a message from $q$, then $p$ sets $V_p[q] = \bot$. The algorithm guarantees that at the end of phase 1 $\forall (p \land q). V_p = V_q$.

Phase 2 generates the final decision value for each process.
Consensus using Perfect Failure Detector $P$

Algorithm for Consensus with $P$ program for process $p$

**define** $V_p, D_p$: array $[0 \cdots n - 1]$ of input values

**initially** $V_p = (\bot, \bot, \cdots, \bot)$

$V_p[p] = p_{inputs}$; $D_p = V_p$; $r_p = 1$

(Phase 1)

**do** $r_p < t + 1$
send \( (r_p, D_p, p) \) to each \( q \neq p \)

wait for \( (r_p, D_p, q) \) from \( \forall q \neq p \) [or \( q \) becomes a suspect]

\textbf{for} \( k = 0...n - 1 \)

\textbf{if}
\[ V_p[k] = \perp \land \exists (r_q, D_q, q) \text{ s.t. } D_q[k] \neq \perp \]
\[ V_p[k] = D_q[k]; D_p[k] = D_q[k] \]
\textbf{fi}

\textbf{end for}

\[ r_p = r_p + 1 \]
Phase 2 Final decision value is the *first* element of $V_p[]$

$V_p[] \neq \bot$
Failure Detectors - Summing Up

- failure detectors arose from extending asynch. models to cope with FLP Impossibility*

- useful for leader election and solving the consensus problem

- service that processes queries regarding process failure

usually implemented using local failure detectors, one for each process

• **unreliable failure detectors**
  
  – return hints: *Unsuspected / Suspected*
  
  – example protocol:
    
    * each process sends “alive” message to everyone else every $T$ seconds
    
    * local failure detector reports *Suspected* if “alive” message not received within $T + D$ seconds of the last one ($D$ estimates the maximum message transmission time)
• **reliable (perfect) failure detectors**
  
  – return *Unsuspected* (hint) / *Failure* (guarantee)
  
  – very useful but difficult to implement in practice