Lecture 4
Failure Detection
Reading: Section 15.1 and parts of 2.4.2
Next Week

• **Next Tuesday: Jack Dorsey**, co-founder of Twitter and founder of Square (mobile payments) is visiting our class 2-2.30pm!

• **Townhall-style Q&A**

• We are using Google Moderator to post questions, and up/down-vote questions

• Please follow the Google Moderator link from the website – vote and post your own questions!
You’ve been put in charge of a datacenter (think of the Prineville Facebook DC), and your manager has told you, “Oh no! We don’t have any failures in our datacenter!”

Do you believe him/her?

What would be your first responsibility?

Build a failure detector

What are some things that could go wrong if you didn’t do this?
Failures are the norm

... not the exception, in datacenters.

Say, the rate of failure of one machine (OS/disk/motherboard/network, etc.) is once every 10 years (120 months) on average.

When you have 120 servers in the DC, the mean time to failure (MTTF) of the next machine is 1 month.

When you have 12,000 servers in the DC, the MTTF is about once every 7.2 hours!
To build a failure detector

- You have a few options

  1. Hire 1000 people, each to monitor one machine in the datacenter and report to you when it fails.
  2. Write a failure detector program (distributed) that automatically detects failures and reports to your workstation.

Which is more preferable, and why?
Two Different System Models

Whenever someone gives you a distributed computing problem, the first question you want to ask is, “What is the system model under which I need to solve the problem?”

- **Synchronous Distributed System**
  - Each message is received within bounded time
  - Each step in a process takes \( lb < \text{time} < ub \)
  - (Each local clock’s drift has a known bound)
  
  Examples: Multiprocessor systems

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

  Examples: Internet, wireless networks, datacenters, most real systems
Failure Model

- 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
    - Special case of crash-stop model (use a new identifier on recovery)

- We will focus on Crash-stop failures
  - They are easy to detect in synchronous systems
  - Not so easy in asynchronous systems
What’s a failure detector?

pi

pj
What’s a failure detector?

Crash-stop failure
(pj is a failed process)
What’s a failure detector?

needs to know about pj’s failure
(pi is a non-faulty process or alive process)

Crash-stop failure
(pj is a failed process)

There are two main flavors of Failure Detectors…
I. Ping-Ack Protocol

needs to know about pj’s failure

- pi queries pj once every T time units
- if pj does not respond within another T time units of being sent the ping, pi detects pj as failed

Worst case Detection time = 2T
If pj fails, then within T time units, pi will send it a ping message. pi will time out within another T time units.
The waiting time ‘T’ can be parameterized.
II. Heartbeating Protocol

needs to know about pj’s failure

- pj maintains a sequence number
- pj sends pi a heartbeat with incremented seq. number after every T time units

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

If \( T \gg \) round trip time of messages, then worst case detection time \(~ 3*T\) (why?)

The ‘3’ can be changed to any positive number since it is a parameter
In a Synchronous System

• The Ping-ack and Heartbeat failure detectors are always correct
  – If a process pj fails, then pi will detect its failure as long as pi itself is alive

• Why?
  – Ping-ack: set waiting time ‘T’ to be > round—trip time upper bound
    » pi->pj latency + pj processing + pj->pi latency + pi processing time
  – Heartbeat: set waiting time ‘3*T’ to be > round—trip time upper bound
Failure Detector Properties

- **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
  - 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 pj, then pj will be mistakenly detected as failed => inaccurate detection
  - How large would the T waiting period in ping-ack or 3*T 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?)
Completeness or Accuracy? (in asynchronous system)

- 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 and catch up.
- 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.
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

Downside?
Ring Heartbeating

\[ p_j, \text{ Heartbeat Seq. } l_{++} \]

No SPOF (single point of failure)

Downside?
All-to-All Heartbeating

pj, 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**: How do bandwidth and detection properties scale with N, the number of processes?

- **Accuracy**
  - Large is good (lower inaccuracy is good)
Accuracy metrics

• False Detection Rate/False Positive Rate (inaccuracy)
  – Multiple possible metrics
  – 1. Average number of failures detected per second, when there are in fact no failures
  – 2. Fraction of failure detections that are false

• Tradeoffs: If you increase the T waiting period in ping-ack or 3*T waiting period in heartbeating what happens to:
  – Detection Time?
  – False positive rate?
  – Where would you set these waiting periods?
Suspicion

- Augment failure detection with suspicion count
- Ex: In all-to-all heartbeating, suspicion count = number of machines that have timed out waiting for heartbeats from a particular machine M
  - When suspicion count crosses a threshold, declare M failed
  - Issues: Who maintains this count? If distributed, need to circulate the count
- Lowers mistaken detections (e.g., message dropped, Internet path bad), e.g., in Cassandra key-value store
- Can also keep much longer-term failure counts, and use this to blacklist and greylist machines, e.g., in OpenCorral CDN
Membership Protocols

- Maintain a list of other alive (non-faulty) processes at *each process* in the system
- Failure detector is a component in membership protocol
  - Failure of pj detected -> delete pj from membership list
  - New machine joins -> pj sends message to everyone -> add pj to membership list
- Flavors
  - Strongly consistent: all membership lists identical at all times (hard, may not scale)
  - Weakly consistent: membership lists not identical at all times
  - Eventually consistent: membership lists always moving towards becoming identical eventually (scales well)
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

process $q$

receive

Incoming message buffer
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 or Crash-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>send</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>
Summary

- Failure Detectors
- Completeness and Accuracy
- Ping-ack and Heartbeating
- Suspicion, Membership
Next Week

- **Next Tuesday: Jack Dorsey**, co-founder of Twitter and founder of Square (mobile payments) is visiting our class 2-2.30pm!
- Townhall-style Q&A
- We are using Google Moderator to post questions, and up/down-vote questions
- Please follow the Google Moderator link from the website – vote and post your own questions!
Next Week

- Reading for Next Two Lectures: Sections 14.1-14.5
  - Time and Synchronization
  - Global States and Snapshots

- HW1 already out, due Sep 20th
- MP1 already out, due 9/16: By now you should
  - Be in a group (send email to us TODAY, subject line: “425 MP group”), use Piazza to find partners
  - Have a basic design.
  - If you’ve already started coding, you’re doing well.