CS 582: Distributed Systems

System Models

Dr. Zafar Ayyub Qazi

Fall 2024

Announcement

- Please signup on slack

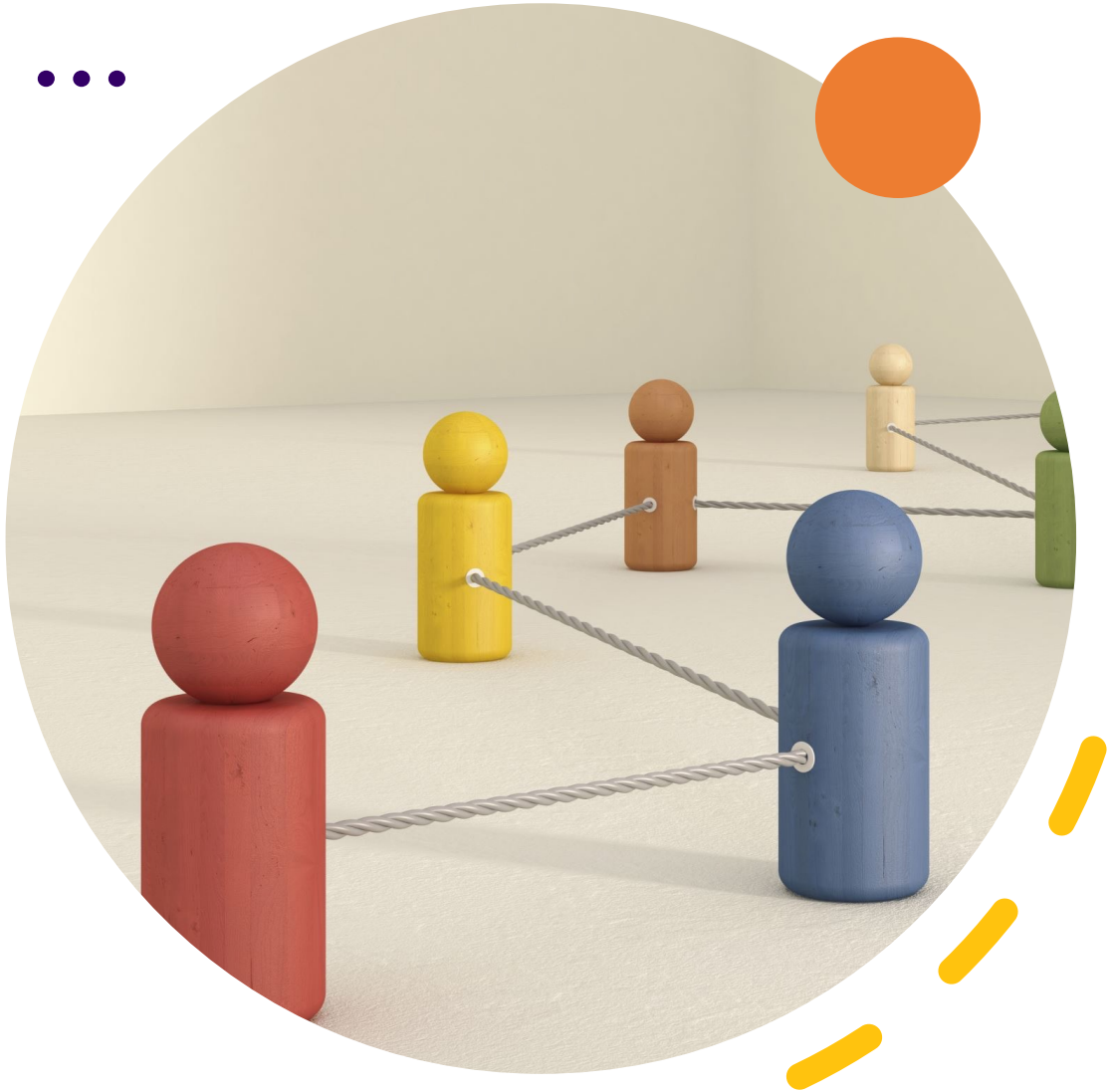
Agenda

- Models of distributed systems
- Failure Detectors

By the end of the class …

You should be able to:

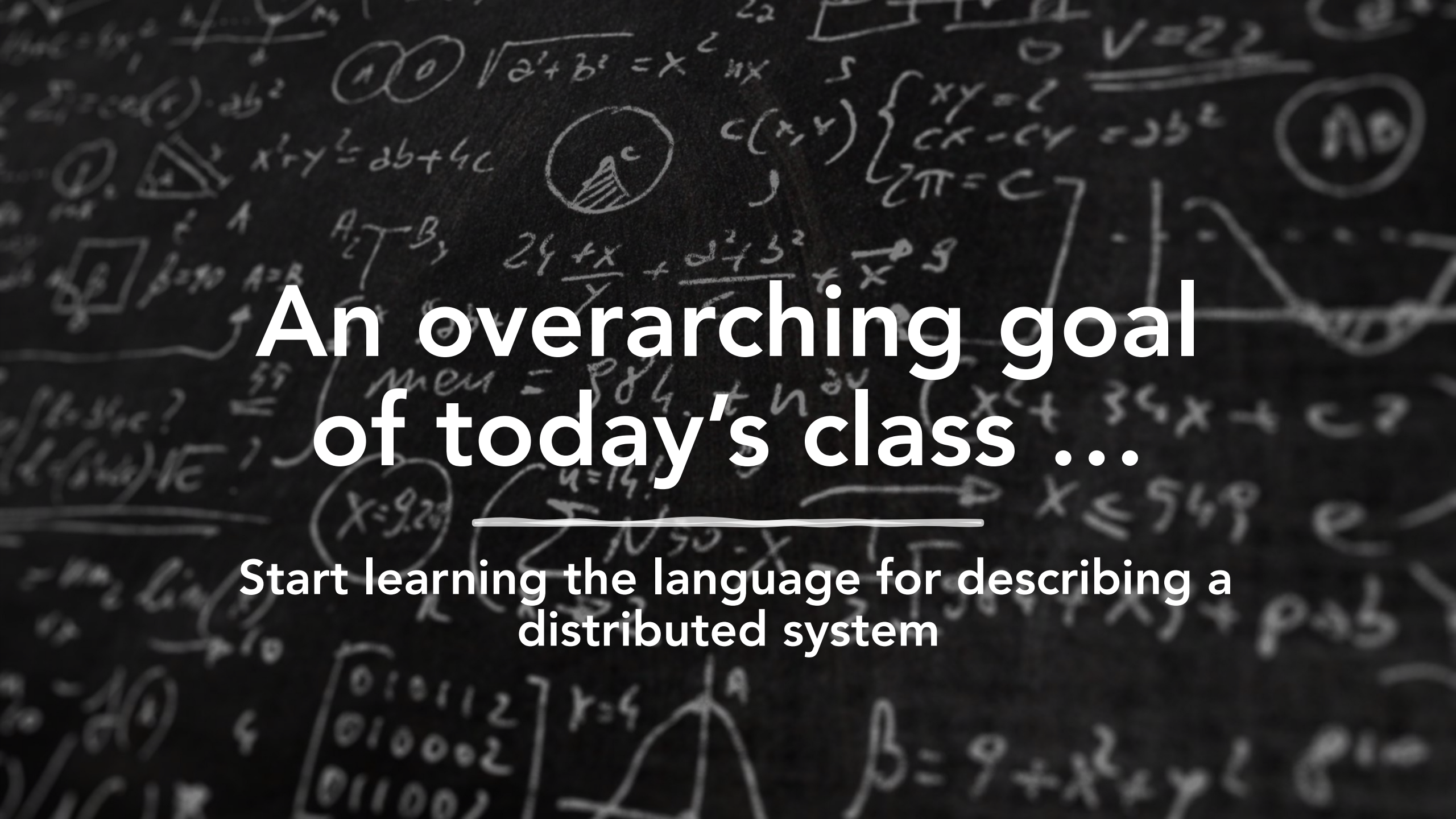
- ❑ Describe different network behaviour models
- ❑ Explain different node behaviour models
- ❑ Explain the properties of a communication channel
- ❑ Compare and contrast different synchrony assumptions
  - Synchronous, asynchronous, and partially synchronous



By the end of the class …

You should be able to:

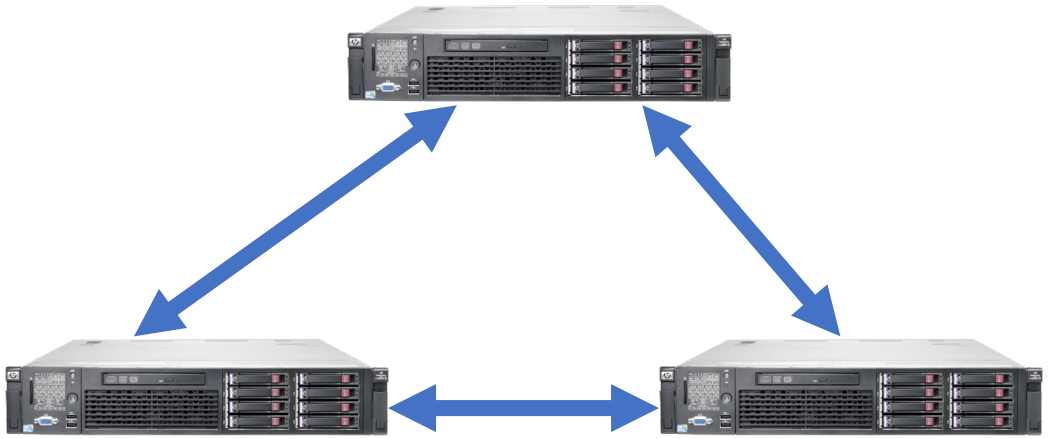
- ❑ Explain and analyze the different types of building blocks for detecting that a process has crashed
- ❑ Explain the important properties of failure detectors
- ❑ Explain if we can accurately detect process failures in asynchronous systems



An overarching goal of today's class …

Start learning the language for describing a distributed system

What is a distributed system?



1. Multiple computers
2. Connected by a network
3. Doing something together

System Model

- Before designing/analyzing a distributed algorithm, it is necessary to make the relevant assumptions about the system explicit

- Captures the assumptions that a distributed algorithm makes about how nodes and networks behave:
  - Network behaviour (e.g., message loss)
  - Node behaviour (e.g., crashes)
  - Timing behaviour (e.g., latency)

- A system model is an abstract description of these properties
  - Which can be implemented by various technologies in practice

Example: Snapshots from the Paxos Paper

2 The Consensus Algorithm

2.1 The Problem

Assume a collection of processes that can propose values. A consensus algorithm ensures that a single one among the proposed values is chosen. If no value is proposed, then no value should be chosen. If a value has been chosen, then processes should be able to learn the chosen value. The safety requirements for consensus are:

- Only a value that has been proposed may be chosen,
- Only a single value is chosen, and
- A process never learns that a value has been chosen unless it actually has been.

We won't try to specify precise liveness requirements. However, the goal is to ensure that some proposed value is eventually chosen and, if a value has been chosen, then a process can eventually learn the value.

We let the three roles in the consensus algorithm be performed by three classes of agents: *proposers*, *acceptors*, and *learners*. In an implementation, a single process may act as more than one agent, but the mapping from agents to processes does not concern us here.

Assume that agents can communicate with one another by sending messages. We use the customary asynchronous, non-Byzantine model, in which:

- Agents operate at arbitrary speed, may fail by stopping, and may restart. Since all agents may fail after a value is chosen and then restart, a solution is impossible unless some information can be remembered by an agent that has failed and restarted.
- Messages can take arbitrarily long to be delivered, can be duplicated, and can be lost, but they are not corrupted.

Network Behaviour

- Networks are unreliable

https://slate.com/technology/2014/08/ shark-attacks-threaten-google-s-undersea-internet-cables-video.html

Message Drops

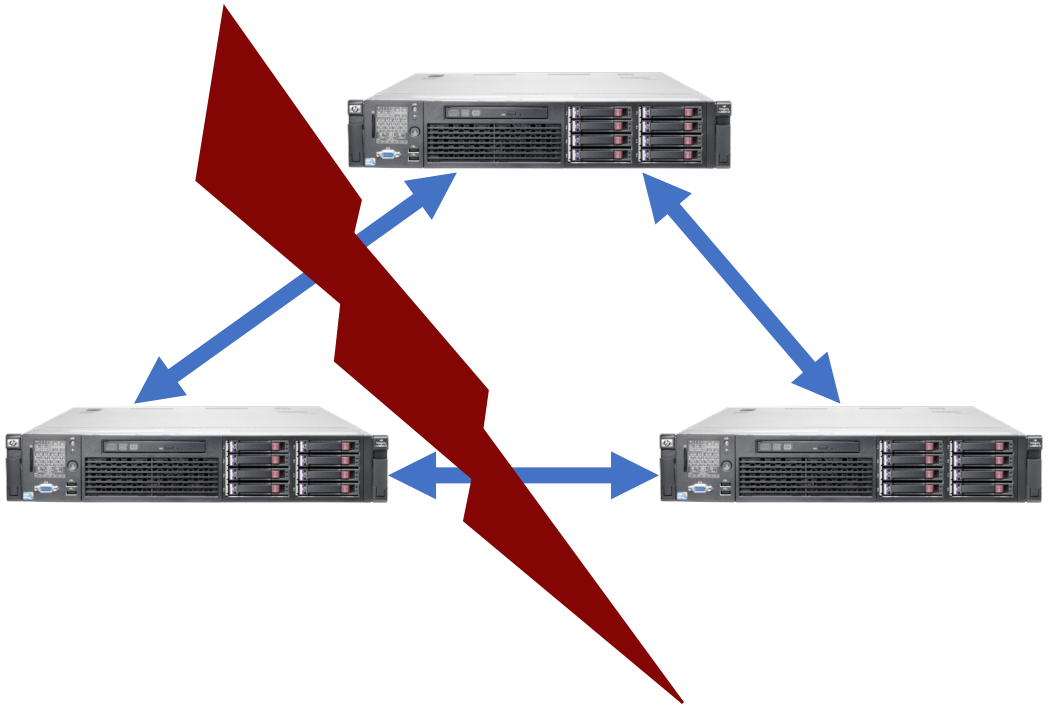
- What could be other causes of message drops besides link failures?

System Model: Network Behaviour

- Assume a bi-directional point-point communication between two nodes, with one of:
  - Reliable (perfect) links:
    - A message is received if and only if it is sent. Messages may be reordered.
  - Fair-loss links:
    - Messages may be lost, duplicated, or reordered.

      If you keep retrying, a message eventually gets through.
  - Arbitrary links (active adversary)
    - A malicious adversary may interfere with messages (eavesdrop, modify, drop, spoof, replay).

Network Partition



Failure Model: Fail-Stop

- A node fails by crashing
  - A machine is either working correctly or it is doing nothing
  - Can (possibly) recover at some later point

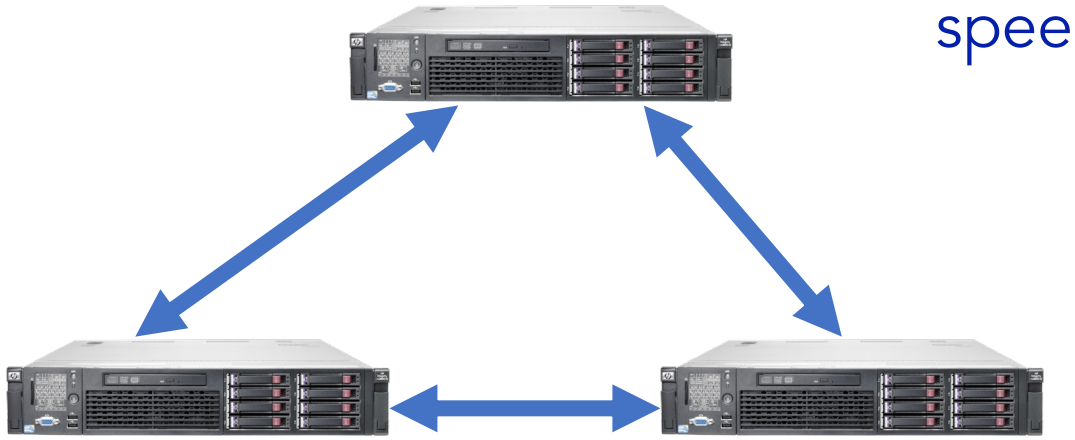
Failure Model: Byzantine (fail arbitrary)

- A node is faulty if it deviates from the algorithm
  - Performs incorrect computation
  - Send different and wrong messages
  - Not send messages at all
  - Lie about the input value
  - Collude with other failed nodes
  - And more …

System model: Node behaviour

- Fail-Stop
  - A node is faulty if it crashes (at any moment)
  - It may resume execution sometime later
- Byzantine (fail-arbitrary)
  - A node is faulty if it deviates from the algorithm
  - Faulty nodes may do anything, including crashing or malicious behaviour
- A node that is not faulty is called correct

Communication Channel Properties



In a distributed system, there is a minimum delay of coordination that cannot be overcome: the speed of light limits how fast information can travel

Communication channel properties

- Latency:
  - Delay in sending a message to the receiver

- Bandwidth:
  - Total amount of information that can be transmitted per unit time

Communication channel properties

- Does **latency** remain constant in a network?

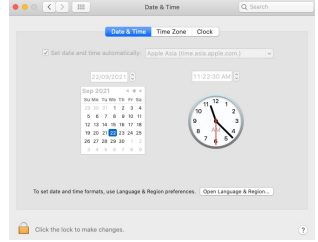
- Does **available bandwidth** remain constant in a network?

Events and their latancies

nanosecond events	microsecond events	millisecond events
register file: 1ns–5ns	datacenter networking: O(1μs)	disk: O(10ms)
cache accesses: 4ns–30ns	new NVM memories: O(1μs)	low-end flash: O(1ms)
memory access: 100ns	high-end flash: O(10μs)	wide-area networking: O(10ms)
	GPU/accelerator: O(10μs)	

Luiz, A., & Patterson, D. (2017). Attack of the Killer Microseconds. *Communications of the ACM*, 60(4), 36-44. https://doi.org/10.1145/3015146

Differing clocks



- Each computer in a distributed has its own internal clock
- Local clock of different computers can show different time values
- Clocks can drift from perfect times at different rates

More on this in later lectures when we study time synchronization …

Ordinary and authoritative clocks

- Ordinary quartz crystal clocks (cheap, laptops/desktops/phones):
  - Drift rate is about 10^{-6} seconds/second
  - Drift by 1 second every 11.6 days
  - Skew of about 30 minutes after 60 years

- High-precision atomic clocks (expensive ~$25K)
  - Drift rate is about 10^{-13} seconds/second
  - Skew of about 0.18ms after 60 years
  - Used as standard for real time
  - Universal Coordinated Time (UTC)* obtained from such clocks

System: Synchrony assumptions

- Synchronous:
  - Known bounds on time taken by each step in a process
  - Known bounds on message passing delays.
  - Known bounds on clock drift rates
- Asynchronous:
  - No bounds on process execution speeds
  - No bounds on message passing delays
  - No bounds on clock drift rates

Synchronous and Asynchronous

- Most real-world systems are asynchronous
  - However, some problems in distributed computing are impossible to solve in an asynchronous model

Partially synchronous systems

- The system is asynchronous for some finite (but unknown) periods of time, synchronous otherwise

Violations of synchrony in practice

- Networks usually have quite predictable latency, which can occasionally increase:
  - Message loss requiring retry
  - Congestion/contention causing queuing
  - Network/route reconfiguration
- Nodes usually execute code at a predictable speed, with occasional pauses
  - Operating system scheduling issues
  - Page fault, swap, thrashing

System Models Summary

- Network:
  - Reliable, fair-loss, arbitrary
- Nodes:
  - Fail-stop or Byzantine
- Timing:
  - Synchronous, partially synchronous, or asynchronous
- These are the basic set of assumptions for any distributed algorithm

Key aspects of a distributed system

- Processes must communicate with one another to coordinate actions. Communication time is variable
- Nodes and communication channels may fail
- Different processes (on different computers) have different clocks!

Questions

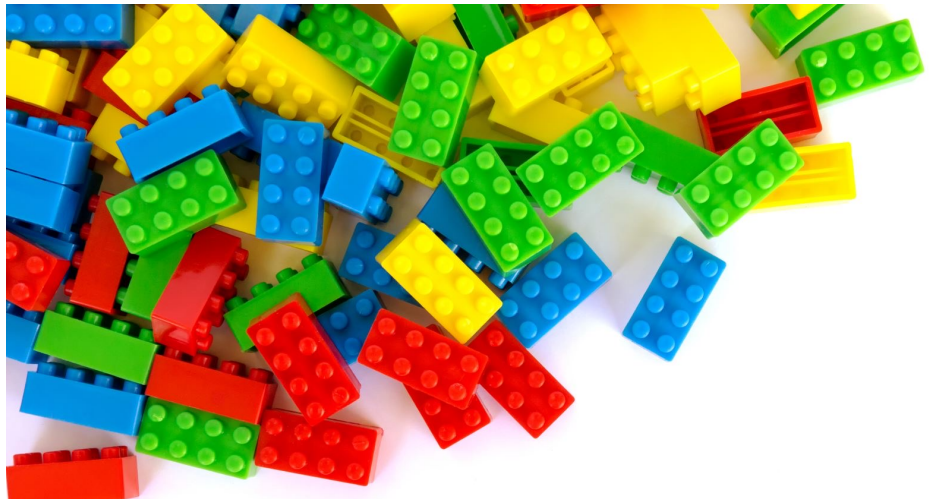
Failure Detectors

What is a failure detector?

- Service that detects failures and reports them
- Key building block in any distributed system setting
  - Cloud/Datacenters
  - Replicated servers
  - Distributed databases

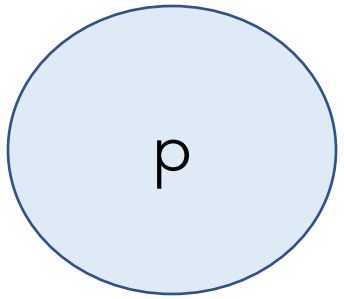
We will focus on **fail-stop** (**crash**) failures

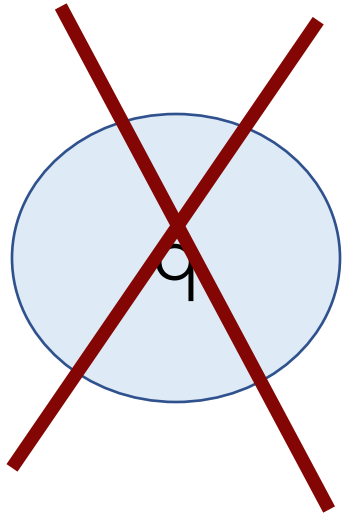
Building Blocks for Designing Failure Detectors



How to detect a process has crashed?

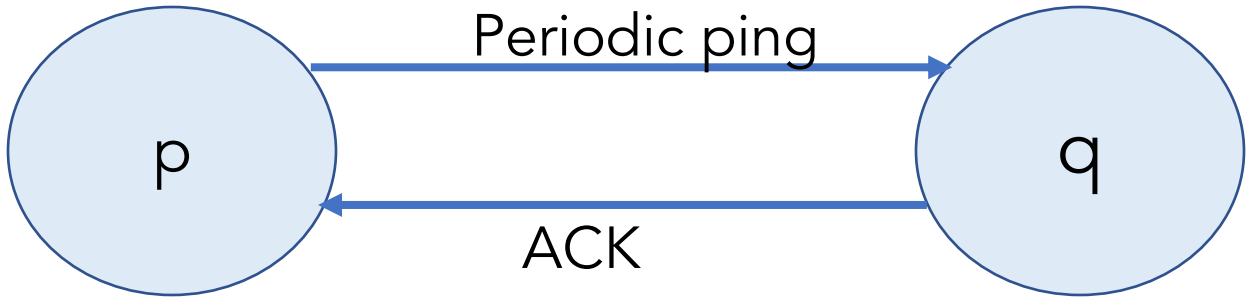
Our assumptions

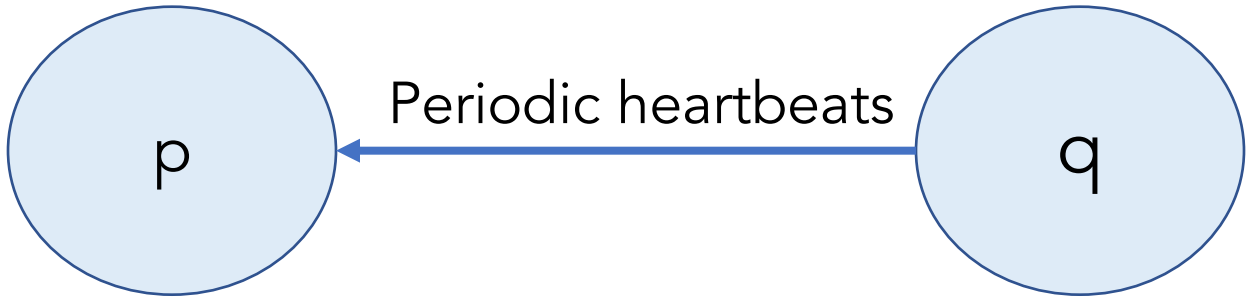




- Fails by crashing
- **Stops working and stays that way**

How to detect a process has crashed?





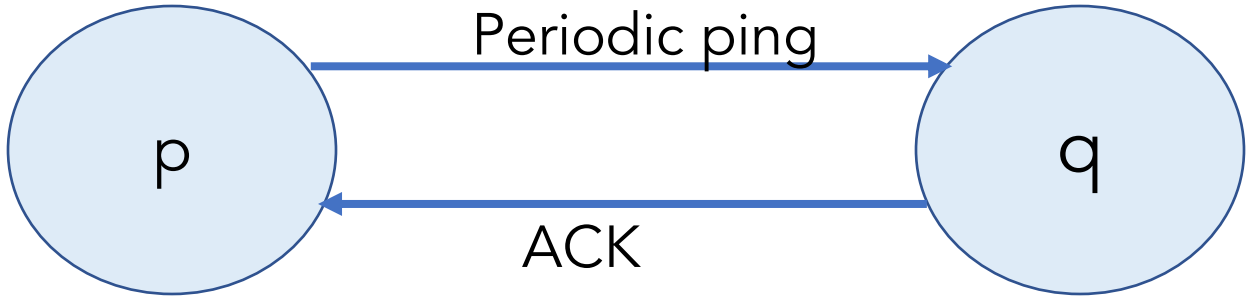
Our assumptions

(1) We will **ignore the processing delays** at p & q

(2) If we consider a **synchronous model**, we will assume that for any message exchanged between p and q, **min ≤ one-way message delay ≤ max**

Pings in a synchronous setting

Assume we know maximum (**max**) and minimum (**min**) one-way network delays. Ignore processing delays at p & q



p send pings every T seconds to q

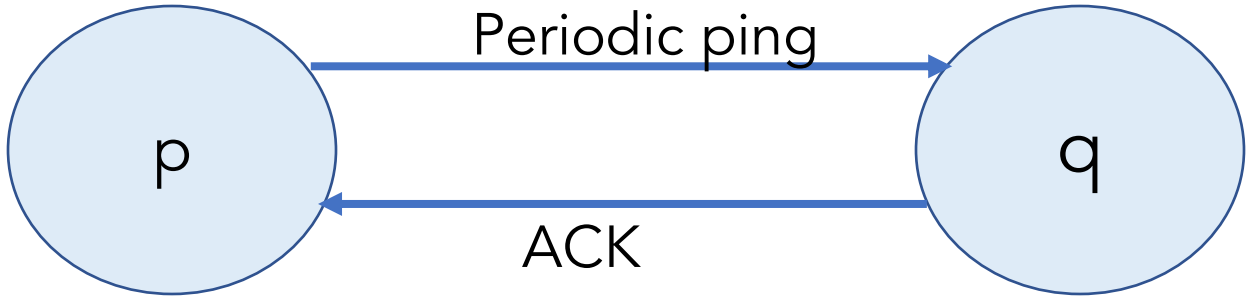
How long should p wait before deciding that q has failed?

Assuming a synchronous distributed system

Lets say Δt is the *timeout* value at p

Pings in a synchronous setting

Assume we know maximum (**max**) and minimum (**min**) one-way network delays. Ignore processing delays at p & q



p send pings every T seconds to q

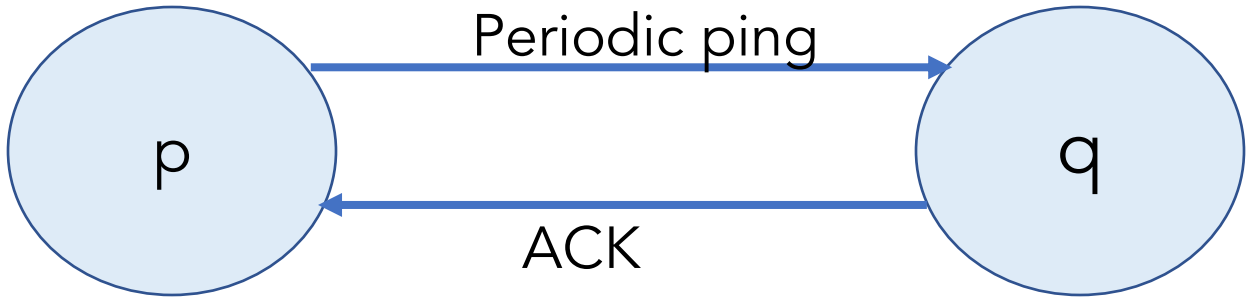
How long should p wait before deciding that q has failed?

Assuming a synchronous distributed system

Lets say Δt is the *timeout* value at p

Δt = 2 * max

Pings in an asynchronous setting



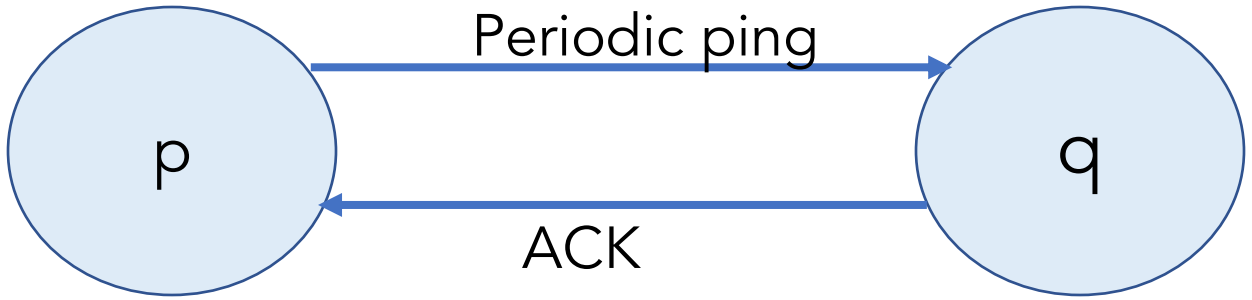
p send pings every T seconds to q

How long should p wait before deciding that q has failed?

Assuming an asynchronous distributed system

Lets say Δt is the *timeout* value at p

Pings in an asynchronous setting



p send pings every T seconds to q

How long should p wait before deciding that q has failed?

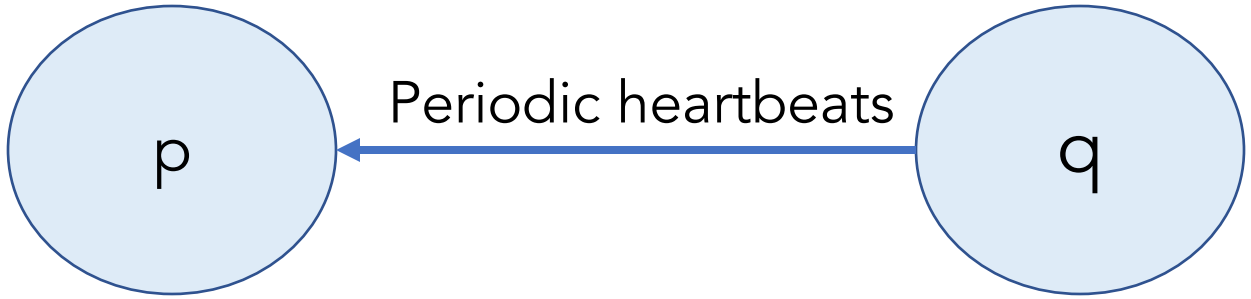
Assuming an asynchronous distributed system

Lets say Δt is the *timeout* value at p

Δt = k × (maximum RTT), where generally k > 1

Heartbeats in a synchronous setting

Assume we know maximum (**max**) and minimum (**min**) one-way network delays. Ignore processing delays at p & q



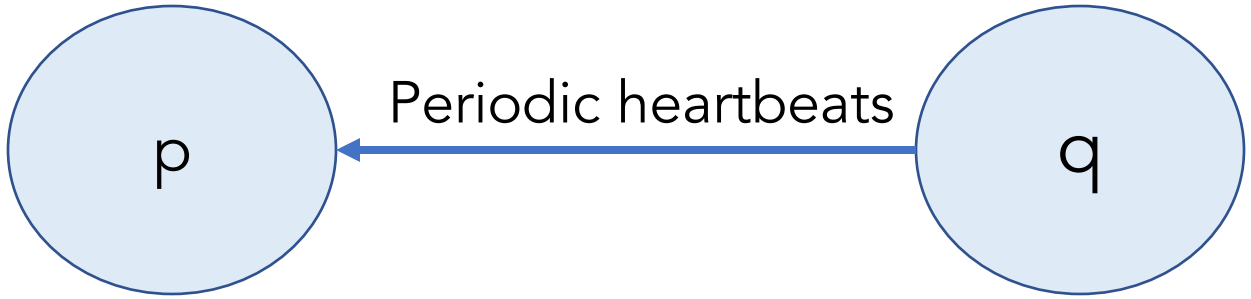
q sends heartbeat msgs every T seconds to p

How long should p wait before deciding that q has failed?

Assuming a synchronous distributed system

Heartbeats in a synchronous setting

Assume we know maximum (**max**) and minimum (**min**) one-way network delays. Ignore processing delays at p & q



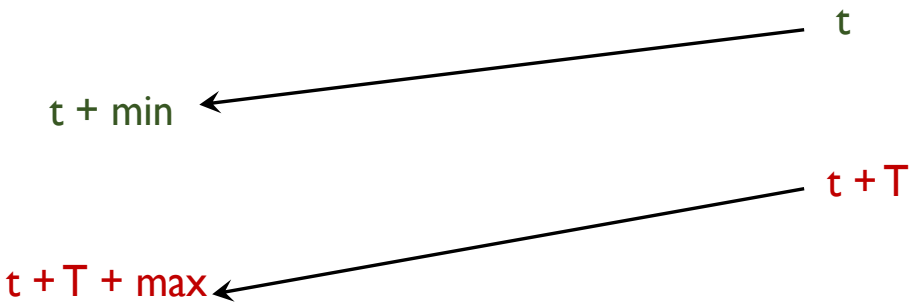
q sends heartbeat msgs every T seconds to p

How long should p wait before deciding that q has failed?

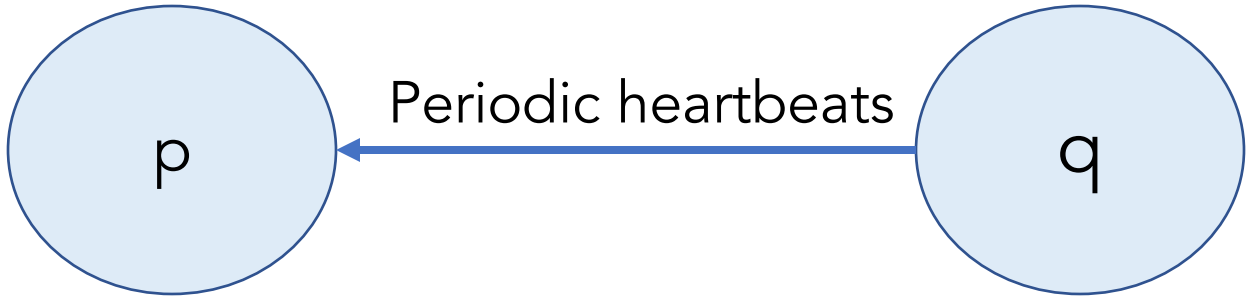
Assuming a synchronous distributed system

Lets say Δt is the *timeout* value at p

$\Delta t = T + (max - min)$



Heartbeats in an asynchronous setting



q sends heartbeat msgs every T seconds to p

How long should p wait before deciding that q has failed?

Assuming an asynchronous distributed system

Lets say Δt is the *timeout* value at p

$\Delta t = k \times$ (max. observed delay between heartbeats), where generally $k > 1$

Correctness properties of failure detection

- Completeness
  - Every failed process is *eventually* detected
- Accuracy
  - Every detected failure corresponds to a failed process (no mistakes)

Is it possible to achieve both completeness and accuracy in failure detection in a **synchronous system**?

Is it possible to achieve both completeness and accuracy in failure detection in an **asynchronous system**?

Correctness properties of failure detection

- Synchronous system
  - Failure detection via ping-ack and heartbeat is both complete and accurate

- Asynchronous system
  - *Our strategy for ping-ack and heartbeat is complete*
  - Impossible to guarantee both completeness and accuracy*

*Proven by [Chandra and Toueg] in the 1990s

What real distributed systems prefer?

- Completeness
- Accuracy

Guarantee

Partial or Probabilistic

What real distributed systems prefer?

- Completeness
- Accuracy
- Speed/Failure detection time
  - Time to first detection of a failure
- Scale
  - Network message load
  - Equal load on each process

Guarantee

Partial or Probabilistic

Time until some process detects a failure

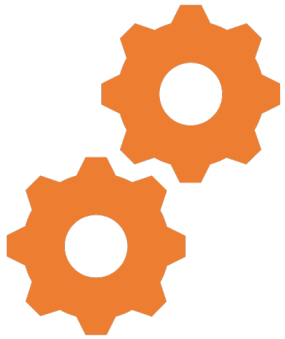
No bottlenecks or single points of failure

Inspite of arbitrary simultaneous process failures

Summary so far

- Failure detection (detecting a crashed process):
  - Send periodic ping-acks or heartbeats
  - Report crash if no response until a timeout
  - Timeout can be precisely computed for synchronous systems and estimated for asynchronous
  - Correctness properties
    - Completeness: Every failed process is *eventually* detected
    - Accuracy: Every detected failure corresponds to a failed process (no mistakes)

Next Lecture



Designing failure detectors for systems with more than 2 processes

Remote Procedure Calls