Course Introduction

COMP 30220: Distributed Systems

Lecturer: Rem Collier

Email: rem.collier@ucd.ie

WHAT IS THIS COURSE ABOUT?

- This course will explore some of the core practical concepts that underpin **distributed systems.**
  - Basic Networking (Sockets)
  - Distributed Objects (RMI)
  - Web Services / SOAP & REST
  - Message-Oriented Middleware / Active MQ + JMS

- We will also try to touch on some "bigger" issues:
  - Enterprise Service Bus / Choreography & Orchestration
  - Microservices / Service Mashups
  - Reactive Systems / Actor-based Systems

Course Assessment

- This course is 100% assessment:
  - Paper Review 20%
    Read an assigned papers and discuss in a reading group. Submit individual reports.
  - Group Presentation 20%
    Reading Groups come together to present assigned papers.
  - Practical Work 20%
    Preparatory programming work.
  - Group Project 40%
    Large Student-Driven Project that uses techniques from module.
- Labs will start in week 2, papers and reading groups will be assigned in week 3.

Course Schedule

Week	Lectures	Labs	Notes
1	Introduction	n/a	
2	Network Programming	Sockets	
3	Distributed Objects	Java RMI	Paper Assignment
4	Web Services / SOAP	Jax-WS	
5	Web Services / REST	Restlets	
6	Message-Oriented Middleware	READING GROUPS	Report Submission
7	Actors	JMS + Apache MQ	Proposal Submission
8	4x PRESENTATIONS	4x PRESENTATIONS	
9	4x PRESENTATIONS	4x PRESENTATIONS	
10		PROJECT SUPPORT	
11		PROJECT SUPPORT	
12		PROJECT SUPPORT	
13	--READING WEEK--		Project Submission

Example Projects

- **Spy Polite**: proxy system for Facebook to allow secure sharing of users credentials / accessing photos of friends of friends.
- **Rotten Tomatoes:** Crowdsourced YouTube service based on REST + GraphQL.
- **Spaceship Tamogatchi**: REST-based tamogatchi space game.
- **Games4gamers**: JMS-based game server & client
- **Distributed Betting Service**: REST-based mashup of betting sites to provide single point of reference.
- **Billy Rubin & the Jets**: JMS-based Distributed File Service

Access to Course Material

- http://csmoodle.ucd.ie/
- Login using UCD credentials
- COMP30220 Distributed Systems 2018-19
- Enrolment Key: “ds18”

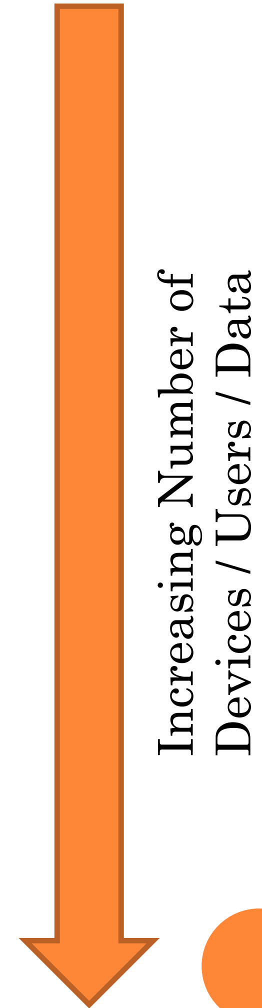
Introduction

A Short History of “Distributed” Computing

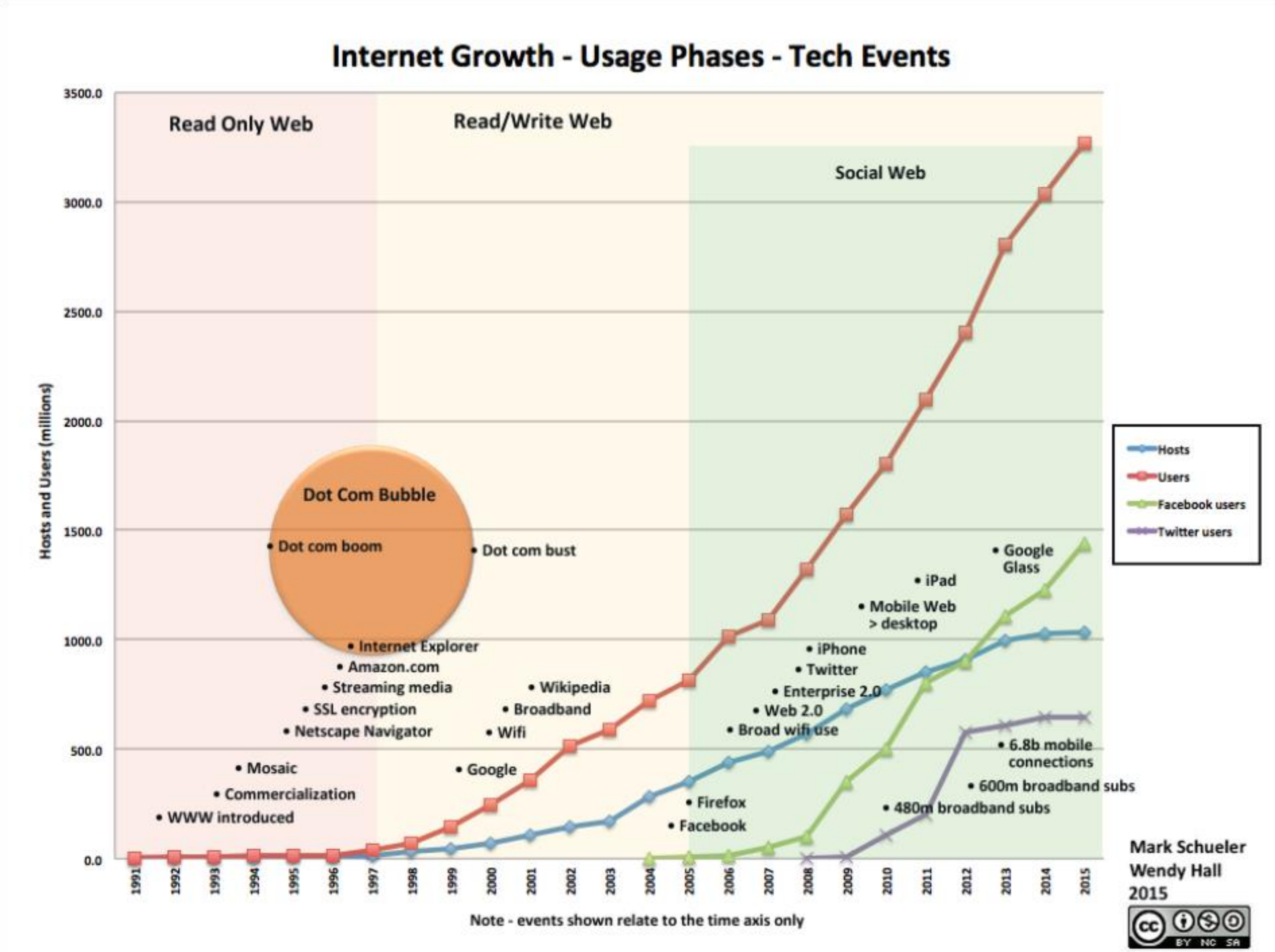
- Humble Beginnings:
  - Bankomat ATM (1968)
  - ARPANET (1969) - File Transfer / Email
  - **Remote Procedure Calls (1970s)**
  - The Internet (1982) - TCP/IP
  - Unix NFS (1980-) / Andrew File System (1982-)
  - The World Wide Web (1991) - URI’s, HTML, HTTP
  - **CORBA: Common Object Request Broker Architecture (1991)**

- The Rise of the Internet:
  - Web Browsers: Mosaic (1993), Netscape (1994), IE (1996)
  - E-commerce: Amazon (1994), ebay (1995), PayPal (1998)
  - Search Engines: Excite (1993), Yahoo (1994), Google (1998)
  - **IIOP (1996), Java RMI (1997), XML-RPC (1998)**
  - P2P: Napster (1999), Gnutella (2000), BitTorrent (2001)

- Web 2.0:
  - Social Media: LinkedIn (2002), MySpace (2003), Facebook (2004), Twitter (2008), WhatsApp (2009), Snapchat (2011)
  - Mobile Apps / Gaming (2008-)
  - **SOAP (2000), REST (2000), JSON (2001), JMS (2002), JSON-RPC (2005)**



Growth of the Internet



Why Distributed Systems?

- **Cost/Performance/Scalability**.
  - Better price/performance as long as commodity hardware is used for the component computers.
  - Resources such as processing and storage capacity can be increased incrementally.
  - Scale Up vs Scale Out
- **Reliability/Fault Tolerance**.
  - Redundancy of functionality / replication of data.
- **Inherent distribution**.
  - Some applications, such as email and the Web (where users are spread out over the whole world), are naturally distributed.
  - This includes cases where users are geographically dispersed as well as when single resources (e.g., printers, data) need to be shared.

HOWEVER…

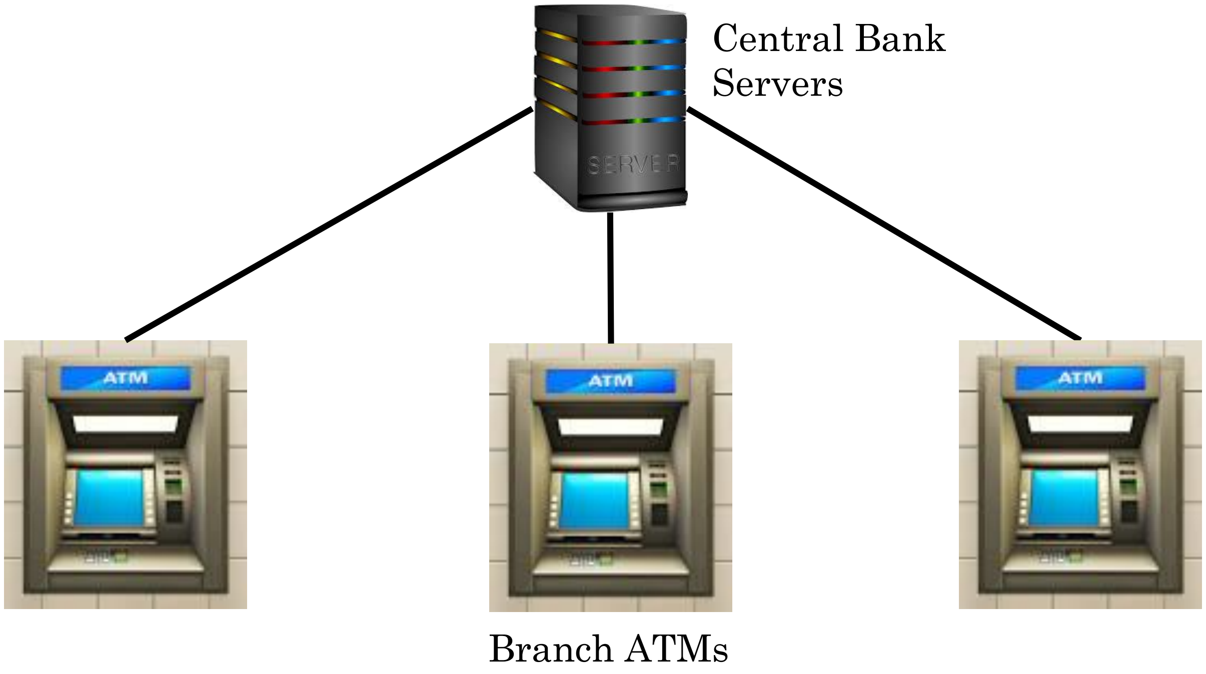
- **New component: network**. Networks are needed to connect independent nodes and are subject to performance limitations. Besides these limitations, networks also constitute new potential points of failure.

- **Security**. Because a distributed system consists of multiple components there are more elements that can be compromised and must, therefore, be secured. This makes it easier to compromise distributed systems.

- **Software complexity**. As will become clear throughout this course - distributed software is more complex and harder to develop than conventional software; this makes it more expensive to develop and there is a greater chance of introducing errors.

Example

- A bank asks you to program their new ATM software
  - Central bank computer (server) stores account information
  - Remote ATMs authenticate customers and deliver money

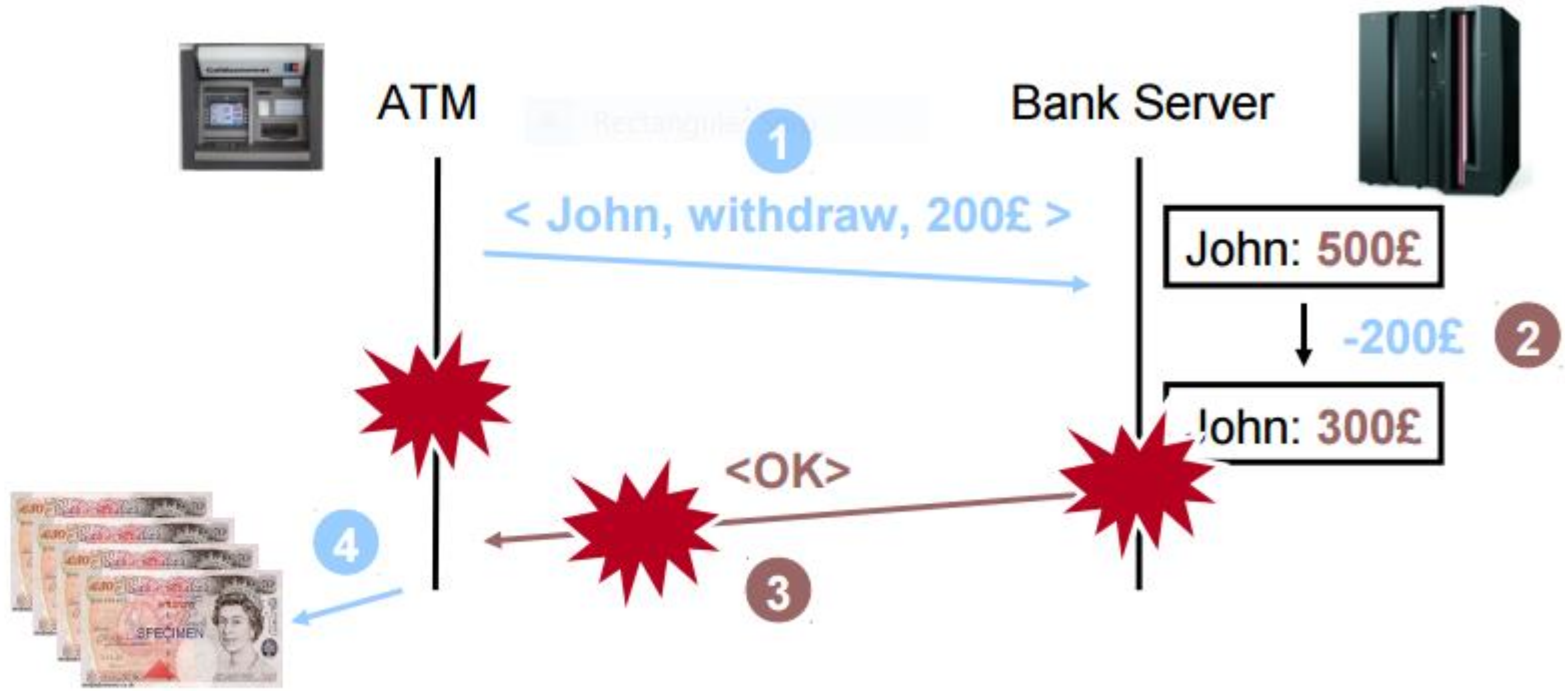


Example: Naïve Solution

- ATM: (ignoring authentication and security issues)
  1. Ask customer how much money s/he wants
  2. Send message with **<customerId, 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 Server:
  1. Wait for messages from ATM:
  2. If enough money withdraw money, send **<OK>**, else send **<refused>**

Example: Naïve Solution



- But …
  - What if the bank server crashes just after 2 and before 3?
  - What if the message gets lost? Takes days to arrive?
  - What is the ATM crashes after 1, but before 4?

Typical False Assumptions

- The network is reliable
- The network is secure
- Everything is homogeneous
- The topology does not change
- Latency is zero
- Bandwidth is infinite
- Transport cost is zero
- There is one administrator

The CAP (Brewers) Theorem

- It is impossible for a distributed data store to satisfy all three of the following guarantees:
  - **Consistency**: Every read receives the most recent write or an error
  - **Availability**: Every request receives a non-error response – without guarantee that it contains the most recent write
  - **Partition tolerance**: The system continues to operate despite an arbitrary number of messages being dropped (or delayed) by the network between nodes.

- In short, designers of distributed data stores must choose between consistency and availability:
  - SQL databases: Consistency over Availability (ACID)
  - NoSQL databases: Availability over Consistency (BASE)

Consistency Models

- **Strict Consistency**: a write to a variable / data store must be seen **immediately** by all nodes.
  - In practice can only be achieved by centralised systems.
- **Sequential Consistency**: writes to variables / data stores must be seen in the **same order**.
  - Requires some form of global clock to ensure a consistent ordering
  - Can lead to poor performance in practice
  - Sometimes necessary – e.g. banking systems
- **Eventual Consistency**: reads eventually return the same value (there is no guarantee that it is the latest write).
  - Good for systems where sequential consistency is not required (messages
  - For example – Twitter Tweets follow eventual consistency (your tweet eventually appears in all your followers feeds).
- **Key Issue**: How to know the order of the writes?

Insert: Global Time

- Timestamps provide an ultimate ordering, but…
  - Local clocks are not consistent – clock drift
  - Berkeley and Cristians Algorithms can fix do it on the intranet…
  - BUT –still have to deal with latency, failure, …
- Universal Coordinated Time (UTC)
  - Global hierarchy of clock servers rooted in a set of synchronised Cesium-133 clocks hosted by accepted authorities.
  - Setting on every computer to synchronise with a Network Time Protocol (NTP) signal that is broadcast on most networks.
- Get Logical
  - Stop trying to get the real time, us logical time instead
  - Lamport Clocks – Synchronisation based on **Happens Before**
  - Vector Clocks – Time is a set of counters (1 per machine)

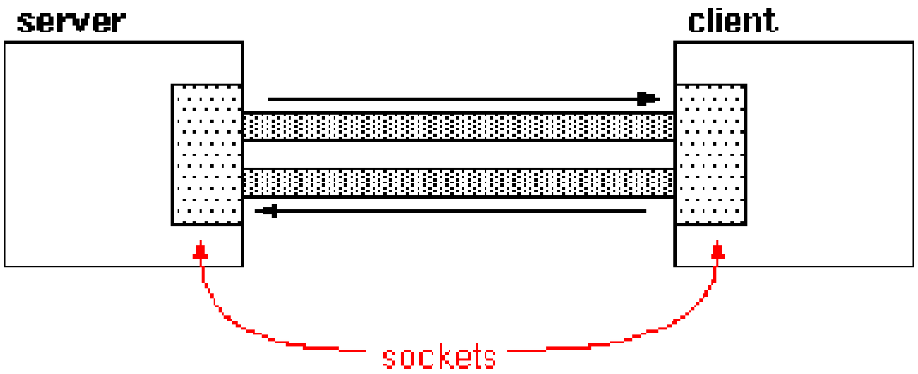
AVAILABILITY

- Availability in the presence of network failure requires replication of data.
  - Each replicas state must be maintained when any update occurs.

- Sequential consistency requires that all updates must be applied in the same order and become visible at the same time.
  - If you have 2 replicas of a piece of data, D, on machines A and B, and D is updated on machine A, then the copy of D on machine B **must be updated before** the update can be read on machine A.

- Eventual consistency requires that all updates eventually be applied (in the same order).
  - In out example above, the A can allow reads of the updated D **before the copy of D has been updated on B**.

FINALLY: REMOTE PROCEDURE CALLS



- Form of Socket-Based **Inter-Process Communication** that mimics a procedure call.
  - Interaction is **synchronous**, based on a **request-response protocol**.
  - The processes are known as the **client** and the **server**.
    - A **client stub** constructs the request and parses the response.
    - A **server stub** parses the request; invokes the procedure and constructs the response.
  - The conversion of parameters/return values to/from some intermediary format is known as **marshalling/unmarshalling**.
    - The intermediary format is known as an Interface Description Language (IDL).