SCC.311: Introduction

Prof. Barry Porter

Welcome to Distributed Systems

- Your teaching team:

Prof. Barry Porter

Dr. Onur Ascigil

Teaching Assistants

- Ben Craine
- Daria Smirnova
- Zsolt Nemeth
- Ellie Farrar
- William Dove
- Jesse Phillips
- Ryan Thomas

Lecture Schedule

- Each week we have two lectures: **Monday 2pm** & **Friday 12 noon**

Week	Tuesday	Friday
1	Introduction	Remote Invocation
2	Group Communication	Security Systems
3	Peer to Peer & Cloud Computing	Software Containment
4	Fault Tolerance I	Fault Tolerance II
5	Google's Infrastructure I	Google's Infrastructure II
6	PAXOS consensus	RAFT consensus protocol
7	PBFT consensus	Eventual Consistency
8	Distributed Ledgers (blockchain)	Time and Ordering
9	Indirect Communication	**No lecture** (final coursework submission)
10	Design Principles and Patterns	Revision & Exam Preparation

Assessment Schedule

- This module is 60% exam and 40% coursework
- The coursework is composed of practical lab exercises which build into an increasingly large system by the end of the term
- There are three stages, with deadlines at:
  - Friday **Week 3**, submit on Moodle
  - Friday **Week 6**, submit on Moodle
  - Friday **Week 9**, submit on Moodle

Assessment Approach

- We use a combination of **automated marking** and source code inspection to mark your work

- For each coursework stage we have a test system to which you can upload your solution for compatibility testing; this tells you whether or not your work **can be marked** by our automated marking system

- You **must** then submit your work to Moodle by the deadline

Course textbooks

- **Distributed Systems. George Coulouris et al. (CKDB)**
  - Link to the digital version of the book is on the course Moodle page

- **Distributed Systems. Andrew Tanenbaum and Maarten Van Steen (TvS)**
  - You can get a free digital version (2023) from: https://www.distributed-systems.net/index.php/books/ds4/

Distributed Systems: An Introduction

Distributed Systems: An Introduction

- What is a distributed system?
- Why distributed systems?
- Major Challenges
- System Models
- Classical System Models
- Middleware

What is a distributed system?

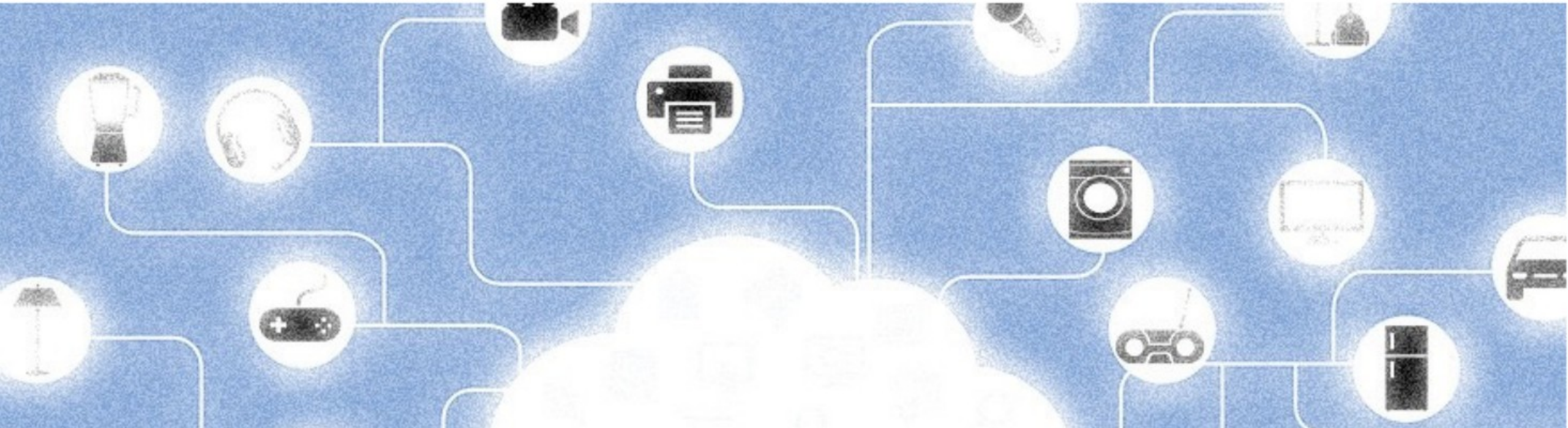
“a collection of independent computers that appears to its users as a single coherent system” [Tanenbaum and van Steen]

“one in which hardware or software components located at networked computers communicate and coordinate their actions only by passing messages” [Coulouris, Dollimore, Kindberg]

“one that stops you getting work done when a machine you’ve never even heard of crashes” [Lamport]

What is a distributed system?

- In its broadest definition, a distributed system is one that comprises more than one computer with the goal of reaching a level of performance and/or providing a service that is quite difficult or infeasible to do on a single computer.



Examples of Distributed Systems

- World Wide Web (WWW)
  - Client/Server system that uses HTTP protocol to access documents and other resources from servers
  - Over 4 billion web pages spread across around 8.2 million web servers
- Web search engines
  - Modern web engines serve over 3.5 billion searches per day!
  - Crawling, indexing dynamic web pages and timely respond to queries using a massively parallel computing infrastructure
- Blockchain Systems
  - A peer-to-peer system with tens of thousands of peers, each maintaining a copy of a distributed ledger (i.e., blockchain)
  - A new block containing transactions broadcast to the network and added to the ledger every ~12 seconds

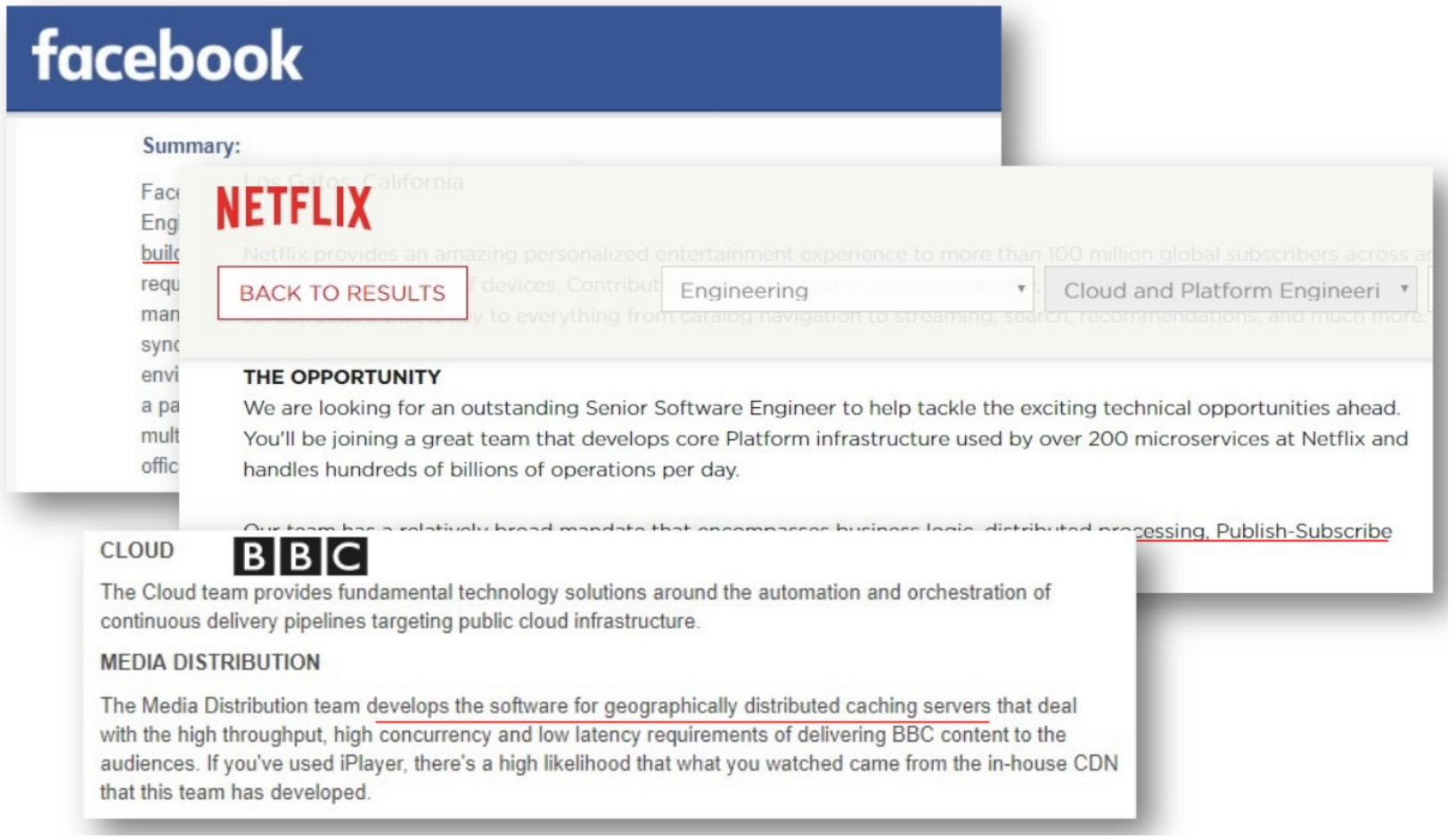
Context

- You have seen multiple processes or threads running on the same computer (SCC.211 Operating Systems)
  - These threads/processes have shared memory
- In this course, we will take things further...
  - Examine systems consisting of multiple communicating computers
- We will often refer to a computer in a distributed system as a **node**
  - A node can be a desktop computer, a mobile device, a server in a data center, a sensor or an embedded device, and so on
  - A node can be any type of computing device with a network interface

Why Distributed Systems?

- Because the world is distributed
  - You want to book a hotel in Sydney, but you are in Lancaster
  - You want to be able to retrieve money from any ATM in the world, but your bank is in London
  - An airplane has 1 cockpit, 2 wings, 4 engines, 10k sensors, etc.
    - Similarly railway networks, and other distributed transport systems
- Because problems rarely hit two different places at the same time
  - As a company having only one database server is a bad idea
  - Having two in the same room is better, but still risky
- Because joining forces increases performance, availability, etc.
  - High Performance Computing, replicated web servers, etc.
  - “Resource” sharing – a resource is anything that can be shared in a networked computer system

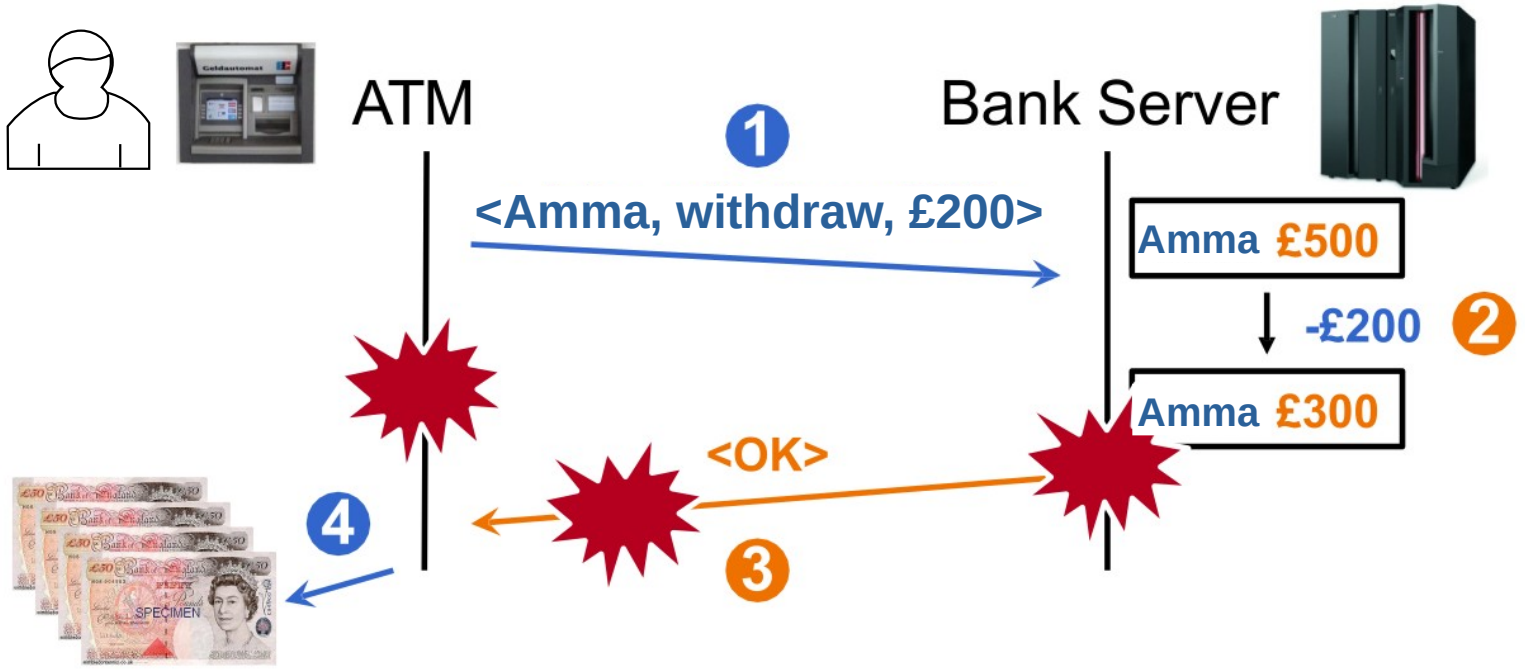
Is this important?



Why is it so hard?

- A bank asks you to program their new ATM software
  - Central bank computer (server) stores account information
  - Remote ATMs authenticate customers and deliver money
- A first version of the program
  - **ATM**: (ignoring authentication and security issues)
  - 1. Ask customer how much money s/he wants
  - 2. Send message with **<customer ID, withdraw, amount>** to bank server
  - 3. Wait for bank server answer: **<OK>** or **<refused>**
  - 4. If **<OK>** give money to customer, else display error message
  - **Central Serve**r:
  - 1. Wait for messages from ATM: **<customer ID, withdraw, amount>**
  - 2. If enough money, withdraw money and send **<OK>**, else send **<refused>**

Why is it so hard?



- But …
  - What if the bank's central server crashes just after 2 and before 3?
  - What if the <OK> message gets lost? Takes too long to arrive?
  - What if the ATM crashes after 1, but before 4?

Challenges (I)

- In a standalone computer, if a component fails (like a RAM module develops an issue), we typically expect the computer to entirely stop working, rather than partly continue functioning.
  - Software doesn't usually need to be explicitly designed to handle a faulty RAM chip.
- However, in a distributed system, it is highly desirable to tolerate certain parts of the system stop working (e.g., crashing), while the system continues to operate.
  - For instance, if one node experiences a crash, the remaining nodes should still be capable of providing the service.
- When a component of a system stops working, it's referred to as a fault. Many distributed systems aim to achieve fault tolerance, meaning the system continues to operate as a whole, despite the fault.
  - Managing faults is what sets apart distributed systems, which can often be more challenging, from programming a single computer.

Challenges (II)

- Nodes can coordinate/communicate their actions ONLY by sending messages over a network
- No global clock! Close coordination depends on the shared idea of time
- The failure of a computer (or crashing of a program running at a node) is not immediately known to other nodes
  - Each component can fail independently!
  - Detection of failure is not straightforward when networks themselves can fail (i.e., lose or excessively delay packets)

System Model

- Description of assumptions that we make when designing an algorithm/software to run in a distributed system
- As we discussed, things can go wrong in a distributed system:
  - Nodes crash
  - Networks fail
- The designers of distributed algorithms must be precise on the kind of failures that are both possible and not possible (i.e., ruled out)

Example System Models

- Node failure modes
  - **Crash-stop** (fail-stop)
    - Node/component can unpredictably fail due to hardware/software fault and stops responding
  - **Crash-recovery** (fail-recovery)
    - Node/component experiences a crash can subsequently recover and resume its operation after the failure has occurred
  - **Byzantine** (fail-arbitrarily)
    - Node/component can exhibit arbitrary, malicious, or unexpected behaviors

Example System Models

- Timing and synchrony assumptions:
  - **Synchronous**: message delivery time < a known upper bound
    - Nodes' clocks are reasonably synchronized.
    - There is an upper bound on processing time for nodes.
  - **Asynchronous**: arbitrary delivery times
    - No guarantees on message delivery time or processing time, and nodes' clocks may drift.
  - **Partially Synchronous**:
    - Unpredictable transitions between synchronous and asynchronous periods
  - **Bounded asynchrony**: assume there is a known upper bound on how long it can take for messages to be delivered.

Example System Models

- Network failure modes:
  - **Reliable** (perfect) links:
    - A message is received if it is sent.
    - Messages may be reordered.
  - **Fair-loss** links:
    - Messages can be lost, duplicated, or reordered.
    - But, if you keep retrying, a message eventually gets through.
  - **Arbitrary** links:
    - An adversary may interfere with messages (eavesdrop, modify, drop, spoof, replay).

Example System Models

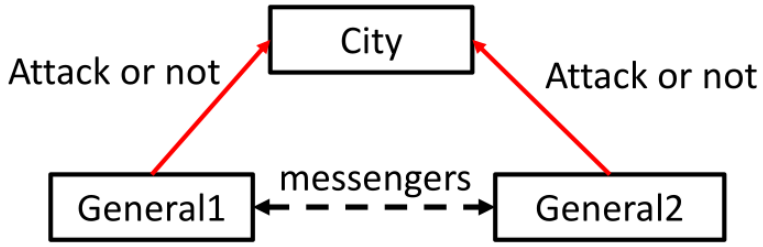
- Two classic thought experiments in distributed systems help to illustrate common system models:
  - The two generals problem
  - The Byzantine generals problem (in a later lecture)

The Two Generals Problem

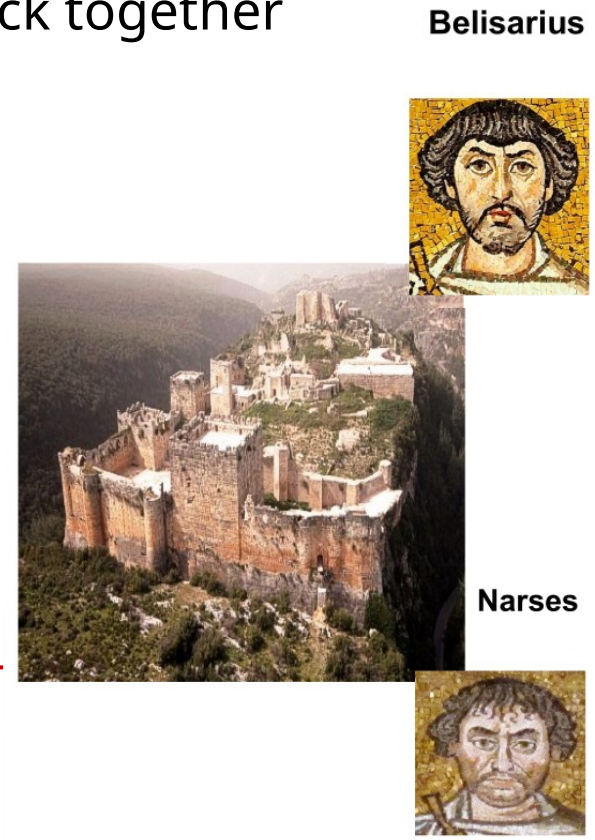
- A common complication in distributed consensus

  Two generals, each leading an army in different parts of the world, want to capture a city. The attack is only successful if both armies attack together

  - However, the communication is not reliable!

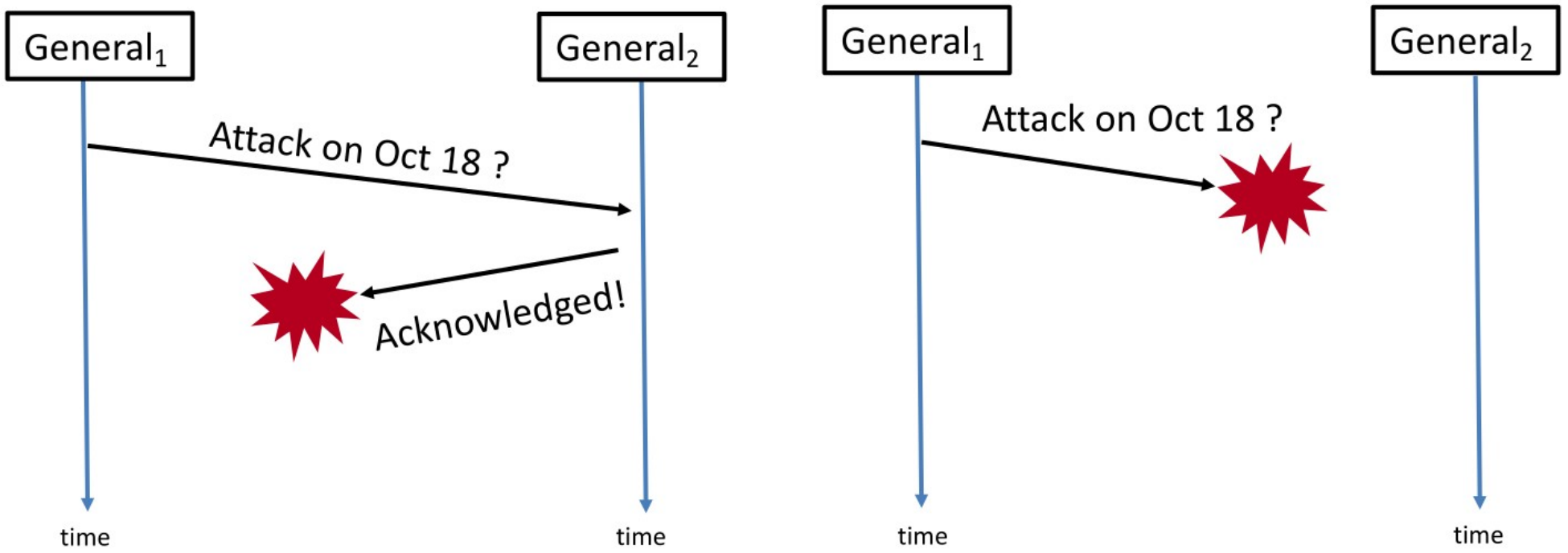


Army1	Army2	Result
No Attack	No attack	Nothing happens
No Attack	Attack	Army2 defeated
Attack	No attack	Army1 defeated
Attack	Attack	City captured



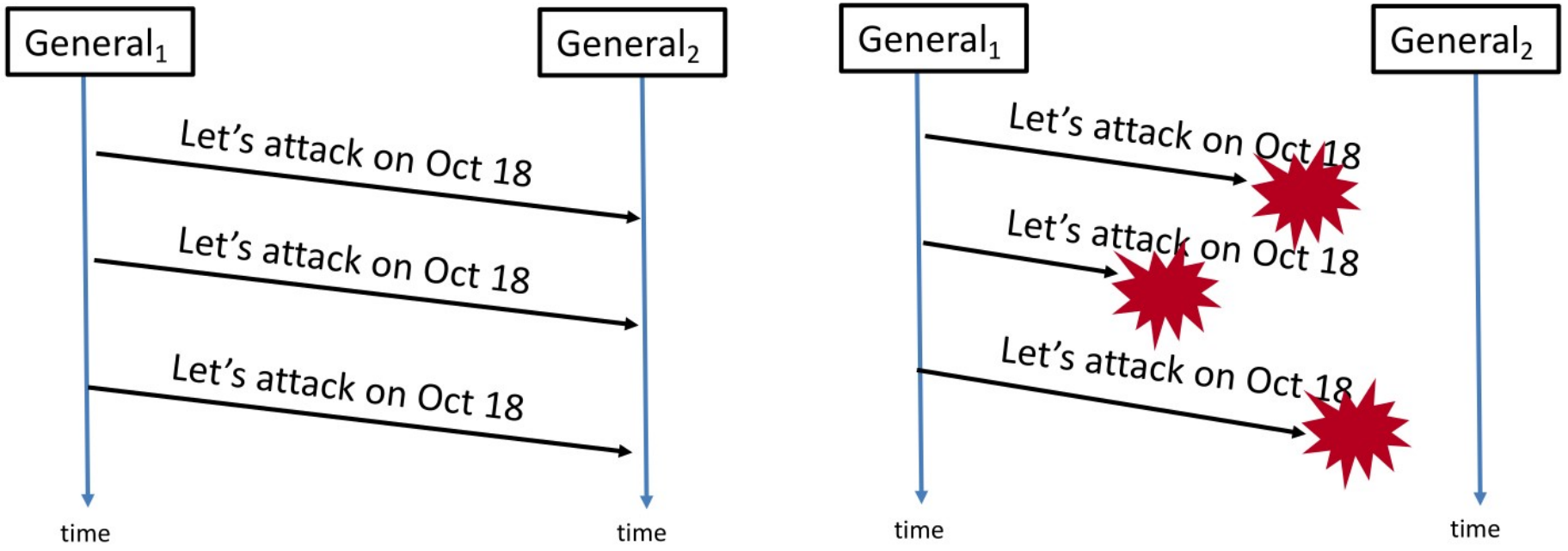
The Two Generals Problem

- Important challenges
  - Unreliable network
  - Receipt of Acknowledgement is necessary
  - Note: “Sequence diagrams” like the one below will be used often in the course



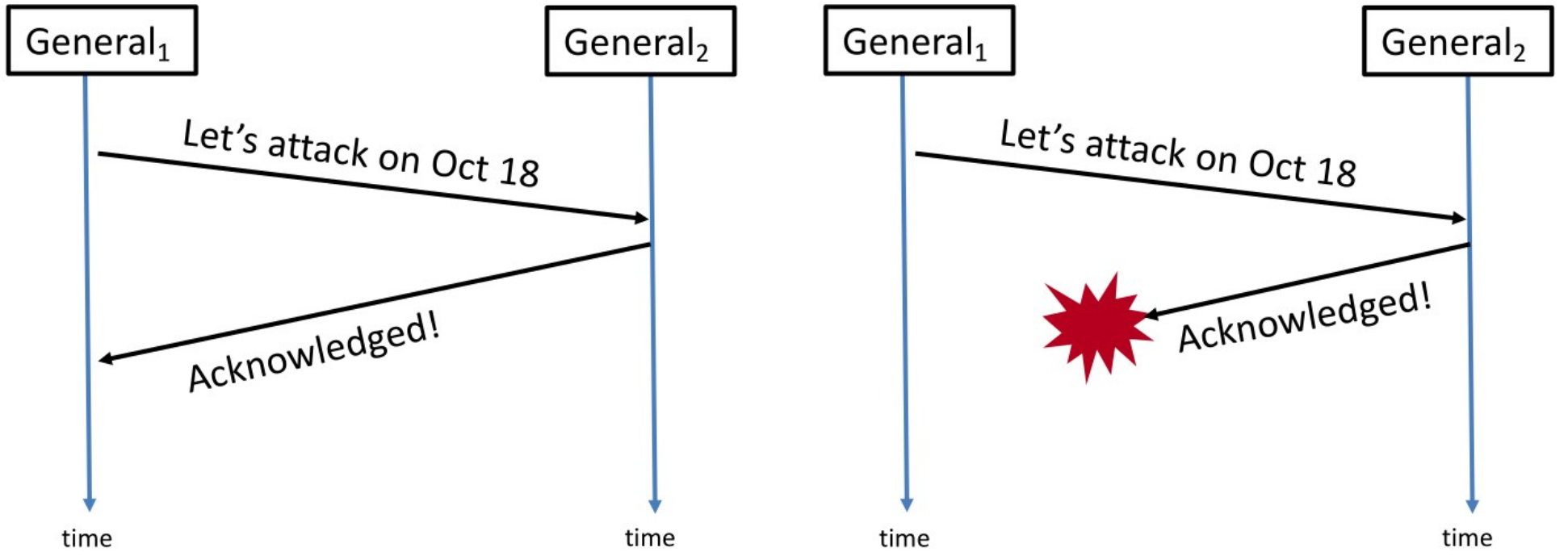
The Two Generals Problem: Approach 1

- General1 attacks at a proposed time **t**, after sending messenger(s) to General2 (who is expected to move at time **t**)
  - General1 sends *a lot* of messengers at once, and then attacks
  - Problem: what if *none* of the messages are delivered?



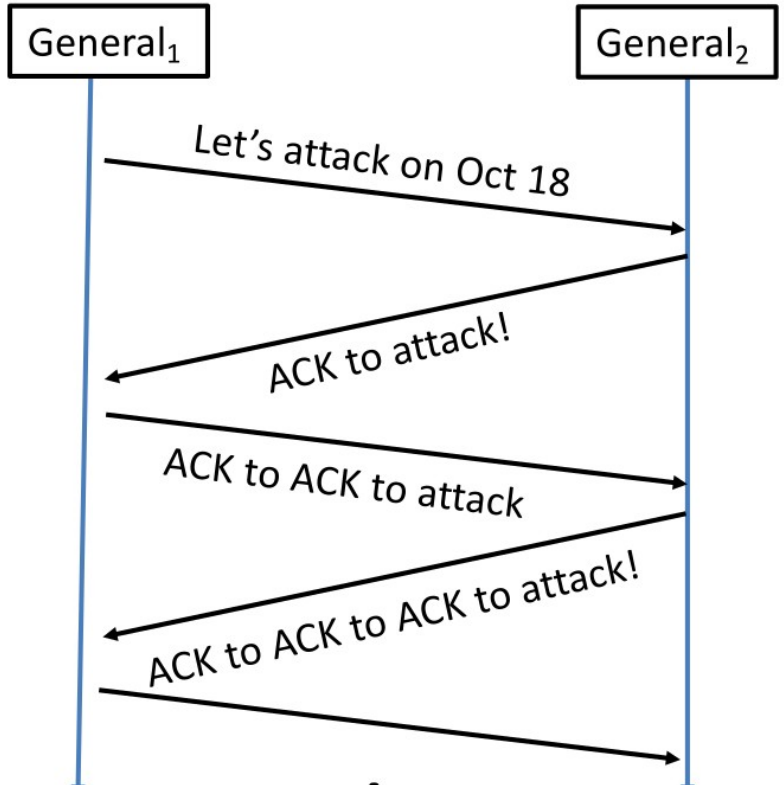
The Two Generals Problem: Approach 2

- General 1 attacks only when an ACK is received, and General 2 attacks after sending the ACK
  - Now General2 will fail in their attack, if ACK is intercepted or delayed



The Problem of Distributed Consensus

- Consensus means to agree on something, such that everyone knows it has been agreed. One node (General1) knows something and the rest of the nodes (General2 and others) know that General1 knows it.
- For General1 and General2 to reach agreement
  - We need an infinite series of messages
  - It's impossible to reach complete certainty



FLP Impossibility Theorem

- FLP impossibility:
  - Fischer, Lynch, and Patterson, Impossibility of Distributed Consensus with One Faulty Process, 1985
- With no upper bound on the time a process takes to complete its work and respond, it's impossible to make the distinction between a process that is crashed and one that is working (but taking very long to respond).
- FLP shows that there is no guarantee for distributed processes to reach consensus in an asynchronous environment, where it's possible for at least one process to crash.
- Equivalently, it's not always possible to detect failure in an asynchronous system with nodes crashing.

Further Complications

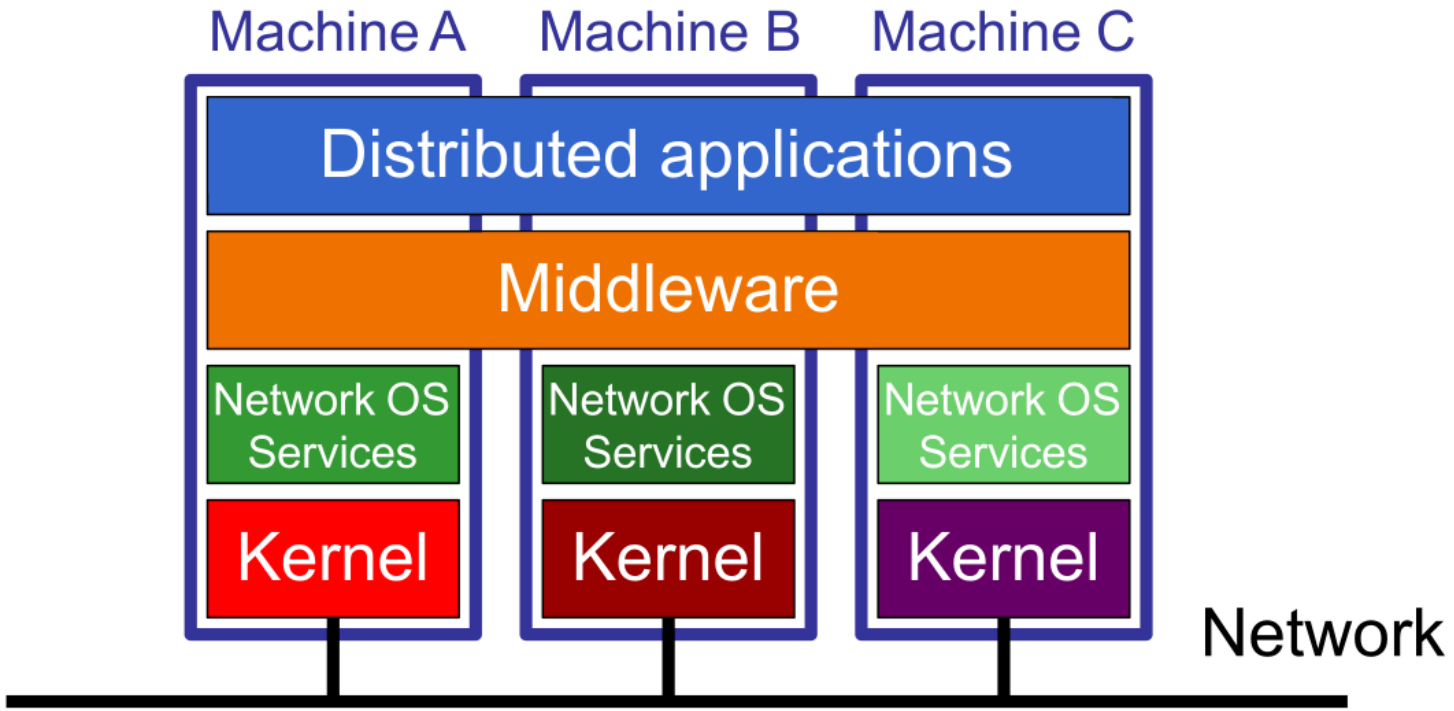
- Heterogeneity
  - Access
  - Platform
  - Format
  - Administration
- Changing nature
  - Increase/decrease in scale
  - Churn
  - Relocation
  - Failure

This is going to be a challenge...

- Developing good software is difficult
- Developing good distributed software is even harder
- To do this you need help!
  - Very hard to build such systems on bare-bones devices
  - Strong need for software platforms -> Middleware

Middleware

- Rationale
  - Provide a high-level programming abstraction
  - Hide the complexity associated with distributed systems (including the underlying forms of heterogeneity)



Goals of a Middleware Platform

- Resource sharing
  - The ability to access and share resources in a distributed environment
  - Distributed locking
- Transparency
  - The ability to view a distributed system as if it were a single computer
  - Varying dimensions of transparency incl. location, access, migration, failure etc.
- Openness
  - The offering of services according to standard rules (syntax and semantics)
  - Openness provides support for the key properties of:
    - Portability: ability to run across different platforms
    - Interoperability: having separate parts harmoniously work together
- Extensibility
  - The ability to be able to introduce new or modified functionality

Goals of a Middleware Platform

- Scalability: the ability to grow/shrink with respect to
  - Size
    - e.g. support massive growth in the number of users of a service
  - Geography
    - e.g. supporting systems across continents (dealing with latencies, etc.)
  - Administration
    - e.g. supporting systems spanning many different administrative organisations
- Dependability/ Quality of Service
  - Security
    - Providing secure and authenticated channels, access control, key management, etc.
  - Fault tolerance
    - Providing highly available and resilient distributed applications and services

Learning Outcomes

- At the end of this session, you should be able to:
  - Define distributed computer systems
  - Explain why distribution is needed
  - Know a few everyday examples of distributed systems
  - Understand some of the challenges of building algorithms/software for distributed systems

Further Reading

- **CDKB**, chapter 1 sections 1.1 -- 1.4
  - also chapter 3 for revision
- **TvS**, chapter 1
- **Computer Systems: A Programmer's Perspective**, by Bryant and O'Hallaron, chapter 1
- Mark Cavage, “**There's Just No Getting around It: You're Building a Distributed System**”, ACM Queue, Vol. 11 No. 4, April 2013. http://queue.acm.org/detail.cfm?id=2482856