COMP90015 Distributed Systems

Introduction

Aaron Harwood

School of Computing and Information Systems

2022 Semester II

Staff

- Aaron Harwood, Lecturer and Subject Coordinator, aharwood@unimelb.edu.au
- Mohammad Goudarzi, Head Tutor
- Jonathan El-Khoury, Tutor
- Tharindu Bandara, Tutor
- Yiwen Zeng, Tutor

Assessment

Projects will be group work with groups of size 2. You may work alone if you wish however you must complete the entire project. The programming language is Java (11). Please start considering your partner now. Project 1 details will be provided in Week 4.

1. Project 1, 20%, software and written report, starting around Week 4 and due around Week 8
2. Project 2, 20%, software and written report, starting around Week 8 and due around Week 12
3. Final Exam, 60%, online

Academic Integrity

All University of Melbourne students are expected to uphold academic integrity in all aspects of every piece of work that they submit for assessment.

- `https://academicintegrity.unimelb.edu.au/`

Traditional Course Overview

This course was originally developed from Coulouris, Dollimore and Kindberg, *Distributed Systems: Concepts and Design*, Edition 5, ©Addison-Wesley 2012, with emphasis on the following chapters:

- Chapter 1 Characterization of Distributed Systems
- Chapter 2 System Models
- Chapter 4 Interprocess Communication
- Chapter 5 Remote Invocation
- Chapter 6 Indirect Communication
- Chapter 7 Operating System Support
- Chapter 11 Security
- Chapter 12 Distributed File Systems
- Chapter 13 Name Services

2021 Semester 2 Course Overview

A tentative plan for the semester:

Week	Date	Lecture	Tutorial
1	25th Jul.	Motivation, Challenges	
2	1st Aug.	IPC, Data Representation	Tutorial 1
3	8th Aug.	Architectures, Fundamental Models	Tutorial 2
4	15th Aug.	Synchronous/Asynchronous Protocols	Tutorial 3
5	22nd Aug.	Middleware, RPC, RMI	Tutorial 4
6	29th Aug.	Indirect Communication Paradigms	Tutorial 5
7	5th Sep.	Encrypting Communication	Tutorial 6
8	12th Sep.	Digital Signatures, Certificates, SSL/TLS	Tutorial 7
9	19th Sep.	Distributed File Systems	Tutorial 8
-	26th Sep.	Non-teaching Week	
10	3rd Oct.	Name Services	Tutorial 9
11	10th Oct.	TBA	Tutorial 10
12	17th Oct.	TBA	Tutorial 11

Learning Outcomes

- Familiarity with distributed system terminology and fundamental concepts
- Develop critical thinking, reasoning, modeling and analysis techniques
- Know a range of solutions to essential distributed system challenges
- Understand high-level distributed system paradigms and demonstrate their appropriate application
- Develop skills in distributed system design and programming implementation
- Develop skills in small group work and written communication

Computer Hardware

In this subject we are mainly interested in distributed systems based on *computer systems*, with a broad applicability:

- Machine/Host/Device/Node: terminology used somewhat synonomously depending on context; "machine room", "server host", "edge device", "cluster node", ... even we use "thing" as in "Internet of Things"
- Physical location: typically stationary (desktop) or mobile (phone or tablet), but technically not much difference – we can move our desktop if we want.
- Hardware devices/specifications:
  - CPUs/cores (clock rate, cache), RAM (capacity, latency)
  - Connectivity: ethernet, wireless, cellular, bluetooth, USB (bitrate/bandwidth)
  - Storage: SSD, HDD (capacity, latency, throughput, read/write iops)
  - Graphics: GPU (resolution, refresh rate)
  - Peripheral devices: keyboard, monitor, mouse, printer, webcam, microphone, etc

Computer Software

We are mainly interested in the higher layers of software, from the Application/User software down to the Operating System.

- Application/User software
- Middleware/Libraries – *Application Programming Interfaces* (APIs)
- Operating System (OS):
  - Kernel (monolithic/micro kernel)
  - Device drivers
  - Modules/services
- Hypervisor (for virtualization)

- Bootloaders, EFI (Extensible Firmware Interface) BIOS (Basic Input Output System)
- ROM (Read Only Memory) and firmware, POST (Power On Self Test)

Platform

A platform is a *layer* of functionality or API across a number of machines, by de facto being a consistent hardware and OS specification combined, but sometimes being a *middleware* layer above the OS primarily to hide differences in OS and hardware on different machines.

- POSIX or Portable Operating System Interface is the standard OS API adopted by most Unix-based OSes. It was defined in 1988 by the IEEE Computer Society.
- The macOS from Apple Inc. adopted POSIX around 2007 when their OS became UNIX-based.

- The Linux OS, Android and a range of other popular OSes for devices such as routers are "mostly" POSIX-compliant.
- Cygwin/MinGW provide a mostly POSIX-compliant development and runtime environment for Microsoft Windows. Conversely, Wine is a compatibility layer for UNIX-like operating systems to run programs written for Microsoft Windows.

- The Windows API or WinAPI is the API for the Microsoft Windows OS, from Microsoft Corporation.
- The Java Virtual Machine executes programs, usually written in Java, which compile to Java bytecode. The JVM provides a consistent API across many different Hardware/OS platforms, which is one aspect which lead to the popularity of the Java programming language.

Virtualization and Emulation

Even commodity OS and computer hardware provides feature rich virtualization and emulation support that can be of interest when studying distributed systems. However the details are often quite technical and OS specific.

- Virtualization: managing access to existing physical devices such that there appears to be more of them or greater capacity from a software perspective (i.e. logically), essentially time sharing.
  - Virtual memory: paging to disk
  - Virtual Cores: time sharing physical cores
  - OS Virtualization: Jails, Containers, Zones
  - Virtual Machine (VM): all devices are virtualized.
    - Hypervisors
    - run an OS in the VM

- Emulation: create the appearance of a device existing, even though it does not:
  - Mobile device emulation for testing and debugging on hardware that is not physically available
  - Network device emulation: switches and routers (Mininet)
  - OS emulation

Computer Networks I

Besides the computer system, the computer network is an essential aspect for our study of distributed systems:

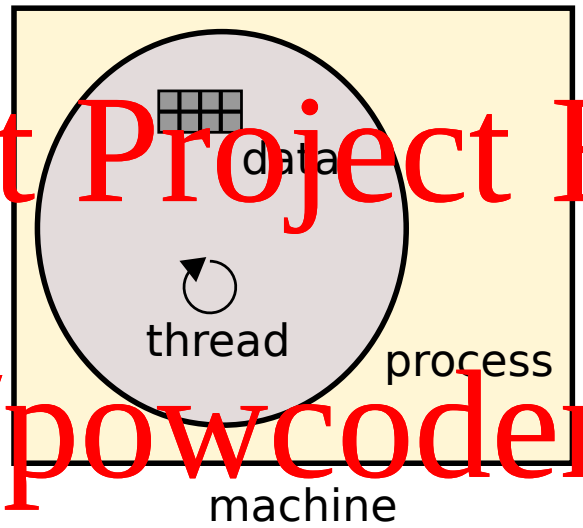
- We are almost exclusively interested in networks that support the TCP/UDP/IP protocol stack, although sometimes we use other network protocols specific to a given technology:
  - Bluetooth: largely for removing peripheral cables, Service Discovery Protocol (SDP)
  - Cellular Networks: mixing traditional telecommunications functionality like SMS with Internet-based applications
- Relevant networking devices include:
  - Wireless Base Stations: provide a way to connect to the network using radio waves anywhere from 900MHz to 60GHz, sometimes called WLAN or WiFi.
  - Switches and Routers: provide a way to connect to the network using cables, e.g. Ethernet at least 1Gbps but also at 10, 25 and 40Gbps
- The well known *Internet*, arising in the 1980's, has become the dominant global computer network and communications platform:
  - Internet Service Providers (ISPs) provide access which among other things provides one or more unique *Internet Addresses*:
    - represent end-points on the *public* Internet,
    - devices with public Internet addresses can be directly communicated with by every other device connected to the Internet.

Computer Networks II

- No one country owns or controls the entire Internet, although some governing bodies such as ICANN (The Internet Corporation for Assigned Names and Numbers) have traditionally controlled some essential aspects, namely the Internet Addresses and Internet Domain Names.
- There are lessor known alternatives arising to ICANN's governance, e.g. Yeti DNS (Chinese government sponsored) and National DNS (Russian government sponsored).

- Organizations can build *private* networks that operate in the same way as the Internet but that are entirely owned and governed by the organization:
  - Private set of network addresses that are unique only to the organization's private network.
  - Local Area Networks (LANs): the smallest unit of network management where typically all of the machines on the LAN are subject to the same network policies.
  - Wide Area Networks (WANs): multiple LANs connected together, providing for differing network policies across an organization.
  - Network Address Translation (NAT): provide an ability for a private network to connect to the public Internet via a machine (typically a router) that has both a public Internet Address and a private network address.
  - Virtual Private Network (VPN): connecting two or more private networks via a secure connection over the public Internet .

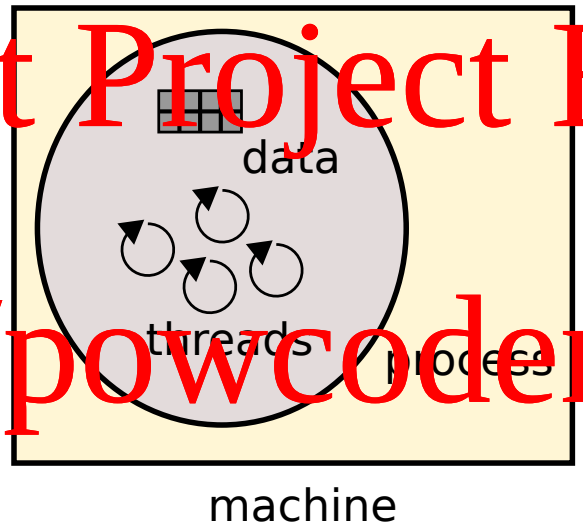
Single process, single thread



- Smallest OS encapsulation:
  - Every application requires at least one process
  - Primarily encapsulates processor and memory resources
  - System calls to access devices on the machine
- One machine
- Single thread:
  - one core
  - one address space
  - no concurrency control

Not a distributed system.

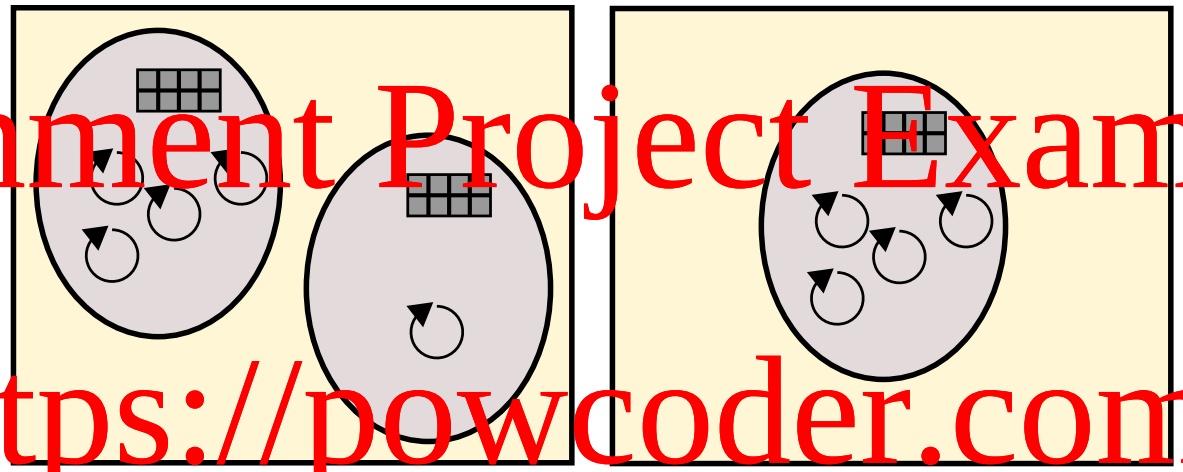
Single process, multiple threads



- One machine, one process
- Multiple threads:
  - multiple cores
  - one address space shared among threads
  - concurrency control required

Not a distributed system – limited to a single machine.

Multiple processes, single/multi-threaded



- One or more machines – each process can run on a different machine
- Each process has its own address space
- *Inter-process communication* (IPC) is required
  - Shared memory, pipes and file-based communication if processes on the same machine
  - TCP/UDP/IP for processes on the same or different machines
- Each process may have one or more threads:
  - One or more cores per process
  - Concurrency control within processes and across processes

Now do we have a distributed system?

What is a Distributed System?

Distributed System definitions – many and varying:

- "A system in which hardware or software components located at networked computers communicate and coordinate their actions only by passing messages" [Coulouris]
- "A collection of independent computers that appears to its users as a single coherent system" [Tanenbaum]

Key aspects:

- a number of components
- communication between the components (implied?)
- synergy; achieve more than the simple sum of individual components

Discussion questions

Question (1): If we have a number of floor cleaning robots, *that never communicate*, but that independently go through a process of cleaning the same floor at the same time (with some smarts like not bumping in to each other, and detecting if some part of the floor needs to be cleaned or can be skipped over), in such a way that they end up collectively cleaning the floor faster than a single independent robot could do so, is this an example of a distributed system? Why or why not?

Question (2): Multiple threads can be seen as working together to solve a problem. So why don't we call a multi-threaded, single process application as a distributed system?

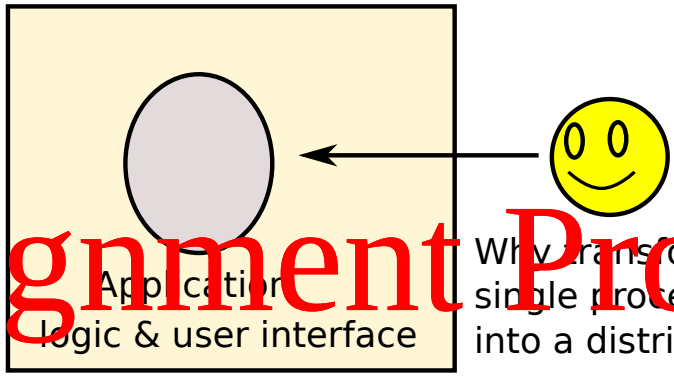
Why would we build a Distributed System? I

Lots of reasons, many of which you would take for granted in today's world:

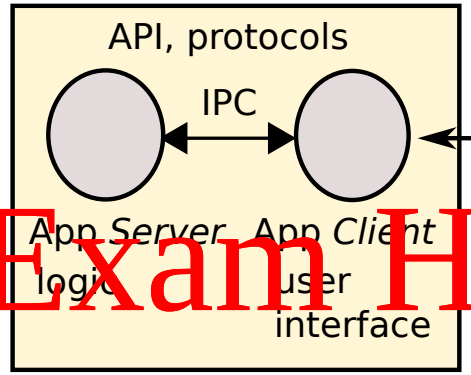
- Communication:
  - users at different computers can communicate with each other
  - multiple users can communicate with each other concurrently: online chat, video conference
- Remote access:
  - user does not need to travel to a given computer to use it, but can access it remotely
  - can use resources on the other side of the Earth, can control a rover on Mars (if it lands successfully)
- Resource sharing:
  - single resource can be shared among multiple users: significant cost saving and increased utilization

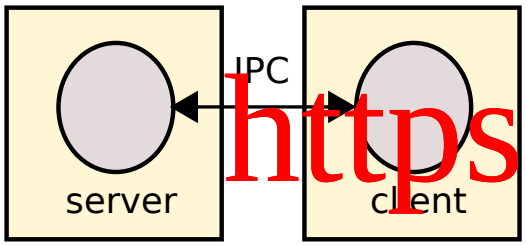
Why would we build a Distributed System? II

- Reliability:
  - increased mean-time-to-failure, redundancy in resources allows the system to continue to operate in the presence of hardware and software failures
- Availability:
  - less downtime, e.g. when upgrading some hardware the system can stay online on remaining hardware
- Scalability:
  - using more resources to provide greater capacity than any single resource can
  - ideally having N resources provides N times the capacity of a single resource



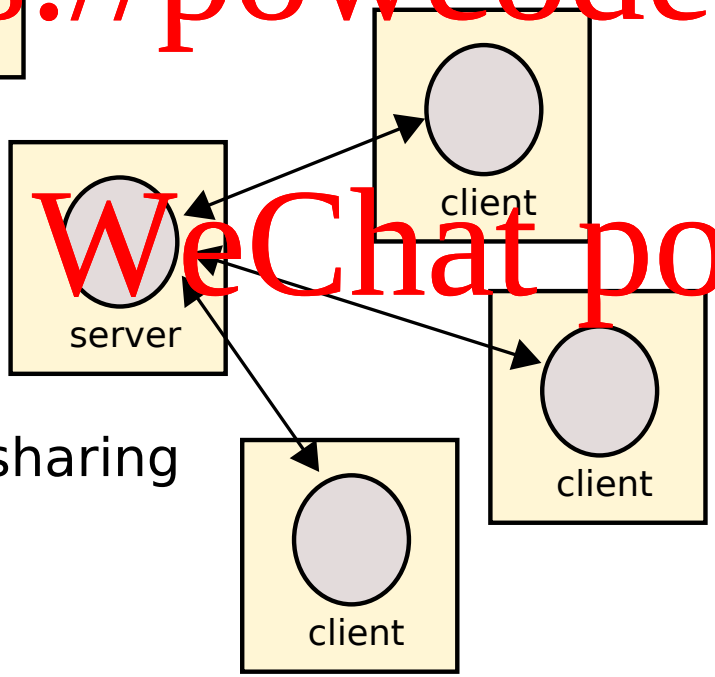
Why transform an existing single process application into a distributed application?



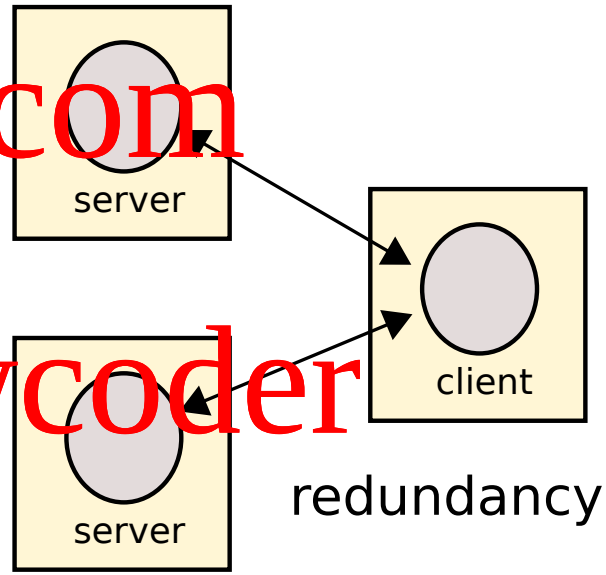


remote access

what benefits do you see?



sharing



redundancy

Consequences of Distributed Systems I

The benefits that distributed systems bring are not free:

- *Communication complexity*. Communication is not free, and is largely an overhead. Processes may waste time waiting for communication to take place. Communication requires additional power consumption. Communication errors can occur, sometimes causing communication to be impossible for significant periods of time.
- *Concurrency*. In a Distributed System computers perform their tasks autonomously and communicate with other computers via message passing when needed. Services provided by distributed systems will be accessed by multiple users simultaneously. Distributed system design should take this into consideration and implement appropriate techniques for handling concurrency.

Consequences of Distributed Systems II

- *No global clock*. Clocks on individual computers operate independently. Since computers communicate by message passing there are limits to the accuracy with which the computers in a network can synchronize their clocks. Therefore, distributed systems have to deal with the fact of not having a global clock and implement time synchronization techniques when needed.
- *Independent failures*. Some components of the system may fail while the others are still running. Failures of participating computers of the system is not known to others immediately.

Interprocess Communication

APIs, protocols and semantics

IPC significantly increases the design and implementation effort over a single process application. IPC generally requires:

- an API which defines the available operations supported by the communicating processes, e.g. a distributed Calendar App:
  - `GetRemoteCalendar(address:InetAddress) : RemoteCalendarRef`
  - `RetrieveCalendarData(cal:RemoteCalendarRef, date:Date) : Event[]`
  - `StoreCalendarData(cal:RemoteCalendarRef, event:Event)`
- *communication semantics* that define what the programmer can expect, e.g.:
  - "all API calls block until the operation has successfully completed exactly once, or else an exception is thrown in which case the operation outcome is undefined"
- a *communication protocol* which defines how the processes will initiate communication and exchange data to implement the API and guarantee the semantics, e.g.:
  - Transmission Control Protocol (TCP) connections and JSON data objects

Distributed system challenges I

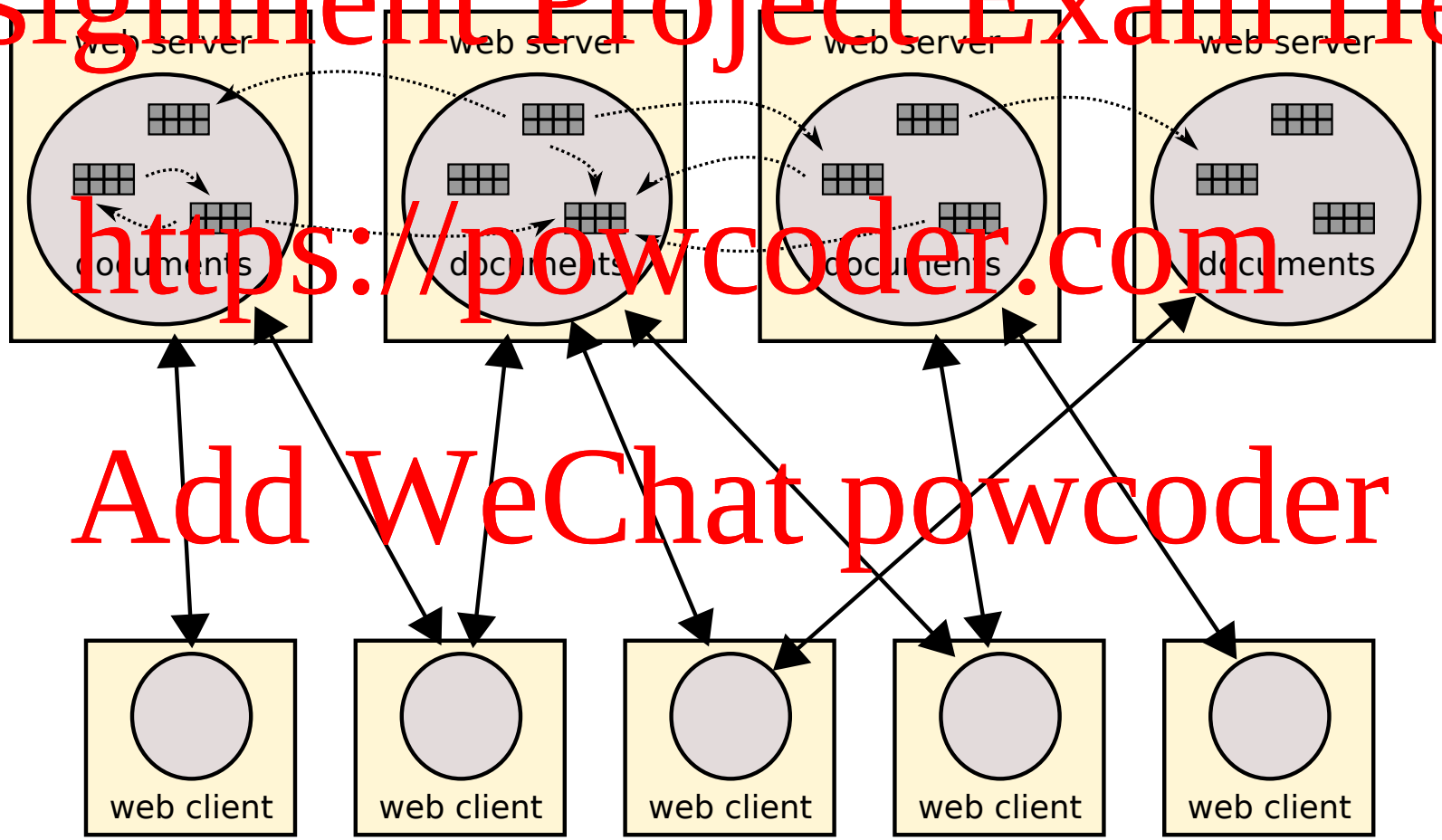
- Heterogeneity – the parts of the system are not consistent (homogenous)
  - Hardware, OS and networking can differ across different components.
  - Processes can be written in different languages, and by different developers.
- Openness – the system can be built upon or accessed by third-party developers, through public APIs and protocols
  - When APIs and protocols need to change they can break third-party systems.
  - APIs should be simple to use, provide all the required features.
- Security - confidentiality, integrity and availability
  - Communication usually takes place over third-party (untrusted) networks which is a significant security risk.
  - System components may become controlled or corrupted by attackers.
  - System components may become targets of denial-of-service attacks.
  - Systems that execute third-party (untrusted) code/scripts are especially high risk.
- Scalability - increasing the number of system components increases the overheads
  - For N system components, overheads may grow geometrically, e.g. proportional to N^2.
  - For a distributed system operation to be scalable we would like the operation's overheads to grow no faster than $\log N$.
  - Avoiding all bottlenecks in a design is non-trivial.

Distributed system challenges II

- Failure Handling - system components fail independently and the communication network can fail as well
  - How can we determine whether a remote component has failed or is simply taking a longer time than expected to respond?
  - How do we know if data has been lost or if some operations were completed or partly completed or not completed at all?
  - What do we do?
    - Try to hide the error e.g. by retrying: *error masking*
    - Ask the higher layer (up to the user) to *tolerate* the error: "please try again later"
    - Try and recover from the error e.g. when data is lost or operations are partially completed: *error recovery*
- Concurrency - multiple processes lead to a higher level of concurrency control requirements
  - data needs to be consistent across the system
  - outcomes need to be deterministic
- Transparency - hiding aspects of the system from the high layers of application
  - Access transparency - APIs for accessing remote resources
  - Location transparency - location of data and resources
  - Concurrency transparency - consistency of data
  - Replication transparency - replication of data and operations
  - Failure transparency - system components and network communication can fail
  - Mobility transparency - system components change their IP address
  - Performance transparency - overheads such as communication delays
  - Scaling transparency - increasing the number of resources

World Wide Web

The World Wide Web is based on Web Servers and Web Clients with documents that embed the location of other documents.



Discussion questions

The WWW began as a simple system for exchanging documents at the European centre for nuclear research (CERN), Switzerland, in 1989. Documents are logically organized using a *hypertext* structure and are connected through *links*. The three main standard technological components used in the web are:

- HypterText Markup Language (HTML) - This is a language for specifying the contents and layout of pages to be displayed by browsers.
- Uniform Resource Locators (URLs) - These identify the resources stored on the web.
- HyperText Transfer Protocol (HTTP) - This is the protocol used for transferring resources between web servers and clients (e.g. browsers).

Question (3): Which distributed system challenges does the WWW address well and which ones does it not address well? Explain your reasoning. **Question (4):** What other aspects of the WWW do you think has lead to its immense success in becomming a defacto platform for most of our distributed applications today?

Discussion questions

Question (5): Give your own examples of distributed applications that you commonly make use of (e.g. social media services, massively multi-player online games, streaming TV services, home automation, etc.) and discuss the most important distributed system challenges that you think these applications face, based on your own experiences using them; try to relate your discussion to concepts given in this lecture.

Summary

- This subject is concerned with computer systems and computer networks as the underlying hardware of our distributed systems
- Distributed systems provide enormous benefits that today are mostly taken for granted
- Communication and resource sharing is a significant motivation for distributed systems.
- There are many challenges associated with the design and implementation of effective distributed systems.