Failure Detection
Key Properties

- Multiple computers
  - Concurrent execution
  - Independent failures
  - Autonomous administrators
  - Heterogeneous capacities, properties
  - Large numbers (scalability)
- Networked communication
  - Asynchronous execution
  - Unreliable delivery
  - Insecure medium
- Common goal
  - Consistency – can discuss whole-system properties
  - Transparency – can use the system without knowing details
Objectives

- How do we detect failures?
- Models
  - Failures
  - Networks
- Properties
  - Guarantees
  - Metrics
- Techniques
What is a failure?

- Process omission failure
  - Crash-stop (fail-stop) – a process halts and does not execute any further operations
  - Crash-recovery – a process halts, but then recovers (reboots) after a while

We will focus on Crash-stop failures

- They are easy to detect in synchronous systems
- Not so easy in asynchronous systems
Synchronous Distributed System
- Each message is received (successfully) within bounded time
- Each step in a process takes $lb < \text{time} < ub$
- (Each local clock’s drift has a known bound)

Asynchronous Distributed System
- No bounds on message transmission delays
- No bounds on process execution
- (The drift of a clock is arbitrary)

Which is more realistic?
- Synchronous: Multiprocessor systems
- Asynchronous: Internet, wireless networks, datacenters, most real systems
What’s a failure detector?

\[ p_i \quad p_j \]
What’s a failure detector?

Crash-stop failure
(p_j is a failed process)
needs to know about \( p_j \)'s failure

(\( p_i \) is a non-faulty process or alive process)

Crash-stop failure

(\( p_j \) is a failed process)

There are two styles of failure detectors
- $p_i$ queries $p_j$ once every $T$ time units
- if $p_j$ does not respond within another $T$ time units of being sent the ping, $p_i$ detects $p_j$ as failed

If $p_j$ fails, then within $T$ time units, $p_i$ will send it a ping message. $p_i$ will time out within another $T$ time units.

Worst case Detection time = $2T$

The waiting time ‘$T$’ can be parameterized.
II. Heartbeating Protocol

- $p_j$ maintains a sequence number
- $p_j$ sends $p_i$ a heartbeat with incremented seq. number after every $T$ time units

-if $p_i$ has not received a new heartbeat for the past, say $3T$ time units, since it received the last heartbeat, then $p_i$ detects $p_j$ as failed

If $T \gg \text{round trip time of messages}$, then worst case detection time $\sim 3T$ (why?)

The ‘3’ can be changed to any positive number since it is a parameter
The Ping-ack and Heartbeat failure detectors are always correct
- Ping-ack: set waiting time $T$ to be $> \text{round-trip time upper bound}$
- Heartbeat: set waiting time $3T$ to be $> \text{round-trip time upper bound}$

The following property is guaranteed:
- If a process $p_j$ fails, then $p_i$ will detect its failure as long as $p_i$ itself is alive
- Its next ack/heartbeat will not be received (within the timeout), and thus $p_i$ will detect $p_j$ as having failed
Completeness = every process failure is eventually detected (no misses)

Accuracy = every detected failure corresponds to a crashed process (no mistakes)

What is a protocol that is 100% complete?
What is a protocol that is 100% accurate?

Completeness and Accuracy
- Can both be guaranteed 100% in a synchronous distributed system (with reliable message delivery in bounded time)
- Can never be guaranteed simultaneously in an asynchronous distributed system

Why?
Satisfying both Completeness and Accuracy in Asynchronous Systems

- Impossible because of arbitrary message delays, message losses
  - If a heartbeat/ack is dropped (or several are dropped) from \( p_j \), then \( p_j \) will be mistakenly detected as failed => inaccurate detection
  - How large would the \( T \) waiting period in ping-ack or \( 3T \) waiting period in heartbeating, need to be to obtain 100% accuracy?
- In asynchronous systems, delay/losses on a network link are impossible to distinguish from a faulty process
- Heartbeating – satisfies completeness but not accuracy (why?)
- Ping-Ack – satisfies completeness but not accuracy (why?)
Most failure detector implementations are willing to tolerate some inaccuracy, but require 100% Completeness.

Plenty of distributed apps designed assuming 100% completeness, e.g., p2p systems:
- “Err on the side of caution”.
- Processes not “stuck” waiting for other processes.

But it’s ok to mistakenly detect once in a while since – the victim process need only rejoin as a new process.

Both Hearbeating and Ping-ack provide:
- Probabilistic accuracy (for a process detected as failed, with some probability close to 1.0 (but not equal), it is true that it has actually crashed).
Failure Detection in a Distributed System

- That was for one process \( p_j \) being detected and one process \( p_i \) detecting failures
- Let’s extend it to an entire distributed system
- Difference from original failure detection is
  - We want failure detection of not merely one process \( (p_j) \), but all processes in system
Centralized Heartbeating

$p_j$, Heartbeat Seq. $l++$

$p_i$

Downside?
Ring Heartbeating

$p_i$, Heartbeat Seq. $l++$

$Downside?$
All-to-All Heartbeating

$p_j$, Heartbeat Seq. l++

Advantage: Everyone is able to keep track of everyone
Downside?
Efficiency of Failure Detector: Metrics

- Bandwidth: the number of messages sent in the system during steady state (no failures)
  - Small is good
- Detection Time
  - Time between a process crash and its detection
  - Small is good
- Scalability: Given the bandwidth and the detection properties, can you scale to a 1000 or million nodes?
  - Large is good
- Accuracy
  - Large is good (lower inaccuracy is good)
**Accuracy metrics**

- **False Detection Rate**: Average number of failures detected per second, when there are in fact no failures.

- Fraction of failure detections that are false.

- **Tradeoffs**: If you increase the T waiting period in ping-ack or 3T waiting period in heartbeating, what happens to:
  - Detection Time?
  - False positive rate?
  - Where would you set these waiting periods?
Other Types of Failures

- Let’s discuss the other types of failures
- Failure detectors exist for them too (but we won’t discuss those)
Processes and Channels

process $p$

send $m$

Communication channel

Outgoing message buffer

Incoming message buffer

process $q$

receive
Other Failure Types

- Communication omission failures
  - Send-omission: loss of messages between the sending process and the outgoing message buffer (both inclusive)
    - What might cause this?
  - Channel omission: loss of message in the communication channel
    - What might cause this?
  - Receive-omission: loss of messages between the incoming message buffer and the receiving process (both inclusive)
    - What might cause this?
Other Failure Types

- Arbitrary failures
  - Arbitrary process failure: arbitrarily omits intended processing steps or takes unintended processing steps.
  - Arbitrary channel failures: messages may be corrupted, duplicated, delivered out of order, incur extremely large delays; or non-existent messages may be delivered.
- Above two are Byzantine failures, e.g., due to hackers, man-in-the-middle attacks, viruses, worms, etc.
- A variety of Byzantine fault-tolerant protocols have been designed in literature!
## Omission and Arbitrary Failures

<table>
<thead>
<tr>
<th>Class of failure</th>
<th>Affects</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fail-stop</td>
<td>Process</td>
<td>Process halts and remains halted. Other processes may detect this state.</td>
</tr>
<tr>
<td>Omission</td>
<td>Channel</td>
<td>A message inserted in an outgoing message buffer never arrives at the other end’s incoming message buffer.</td>
</tr>
<tr>
<td>Send-omission</td>
<td>Process</td>
<td>A process completes <code>asend</code>, but the message is not put in its outgoing message buffer.</td>
</tr>
<tr>
<td>Receive-omission</td>
<td>Process</td>
<td>A message is put in a process’s incoming message buffer, but that process does not receive it.</td>
</tr>
<tr>
<td>Arbitrary (Byzantine)</td>
<td>Process or channel</td>
<td>Process/channel exhibits arbitrary behaviour: it may send/transmit arbitrary messages at arbitrary times, commit omissions; a process may stop or take an incorrect step.</td>
</tr>
</tbody>
</table>
Failure detectors are required in distributed systems to keep system running in spite of process crashes.

Properties – completeness & accuracy, together unachievable in asynchronous systems but achievable in synchronous systems

- Most apps require 100% completeness, but can tolerate inaccuracy
- 2 failure detector algorithms - Heartbeating and Ping
- Distributed FD through heartbeating: Centralized, Ring, All-to-all
- Metrics: Bandwidth, Detection Time, Scale, Accuracy
- Other Types of Failures
Next Week

- Reading for Next Topics:
  Sections 11.1-11.5
  - Time and Synchronization
  - Global States and Snapshots