CSE 486/586 Distributed Systems

Failure Detectors

Slides by: Steve Ko
Computer Sciences and Engineering
University at Buffalo

Administrivia

- Programming Assignment 2 is out
- Please continue to monitor Piazza
  - OS and networking background resources are up
  - Help and hints for successful completion/submission/*etc.*
- Make sure you submit correctly!

Today

- How do we handle failures?
  - Cannot answer this fully (yet!)
- You'll learn new terminologies, definitions, etc.
- Let's start with some new definitions.
- One of the two fundamental challenges in distributed systems
  - Failure
  - Ordering (with concurrency)

Two Different System Models

- Synchronous Distributed System
  - Every message is received within bounded time
  - Each step in a process takes lb < time < ub
  - (Each local clock's drift has a known bound)
  - Example: 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
- These are used to reason about how protocols would behave, *e.g.*, in formal proofs.

Failure Model

- What is a failure?
- We'll consider: process omission failure
  - A process disappears.
  - Permanently: crash-stop (fail-stop) – a process halts and does not execute any further operations
  - Temporarily: crash-recovery – a process halts, but then recovers (reboots) after a while
- We will focus on *crash-stop* failures
  - Meaning, we assume *there's no other failure* (*e.g.*, network error).
  - More failure types at the end of this lecture.
  - They are easy to detect in synchronous systems
  - Not so easy in asynchronous systems

Why, What, and How

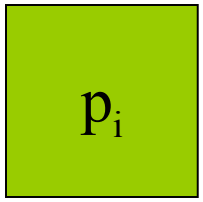
- Why design a failure detector?
  - First step to failure handling
- What do we want from a failure detector?
  - Failures are always detected (completeness)
  - No false positives (accuracy)
- How do we design one?

What is a Failure Detector?

What is a Failure Detector?

Crash-stop failure

(p_j is a *failed* process)

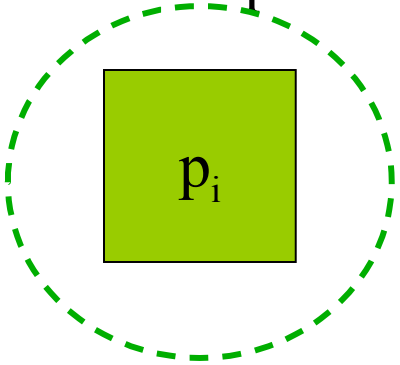


What is a Failure Detector?

needs to know about p_j's failure

(p_i is a *non-faulty* process

or *alive* process)

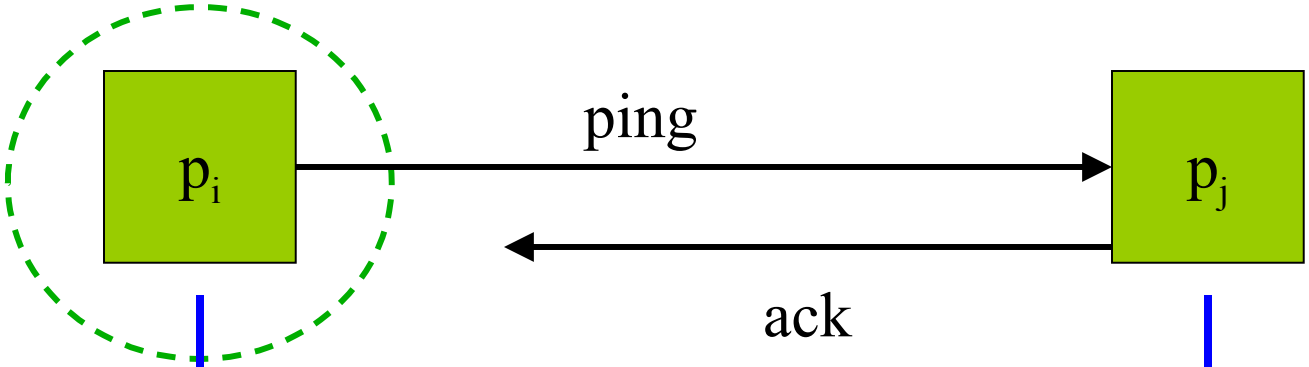


Crash-stop failure

(p_j is a *failed* process)

There are two styles of failure detectors

I. Ping-Ack Protocol

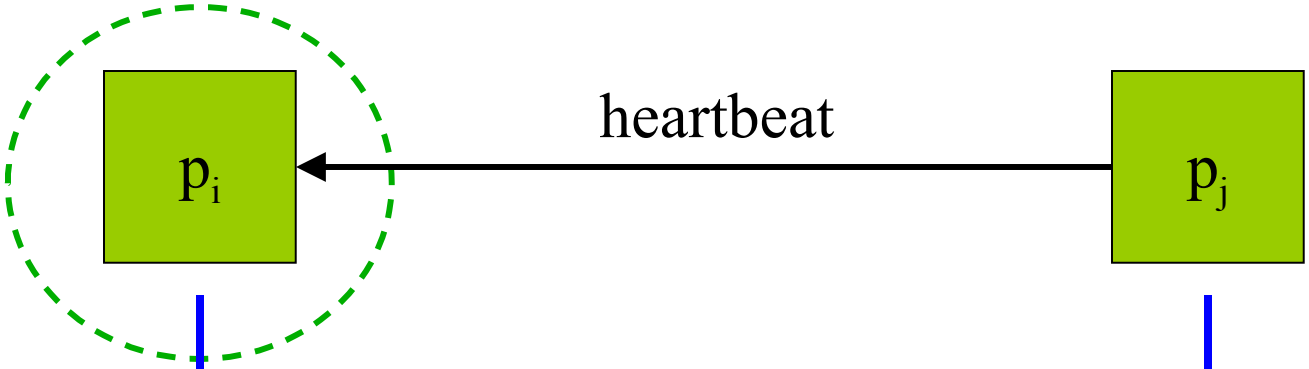


- p_i queries p_j once every T time units
- If p_j does not respond within another T time units of being sent the ping, p_i detects/declares p_j as failed

- p_j replies

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



- If p_i has not received a new heartbeat for the past, (say) 3T time units, then p_i detects p_j as failed

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

If T ≫ one-way delay of messages, then worst case detection time ~ 3T (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. For example:
  - Ping-Ack: set waiting time ‘T’ to be > transmission upper bound
  - Heartbeat: set waiting time ‘3*T’ to be > transmission 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

Failure Detector Properties

- What does it mean for a failure detector to be correct?
- Completeness: every process failure is eventually detected
- Accuracy: every detected failure corresponds to a crashed process
- What is a protocol that is 100% complete?
- What is a protocol that is 100% accurate?
- Completeness and Accuracy
  - Both can be guaranteed in a synchronous distributed system (with reliable message delivery in bounded time)
  - Can never be guaranteed simultaneously in an asynchronous distributed system
  - Why?

Completeness and Accuracy in Asynchronous Systems

- Impossible because of arbitrary message delays
  - Packet loss can be indistinguishable from host failure
  - How large would the T waiting period in ping-ack or 3T waiting period in heartbeating need to be to be 100% accurate?
  - In asynchronous systems, network delays are impossible to distinguish from process failure
- Heartbeating – satisfies completeness but not accuracy (why?)
- Ping-Ack – satisfies completeness but not accuracy (why?)
- Point: You can't design a perfect failure detector!
  - You need to think about what metrics are important.

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
- It's ok to mistakenly detect once in a while
  - The victim process need only rejoin as a new process
- Both Heartbeating and Ping-Ack provide
  - Probabilistic accuracy: for a process detected as failed, with some probability close (but not equal) to 1.0, it is true that it has actually failed.

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
- Why should we do this? How?

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 bandwidth and detection properties, can you scale to a 1000 or a million nodes?

- Large is good

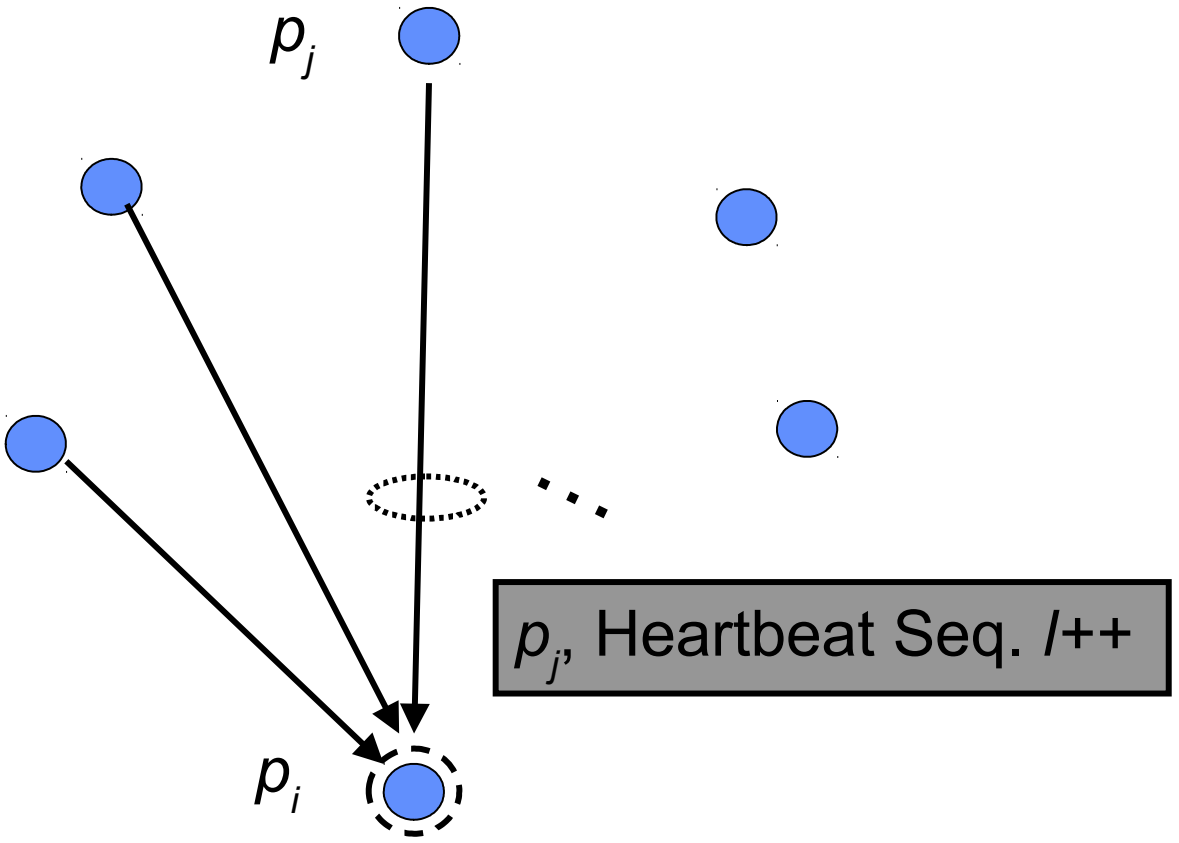
Accuracy:

- Large is good (lower inaccuracy is good)

Accuracy Metrics

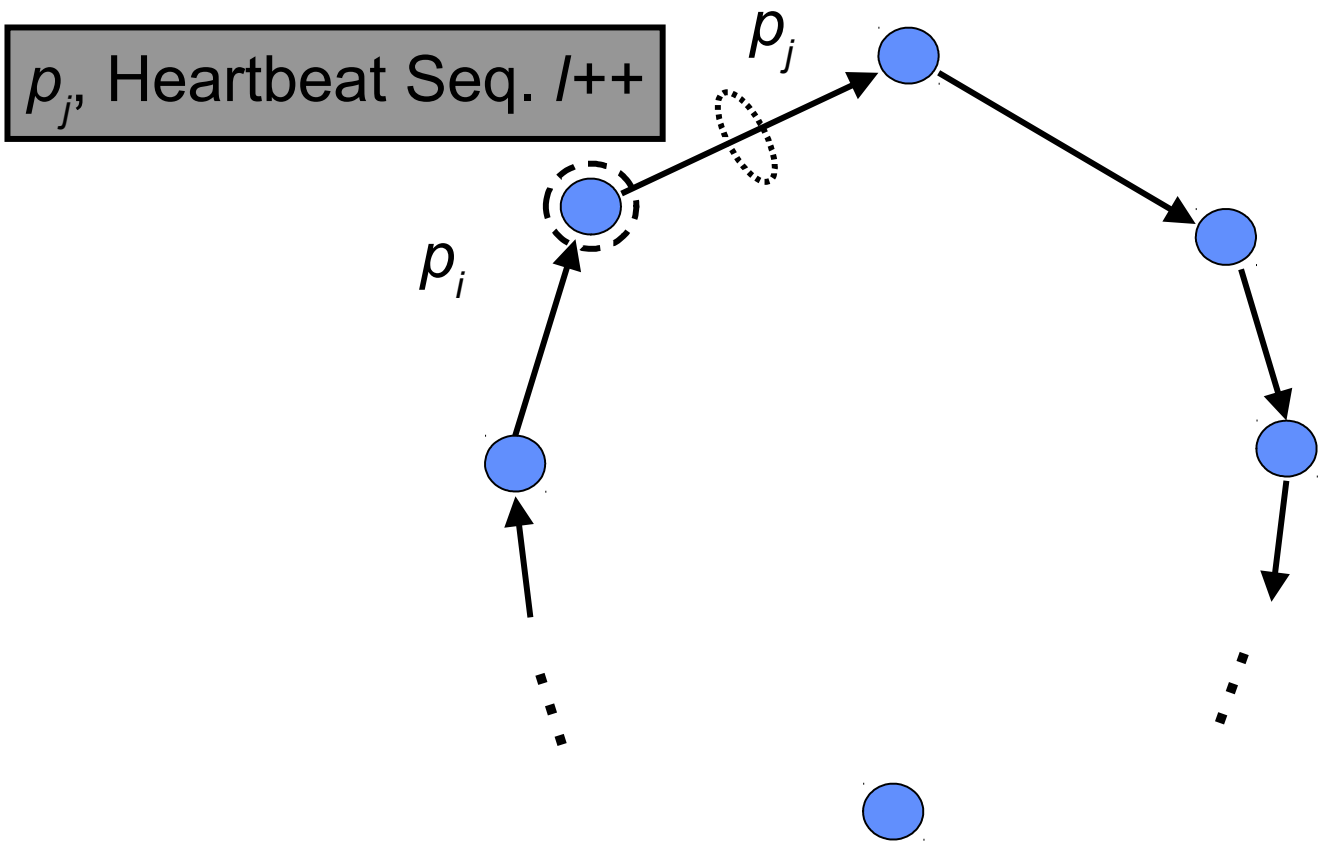
- False Detection Rate: Average number of failures detected per second, when there are in fact no failures
- Fraction of failure detections that are false
- Tradeoffs: If you increase the T waiting period in ping-ack or 3*T waiting period in heartbeating, what happens to:
  - Detection Time?
  - False positive rate?
  - How would you set these waiting periods?

Centralized Heartbeat



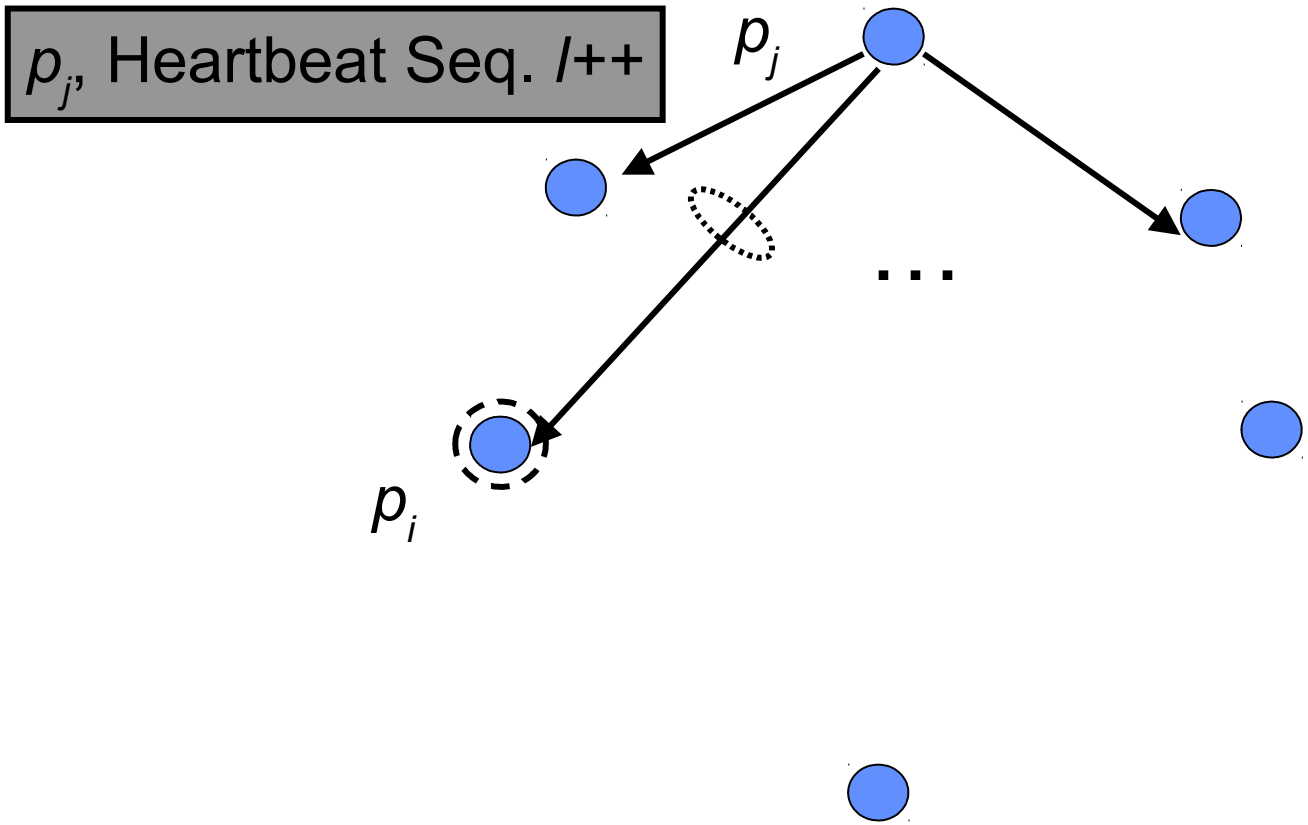
Downside?

Ring Heartbeat



Downside?

All-to-All Heartbeat



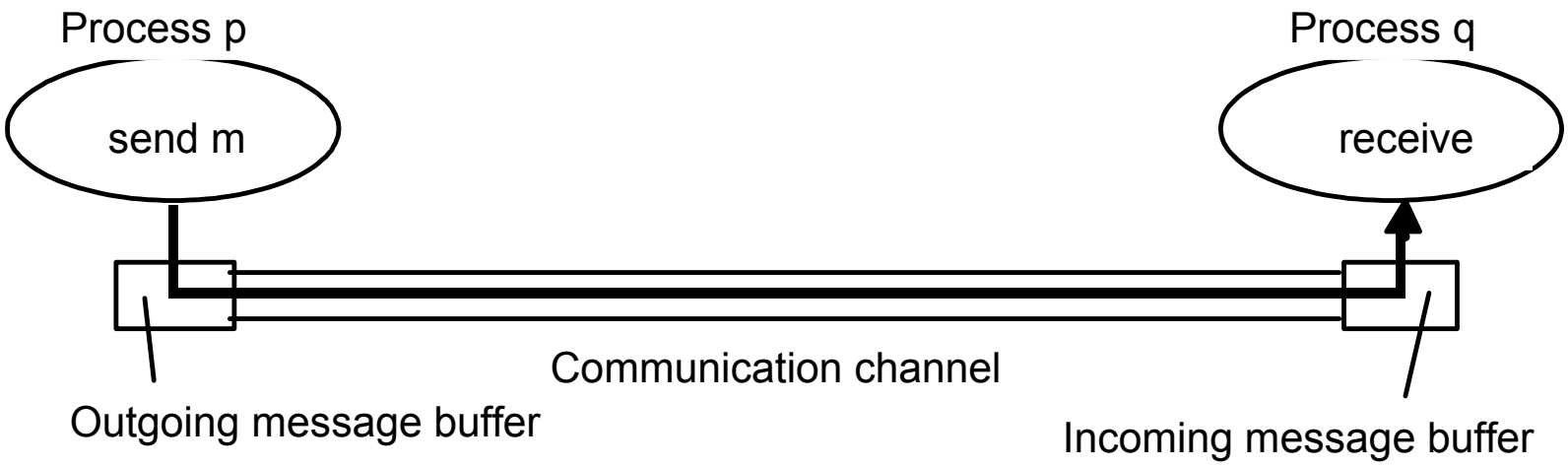
Advantage: Everyone is able to keep track of everyone

Downside?

Other Types of Failures

- Let's discuss other types of failures
- Failure detectors exist for them, too
  - (but we won't discuss them)

Processes and Channels



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.
- These 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 the literature!

Omission and Arbitrary Failures

Class of failure	Detects	Description
Fail-stop	Process	Process halts and remains halted. Other processes may detect this state.
Omission	Channel	A message inserted in an outgoing message buffer never arrives at the receiving message buffer.
Send-omission	Process	A process completes a *send,* but the message is not placed in its outgoing message buffer.
Receive-omission	Process	A message is placed in a process's incoming message buffer, but the process does not receive it.
Arbitrary (Byzantine)	Process *or* Channel	A process or channel exhibits arbitrary behavior. It may send/transmit messages at arbitrary times, commit omissions; a process may stop or behave incorrectly.

Summary

- Failure detectors are required in distributed systems to keep systems 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 - heartbeat and ping
- Distributed FD through heartbeat: centralized, ring, all-to-all
- Metrics: bandwidth, detection time, scalability, accuracy
- Other types of failures
- Next time: the notion of time in distributed systems

References

Required reading:

- Textbook sections 2.4.2 and 15.1

Acknowledgements

- These slides are by Steve Ko (with permission), lightly modified by Ethan Blanton.
- These slides contain material developed and copyrighted by Indranil Gupta at UIUC.