SR (Systèmes Répartis)

Overview

François Taïani

The lecturer

- François Taiani
  - ➔ francois.taiani@irisa.fr

- Background

- My interests
  - ➔ system software (OS, middleware)
  - ➔ large scale distributed computing & algorithms
  - ➔ Self-organisation

The module

- SR (Systèmes Répartis)

- Aims: introduction to
  - ➔ foundations of distributed computing
  - ➔ common distributed problems and their solutions
  - ➔ common distributed services and concepts
  - ➔ construction of distributed computing programs

Assessment

- 2 in-class test (25% + 25%, 50% in total)
  - ➔ 25% for each test
  - ➔ Test 1: Monday, 20 October 14 at 10:15 **(1h, 50%)**
  - ➔ Test 2: in January **(1h, 50%)** (Date TBA!)

- 1 monitored project (2 marking sessions, 50% in total)
  - ➔ marked during lab sessions
  - ➔ project : distributed fault-tolerant game
  - ➔ schedule TBA
  - ➔ first session: Wednesday, 15 October 14 at 10:15

Outline: Part I Foundations

- Unit 1a Intro
- Unit 1b Middleware
- Unit 2 Basic mechanisms and properties: Synchrony, Asynchrony, Reliable Channels
- Unit 3 One fundamental problem: Distributed Snapshots
- Unit 4 Time in distributed systems
- Unit 5 Broadcast and Ordering in Distributed systems
- Unit 6 Synchronization in distributed systems (+ **test 1h**)

Outline: Part II Engineering

- Unit 7 RPC and Indirect Communication
- Unit 8 Fault Tolerance (I)
- Unit 9 Fault Tolerance (II)
- Unit 10 (a) JMS (b) Jgroups
- Unit 11 Perspective: Cloud computing
- Unit 12 **Test (1h)**

Outline: Part I Foundations

- **Unit 1a Intro**
- **Unit 1b Middleware**
- Unit 2 Basic mechanisms and properties: Synchrony, Asynchrony, Reliable Channels
- Unit 3 One fundamental problem: Distributed Snapshots
- Unit 4 Time in distributed systems
- Unit 5 Ordering in distributed systems
- Unit 6 Synchronisation in distributed systems (+ **test 1h**)

SR (Systèmes Répartis)

Unit 1: Introduction & Motivation

François Taïani

Unit Objectives

- Today: For **you** to be able to define **distributed computer systems** and
  - ➔ explain **why** distribution is needed
  - ➔ present **everyday examples** of distributed systems
  - ➔ understand **how** they evolved into their current form
  - ➔ understand the role and importance of **middleware**

Outline of Unit 1

A - Distributed Systems

- ➔What and why?
- ➔Brief history
- ➔Challenges

B - Middleware

- ➔ What is it? Why should we care?

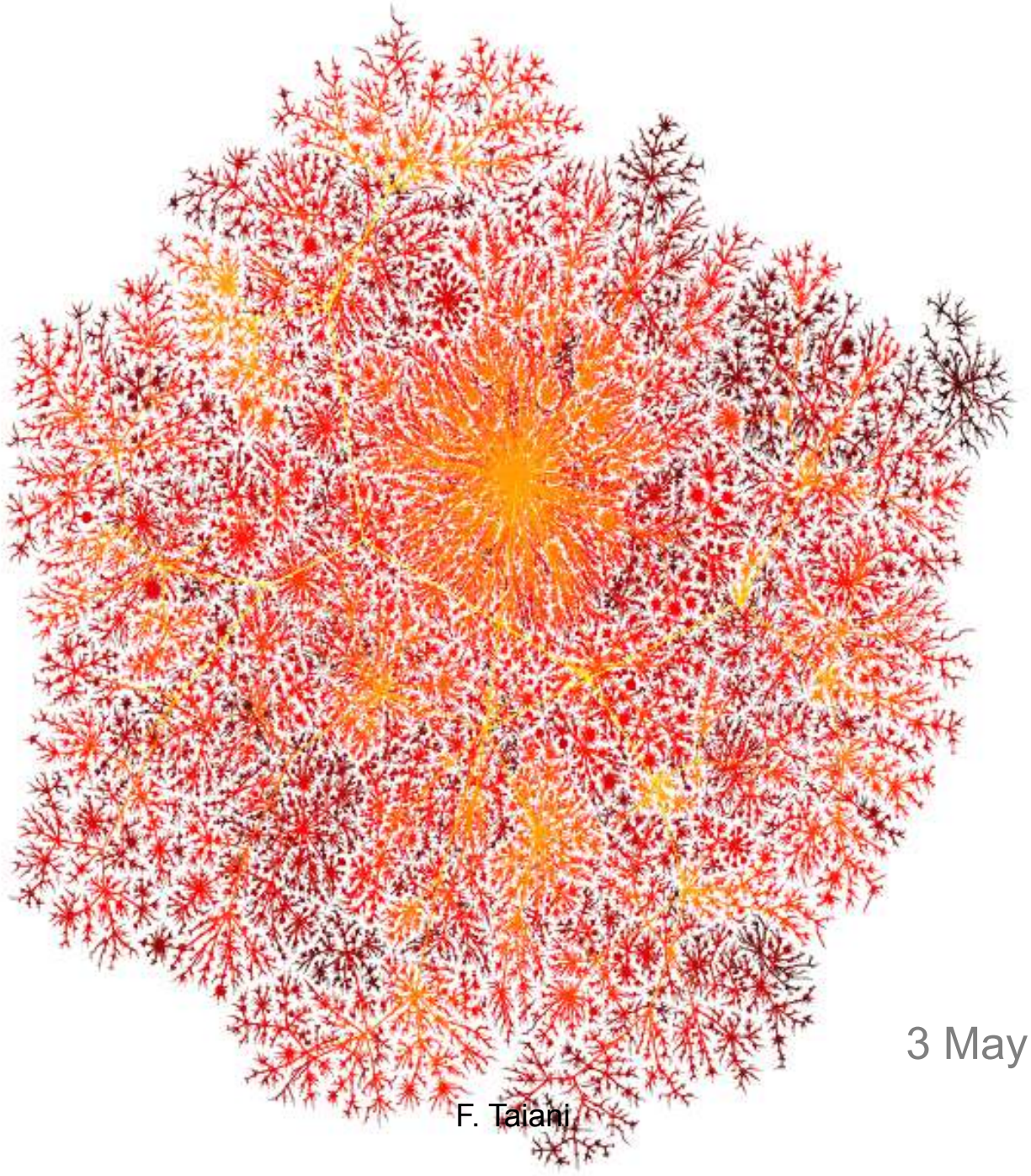
Outline of Unit 1

A - Distributed Systems

- ➔ **What and why?**
- ➔ **Brief history**
- ➔ **Challenges**

B - Middleware

- ➔ What is it? Why should we care?



[source: http://www.cheswick.com/ches/map/]

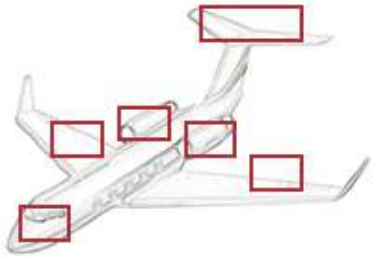
3 May 1999

Examples of Distributed Systems

- **Web search**
  - ➔ The web consist s of over 60 trillion web pages and modern web engines must service over 10 billion queries per month
    - Major distributed systems challenges
- **Massively multiplayer online games**
  - ➔ Online games such as Eve Online or WoW support very large numbers of users viewing a common world
  - ➔ Need for very low latencies to support game
- **Financial trading**
  - ➔ Support for financial trading systems
  - ➔ Dissemination and processing of events
- **Mobile Apps**

Why Distributed Systems?

- Because the **world** is distributed
  - ATM networks
  - railway networks
  - An airplane has one cockpit, but 2 wings



- Because **problems** rarely hits two different places at exactly 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.

Looking for a Definition

- What do these systems have in **common**?
- Can **lessons** learnt from one system be applied to the next?

- **Leslie Lamport** (presently at Microsoft Research)
  - “ *A distributed* [computer] *system is one in which the failure of a computer you didn't even know existed can render your own computer unusable.* ” [1987, e-mail]
- **Tanenbaum & Steen**
  - “ *A collection of independent computers that appears to its users as a single coherent system.* ” [DS Book]

Definition: Key Points

- A DS is made of **autonomous** machines
  - ➔ Implies “physical” distribution, sometimes over large distances
  - ➔ A bi-processor computer is (usually) not considered as a DS

- Appears as a **single entity** to the user
  - ➔ You don’t need to know about hubs, routers, and DNS servers to use a cloud computing application
  - ➔ Implies programs executing at the same time
  - ➔ Interaction/coordination is needed between the computers (just putting them side by side does not make a DS)

- Things can easily go **wrong**!

A Short History of DSs

- Computers are far more **recent** than empires

- And distributed computing is **even more** recent

A Short History of DSs

- Distributed *Computer* Systems are largely concerned with:
  - ➔ Data processing/management/presentation ("**computing**" side)
  - ➔ Communication/ coordination ("**distributed** side")
- Those concerns existed well before computers were invented
  - ➔ **Ancient empires** needed efficient communication systems:
    - cf. the Postal Service of the Persian Empire (6th century BC), cf. the Roman roads (many still visible today), etc.
    - max message speed: ~ 300 km/day in the Persian system
    - assuming a good infrastructure (roads, horses, staging posts)
  - ➔ **Delays** impose distributed organisations
    - Persian and Roman empires extended over 1000s of miles
  - ➔ **Trust / secrecy / reliability** issues
    - Am I sure Governor X is doing what he says he is?

- This all has not really changed! Things have only speeded up!

A Short History of DSs

- During the 80's computer networks mainly remained an **academic** affair

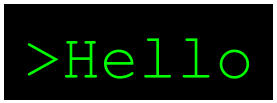
- And then in 1990 came the **Web**

A Short History of DSs

- Computers are far more **recent** than empires
  - The first "modern" computers appeared just after WWII
  - They were slow, bulky, and incredibly expensive
    - The ENIAC (1945), used by the US army: 30 tons, 170 m^2 footprint, 180 kilowatts, 18,000 vacuum tubes, and 5,000 additions/second (5KHz),
    - Price: $500,000 (in US$ of the time, would be roughly $5,000,000 today)
- And distributed computing is **even more** recent
  - For a long time, only very few computers around anyway
  - No practical technology to connect them
  - This all changed in the 80's:
    - The rise of the **micro-computers** (PC, Mac, etc.)
    - The launch of the **"Internet"** (1982, TCP/IP), after 10 years of development
  - (Almost) everybody could have a computer
  - And there was a way to connect them!

A Short History of DSs

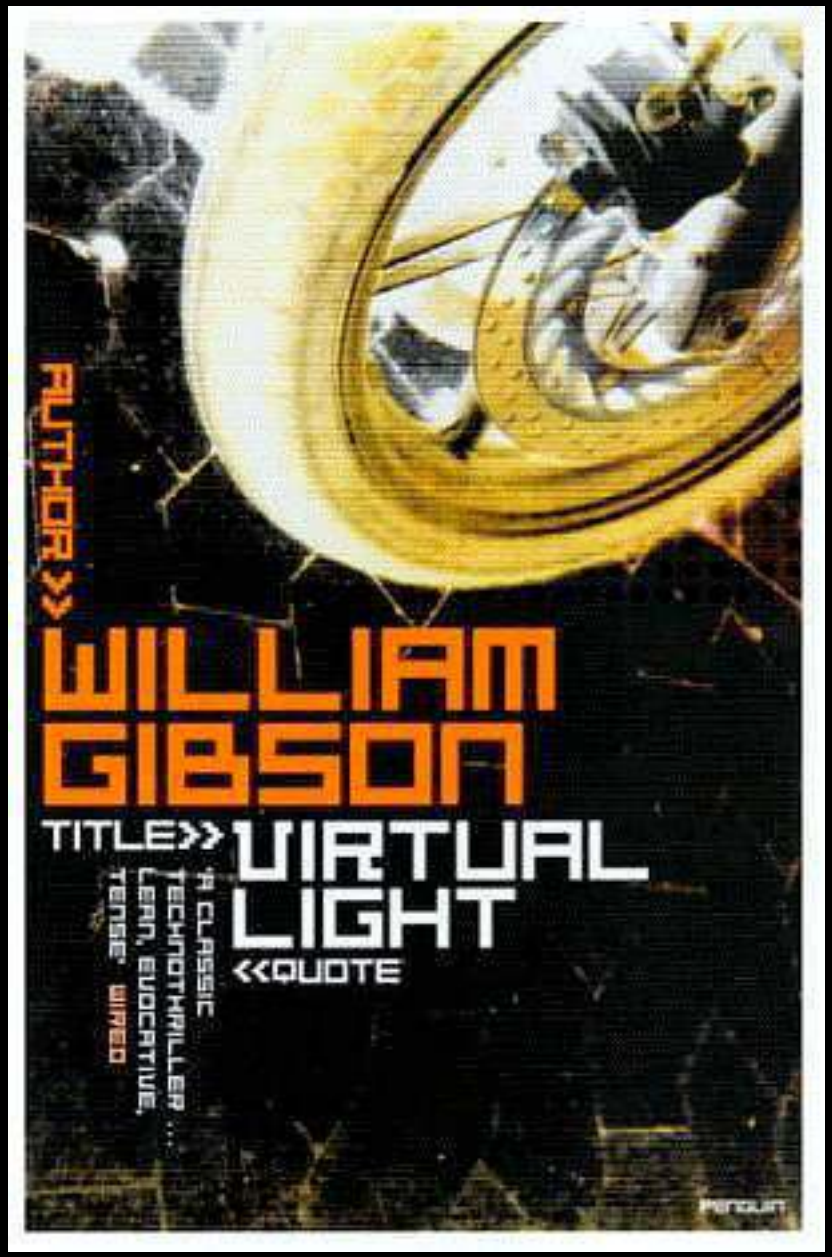
- During the 80's computer networks mainly remained an **academic** affair
  - ➔ Competing networks and technologies, not always compatible
    - ARPANET/Internet was one of them, but not always the biggest.
    - Who remembers BITNET? Was quite big at the time.
  - ➔ Not particularly user friendly
    - User programs were text based (news (Usenet), e-mail)
    - You had to know on which "network" a recipient was to sent her a e-mail.

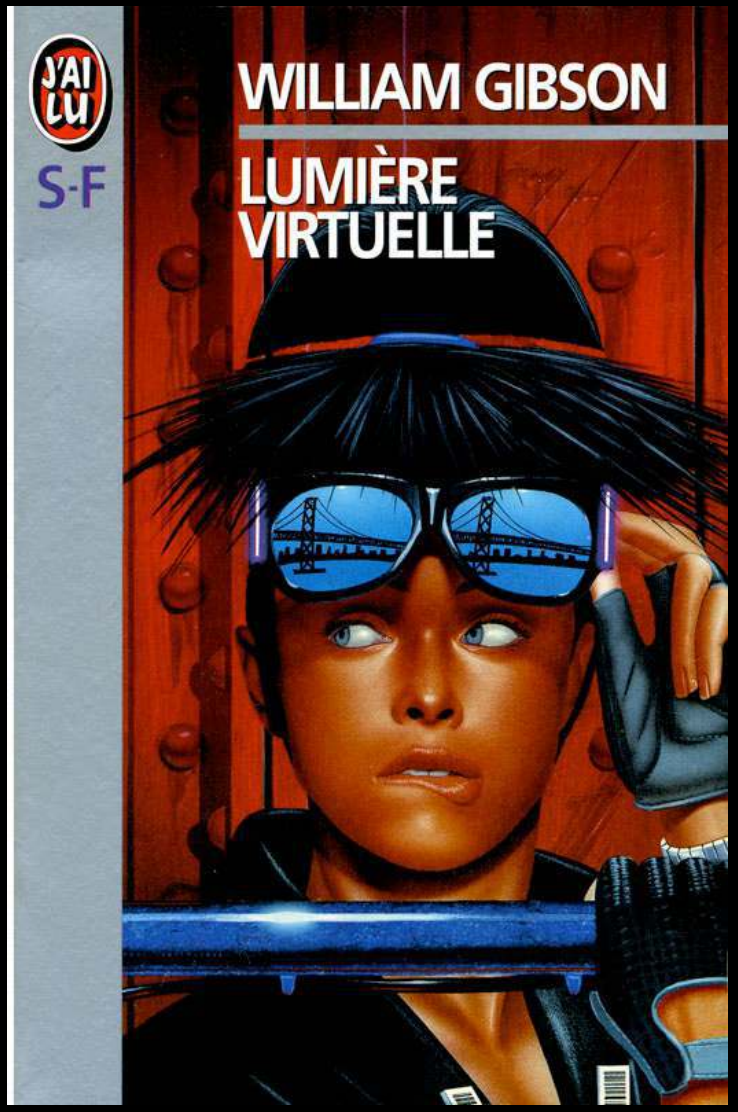


- And then in 1990 came the **Web**
  - ➔ At **CERN** (European Organization for Nuclear Research)
  - ➔ Internet + hypertext (hyperlink) - allowed text-based browsing!
  - ➔ Triggered developments that made the modern web
    - HTML, HTTP, Graphical browsers, search engines, XML, RSS, blogs,….
  - ➔ Assured the domination of the Internet over other networks

“Recent” Developments

- **Client-Server** Applications
  - ➔ Distributed Databases (end of the 80’s)
- Distributed **Object Oriented** Frameworks (90’s)
  - ➔ Adapt ideas from Object Orientation to distributed programming
  - ➔ Several technologies: CORBA, RMI, Web Services (still evolving)
- **Fusion** telephony/television/web/mobility
  - ➔ Allowed by ↑ bandwidth, ↑ processing, ↓ prices
  - ➔ Creates opportunity for new applications:
    - Online games, augmented reality, Internet of Things
- **Cloud computing**
  - ➔ Build on many previous technologies
  - ➔ Key role of **virtualisation** and **web**


AUTHOR>>
WILLIAM GIBSON
TITLE>>
VIRTUAL LIGHT
<<QUOTE
'A CLASSIC TECHNOTHRILLER ... LEAN, EVOCATIVE, TENSE' WIRED
PENGUIN


J'AI LU
S-F
WILLIAM GIBSON
LUMIÈRE VIRTUELLE

Iran Fights Malware Attacking Computers

By DAVID E. SANGER
Published: September 25, 2010

WASHINGTON — The Iranian government agency that runs the country's nuclear facilities, including those the West suspects are part of a weapons program, has reported that its engineers are trying to protect their facilities from a sophisticated computer worm that has infected industrial plants across Iran.

RECOMMEND
TWITTER
E-MAIL
SEND TO PHONE
PRINT
REPRINTS
SHARE

Related

Bits: Malicious Software Program Attacks Industry (September 24, 2010)

The agency, the Atomic Energy Organization, did not specify whether the worm had already infected any of its nuclear facilities, including Natanz, the underground enrichment site that for several years has been a main target of American and Israeli covert programs.

But the announcement raised suspicions, and new questions, about the origins and target of the worm, Stuxnet, which computer experts say is a far cry from common computer malware that has affected the Internet for years. A worm is a self-replicating malware computer program. A virus is malware that infects its target by attaching itself to programs or documents.

What's Popular Now

A Push to Ban Soda Purchases With Food Stamps

Joan Sutherland, Opera Soprano, Dies

MOST E-MAILED | MOST VIEWED

1. PAUL KRUGMAN
Hey, Small Spender


2. CULTURAL STUDIES
The Playground Gets Even Tougher

3. NICHOLAS D. KRISTOF
Test Your Savvy on Religion

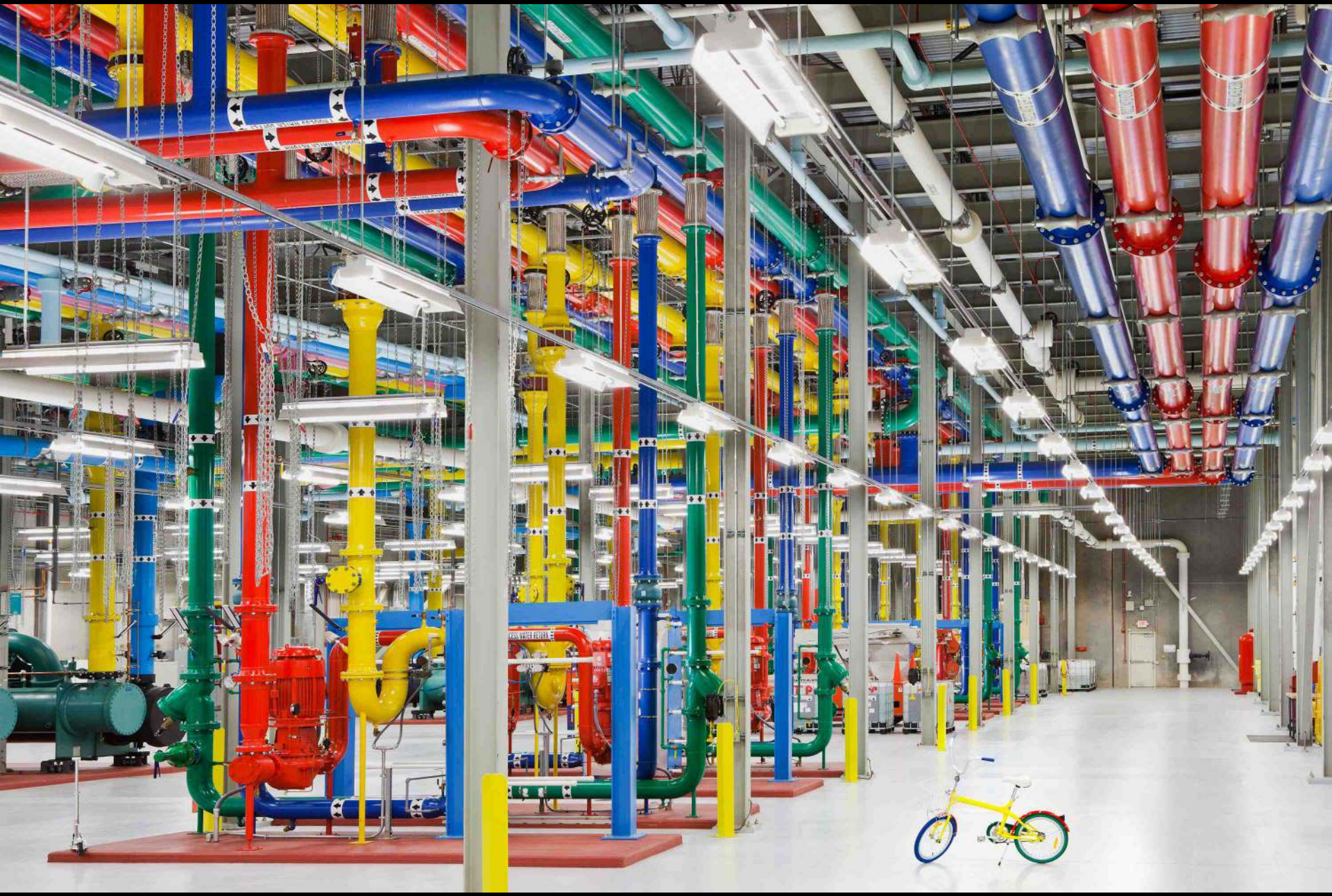

4. **In Life's Latest Chapter, Feeling Free Again**


5. **New Web Code Draws Concern Over Privacy Risks**


http://www.nytimes.com/2010/09/26/world/middleeast/26iran.html

http://www.google.com/about/datacenters/gallery/#/

http://www.google.com/about/datacenters/gallery/#/



http://www.google.com/about/datacenters/gallery/#/

Why care about DS?

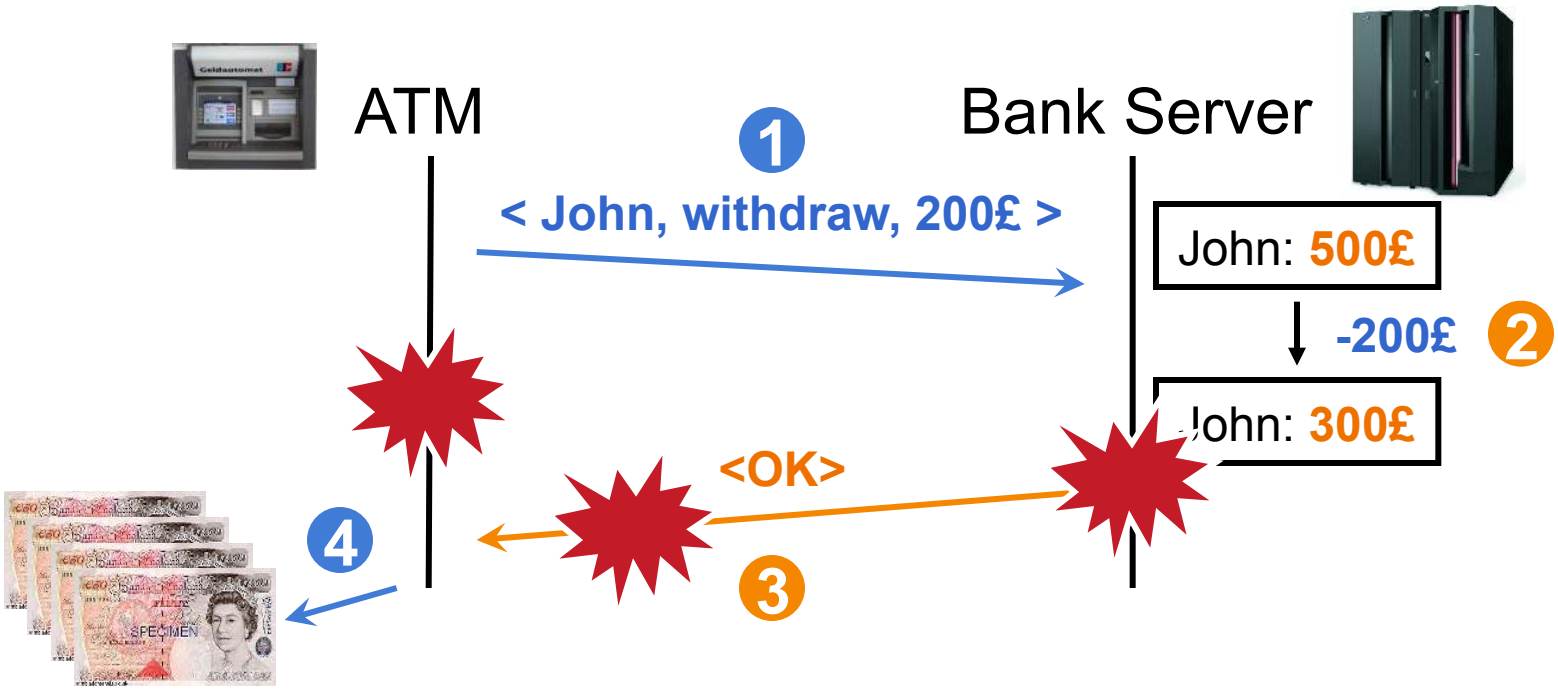
- **Distribution** at the core of almost all recent ICT **revolutions**
  - ➔ mobile telephony (Nokia, iPhone)
  - ➔ search (Google)
  - ➔ social computing (Facebook, Tweeter)
  - ➔ cloud computing
  - ➔ mobile apps

- But **developing** good distributed systems is terribly hard
  - ➔ DS are software intensive (~ 90% of failures at Google)
  - ➔ Developing good distributed software is tough (even for Google)

Why is it hard? Example

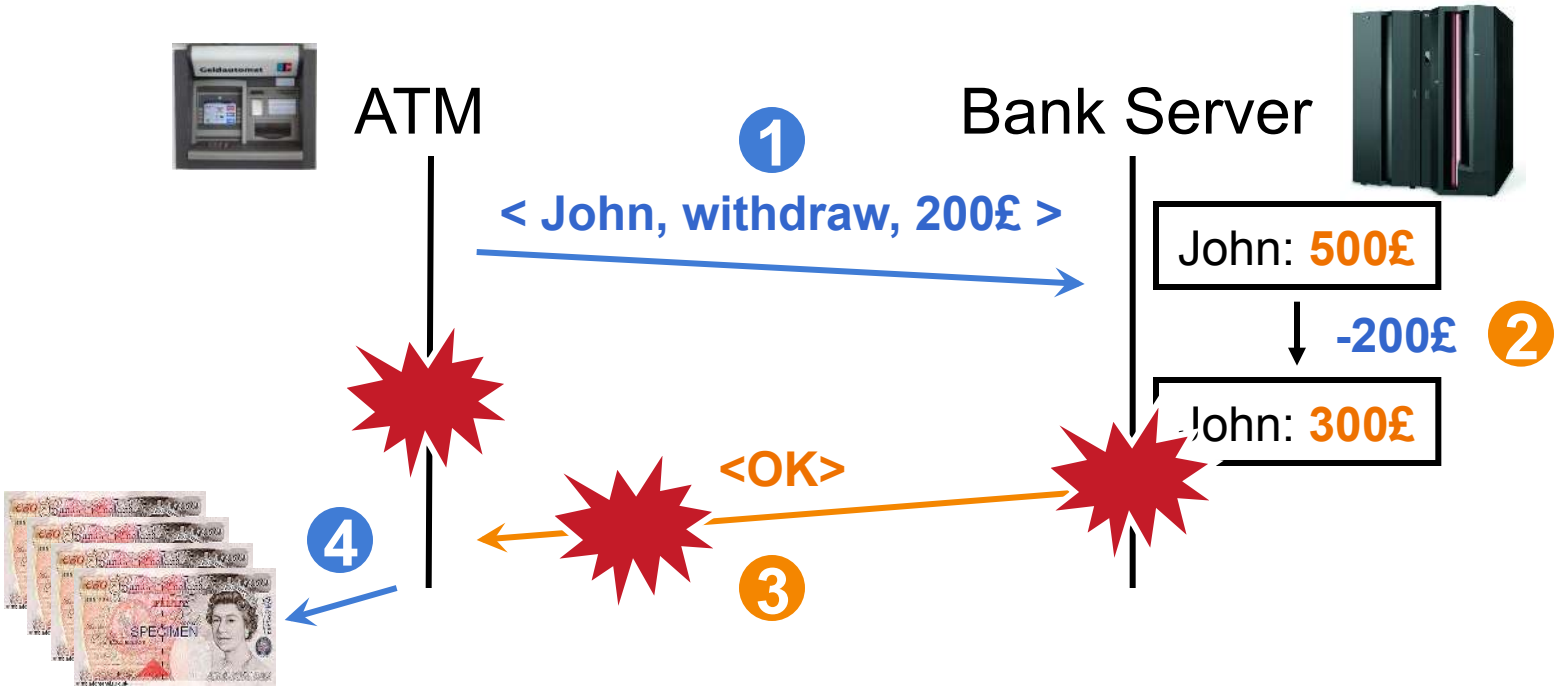
- A bank asks you to program their new **ATM software**
  - ➔ Central bank computer (server) stores account information
  - ➔ Remote ATMs authenticate customers and deliver money
- **A first version** of the program
  - ➔ ATM: (ignoring authentication and security issues)
    1. Ask customer how much 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 Server:
    1. Wait for messages from ATM: **<customer ID, withdraw, amount>**
    2. If enough money withdraw money, send **<OK>,** else send **<refused>**

Why is it hard? Example



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

Why is it hard? Example



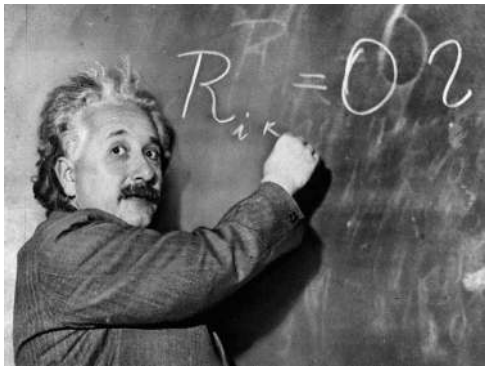
- This problem is known as the **distributed commit problem**
  - ➔ Everybody act or nobody does (atomicity), even if problems
- Requires **fault-tolerance**
  - ➔ System keeps working even when subcomponents fail

Why is it hard? Other Issues

- Other fault-tolerance/availability concerns
  - Replication, caching & consistency issues
  - Reliable communication (multi-cast, message ordering, etc.)
- But fault-tolerance/availability not the only concerns in DS:
  - **Heterogeneity**: How to "glue" different applications on different OS, written in different languages, from different vendors?
  - **Evolvability**: How to change parts of a DS or add new parts without stopping the whole system?
  - **Scalability**: Can a DS grow smoothly without disruption? Are they inherent size limitation in the techniques involved?
  - **Separation of Concerns**: Can development effort be split easily between teams?
  - **Security**: Risks? Vulnerabilities? Which level of integrity, confidentiality, robustness does the system present?

Goals of DS Computing

- Aim: To fight two general laws of physics
  - ➔ increasing **chaos and disorder** (3rd law of thermodynamics)
  - ➔ **information's finite speed** (general relativity)

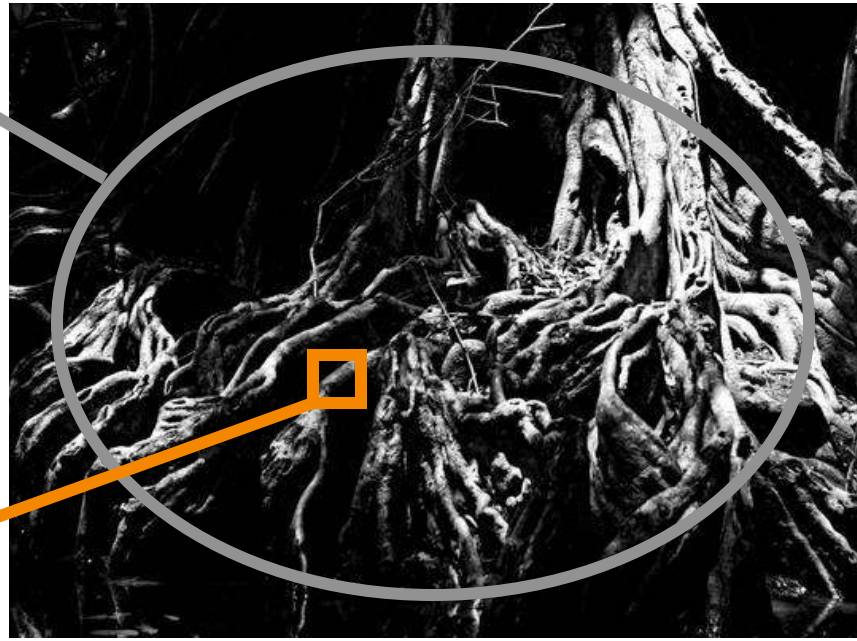


- Approach: **programming tools** that help with the above
  - ➔ Main objective: To abstract away from problems
  - ➔ Many forms: algorithms, programming language, standards
  - ➔ Driven by the mental models programmers like to use
  - ➔ Result encapsulated in middleware technology

Today's Outline

A - Distributed Systems

- ➔What and why?
- ➔Brief history
- ➔Challenges

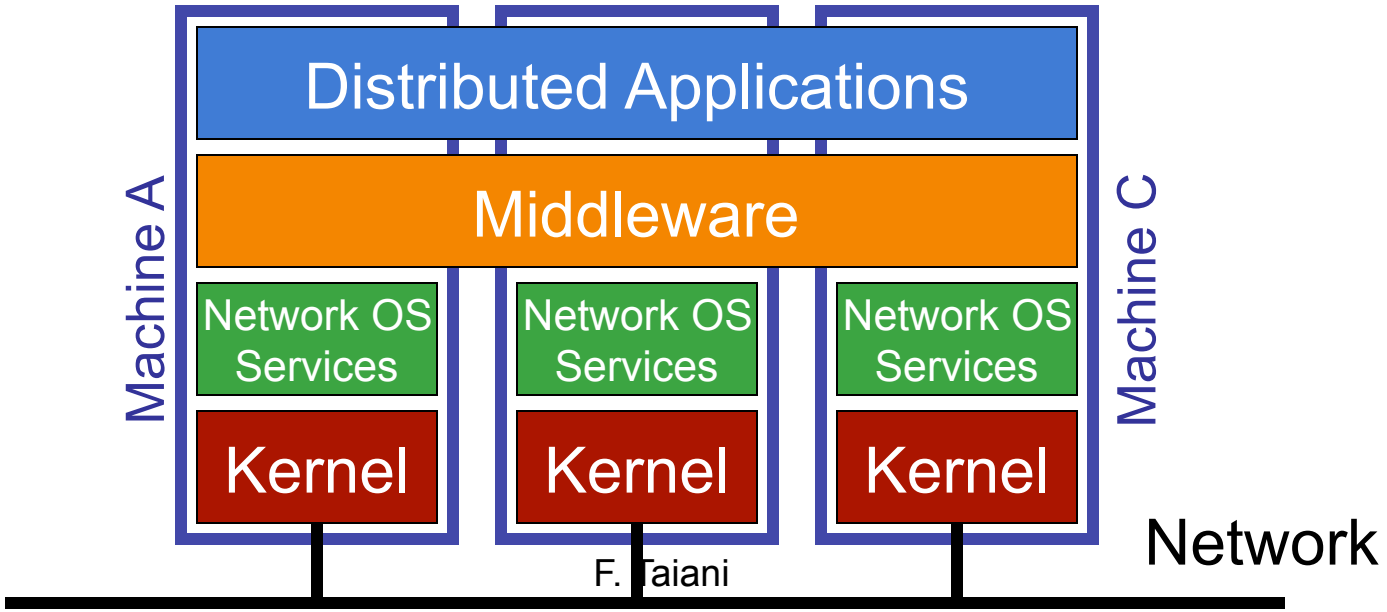


B - Middleware

- ➔ **What is it? Why should we care?**

Middleware

- Key element of recent trends such as **cloud computing**
  - ➔ large amount of middleware central to cloud computing
- A layer providing **"nice"** programming abstractions
  - ➔ to "easily" construct distributed application
  - ➔ different flavours of middleware with different abstractions
  - ➔ all captures in a set of API + specific tools



Quiz 1

- What was the turnover of the middleware industry in 2007?

Quiz 1

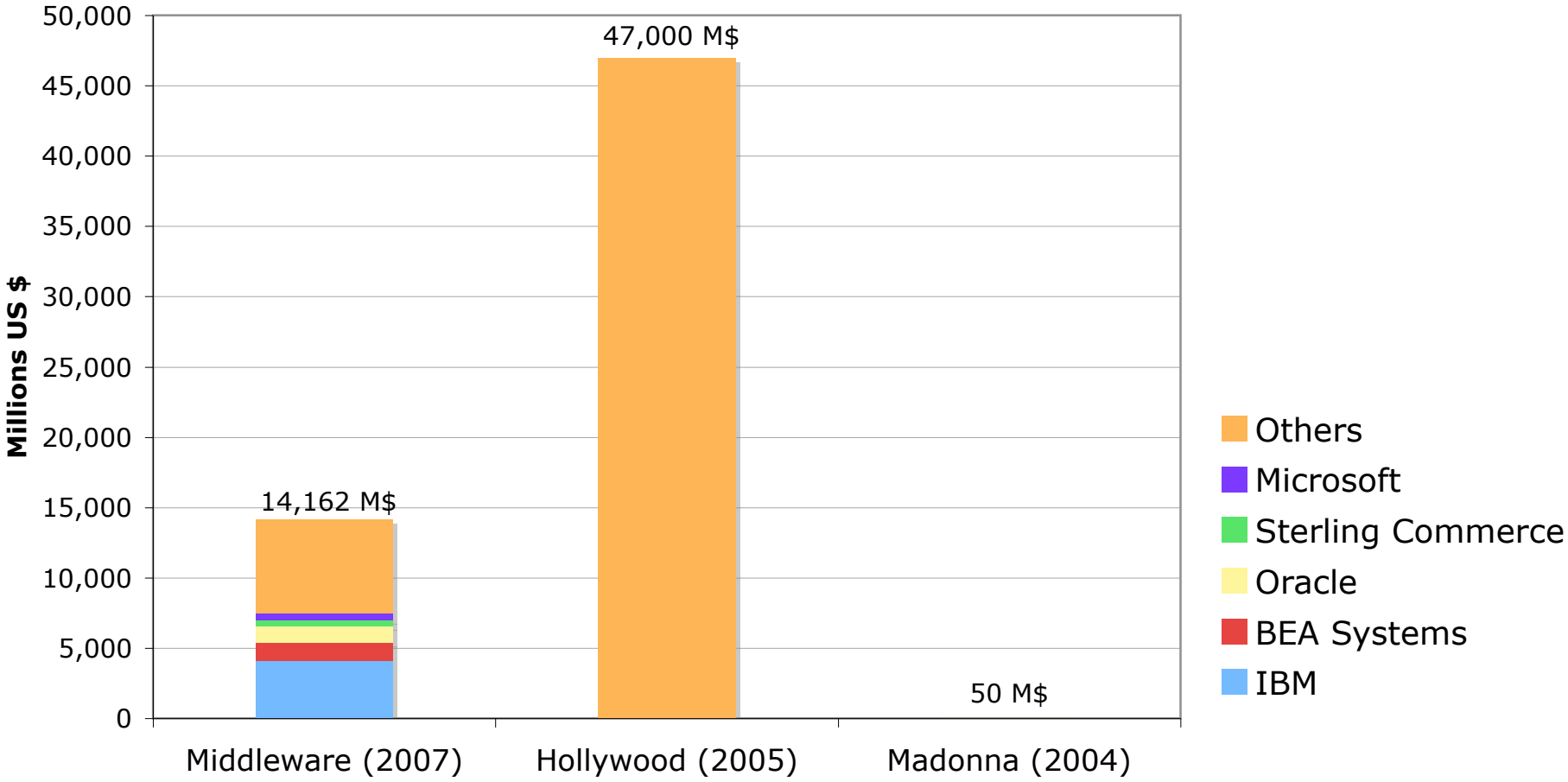
- What was the turnover of the middleware industry in 2007?
  - ➔ 14.1 billion US $ (Gartner Group)
  - ➔ Key players: IBM, BEA, Oracle, Microsoft, Sterling Commerce (AT&T)

[1] Emmerich, W., Aoyama, M., and Sventek, J. 2007. *The impact of research on middleware technology* http://doi.acm.org/10.1145/1228291.1228310

Comparison

- More or less than Hollywood (2005) ?
- More or less than Madonna (2004) ?

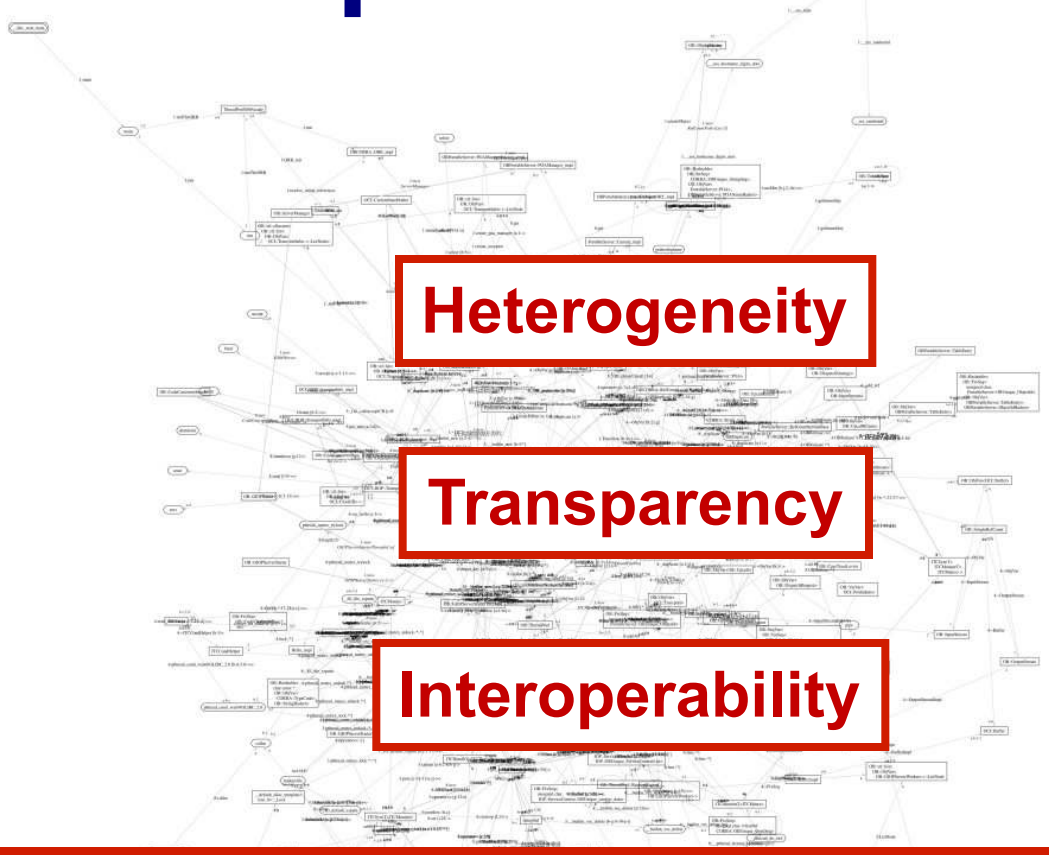
Comparison


50,000
45,000
40,000
35,000
30,000
25,000
20,000
15,000
10,000
5,000
0
Millions US $
47,000 M$
14,162 M$
50 M$
Middleware (2007)
Hollywood (2005)
Madonna (2004)
Others
Microsoft
Sterling Commerce
Oracle
BEA Systems
IBM

Quiz 2

- Orbacus: CORBA middleware
  - ➔ Implemented in C++
  - ➔ On top of Linux (POSIX)
  - ➔ One “Hello world” request processed
- How many C++ classes and C functions involved (server)?
- How many local invocations (server)?
- How big is the source code (LoC)?

Example: ORBacus (4.1.2)



Heterogeneity

Transparency

Interoperability

Standards, Configurability, Portability, …

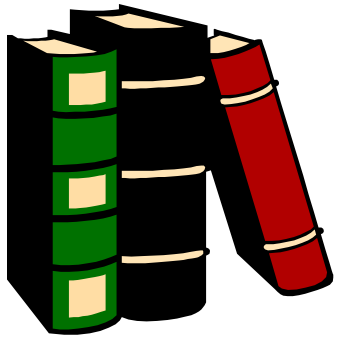
ORBacus Request Processing

one request,
2065 individual invocations,
over 50 C-functions and 140 C++ classes.

The Crucial Problem of Heterogeneity

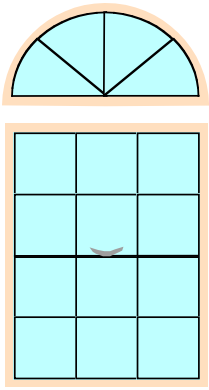
- Modern systems are inevitably heterogeneous
  - ➔ Hardware, OSs, programming languages, etc.

- Emergence of open distributed processing
  - ➔ Support for interoperability and portability
  - ➔ Importance of standards
    - De facto
    - De jure



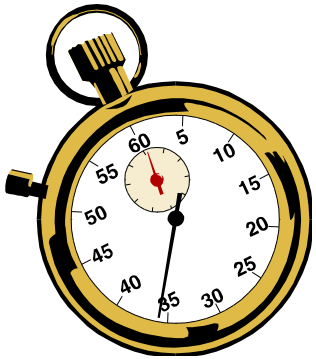
The Issue of Transparency

- Definition
  - ➔ **Transparency** (in dist. systems) = **hiding aspects** of distribution, e.g. *location transparency*
- Example: Transparency in Remote Procedure Call
  - ➔ Ultimate goal, but generally compromised by:
    - overall **cost** of an RPC
    - **extra exceptions** that are raised
    - **parameter** passing is different, e.g. pointers



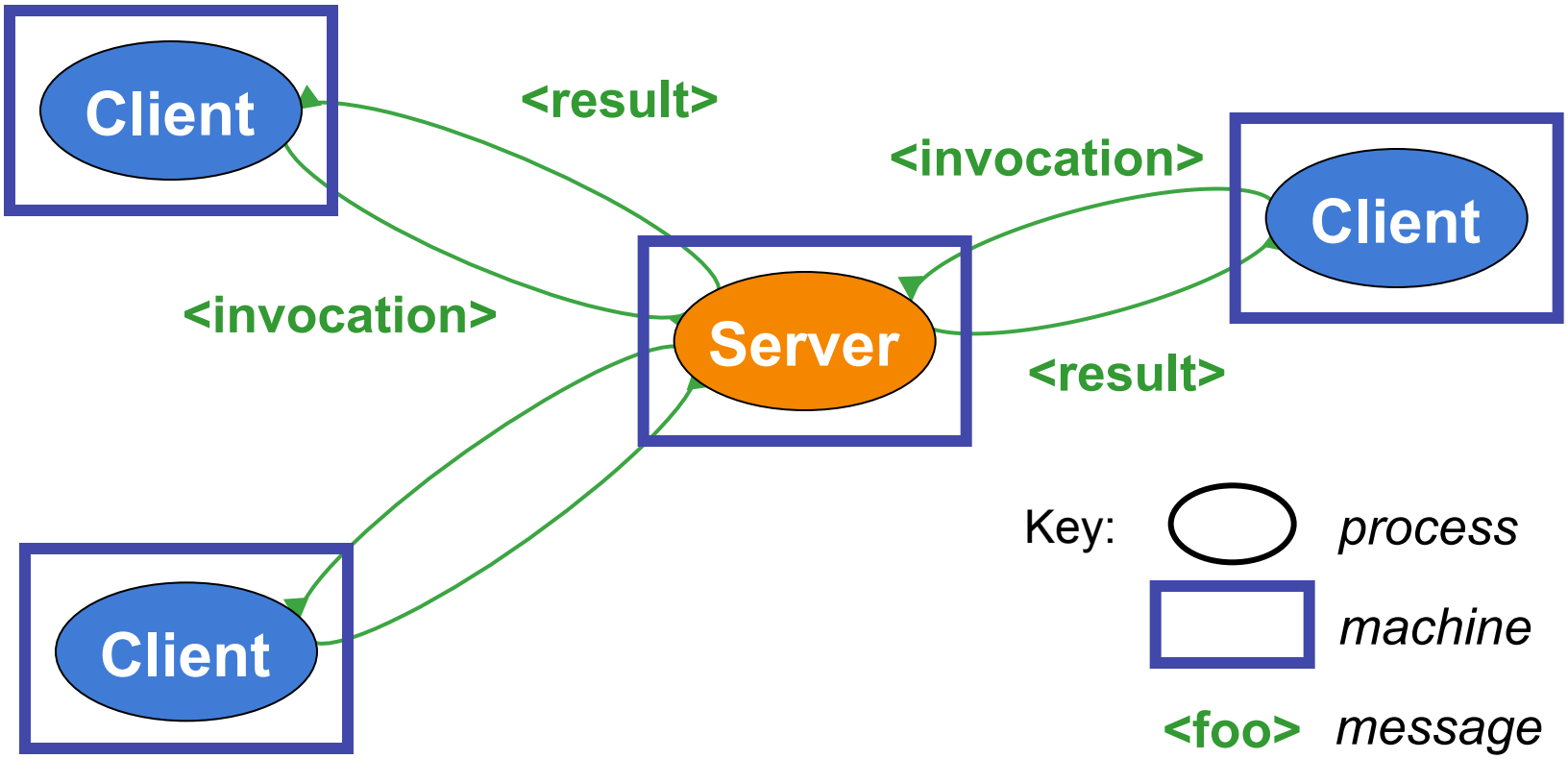
A History of Middleware

- First generation middleware
  - ➔ Based exclusively on the **client-server** model, and approaches such as 2- or 3- tier architectures
  - ➔ Examples include the Open Group's DCE
- Second generation middleware
  - ➔ Based on **distributed object** technology
  - ➔ Examples include CORBA and Java RMI

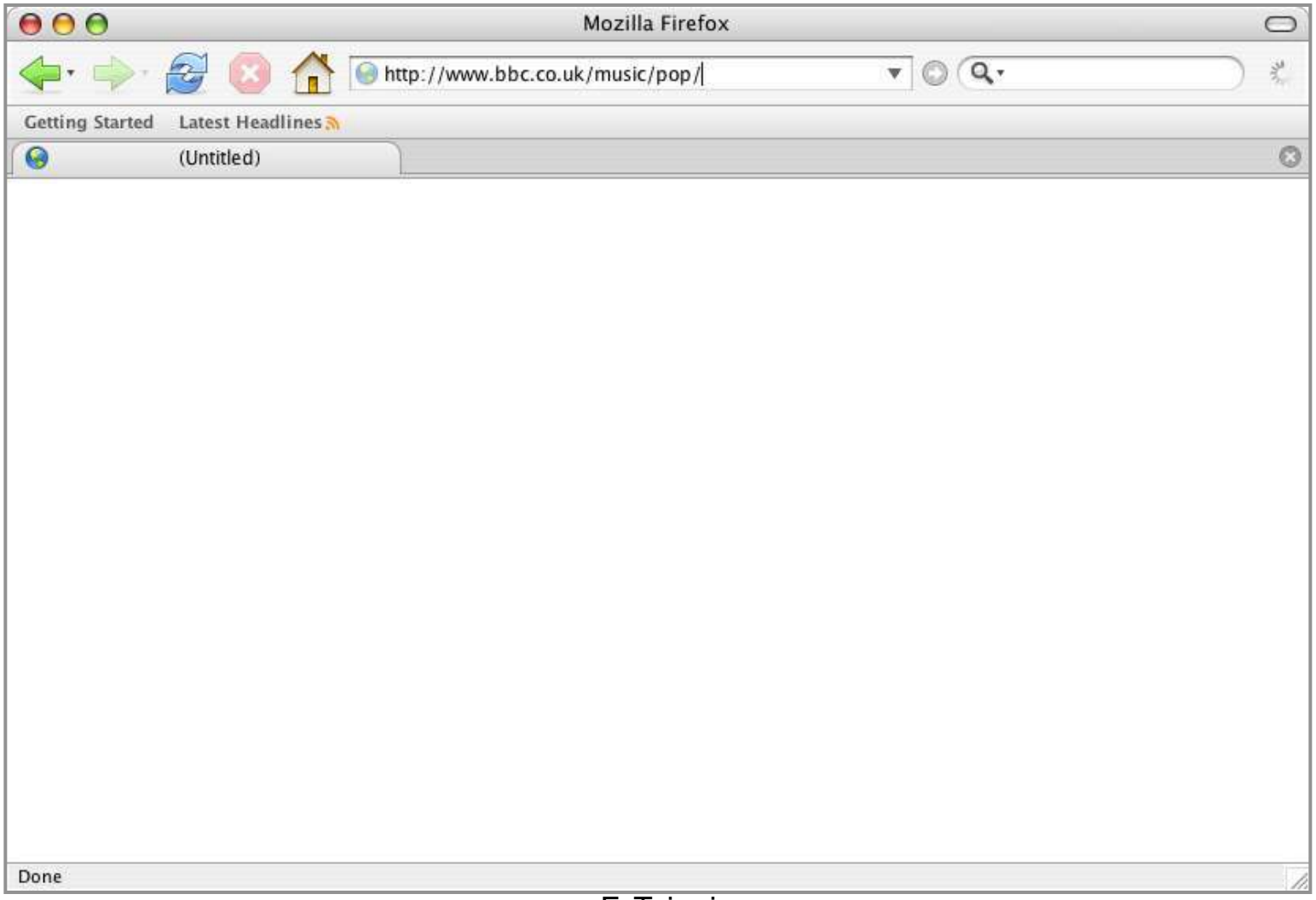


- Third generation
  - ➔ Based on **component technology** and integration **frameworks** e.g. Enterprise Java Beans, OSGi, Spring, .Net
- Fourth generation
  - ➔ Middleware as an on-line service = **cloud computing**

The Client-Server Model



Example: Surfing the Web



Example

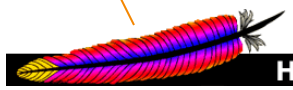
Firefox

```
<!DOCTYPE HTML ...>
<html><head>
<title>
BBC - Music - Pop
</title>
[... etc. ...]
```

Mozilla

Apache

GET /music/pop/



MS IE

www.bbc.co.uk
(IP Address: 212.58.224.89)

Extra Note on Web Surfing

- There is no direct “wire” from “mymachine” to the BBC server
  - Messages go through a series of **intermediaries**: hubs, routers
  - Transmitting the information done the **networking stack**
- In “GET /music/pop/”, GET is an operation of the **http** protocol
  - Client and server need to agree on a protocol to interact
  - HTTP is a simpler mechanism that RPC or RMI
    - with RPC / RMI you do not see the protocol but there is one down deep!
- “www.bbc.co.uk” is a **symbolic name**
  - It can’t be used directly by the TCP/IP stack
  - It needs to be mapped to an “**IP Address**” to be usable
  - This is performed by a distributed **naming service**: **DNS** service

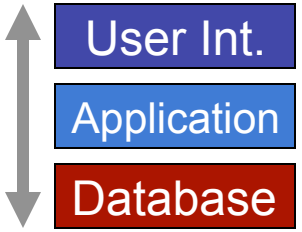
Architectural decomposition

General elements composing a distributed systems

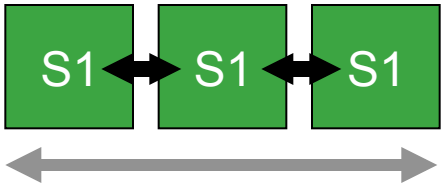
- **Entities**
  - ➔ logical units of behaviour that communicate
- **Communication paradigms**
  - ➔ How do they communicate
- **Roles**
  - ➔ What roles and responsabilities do they have?
- **Placement**
  - ➔ Where are they placed on the physical infrastructures?

Revisiting Client-Server (1)

- C/S architectures based on a layering model
  - ➔ Presentation, business logic, database, etc.
  - ➔ Different types of **entities**, with different **roles**

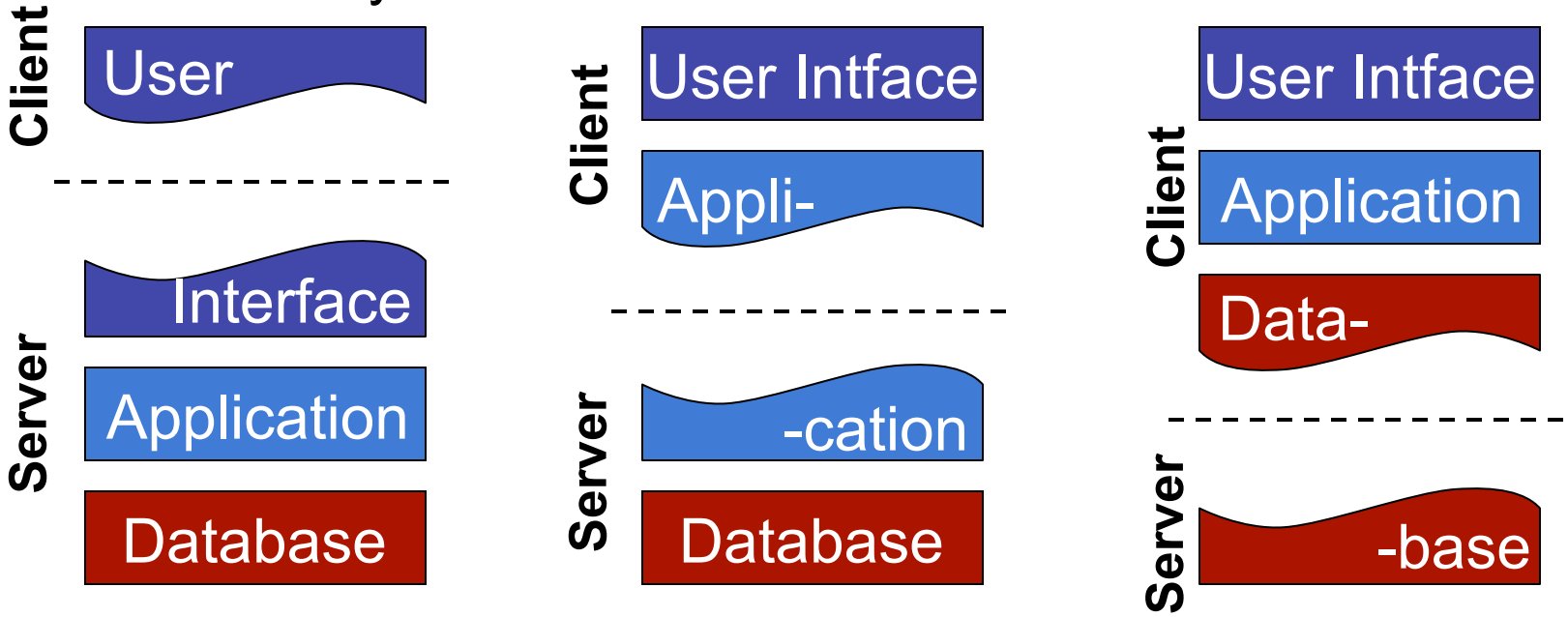


- Not the only way of organising a DS
  - ➔ An application can also be physically split into **logically equivalent entities** executing on different machines
  - ➔ Very useful for **load balancing** and **scalability**
  - ➔ Focus is then on how to coordinate those cooperating entities



Revisiting Client-Server (2)

- **Roles** between client and server can be **split** in many various ways



- The above cases are all called **2-tier** architectures
  - ➔ Two **entities** usually **placed** on 2 kind of **machines**: clients and servers

Summary

- Distributed Systems
  - ➔ multiple machine
  - ➔ exchange messages: messages take time
- Motivated by:
  - ➔ geographic / spatial distribution
  - ➔ need for computing power
  - ➔ need for resilience
- Two sides
  - ➔ theoretical: what we can and cannot do, how
  - ➔ applied: middleware technologies to construct DS


Questions?

SR (Systèmes Répartis)

Unit 2: Basic mechanisms and properties: Synchrony, Asynchrony, Reliable Channels

François Taïani

Distributed Algorithms

- Distributed Algorithms look at
  - ➔ fundamental problems of **distributed coordination**
  - ➔ for instance: agreement, mutual exclusion, leader election...
  - ➔ in an **abstract way** (abstract model of reality)
- Sometimes assuming some **adverse conditions**
  - ➔ participants may behave somewhat erratically
  - ➔ messages may get lost
- Goal of the study of distributed algorithms
  - ➔ find out **whether something is possible** under which conditions
  - ➔ for solvable problems, **prove** that a particular solution works
  - ➔ **compare** correct solutions to the same problems

Goal of this session

- A first contact with **Distributed Algorithms**
- Explore some fundamental aspects of distributed systems
  - ➔ distinguish between different kind of distributed systems
  - ➔ an example of what can and cannot be done
- 3 problems presented as metaphors
  - ➔ The cursed monastery
  - ➔ The royal wedding
  - ➔ The 2 generals
- We will discuss each of them, try to find solutions and see what this teaches us on distributed systems

Group Solving Session

The Cursed Monastery

- A visitor comes to a remote monastery and announces:
  - " *Some of the monks have been cursed by the local wizard and marked by a point on their forehead. They must all leave the monastery, or the whole community will perish.* "
- This monastery obeys a very strict rule:
  - ➔ There are no mirrors in the monastery.
  - ➔ Monks do not communicate in any way.
  - ➔ They only meet once a day for dinner.

- The visitor makes his announcement at diner.
- How many days does it take for all the cursed monks to leave the monastery and why?
  - ➔ Hint: the monks have studied distributed algorithms

The Royal Wedding

- A king would like to marry his son to the princess of a neighbouring kingdom
- By tradition, if the alliance is agreed, the wedding will take place in a remote monastery, on the border between the two kingdoms
- It is all right if the parties do not arrive at the same time at the monastery
- Messengers travel by horses, and may get lost to thugs
  - ➔ however they have a non-zero chance of getting through
- Design an algorithm that allows the wedding to take place if both parties agree (if not, nothing should happen)

The 2 Generals

- 2 allied generals have surrounded their common enemy
- Their camps are 1 day apart by horse from each other
- They want to agree on when to attack
- Each can send the other one only one message per day
- Messengers might get attacked by thugs and get lost
- Design an algorithm for the 2 generals to reach an agreement and attack simultaneously

What have we learnt?

- Whether you know or not **how long your messages will take** makes a huge difference
  - ➔ No bound on communication delays: **asynchronous** systems
  - ➔ Bounded communication delays: **synchronous** systems
- With bounded delays + global clock (monastery)
  - ➔ **Not doing something** can mean a lot
- Some problems have **no solution**
  - ➔ timely coordination with lossy channels impossible (the generals)
- If **communication** channels are **faulty**
  - ➔ possible to **make** them **perfect** (the royal wedding)
  - ➔ but a **price** to pay: communication delays can get **arbitrary long**
  - ➔ this is how **Ethernet** & **TCP/IP** work
  - ➔ does **not** work for **real time** systems

SR (Systèmes Répartis)

Unit 3: One fundamental problem: Distributed Snapshots

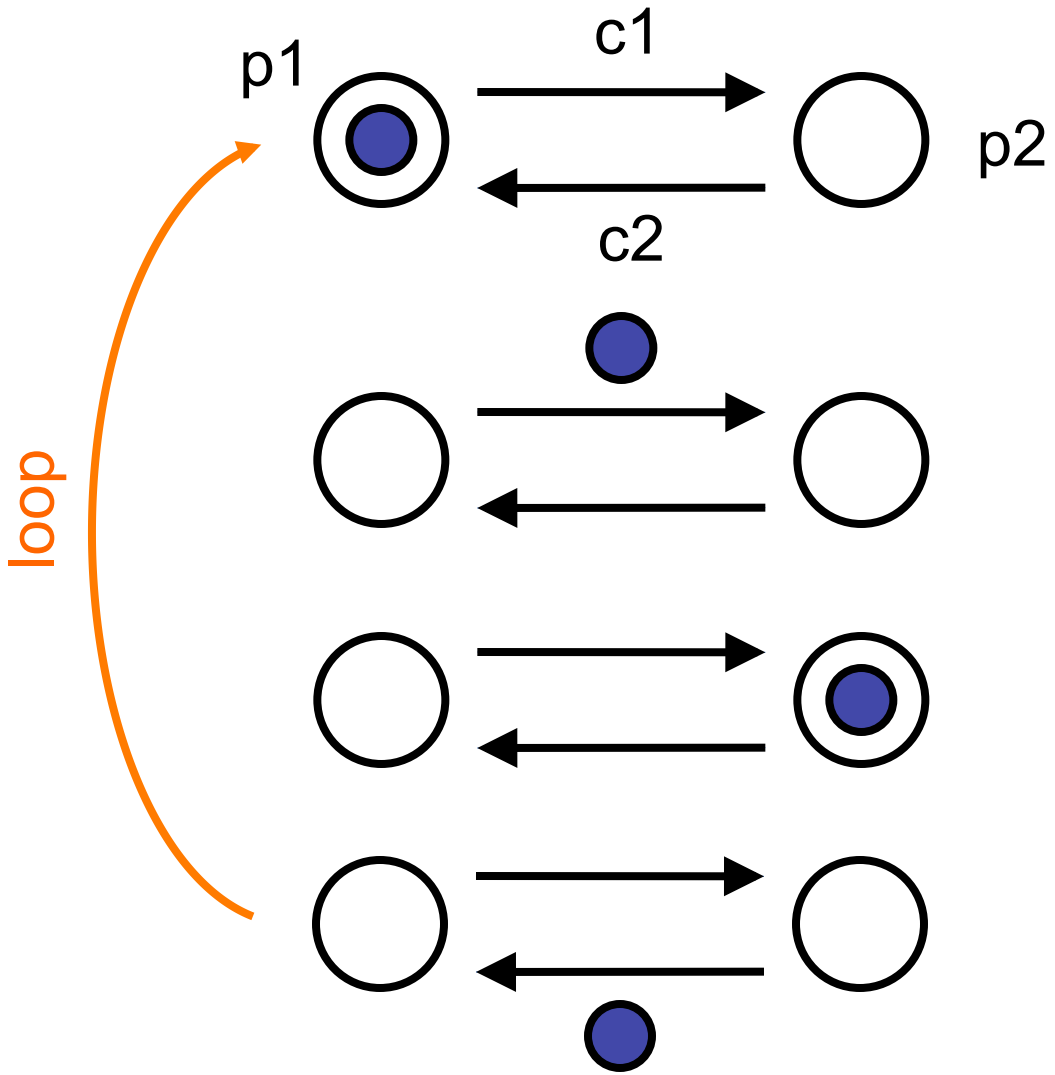
François Taïani

Introduction

- Problem studied in the 70s, 80,s 90s
- Highlights some of the challenges of distributed computing
- Assumptions
  - ➔ set of distributed processes (fixed number, N)
  - ➔ that communicate by message passing (sent, and received)
  - ➔ messages sent over channels (graph of communication)
  - ➔ today: we assume processes do not crash (!)
  - ➔ channels: reliable & FIFO
  - ➔ time of propagation: unbounded (“asynchronous” system)
  - ➔ time of computation: unbounded (process can be very slow)
  - ➔ no global time, no global state (e.g. no shared memory)

Example

- Token exchange
  - ➔ processes p1, p2, channels, c1, c2



Notes

- This is a model, so it abstracts away details
  - ➔ e.g two channels: one socket connection in a real system
  - ➔ previous example: each process with a Boolean state variable + one type of message (e.g. “THE_TOKEN_IS_YOURS”)

Pb of Distributed Snapshot

Goal:

- have each process take a snapshot of
  - ➔ their local state
  - ➔ and that of their associated channels
- so that union of snapshots = valid state of application
  - ➔ a state that is possible for the underlying application
- snapshot protocol should
  - ➔ be generic (i.e. no assumption on underlying application)
  - ➔ run as a controlling agent on top of underlying application
- controlling agent means
  - ➔ can observe, delay application messages, inject own messages, but not interfere with application (transparent)

A First Algorithm

- Assume all processes are connected pair-wise
  - ➔ case on the Internet
- Assume each process knows all other processes
  - ➔ not always the case: e.g. Internet
- Pick one special coordinating process “A” and do on A
  - ➔ for each process p in system {
    send p message “now take your state”
    }
- On receiving “now take your state”, each process do so
- Does this work? Why?
  - ➔ Provide a concrete example using the token application

Simplifying the problem

- Consider a system where communication graph = tree
  - ➔ directed tree, because channels are directed
- System is executing (and might execute permanently)
- Root decides to start a snapshot
  - ➔ what can it do to get others to take a snapshot too?
  - ➔ can you derive an algorithm that works on trees?

Handling cycles

- Consider the simplest of cyclic topologies: a ring
- Does your previous approach still work?
- How can you adapt it so that it does?

Generic Dist. Snapshot Algo

- Chandy & Lamport (1985)

One process initiates the snapshot
(constrain: all other processes reachable from this one)

- ➔ takes a snapshot of its state
- ➔ sends a marker on all outgoing channels
- ➔ starts recording messages received on other channels

On receiving a marker on channel **c** each process does

- if first marker as above
  - ➔ + records state of *channel* **c** on which marker received = empty
- if not first marker (has already recorded its state)
  - ➔ records state of **c** = messages received on **c** since first marker

Does it work?

- Claims
  - ➔ protocol takes snapshots of all processes and channels exactly once
  - ➔ protocol terminates
  - ➔ once terminated, consistent global snapshot of application
- Proofs?
  - ➔ what kind of arguments would you give to support each of the above points?

Proof 1

"Protocol takes snapshot of all process + channels exactly once"

- First part: at most once
  - ➔ processes take their snapshot only when marker = first time
  - ➔ by construction only happens at most once
  - ➔ channels: snapshot only taken when marker comes out of it
  - ➔ marker put on channel by sending process, only happens at most once
  - ➔ so channels have their snapshot taken only at most once
- Second part: exactly once
  - ➔ by contradiction: assume process with snapshot not taken
  - ➔ consider path back to initiator
  - ➔ similarly for channels

Proof 2

- Protocol terminates
  - ➔ protocol made of reactive parts: each time marker received
  - ➔ no blocking statement in the reactive parts of the protocol
  - ➔ marker only sent once on each channel
  - ➔ finite number of channels -> finite number of markers
  - ➔ once all markers received and consumed, protocol terminated

Proof 3

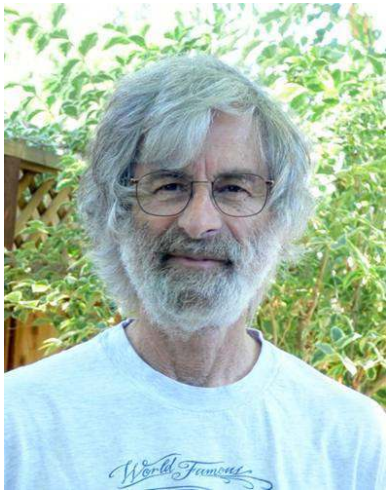
"Global snapshot obtained = consistent state"

- Hardest to prove
- Intuition
  - ➔ consider first marker received by each process
  - ➔ with initiator = spanning tree
  - ➔ if you imagine an execution where each process freezes (becomes very slow) after its state taken = process + channel on tree are consistent
- What of other channels?
  - ➔ recorded state: message accumulated since "freeze"
  - ➔ resulting overall state = consistent ("could" occur in a normal execution)

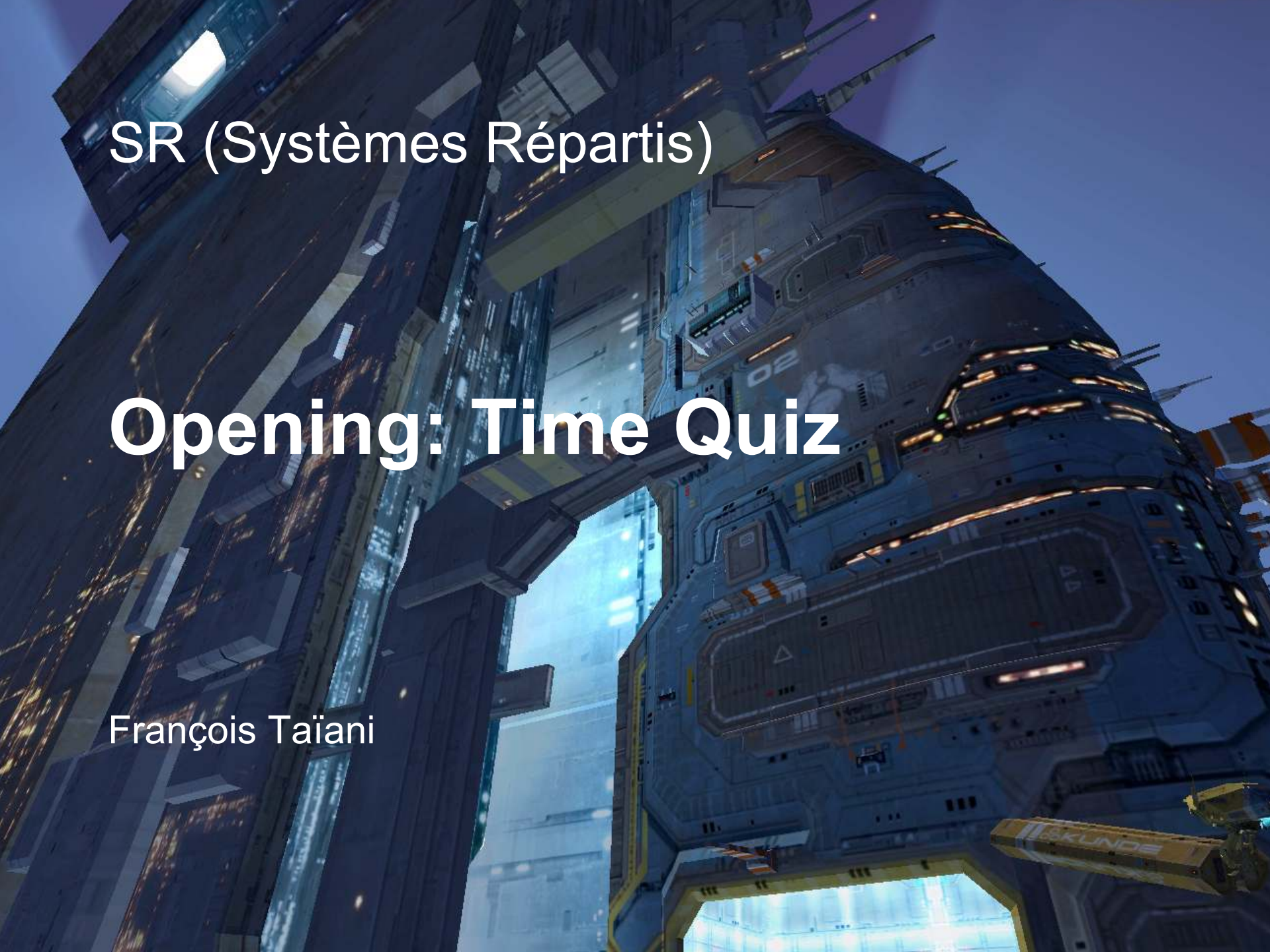
Some Comments

- Design: based on flooding
  - ➔ sends a marker once on all channels (a kind of “wave”)
  - ➔ design found in many other systems (P2P) with variants (epidemic)
- Key difficulties
  - ➔ no global time + message have unbounded delays
- If global time
  - ➔ decide to take snapshot at a particular “exact “ time
  - ➔ record message sent + message received since last time
  - ➔ off band you can then compute diff: gives state of channel
- If bounded delays for channels
  - ➔ send all processes (including itself) a “stop” message
  - ➔ block and record incoming messages
  - ➔ wait 2 x max delay, record state taken at “stop” + msg received

More Comments



- Leslie Lamport did not study comp. science
  - ➔ like many CS pioneers
  - ➔ it just did not exist as a discipline of study
- He studied Math
  - ➔ and taught a bit about relativity
- The snapshot algorithm = inspired by general relativity
  - ➔ what counts is causality, not absolute time
  - ➔ we'll see more of that with logical time (next session)
- More notes on his works by himself (quite interesting on how science works!)
  - ➔ http://research.microsoft.com/en-us/um/people/lamport/pubs/pubs.html



SR (Systèmes Répartis)

Opening: Time Quiz

François Taïani

Time Quiz

- **How old are the oldest remaining calendars in Europe?**
  - ➔ A. before -2000 BC
  - ➔ B. 43 AD
  - ➔ C. 1214 AD
  - ➔ D. 1509 AD

Time Quiz

- **In 1772, King George III said to John Harrison, a genial clockmaker, "By God, Harrison, I'll see you righted!", and saw to it that Harrison received the £20,000 (roughly £3 million in today's terms) from parliament for the clock he had spent his life perfecting.**

  Why was clock accuracy so important at the time that Harrison's plight got the King's attention?

  - ➔ A. Clocks were used to predict religious celebrations
  - ➔ B. Clocks were used for shipping
  - ➔ C. Clocks were employed in early manufactures to organise work
  - ➔ D. Clocks were needed for scientific research

Time Quiz

- **What is the official world time reference?**
  - ➔ A. Coordinated Universal Time (UTC)
  - ➔ B. International Atomic Time (TAI, from French “Temps Atomique International”)
  - ➔ C. UTC and TAI are two names for one and the same thing

Time Quiz

- **What the physical definition of a second?**
  - ➔ A. A second is one 86,400th of the duration of an Earth rotation (Earth day)
  - ➔ B. A second is the fraction 1/31,556,925.9747 of the tropical year for 1900 January 0 at 12 hours ephemeris time
  - ➔ C. A second is the time a caesium-133 atom takes to vibrate 9,192,631,770 times between its two ground states
  - ➔ B. A second is the time an object takes to fall on the ground from an height 4.9m, in a vacuum container, at sea level, on Earth

Time Quiz

- **How many atomic clocks can be found in each GPS satellite**
  - A. 1
  - B. 2
  - C. 3
  - D. 4
  - E. none, atomic clocks are far too expensive to send in space

SR (Systèmes Répartis)

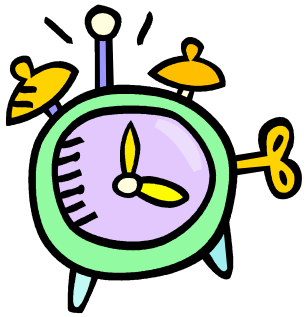
Unit 4: Time in distributed systems

François Taïani

Overview of the Session

- Why Time and Distribution?

- Clocks and their imperfections
  - ➔distributed clock synchronisation
  - ➔logical clocks



Time and Distribution: Why?

- Two main motivation
  - ➔ **internal**: temporal information used for internal mechanisms
  - ➔ **external**: temporal information *about* the distributed systems is needed for activities / processes beyond the system itself

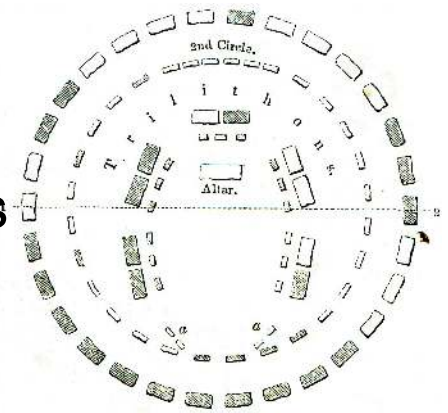
- **Internal reasons**: many distributed algorithms use time, e.g.
  - for security mechanisms (e.g. **Kerberos** timestamps)
  - to **serialise transactions** in databases
  - to **minimise updates** when replicating data
  - ➔ the overall system must be **in sync internally**
  - ➔ No need to be synchronised on an external time reference

Time and Distribution: Why?

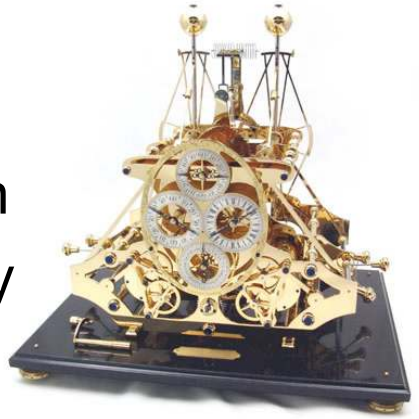
- **External reasons**: We often want to measure time accurately
  - For **billings**: How long was computer X used?
  - For **legal reasons**: When was credit card W charged?
  - For **traceability**: When did this attack occurred? Who did it?
  - ➔ System must be **in sync** with an external time reference (Usually the world time reference: UTC — Universal Coordinated Time)

An (extremely) Brief History of Time

- (... or rather of the measure of time)
- We've always been very interested in time
  - ➔ Many **prehistoric** calendars have reached us



- Measuring time **accurately** always essential
  - ➔ Predict seasons, crops, schedule markets
  - ➔ **Synchronising** time not a new issue
    - In 1582, Pope Gregory ordered 10 days to disappear
- In particular for measuring **locations**
  - ➔ In the 18th century the English had the world best **maritime clocks**. A key technology then
  - ➔ Nowadays the accuracy of the **GPS** is directly linked to its atomic clocks



Imperfect Clocks

- Human-made clocks are imperfect
  - ➔ They run slower or faster than “real” physical time
  - ➔ How much faster or slower is called the drift
  - ➔ A drift of 1% (i.e. $1/100=10^{-2}$) means the clock adds or looses a second every 100 seconds

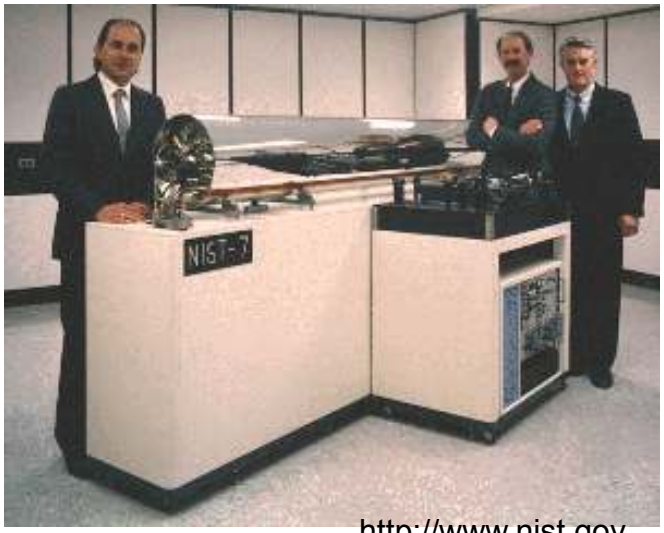
Imperfect Clocks

- Example:
  - A hardware clock generates periodical **hardware interrupts**
  - Every **100** interrupts it increments a counter: **clock_time**++
  - The clock is imperfect:
    it does between **97** and **103** interrupts every second
  - The clock start at 00h00:00 with a counter set to 0 (**clock_time**=0), how much will it **drift** after **one day**?

Imperfect Clocks

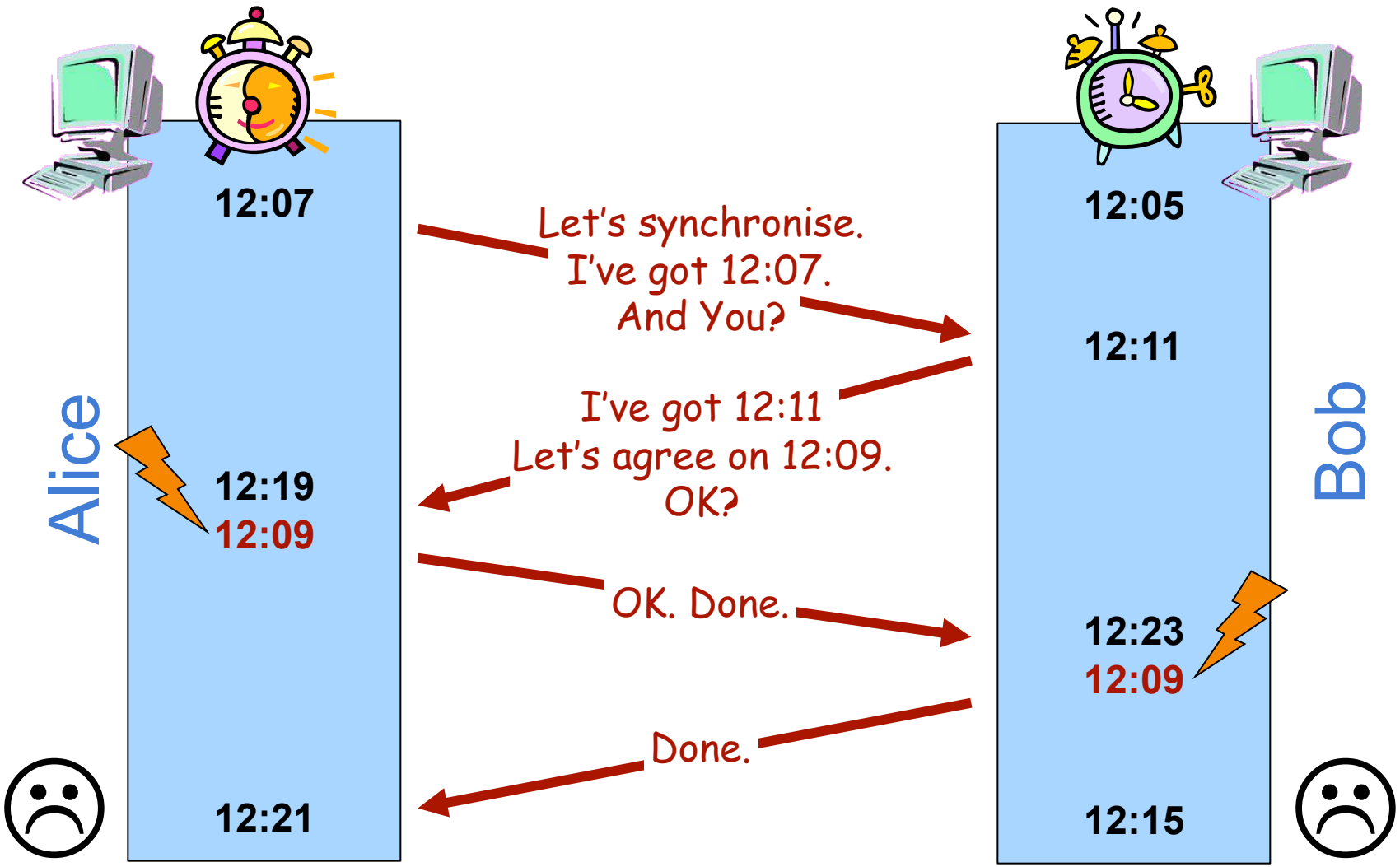
- **Atomic-clocks** are the most accurate ever built
  - ➔ For the best ones drift that is less than one second every 6 million years
  - ➔ But there are **expensive** (up to £30 000)
  - ➔ And can be **quite big** (see NIST-7 below)
  - ➔ Atomic clocks are mounted in all **GPS** satellites

- Typical hardware clocks are not that good
  - ➔ drift around 10^{-5}: ~ 30s / month
- **Synchronisation** required!

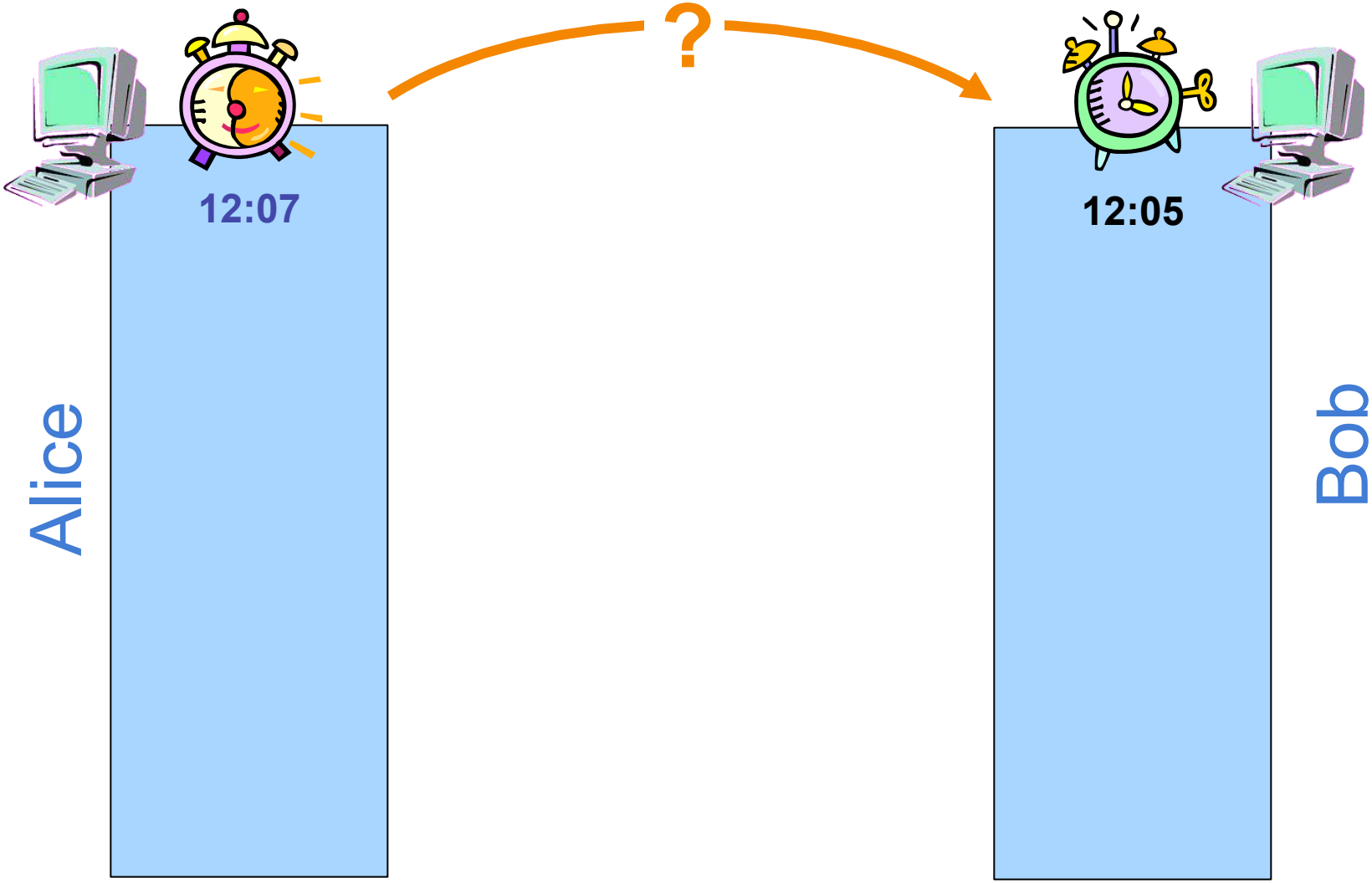


http://www.nist.gov

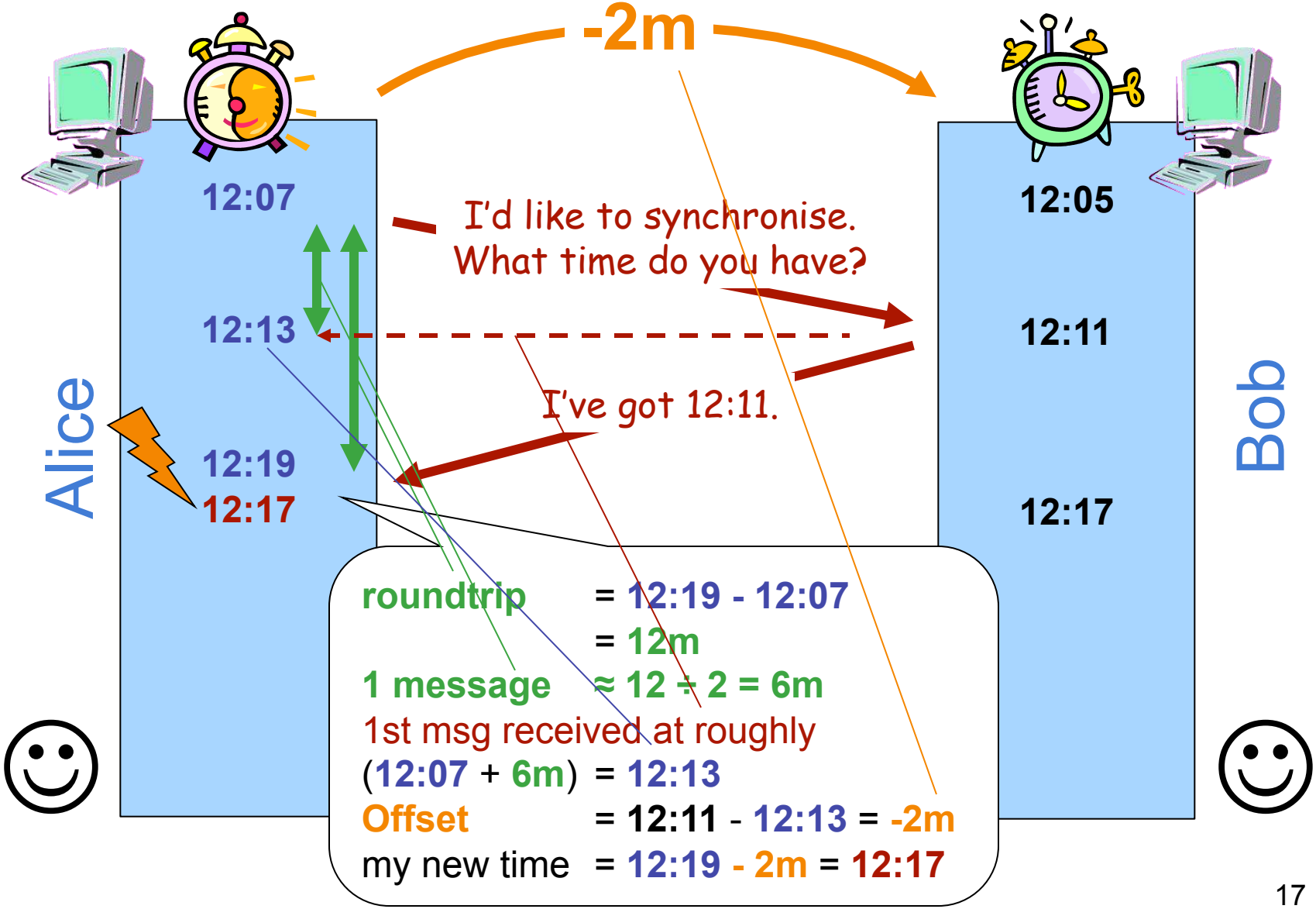
Clock Synchronisation



Clock Synchronisation



Clock Synchronisation

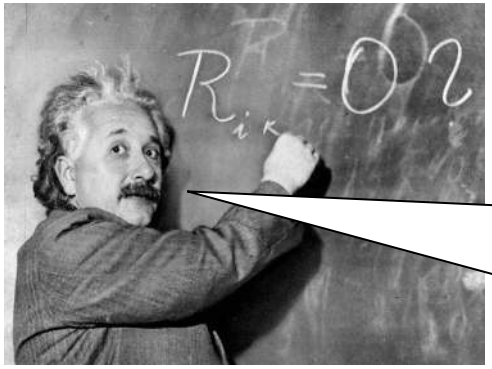


Clock Synchronisation

- Key idea in previous algorithm: estimate **message latency**
  - ➔ Communication delays can then be factored out
- Key **assumption**: request & reply messages = same latency
  - ➔ In typical networks (internet) almost true most of the time
  - ➔ The offset never completely reduced to 0 due to **jitter**
  - ➔ But conditioned by variability in message latency (very small) rather than latency itself (much bigger: 10-400ms)
- Approach used by the **Network Time Protocol (NTP)**
  - ➔ Hierarchy of time servers, synchronised with each other
  - ➔ The "top" servers connected to atomic clocks (UTC)
  - ➔ List of public time servers: http://ntp.isc.org/bin/view/Servers
  - ➔ Accuracy of up to 1ms (with "adaptive" clocks)

Logical Clocks

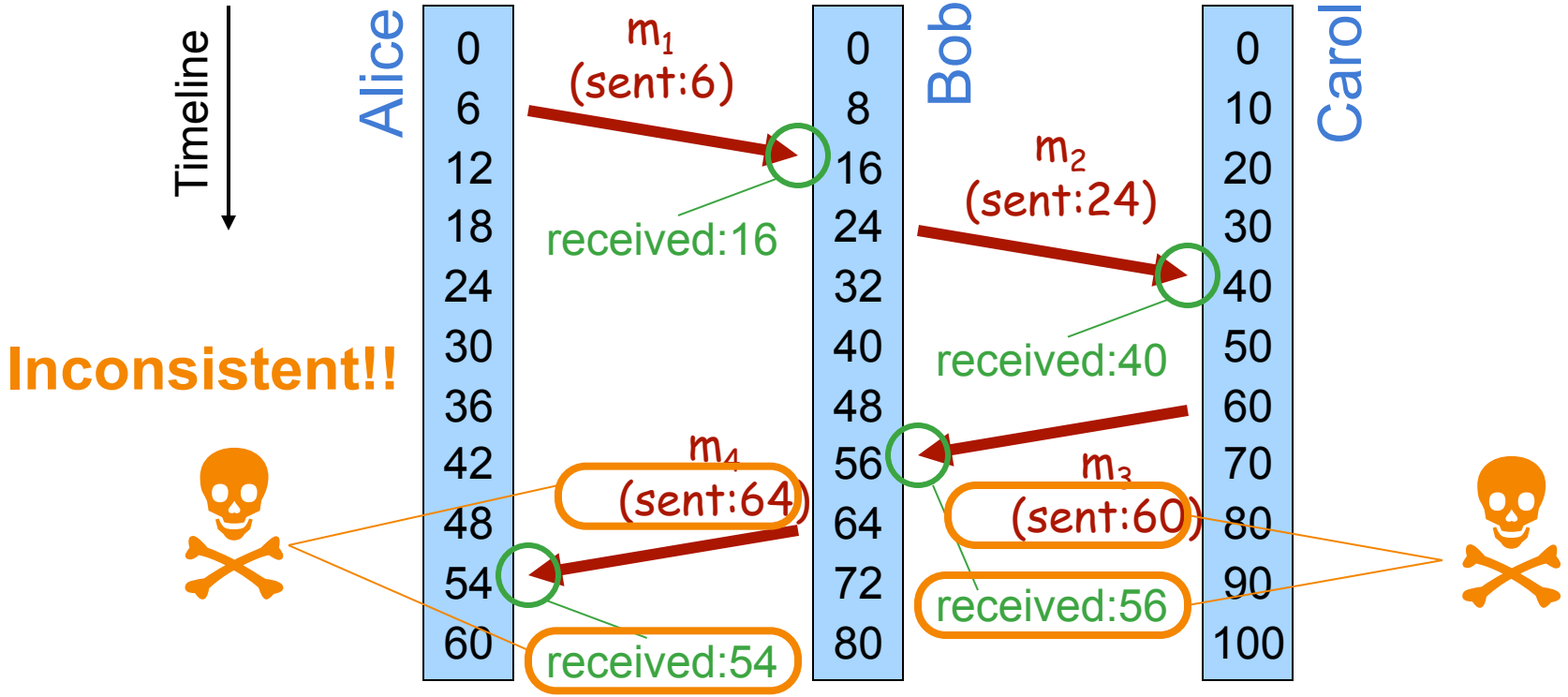
- In many cases “complete” synchronisation not needed
  - ➔ If 2 nodes don’t interact → no need to be in sync
  - ➔ No need for global agreement on time
  - ➔ Rather nodes need to agree on the order of events
- **Logical Clocks** [Leslie Lamport, 1978] do just this
  - ➔ All events (message sending & reception) are time-stamped
  - ➔ Time-stamps granted locally but respect “global” consistency
  - ➔ Inspired by **relativistic physics**: no global time base



unrelated events might be observed
in different orders by different observers
but causallity is respected: an effect always
observed before its cause

Global Time-Stamping (I)

- A first naive implementation that does not work
  - ➔ All processes have a local clock (a counter)
  - ➔ Messages time-stamped with the (local) sending time
  - ➔ Timestamps should have global meaning (but they don't)



Why it does not work

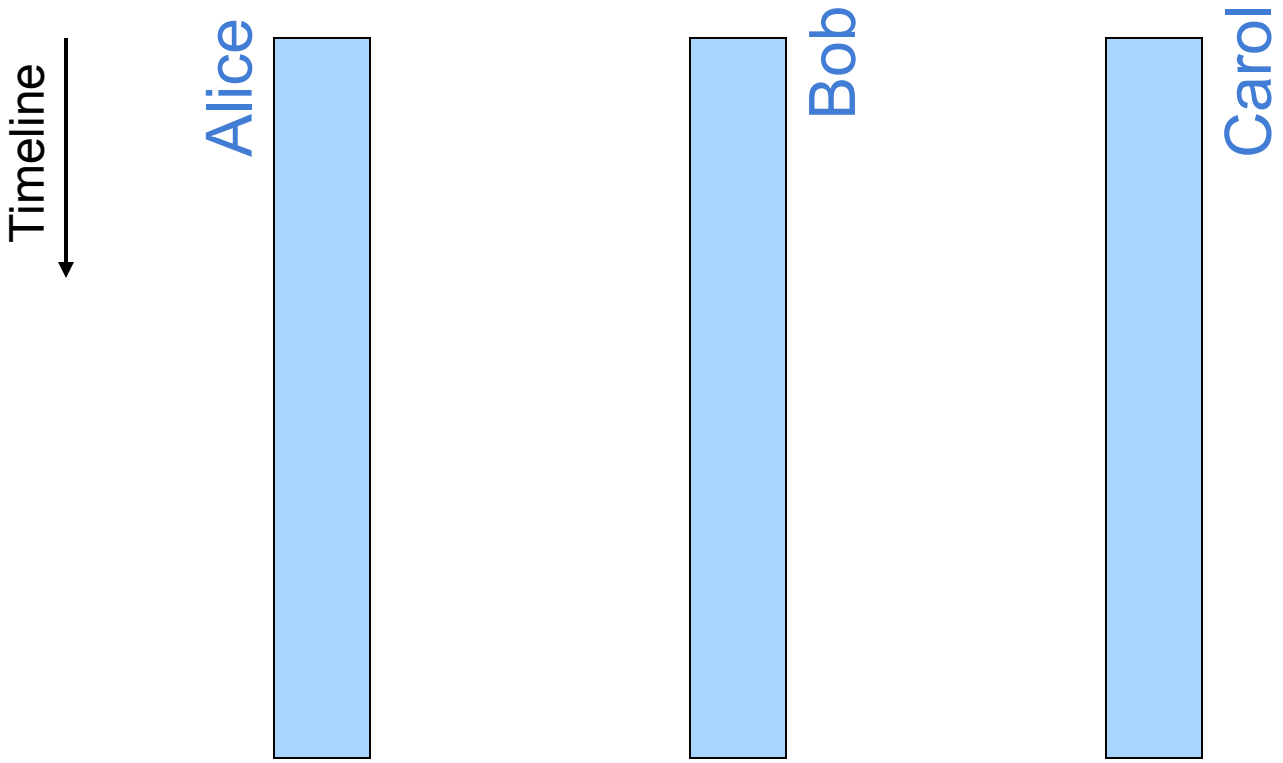
- Logical time-stamps should respect **causality**:
  - ➔ The reception of a message should not get a lower time-stamp than its sending: **C(sending m_i) < C(reception m_i)**
- This can be captured by the "**happened before**" relation
  - ➔ x "happened before" y (noted **x ← y**) **if and only if**
  1. x is the sending and y the reception of the same message
  2. or x and y happened on the same host and x occurred before y
  3. or there is an event z so that: **x ← z ← y** (transitivity)
  - ➔ **x ← y** intuitively means that x "**belongs to the past**" of y
- A **set of logical clocks** time-stamp events consistently if they respect the "**happened before**" relationship
  - ➔ I.e. if x ← y then C(x) < C(y)

Global Time-Stamping (II)

- Revised algorithm for logical time-stamping:
  - ➔ All processes have a local clock (a counter)
  - ➔ Messages time-stamped with the (local) sending time
  - ➔ On receiving a message with sending time **C(sending)**

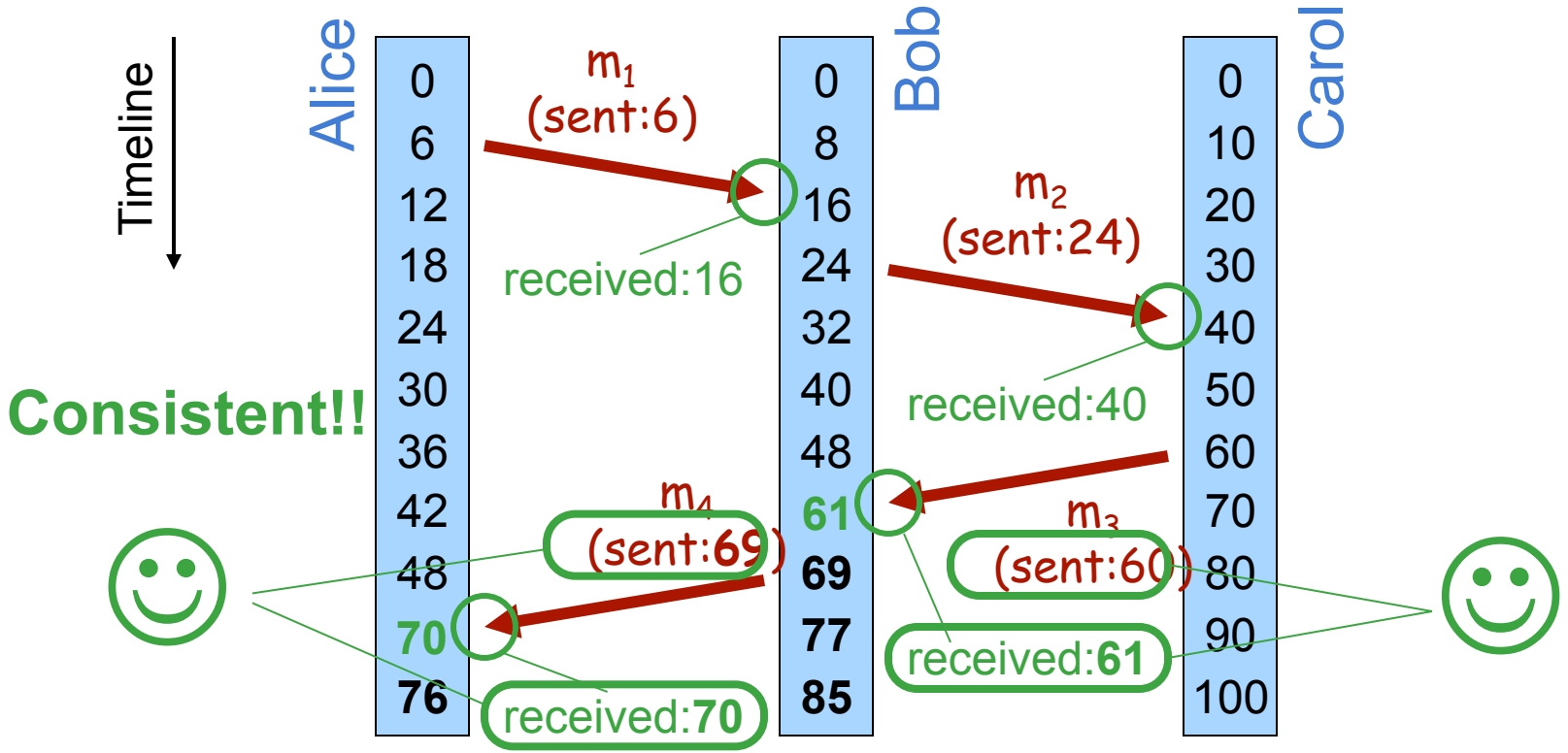
Global Time-Stamping (III)

- Back to our previous example:
  - ➔ every time a local clock could cause an inconsistency it is “moved” forward accordingly



Global Time-Stamping (III)

- Back to our previous example:
  - ➔ every time a local clock could cause an inconsistency it is "moved" forward accordingly



Global Time-Stamping (IV)

- Previous algorithms ensures that if event x "**belongs to the past of event y**", then x gets a **lower** time stamp that y
  - "belonging to the past" = "happened before" relation
  - some events might be **unrelated** (none belongs to the past of the other): timestamps are then **arbitrary**
  - 2 unrelated events might even get the same timestamp
  - To get **unique** timestamps, simply add the process IDs as decimal part to the local clock value
    - 6.1, 12.1, 18.1... for Alice, 8.2, 16.2, 24.2... for Bob, etc.
- Global time-stamping of events can be used as a **basis** for other algorithms
  - Can be used for **message ordering**

Group Comm

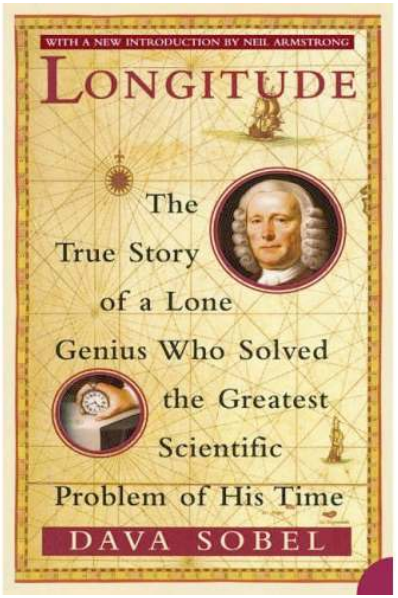
Wrapping Up

At the end of this Unit: You should be able to

- motivate the need for time and clock synchronisation in distributed systems
- compute the drift of an imperfect clock
- present and explain how to synchronise the clocks of two distributed computers
- explain Lamport's logical clocks algorithm and its key ideas

For more Information

- GPS
  - ➔ http://science.nasa.gov/science-news/science-at-nasa/2002/08apr_atomicclock/
- Maritime Clocks: John Harrison
  - ➔ http://en.wikipedia.org/wiki/John_Harrison
  - ➔ *Longitude: The True Story of a Lone Genius Who Solved the Greatest Scientific Problem of His Time* (by Dava Sobel)



SR (Systèmes Répartis)

Unit 5: Broadcast and Ordering in Distributed systems

François Taïani

Overview of the Session

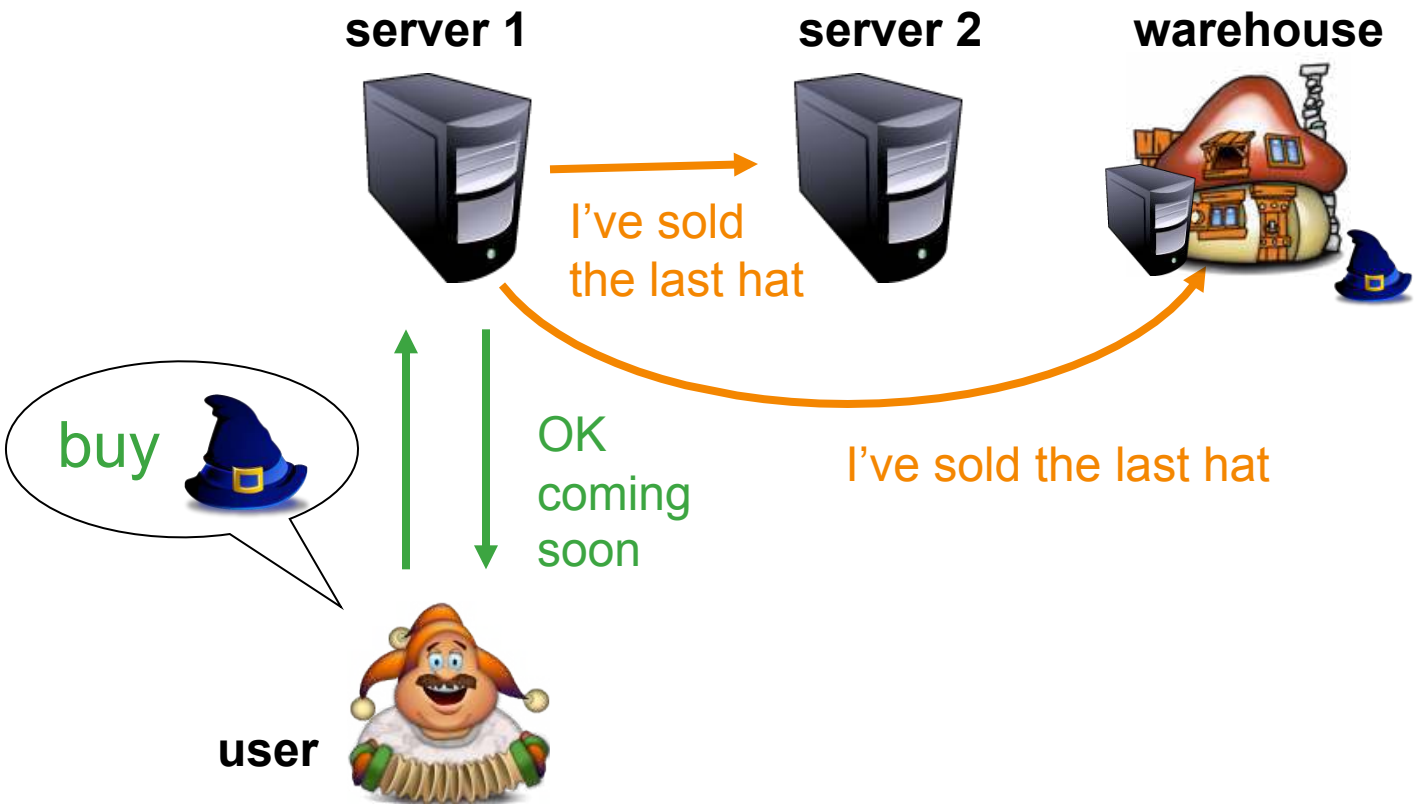
- Broadcast
- Reliability Guarantees
  - Unreliable broadcast
  - Reliable broadcast
  - Atomic broadcast
- Ordering Guarantees
  - Unordered
  - FIFO ordering
  - Total Ordering
  - Causal Ordering

Broadcast / Multicast

- In many distributed systems
  - → same messages send to several machines
  - → e.g.: reserving a resource (file, server, ID, key)
  - → e.g.: distributed replication
- Requires broadcast and multicast services
  - → broadcast: all nodes in the system receive the message (in some algorithms sender also receives its message)
  - → multicast: each message sent to a set of nodes (may vary)
- Provided programmatically through group communication
  - → more on it in Unit 8
  - → today fundamentals of algorithms

Example

- When replicating servers: collective coordination needed
  - ➔ either for fault-tolerance or scalability



Problems

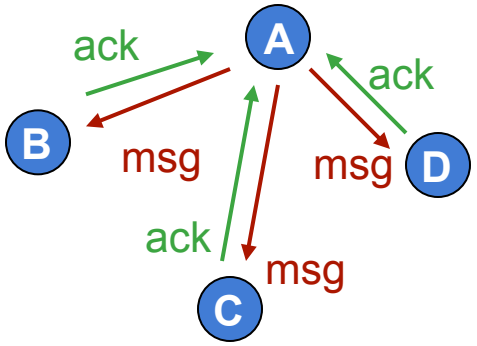
- Reliability? (against loss / crash)
- Scalability?
- Ordering?

Reliability Guarantees

- **Unreliable broadcast**
  - Message sent to all members and may or may not arrive
- **Reliable broadcast: Protection against faulty network**
  - Reasonable efforts are made to ensure delivery in spite of message losses
  - Can be based on positive or negative acknowledgements
  - No guarantees is the sender crashes during broadcast
- **Atomic broadcast: Protection against faulty participants**
  - All members receive message, or none do
  - Main issue: tolerate the sender's crash

Implementing Reliable Broadcast

- **1)** Originator sends message to each member of the group, and awaits acknowledgements (ACK)
- **2)** If some acknowledgements are not received in a given period of time, re-send message; repeat this n times if necessary
- **3)** If all acknowledgements received
  - ➔ then report success to caller



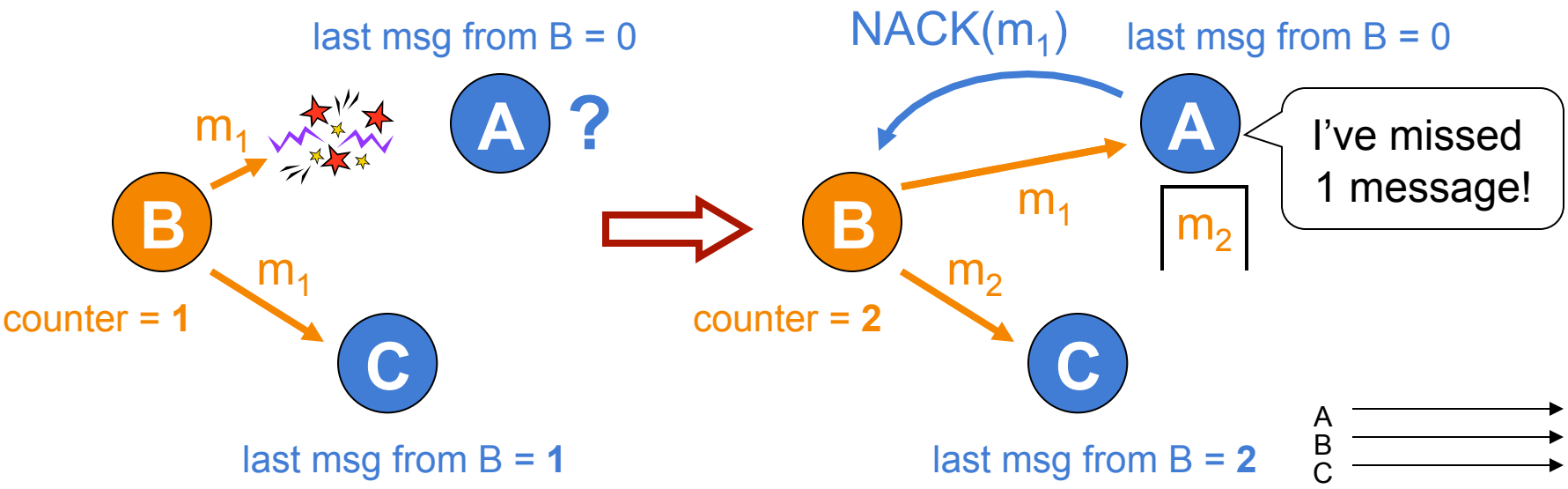
- **Works fine if:**
  - ➔ Network problems are transient (msgs eventually get through)
  - ➔ No crash (and no spurious behaviour)
- **Not very scalable: ACK explosion!**

Avoiding ACKs Explosion

- Using negative ACKs (abbreviated in NACKs)
  - ➔ If everything is fine the receiver does say anything
  - ➔ If a message is lost the receiver complains to the sender
- **Problem:** How do we know that a message should be there?

Avoiding ACKs Explosion

- Using negative ACKs (abbreviated in NACKs)
  - ➔ If everything is fine the receiver does say anything
  - ➔ If a message is lost the receiver complains to the sender
- **Problem:** How do we know that a message should be there?



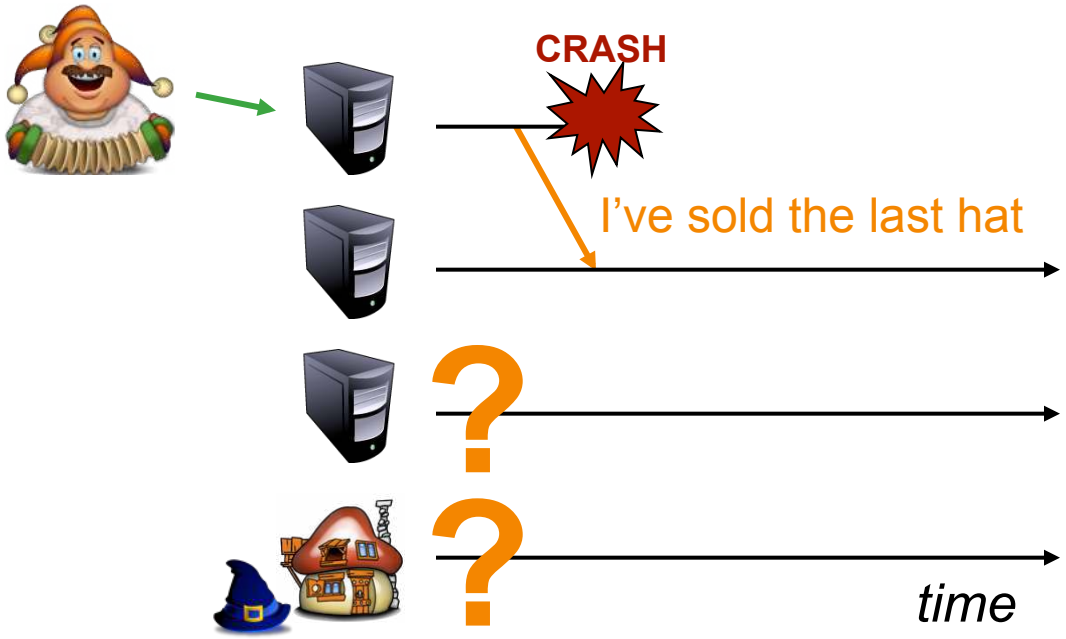
Homework: represent the above protocol on a time-sequence diagram

NACK Mechanism: Notes

- **ACK explosion** is avoided
  - ➔ A NACK explosion might still occur but is less likely
- **Garbage collection** problem
  - ➔ In theory sender should keep all past messages indefinitely
  - ➔ In practice past messages only kept for a “long enough” period
- More advanced schemes possible
  - ➔ Limiting NACK instances using **NACK broadcast**
  - ➔ **Hierarchical** Feedback Control

Death of a sender

- Reliable broadcast caters for network problems …
- But what if participants fail??



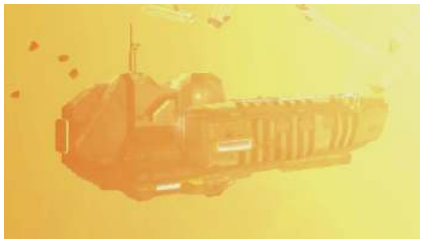
➔ The customer never gets his hat

➔ Worst: the surviving servers **disagree** on what has happened!!

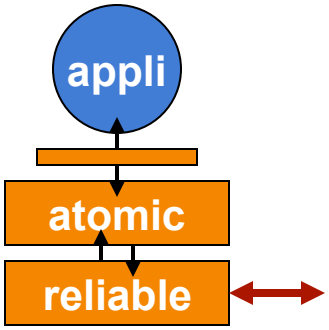

Enters Atomic Multicast

Atomic broadcast

- We need a **stronger** broadcast mechanisms
  - ➔ Either everybody in the group gets the message
  - ➔ Or nobody (who's surviving) does

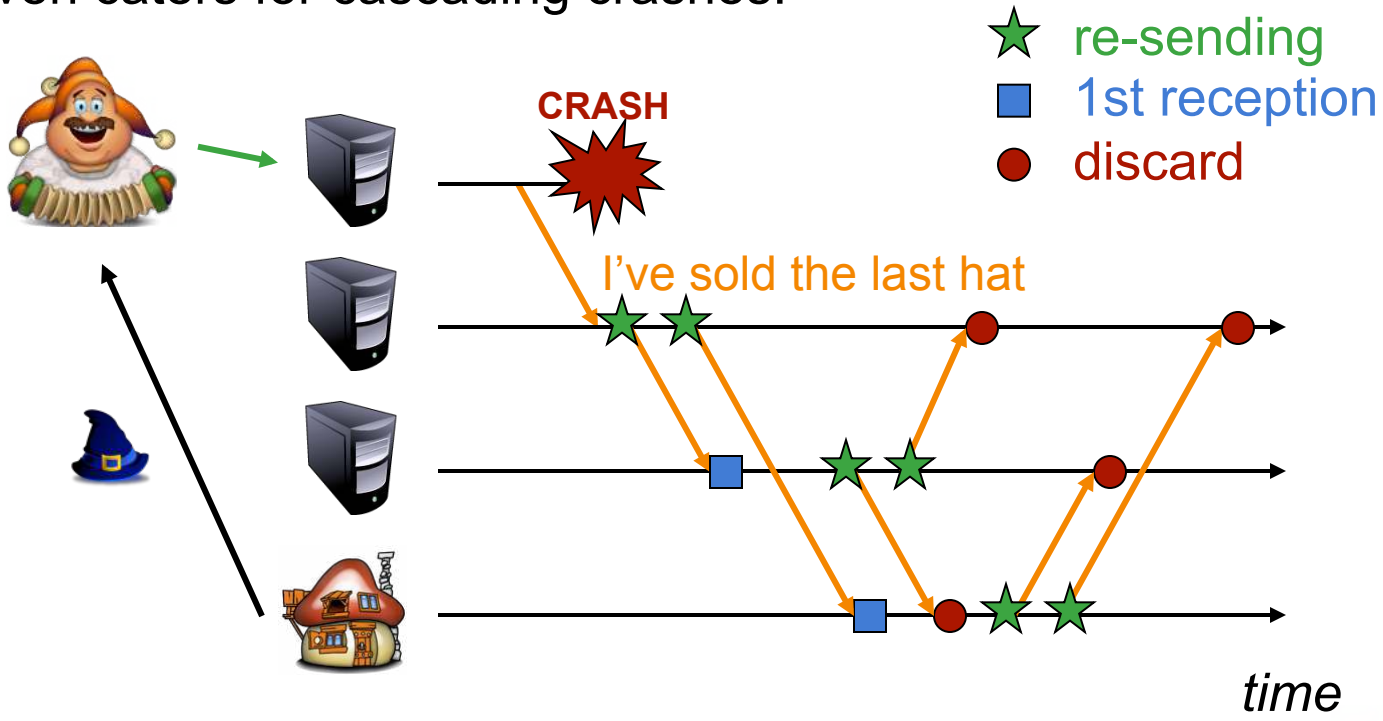


- All or nothing: **Atomic ('which can't be cut')**
- Atomic broadcast protocol
  - ➔ **Sender**: reliably broadcast message to rest of group
  - ➔ **All participants**:
    On receiving message:
    - **If first time I'm seeing this message** then {
      reliably broadcast again;
      and deliver to application; }
    - **else** { discard copy ; }



It's Atomic!

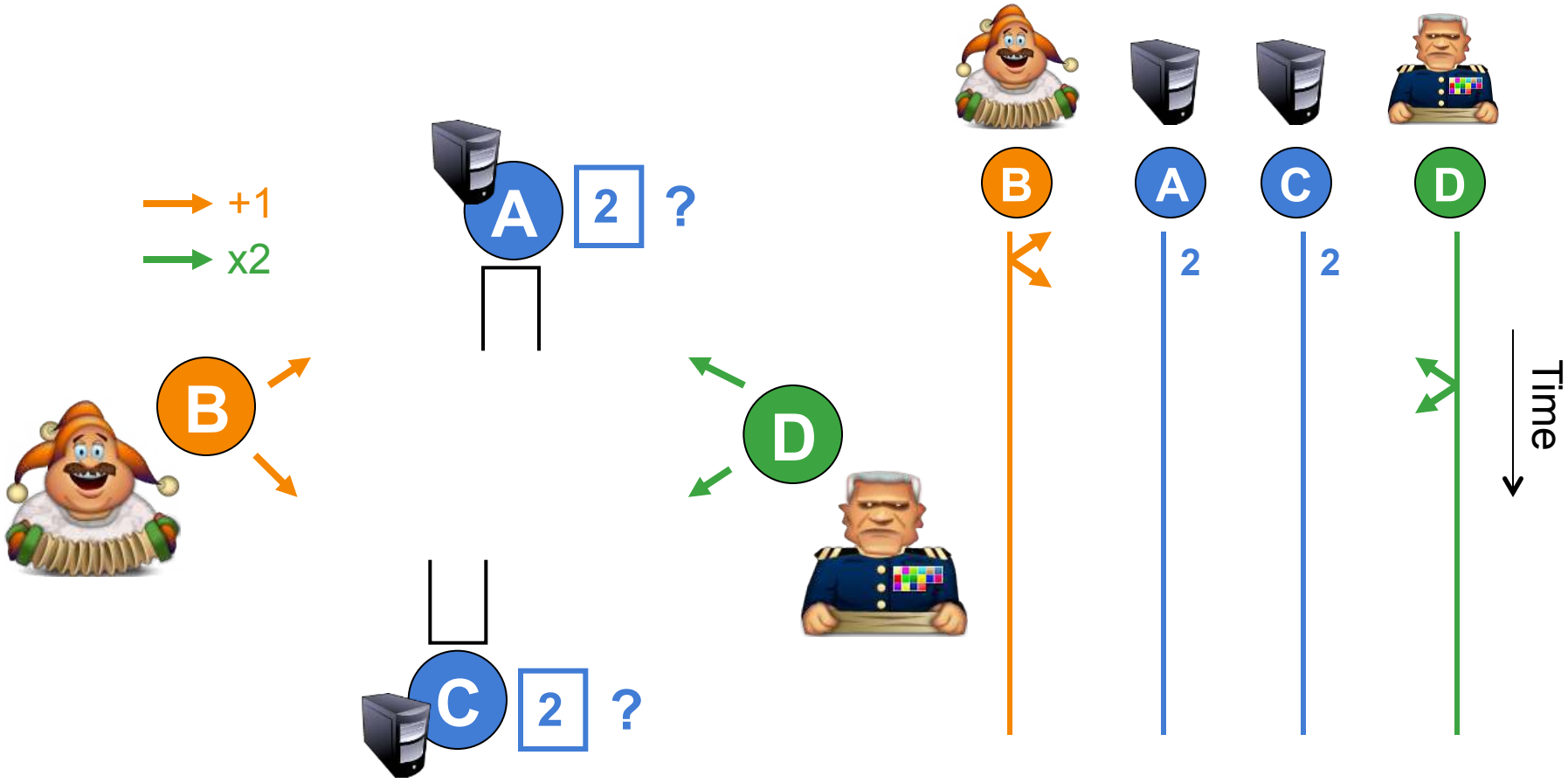
- Everybody gets the message in spite of sender's crash
  - ➔ Even caters for cascading crashes!



- There's a cost: Far more many messages (how many?)
  - ➔ Can be optimised (left as a home exercise) to send less in 'good' runs

Is Atomic all we need?

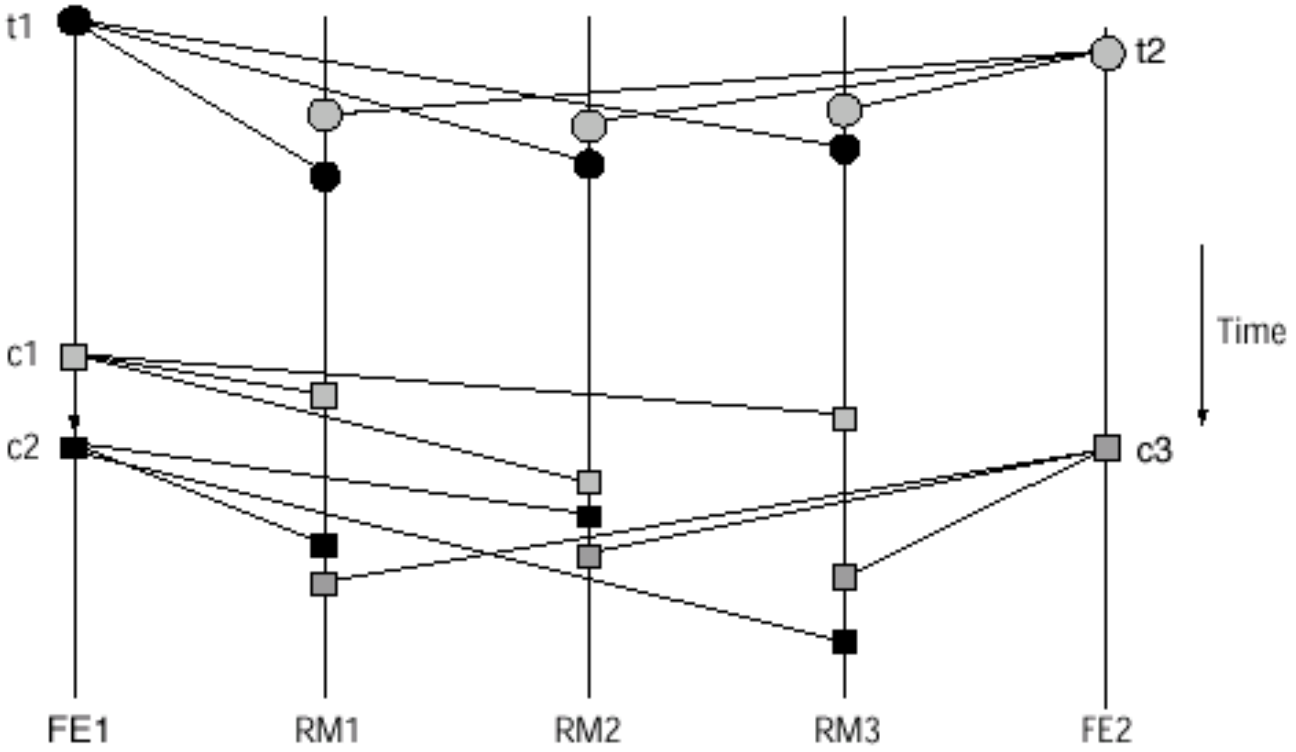
- **Ordering** often needed, and **not** provided by **atomic broadcast**
  - ➔ Participants need to **agree** on how messages should be ordered



Ordering Guarantees

- **Unordered broadcast**
  - No guarantees
- **FIFO (First In, First Out)**
  - Messages sent from the same process are delivered in the order they were sent at different sites
  - Messages sent from different processes may be delivered in different orders at different sites
- **Totally ordered broadcast / multicast**
  - Consider messages m1 and m2 broadcast (resp. multicast) by (potentially) different processes
  - If two process p and q receive m1 and m2, than they receive them in the same order
- **Causally ordered broadcast / multicast**
  - As above, except the ordering of m1 and m2 is only enforced consistently if a “happened-before” relationship exists between the messages

Total Ordering vs Causal Ordering

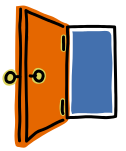


Notice the consistent ordering of *t1* and *t2*, the consistent ordering of the causally related operations *c1* and *c2*, and the arbitrary relative ordering of *c2* and *c3*, which are unrelated.

Implementing Total Ordering

- The sequencer approach (centralised)
  1. All requests sent to a sequencer, where they are given an ID
  2. The sequencer assigns consecutive increasing IDs
  3. Requested arriving at sites are held back until they are next in sequence
  - ➔ Problems: sequencer = bottleneck + single point of failure
- Other approaches
  - ➔ Assign timestamps from a (global) logical or physical clock
  - ➔ Distributed agreement to generate ids (consensus)

Total Ordering with Time-Stamps

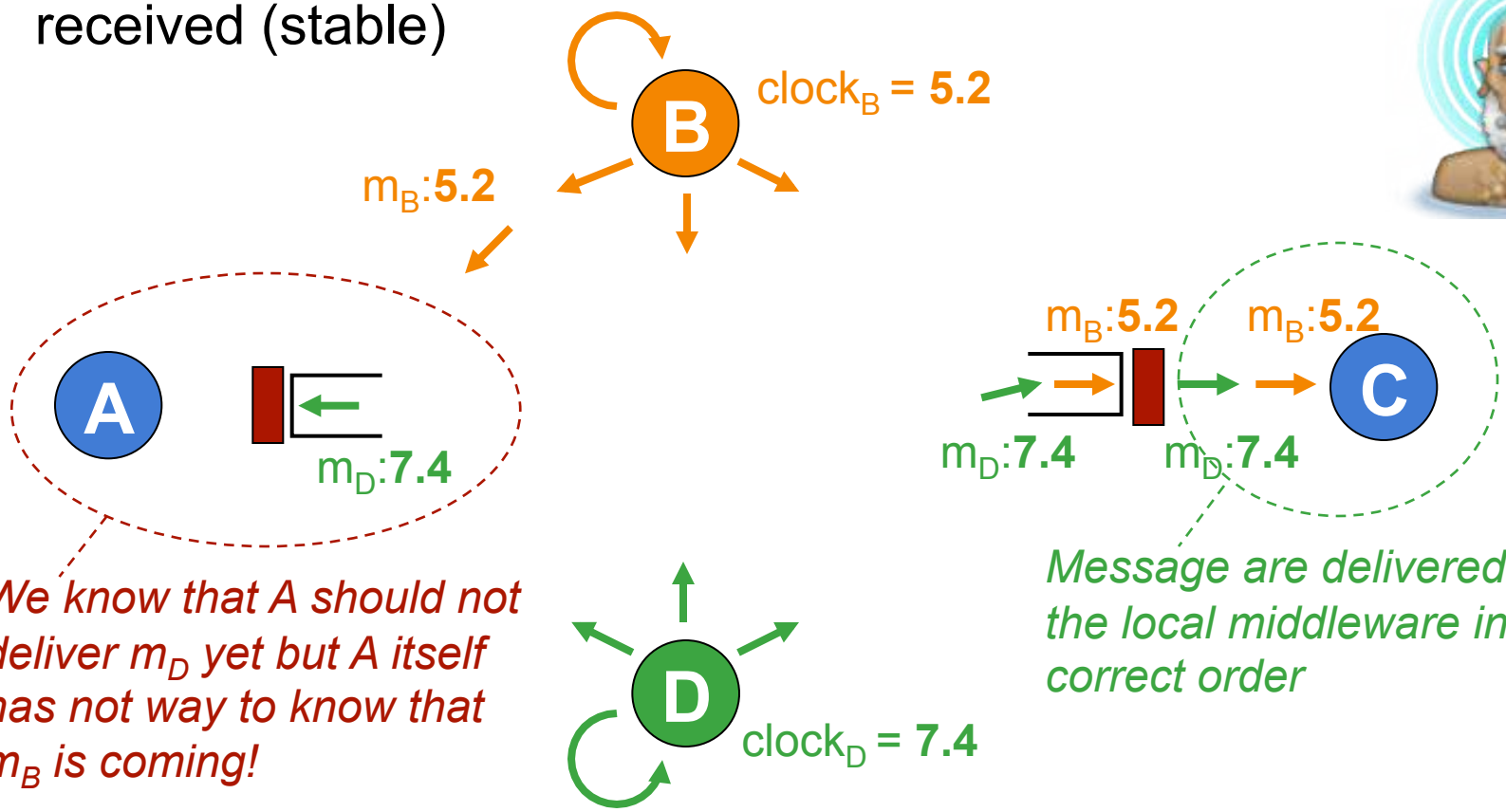


- Based on Lamport's logical clocks (cf. last unit)
  - ➔ Messages belonging to same broadcast operation are given the same timestamp
  - ➔ At destination messages delivered according to their timestamp
  - ➔ Logical clocks are needed to make sure delivery order is consistent with "happened before" relation

- The key of the algorithm is to solve the stabilisation problem

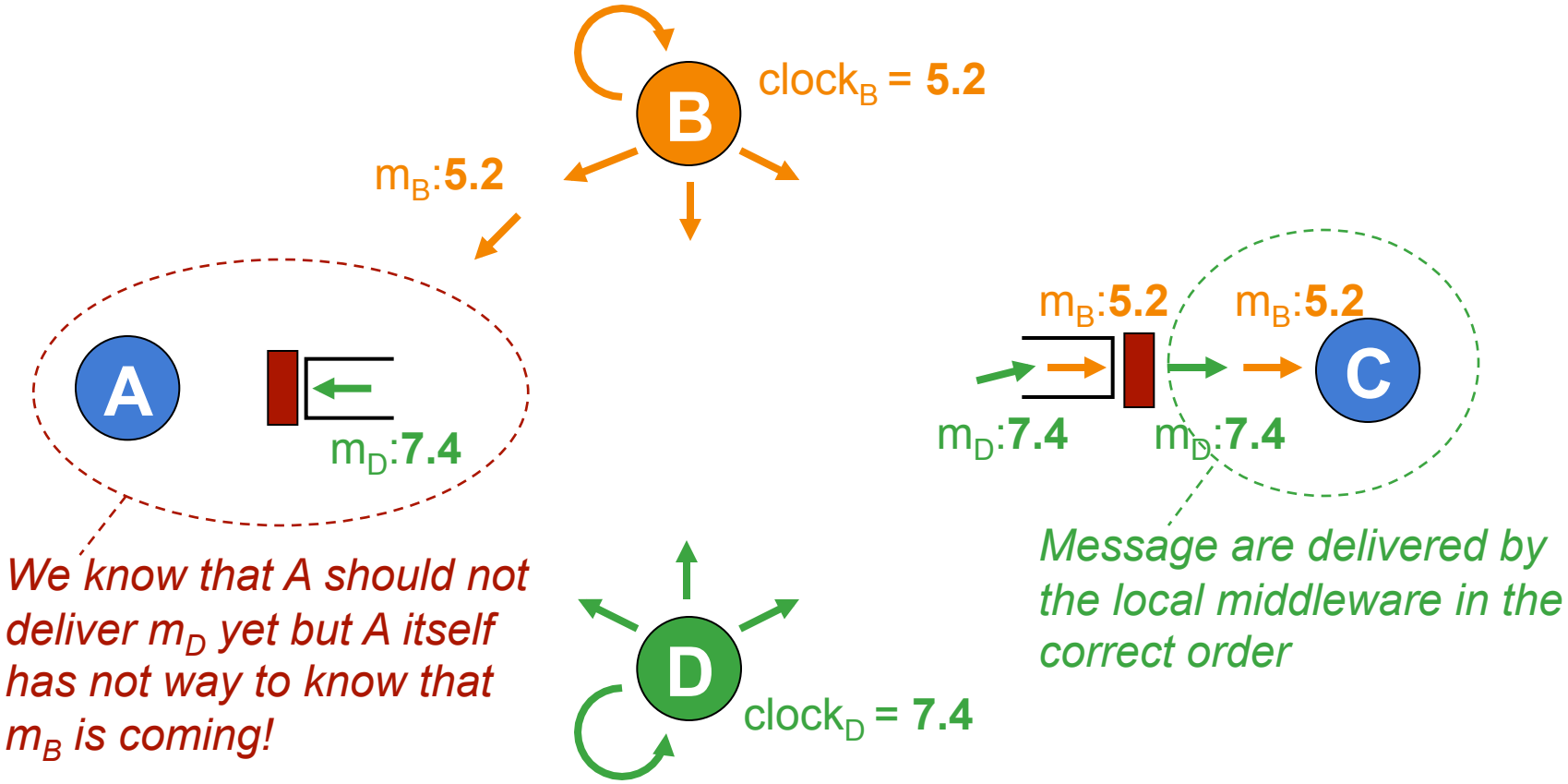
Delivery problem

- A message can only be delivered when the local middleware is sure no other messages with lower timestamps will be received (stable)



Delivery problem

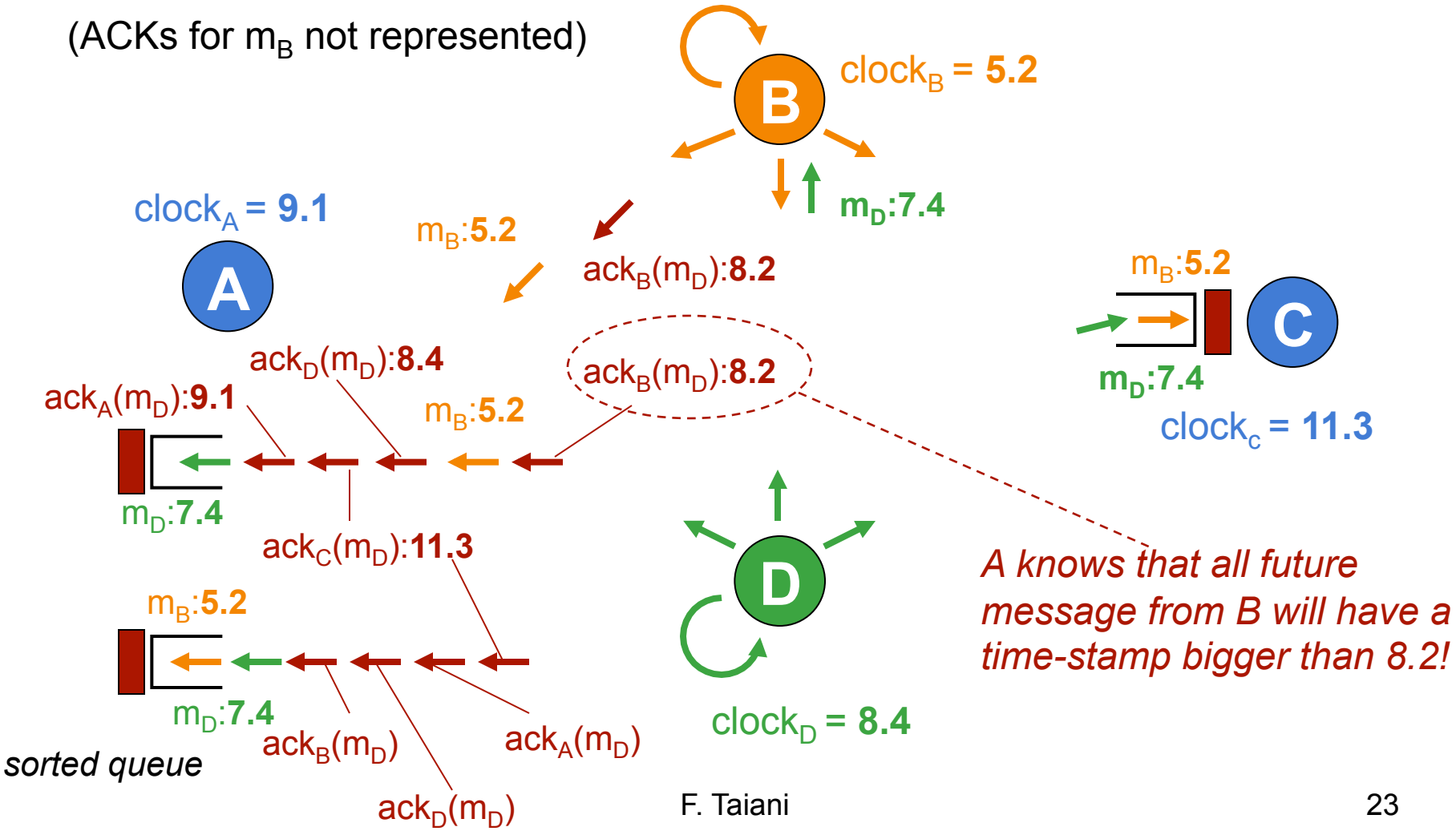
- **Problem:** How would you solve the stabilisation problem with Lamport Clocks?



Delivery Problem: A Solution

- Assumption: underlying network provides FIFO delivery
  - Messages from same sender delivered in same order as sent
  - If not provided can be implemented by consecutive IDs
- On receiving a message m_X a process
  - Puts the message in its local middleware queue
  - Broadcasts ACK(m_X) to all other group members
    - The acknowledgement is time-stamped with a higher timestamp than m_X
- A process delivers a message m_X to the application when
  - This message is at the top of the local middleware queue
    - I.e. it has the lowest timestamp of all the messages in the queue
  - And all ACK(m_X) for this message have been received

Total Ordering with Time-Stamps: Final Algorithm



Notes On Previous Algo

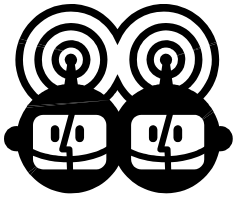
- Completely decentralised
- Rapidly becomes messy
  - ➔ Distributed algorithm are hard to design and understand
- Still not fault tolerant
  - ➔ All ACKs are needed. Any process crash blocks the system
  - ➔ Can be dealt with by using even more complex and more costly algorithms providing fault-tolerant consensus
- Not scalable
  - ➔ ACKs needed from all group members!
  - ➔ Can be improved if we assume message latency is bounded
  - ➔ Or if we assume messages are sent continuously

A Matter of Vocabulary

- For some "Atomic Broadcast" means
  - Atomic delivery
  - *And* total ordering

  } Atomic Broadcast
  - *Tanenbaum and Van Steen* use this definition

- **We** use the meaning of **Coulouris et al.**
  - "A-tomic" originally means "that can't be cut", all-or-nothing
  - For us **does not** imply total-ordering
- It's only a matter of vocabulary
  - The concepts and algorithms are the same, the names change
  - But be sure to know what someone means when he/she says "atomic broadcast"



Implementing Causal Ordering

- Total ordering implies causal ordering
  - ➔ so just use a total ordering algorithm
  - ➔ but: expensive, and not needed
  - ➔ some messages unduly delayed when they don't need to
- One solution: Vector Clocks (C. Fidge & F. Mattern 88)
  - ➔ a different kind of logical clock
  - ➔ not one number, but a vector: one entry per process
- Each process maintains its own vector clock
  - ➔ $C_A[B]$ entry of A's clock for process B
  - ➔ $C_A[B] = 10$ means
    "A has seen effect of B's behaviour up to B' logical time point n. 10"

Maintaining Vector Clocks

- All entries on all node start at 0
- Local entry $C_A[A]$ incremented for each internal event
- When sending a message at node A
  - ➔ $C_A[A]$ incremented
  - ➔ sending of message m time-stamped with $C(Send_m)=C_A$ (whole clock)
- When receiving message m at node B
  - ➔ $C_B[B]$ incremented
  - ➔ $C_B[X] = \max(C_B[X], TSend_m[X])$
  - ➔ reception of message time-stamped with $C(Receive_m)=C_B$

Comparing Vector Clocks

- If C1[] and C2[] are two vector clocks, C1 < C2 iff
  - ➔ C1[X] ≤ C2[X] for all X
  - ➔ there exists one Y such that C1[Y] < C2 [Y]
- Property:
  - ➔ event e happened before event f **iff** C(e) < C(f)
  - ➔ (quiz: What did we have we Lamport Clocks?)

Using Vector Clocks

- Vector clocks can be used to insure causal ordering
- Following work in a broadcast situation when
  - ➔ each node sends to all other nodes (but not itself)
- Slightly different design
  - ➔ internal events ignored, only sending and receiving
  - ➔ local clock not incremented when receiving (only when sending)
- Delay policy:
  when B receives m from A with C_m, delay delivery until
  - ➔ $C_B[A] = C_m[A]-1$
  - ➔ $C_B[X] \geq C_m[X]$ for $X \neq A$
- If not all messages are broadcast: matrix clocks needed!

Expected Learning Outcomes

At the end of this :

- You should be able to define what distributed broadcast is, and why it is relevant to distributed systems
- You should be able to explain and motivate the different reliability and ordering guarantees discussed today
- You should appreciate scalability issues involves in reliable broadcast
- You should be able to describe and analyse the reliable and atomic broadcast mechanisms we've presented
- You should be able to distinguish total from causal ordering
- You should have some insights on how total and causal ordering can be implemented using logical clocks

SR (Systèmes Répartis)

Unit 6: Synchronization in distributed systems

François Taïani

Introduction

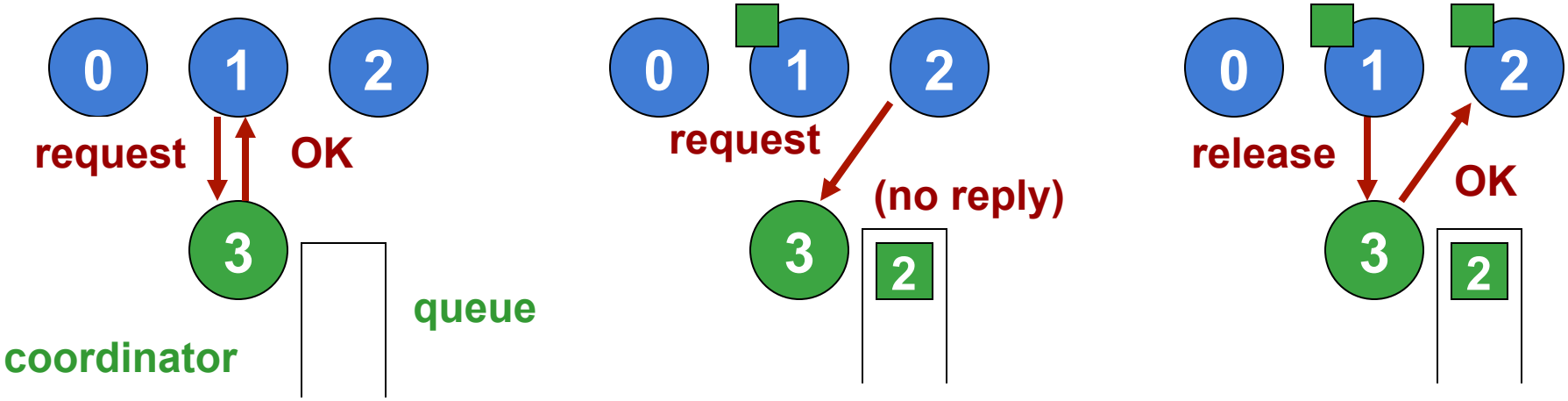
- Distributed execution -> coordination needed
  - ➔ but not more than needed
  - ➔ as potentially very costly
  - ➔ must handle potential problems of DS (notably failures)

- Different types of coordination for different properties
  - ➔ ordering (consistency of message order, Unit 5)
  - ➔ mutual exclusion (this unit)
  - ➔ distributed transactions (this unit)

Mutual Exclusion

- **Mutual exclusion**
  - ➔ only one process can use it or access it at a time
  - ➔ example: a printer, a flight seat, a communication link
- Mutual exclusion not limited to distributed systems
  - ➔ cf. concurrency programming (multithreading)
  - ➔ cf. databases and transaction processing
- Mutual exclusion very important in distributed systems
  - ➔ multiple hosts/processes executing in parallel
  - ➔ some orchestration needed to avoid chaos

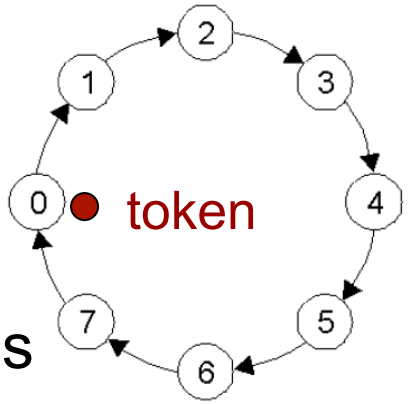
Mutual Exclusion: A Centralized Algorithm



- A central server processes all lock allocation requests
- ☺: Easy to realise
- ☹: Not scalable, not fault-tolerant
  - ➔ client 1 could die or keep the lock (can be addressed)
  - ➔ server is a point of failure & a bottleneck (replication can help, e.g. Chubby at Google)

A Token Ring Algorithm

- Distributed processes organised in a ring
- A logical token circles the ring
  - ➔ only one process has the token at any time
- If a process want to mutual-exclusive access
  - ➔ **waits** to get the **token**
  - ➔ **keep** the **token** as long as mutual access needed
  - ➔ to release mutual access, **release** the **token**
- ☺: Process crashes easier to handle
  - ➔ "just" detect and rebuild the ring
- ☹: No bottleneck but not really scalable (ring size)
- ☹: Token can get lost (solutions exist)



(Distributed) Transactions

- We must first understand what a transaction is
  - ➔ see database module
- **Transaction**: **Synchronisation** mechanisms to access (and modify) shared data concurrently
  - ➔ heavily used in **databases**
- Similar to a **commercial business "transaction"**
  - ➔ first negotiations on what is to be done
  - ➔ at any point during negotiations any party may back out
  - ➔ if agreement is found, all parties need to commit (contract)
- Computer Science: the same
  - ➔ between a client and a data repository (database usually)

(Distributed) Transactions

- A transaction engine provides the following operations
  - ➔Begin_Transaction: start a transaction
  - ➔End_Transaction: end transaction + try to commit
  - ➔Abort: kill transaction & forget everything about it
- Examples of client behaviour:

Begin_Transaction
  book flight MAN-NYC → OK
  book hotel NYC 1 week → OK
End_Transaction

Begin_Transaction
  book flight MAN-NYC → OK
  book hotel NYC 1 week → full
Abort

ACID Properties

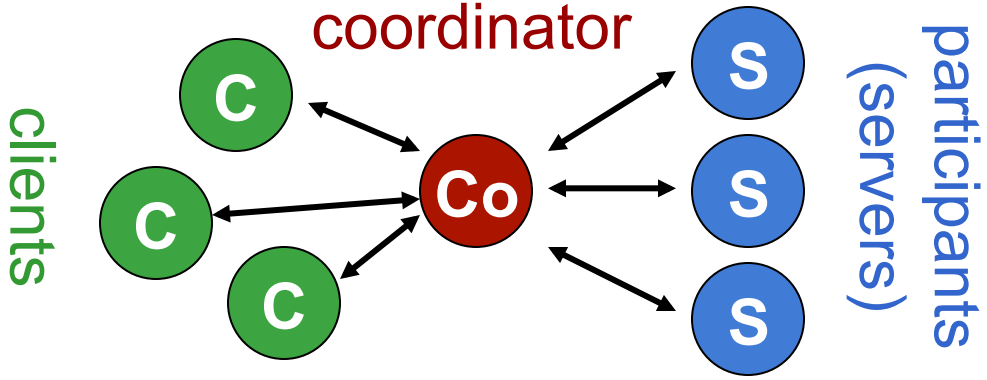
- ACID central to the DB course

- Backstage the transaction engines ensures that
  - ➔transactions are atomic: all or nothing
  - ➔consistent: leave the system in a valid state
  - ➔isolated: don't interfere with each other
  - ➔durable: once successful, changes permanent

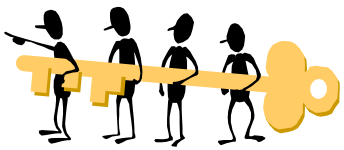
ACID

Distributed Transactions

- What is a distributed transaction?
  - ➔ A transaction where operations involve multiple servers (e.g.: **airline** + **hotel**)



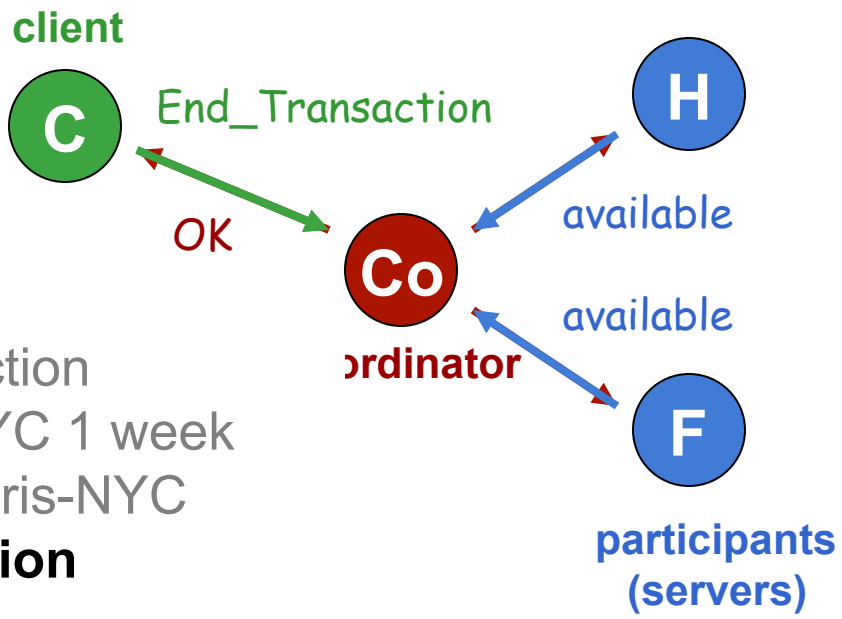
- Issues
  - ➔ Need distributed algorithms for concurrency control and recovery schemes mentioned above
  - ➔ Must deal with added difficulty of distributed deadlock
  - ➔ Crucial issue of **atomic commit protocols**



The 2-Phase Commit Protocol

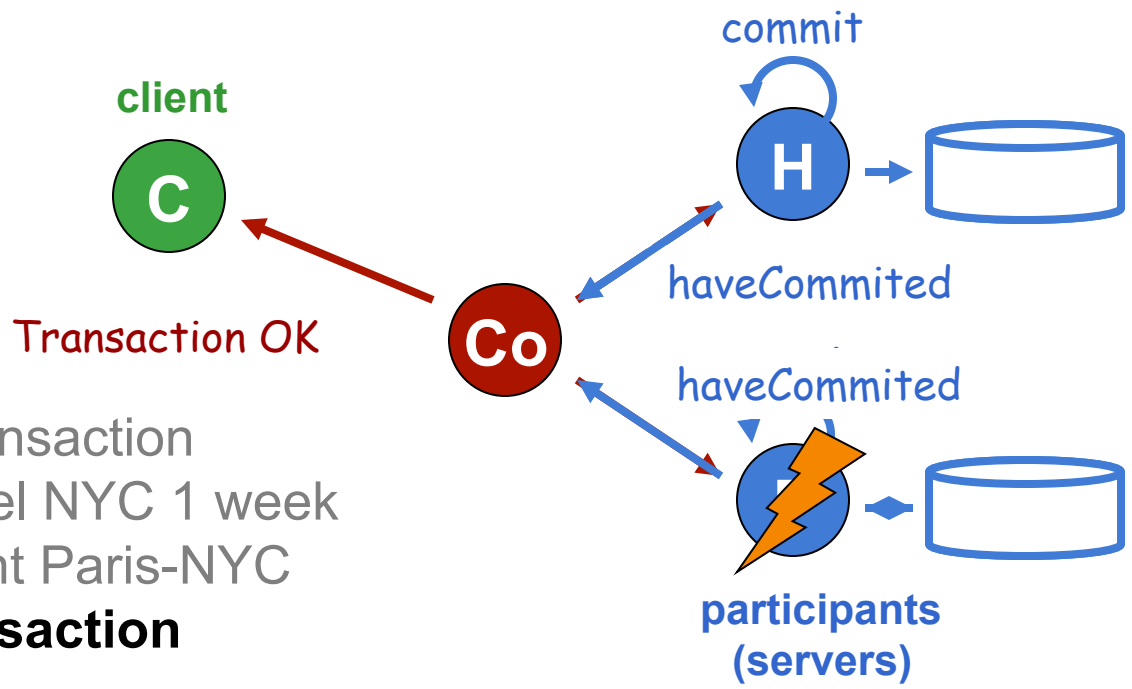
- **Phase 1** (voting phase):
  1. coordinator sends a canCommit? request to all participants
  2. on receiving canCommit? each participant replies yes or no
     if yes it saves the transaction state into **permanent storage** *before* sending the yes reply
     if no it aborts immediately
- **Phase 2** (completion according to vote)
  3. **if** all participants answer yes (including the coordinator) the coordinator sends doCommit to all participants
     else coordina[tor] sends doAbort to participants that voted yes
  4. yes-voting participants wait for a doCommit or doAbort from coordinator, and act accordingly. In case of doCommit they send a haveCommited acknowledgement

Distributed Transaction: An Example



Begin_Transaction
  book hotel NYC 1 week
  book flight Paris-NYC
End_Transaction

(a) Commit Succeeds

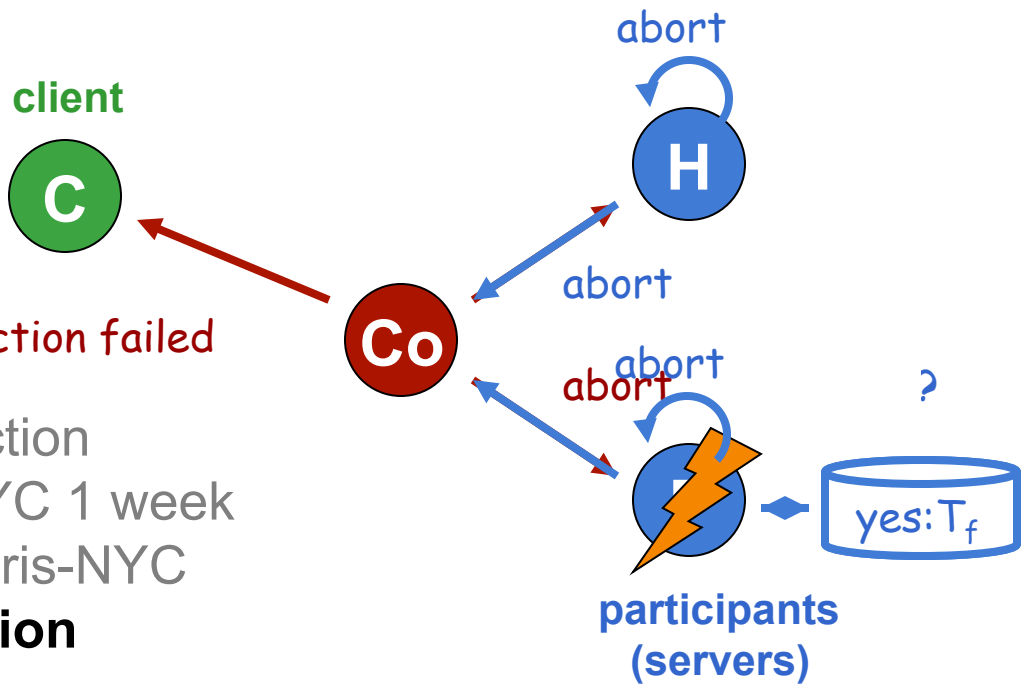


Begin_Transaction
  book hotel NYC 1 week
  book flight Paris-NYC
End_Transaction

Important Points

- Once the coordinator has sent **doCommit** only a commit is possible
- If one of the participants fails:
  - Transaction is **blocked** until the participant resumes
  - **Consistency** is ensured because of transaction state being saved on **stable storage**
  - On resuming the failed participants check its disk to know which transactions to ask the coordinator about
- Many details *not* represented
  - The coordinator uses stable storage as well against failure
  - Distributed locking protocol omitted
  - Typically *multiple* clients performing transaction at the same time

(b) Commit Fails



Begin_Transaction
  book hotel NYC 1 week
  book flight Paris-NYC
End_Transaction

Commit fails: important points

- Same basic mechanisms as when all agree
- Except here **no need to wait** for failed servers:
  - ➔ It is their responsibility to catch up
- Failed servers still need to **check** with coordinator
  - ➔ They do not know the outcome of the transaction
  - ➔ The transaction could have succeeded

Transactional Middleware: Transaction Processing Monitors

- Standards
  - ➔X/Open DTP, OMG Corba OTS, J2EE
- Key products
  - ➔IBM's CICS and Encina (Transarc)
  - ➔Oracle's Tuxedo
  - ➔Microsoft's MTS (included in COM+)
  - ➔SUN's Enterprise JavaBeans (EJB)
  - ➔JBoss, JOnAS (open source J2EE products)

Summary

- Two types of coordination for distributed systems
  - ➔ mutual exclusion
  - ➔ distributed transactions
- Solution for mutual exclusion
  - ➔ centralised
  - ➔ token-based
  - ➔ versions we have seen = no fault tolerance!
- Solution for distributed transactions
  - ➔ 2PC (OK if coordinator not permanently failed)
  - ➔ better protocol (not seen): 3PC

SR (Systèmes Répartis)

Unit 8: Fault Tolerance I

François Taïani

Overview of the session

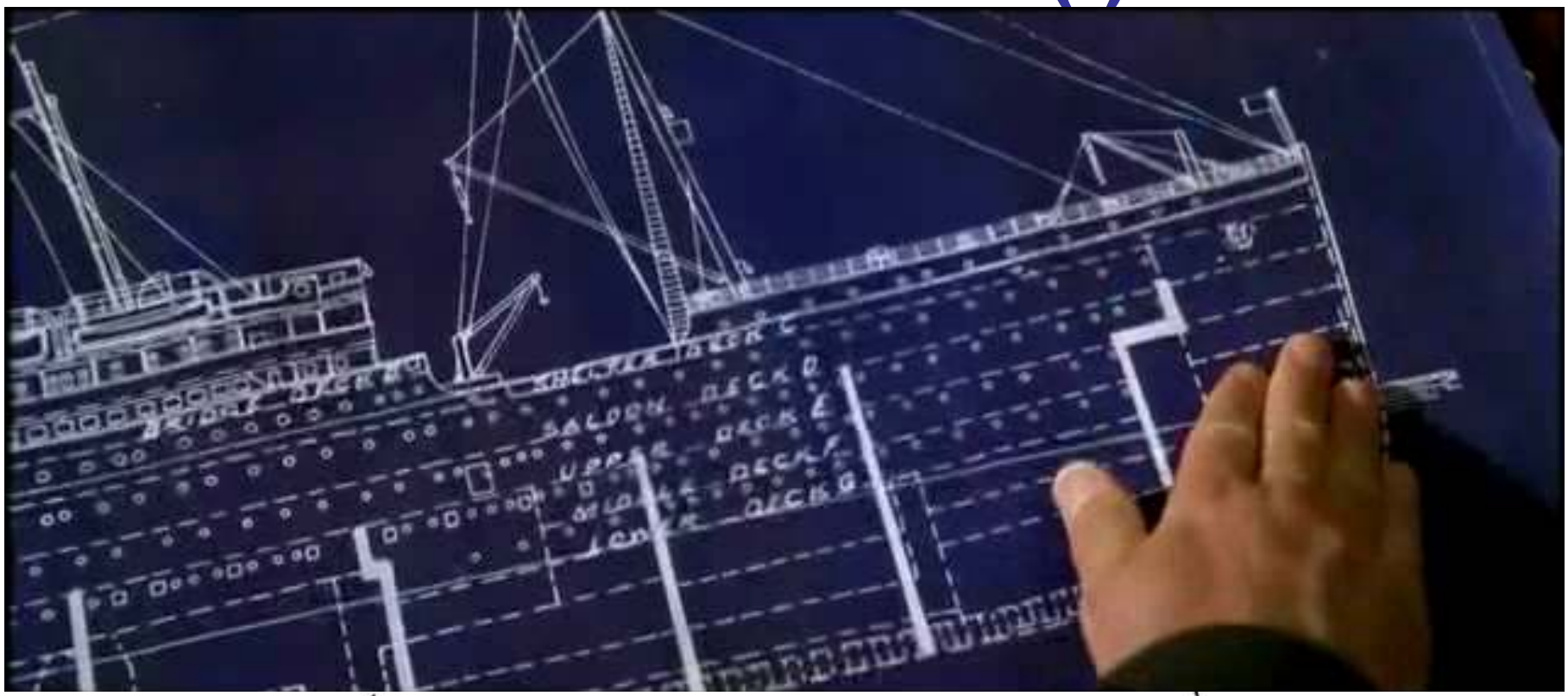
- A famous example (video)
- Basic dependability concepts
- Fault-tolerance at the node level: Replication
- Fault-tolerance at the network level: TCP
- Fault-tolerance at the environment level: Data centres

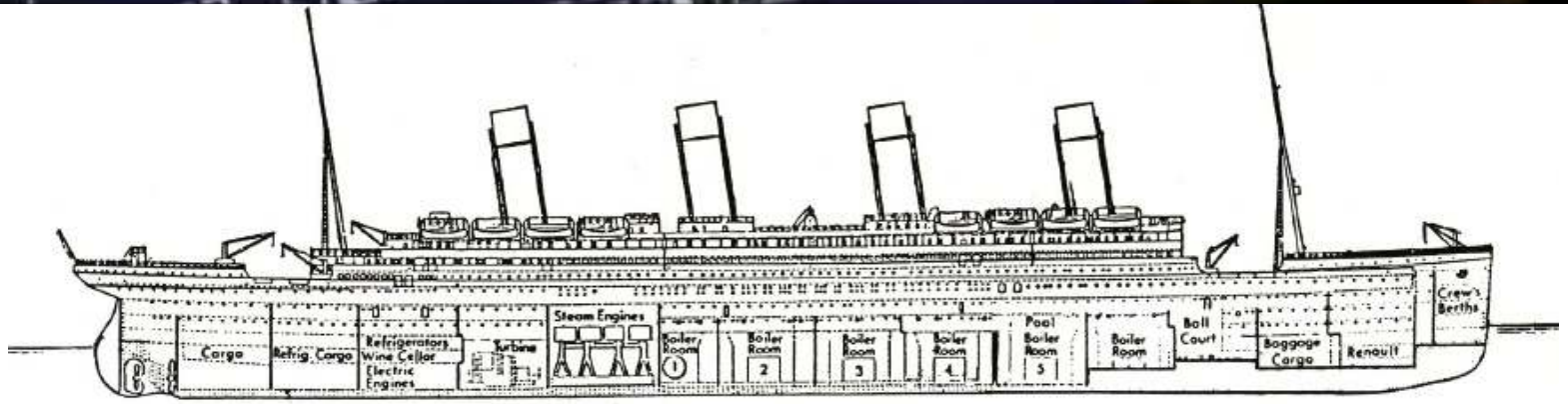
Associated Reading: Chapter 7. Fault Tolerance of Tanenbaum & van Steen; Chapter 15 and 18 Coulouris & al (2012 edition)

A Famous Example

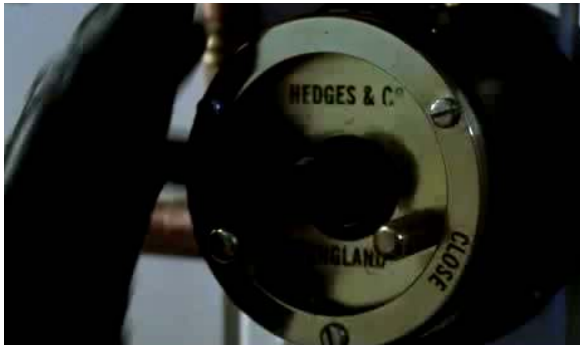
- April 14, 1912

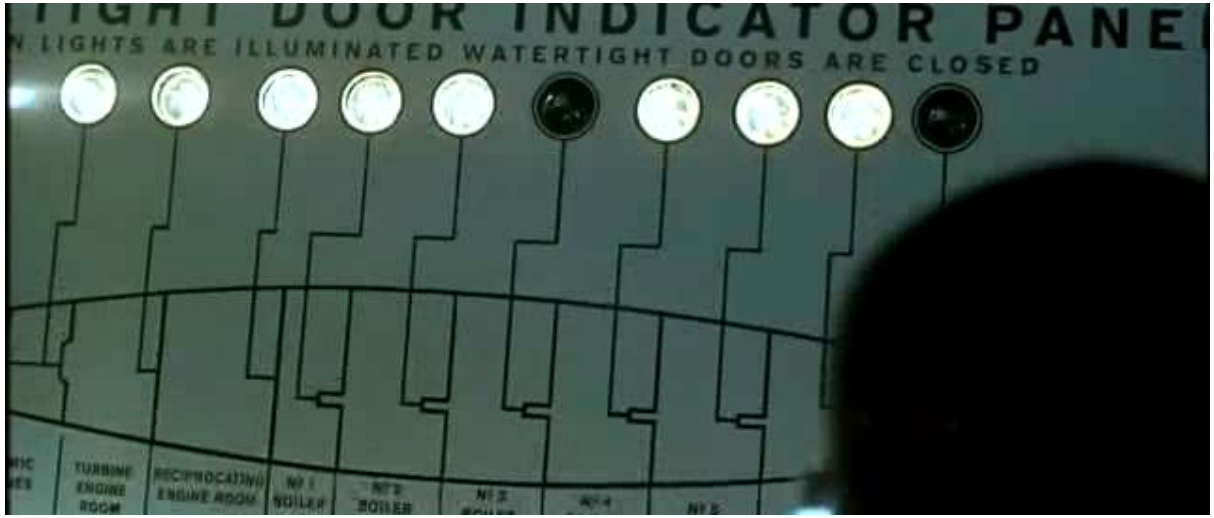
Video Clues (I)





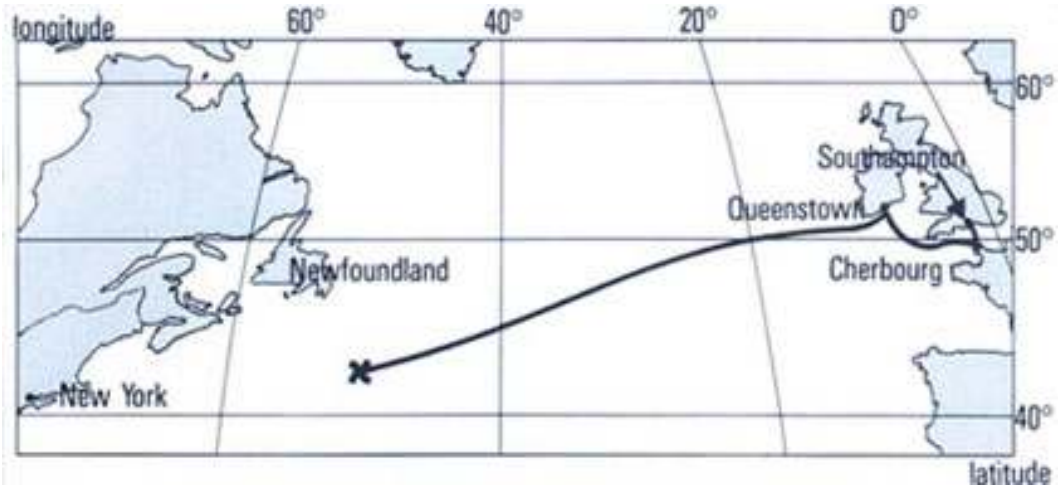
Video Clues (II)





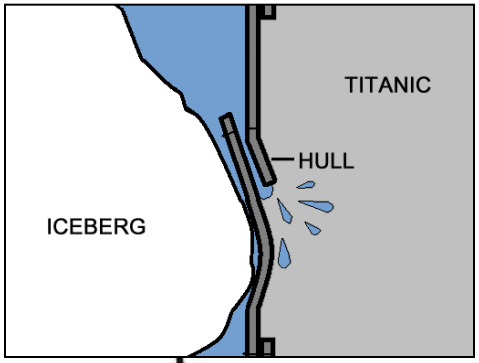
A Major Disaster

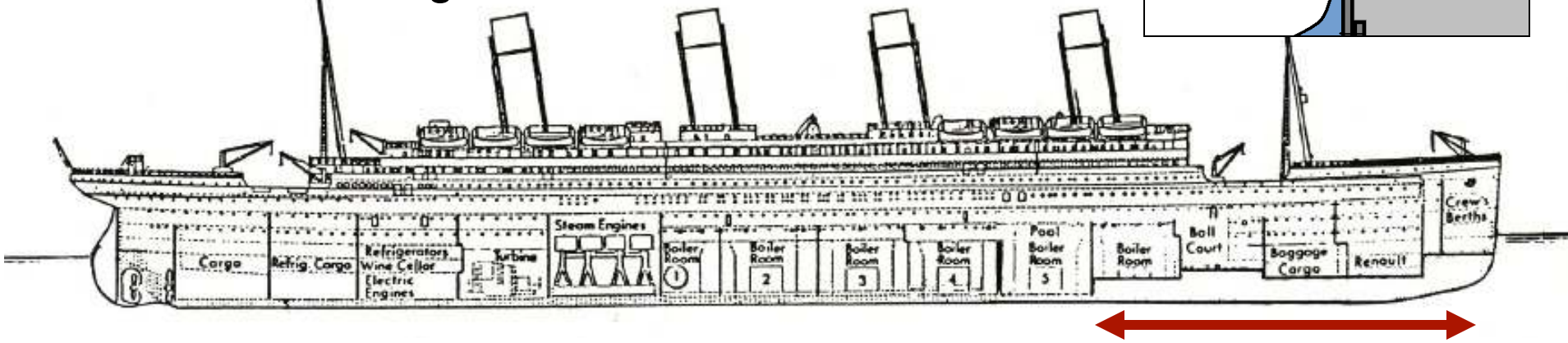
- The worst peacetime maritime disaster
  - ➔ more than 1500 dead
  - ➔ only 706 survivors
- The wreck only discovered in 1985
  - ➔ 2 1/2 miles down on the ocean floor



The Titanic and Fault-Tolerance

- How fault-tolerant was the Titanic?
  - ➔ best techno of the time: deemed “practicably” unsinkable
  - ➔ 16 watertight compartments
  - ➔ *Titanic* could float with its first 4 compartments flooded
- Some flaws in the design:
  - ➔ poor turning ability
  - ➔ bulkheads only went as high as E-deck
  - ➔ not enough lifeboats!





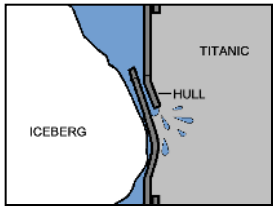
What does It Teach Us?

- Good to plan ahead for problems
  - ➔ Like collision and hull damage for a ship
- Replication / **redundancy** is the key to survival
  - ➔ Titanic contained 16 watertight compartments
  - ➔ Could still float with the first 4 flooded (& other combi.)
  - ➔ But **always a limit** to what you can tolerate
- But you also need **diversity** / independence
  - ➔ Compartments are *watertight*
  - ➔ Critical issue: **prevent water propagation** (deck E pb)
- The case of the titanic applicable to many other sys.
  - ➔ Generic ideas still apply

Dependability Concepts

- **Failure**: a failure of the delivered service.
  - ➔ Titanic: sinking and killing people
  - ➔ Distributed System: stop functioning correctly

- **Errors**: *erroneous internal state* that can lead to a failure
  - ➔ Titanic: damaged hull, water in compartment
  - ➔ DS: incorrect state, inconsistent internal behaviour



- **Fault**: a pb that *could cause* errors, & ultimately a failure
  - ➔ Titanic: the iceberg, design flaws
  - ➔ Distributed System: hardware crash, software bug

Quality of Service

- Notion of **'failure'** is a grey area!
  - ➔ E.g. a web server taking 5 min. to deliver each request
  - ➔ It fulfils its functional specification (delivering web pages)
  - ➔ It works better than a server that would not reply at all
  - ➔ But can it be considered to be functioning correctly?
- Lots of **intermediary** situations

optimal service  complete failure

  - ➔ **Quality of Service**: how 'good' a system is
    Multiple facets: latency, bandwidth, security, availability, ...
  - ➔ 'Failure' usually means a service becomes unusable
  - ➔ Goal: highest QoS in spite of faults (and at the lowest cost)

Graceful Degradation

- **Ideally** internal failure/fault -> completely masked
  - ➔ A router crashes but you do not even notice
- In practice often not possible
  - ➔ Essentially because of cost + fault assumptions
- Goal is usually **to minimise impact** of faults on users
  - ➔ Avoid complete failure
  - ➔ Maintain highest possible QoS in spite of faults
- This is called "**Graceful Degradation**"
  - ➔ Crucial: leaves time to administrators to react
  - ➔ Service still usable, albeit with a lower **QoS**
  - ➔ Requires trade-off between business impact and costs

Distribution and Fault Tolerance

- **Fault tolerance** requires **distribution**
  - ➔ For replication and independence of failure modes
  - ➔ I.e. to tolerate earthquakes
    → not all servers in San Francisco

- **Distribution** requires **fault-tolerance**
  - ➔ Large distributed systems
    → partial node failures unavoidable
  - ➔ Single points of failure
    → to be avoided as much as possible

What Can Go Wrong in a DS?

- **Nodes** (servers, clients, peers)
  - Faulty hardware → crash or data corruption
  - Power failure → crash (and sometimes data corruption)
  - Any "physical" accident: fire, flood, earthquake, ...

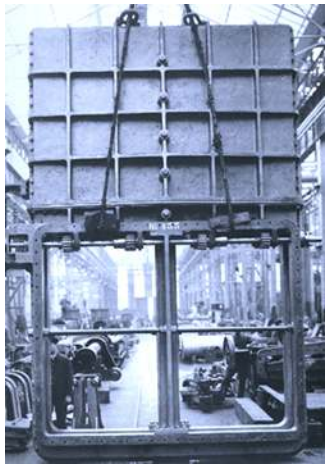
- **Network**
  - Same as nodes: Routers gateways: whole subnet impacted
    Name servers: whole name domain impacted
  - Congestion (dropped packets)

- **Users / People**
  - Involuntary mistakes
  - Security attacks (external & *internal*, i.e. by legitimate users)

Main Steps of Fault Tolerance

- Error **Detection**
  - ➔ Detecting that something is wrong (eg. water in the keel)
  - ➔ The earlier the better
  - ➔ The earlier the more difficult

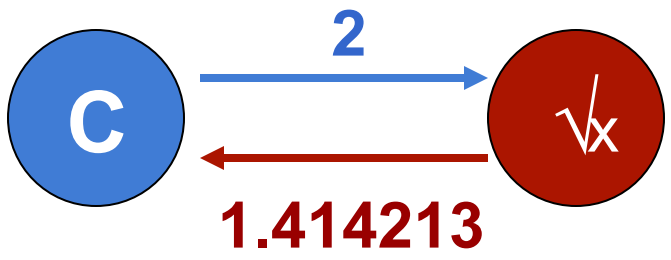
- Error **Recovery**
  - ➔ Preventing errors from propagating (watertight door)
  - ➔ Removing errors
  - ➔ Does not mean root cause (fault) removed



- Fault **treatment** (usually off-line, manually)
  - ➔ Fault Diagnostic
  - ➔ Fault Removal

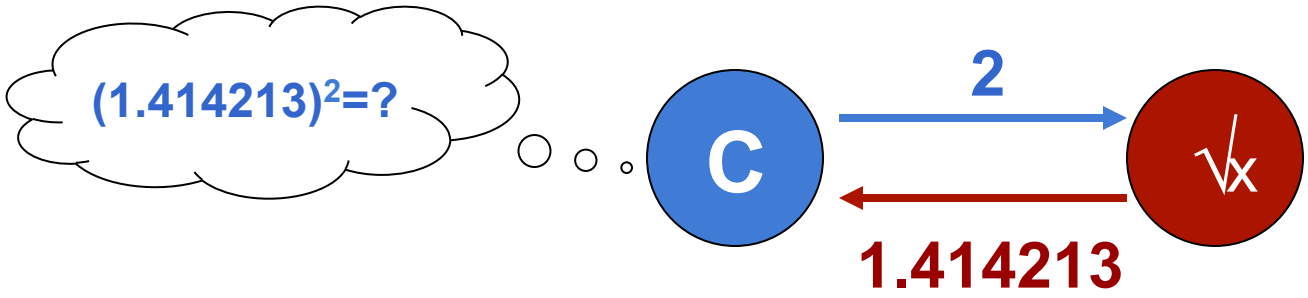
Error Detection

- Error detection in the Titanic quite “easy”
  - ➔ No water should leak inside the boat (safety invariant)
- Distributed Systems: more difficult
  - ➔ How would you detect errors in the following service?



Error Detection

- Error detection in the Titanic quite “easy”
  - ➔ No water should leak inside the boat (safety invariant)
- Distributed Systems: more difficult
  - ➔ How would you detect errors in the following service?



  - ➔ In some cases sanity or correctness check possible
  - ➔ In many cases some form of redundancy needed

Error Recovery

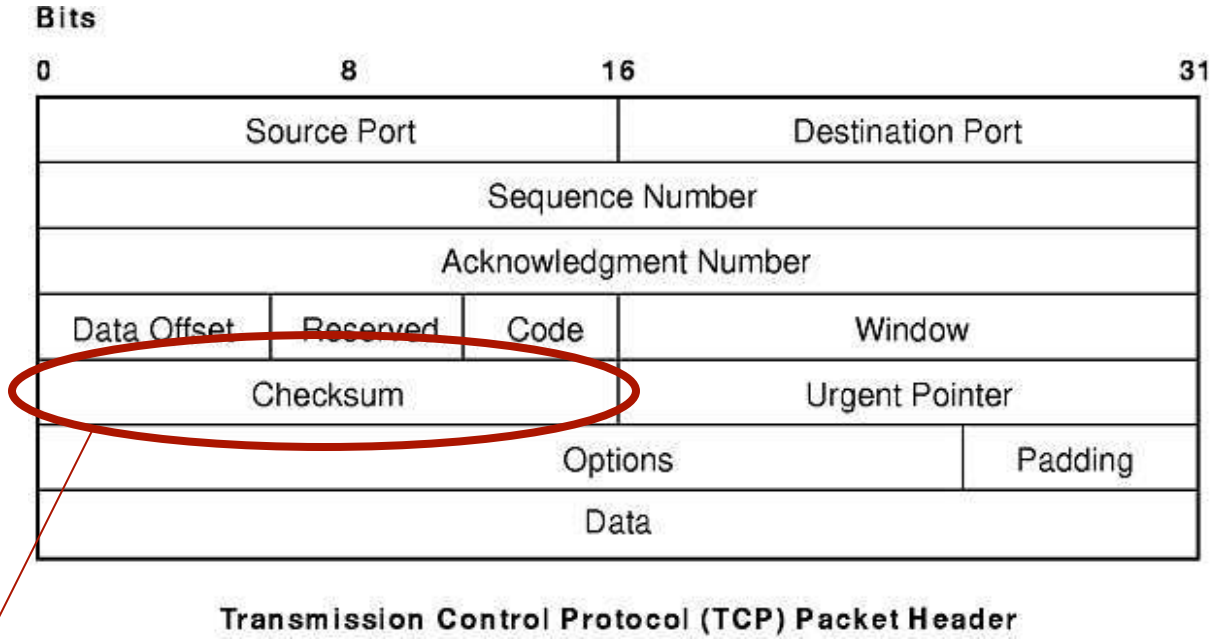
- **Backward** error recovery
  - Return sys. into a previous error-free (hopefully) state
  - Generic approaches (**replication**, **checkpointing**)
  - Potential risk of inconsistency
  - Can be minimised (see next unit)

- **Forward** error recovery
  - Bring system in a valid future state
  - Highly application dependent
  - Makes sense as soon as real-time is involved
  - Examples: video streaming, flight control software

Network FT: TCP

Fault-model

- TCP is a fault-tolerant protocol
  - ➔ correct stream arrives even if packets get lost or corrupted
- A TCP packet:



Transmission Control Protocol (TCP) Packet Header

Error detection against corruption

Networking

TCP Checksum

- Checksum: 16^{bit} blocks of packet added & complemented
  - ➔ 2^{16} (~65 000) possible combinations
  - ➔ TCP packets ~ around 1500 bytes → 2^{12000} combinations
  - ➔ Several packets bounds to have the same checksum
    - Roughly: $2^{12000} \div 2^{16} = 2^{11984} \sim 16 \times 10^{3594}$

- The Trick
  - ➔ Usual corruption unlikely to produce a consistent checksum
  - ➔ Notion of **hamming distance**
  - ➔ Additional checksums at network (IP) & link layers (Ethernet)

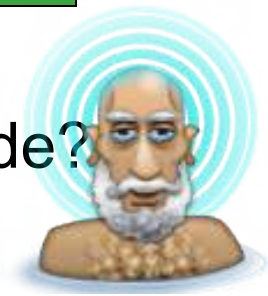
Hamming Distance

- Between 2 string of bits:
  - ➔ the number of bit switches required to go from 1 to the other
- Distance between 1011101 and 1001001?

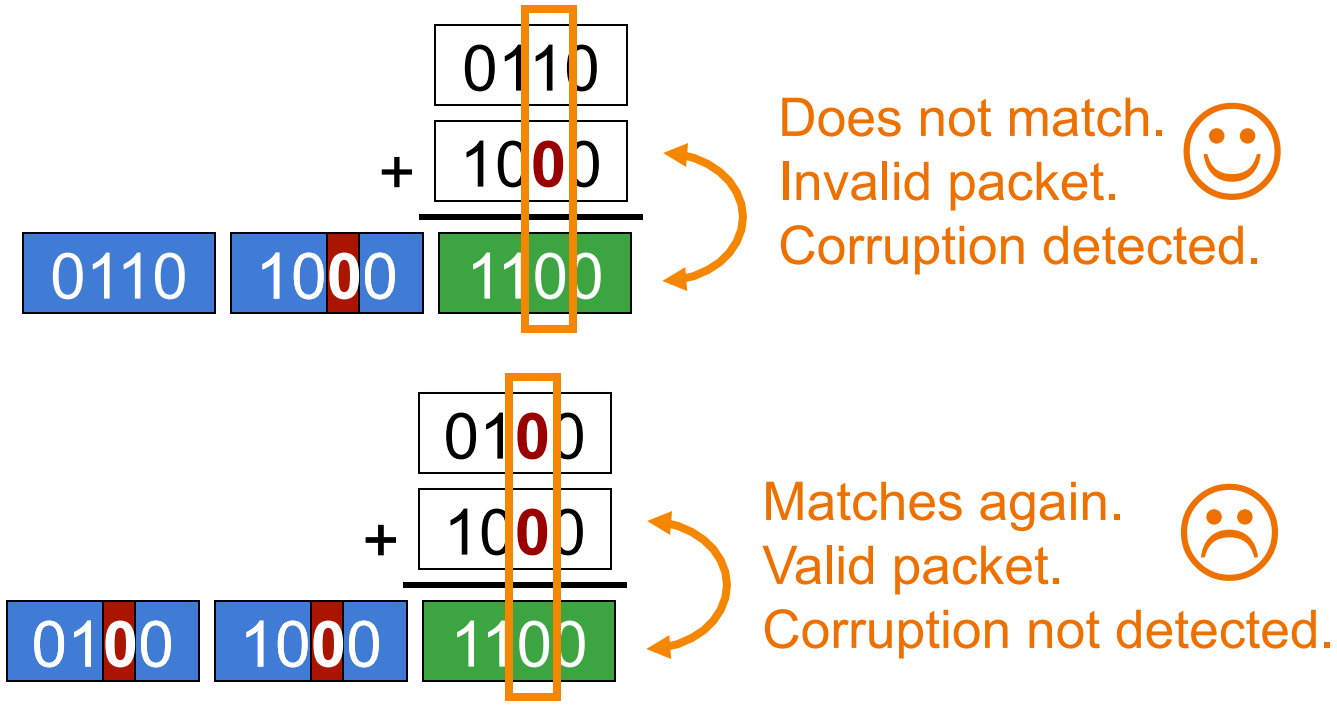
Hamming Distance

- Between 2 string of bits:
  - ➔ the number of bit switches required to go from 1 to the other
- Distance between 1011101 and 1001001? ➔ 2
- Minimum Hamming Distance of a code 
  - ➔ minimal hamming distance between 2 valid "code words"
  - ➔ i.e. minimal number of corruptions to fall back on valid packet
- Simple checksum:
  - ➔ packets of 8 bits
  - ➔ checksum on 4 bits:
    XOR the two 4-bit blocks

- What is the minimal hamming distance of this code?



Hamming Distance



- Hamming distance is 2
  - ➔ Note that the 2 corruptions must obey a particular pattern
  - ➔ In a network corruptions usually occur in bursts

What Is in a Checksum?

- The TCP checksum is a form of **error detection** code
  - ➔ Min. Hamming distance reflects strength of the detection
  - ➔ Far more complex error detection codes exist
  - ➔ For instance CRC: cyclic redundancy check
- It is a form of **redundancy**
  - ➔ info. redundancy: if packet OK no need for checksum
  - ➔ partial redundancy: cannot recover packet w/ checksum
- When **code used "strong" enough** (min. Hamming dist > 2) possible reconstruct "most likely" correct packet
  - ➔ Error correction code (not seen in this course)
  - ➔ Provides error recovery
  - ➔ Similar to RAID

Business Example: Data Centres

- **Mission critical systems** (your company's web server)
  - ➔ Little point in addressing only one type of fault (like bugs)
  - ➔ Major risks: power failure & overheating
- Data Centres: rents luxury "parking places" for servers
  - ➔ Multiple **network** operators
  - ➔ **Secured** physical access
  - ➔ Multiple **power** sources (including power generator)
  - ➔ Highly robust **air conditioning**

- Data Centres give you: **diversity + hardware redundancy**
  - ➔ **But** no software replication
  - ➔ And no consistency
  - ➔ For that you need **software replication** mechanism!

Example: TelecityGroup

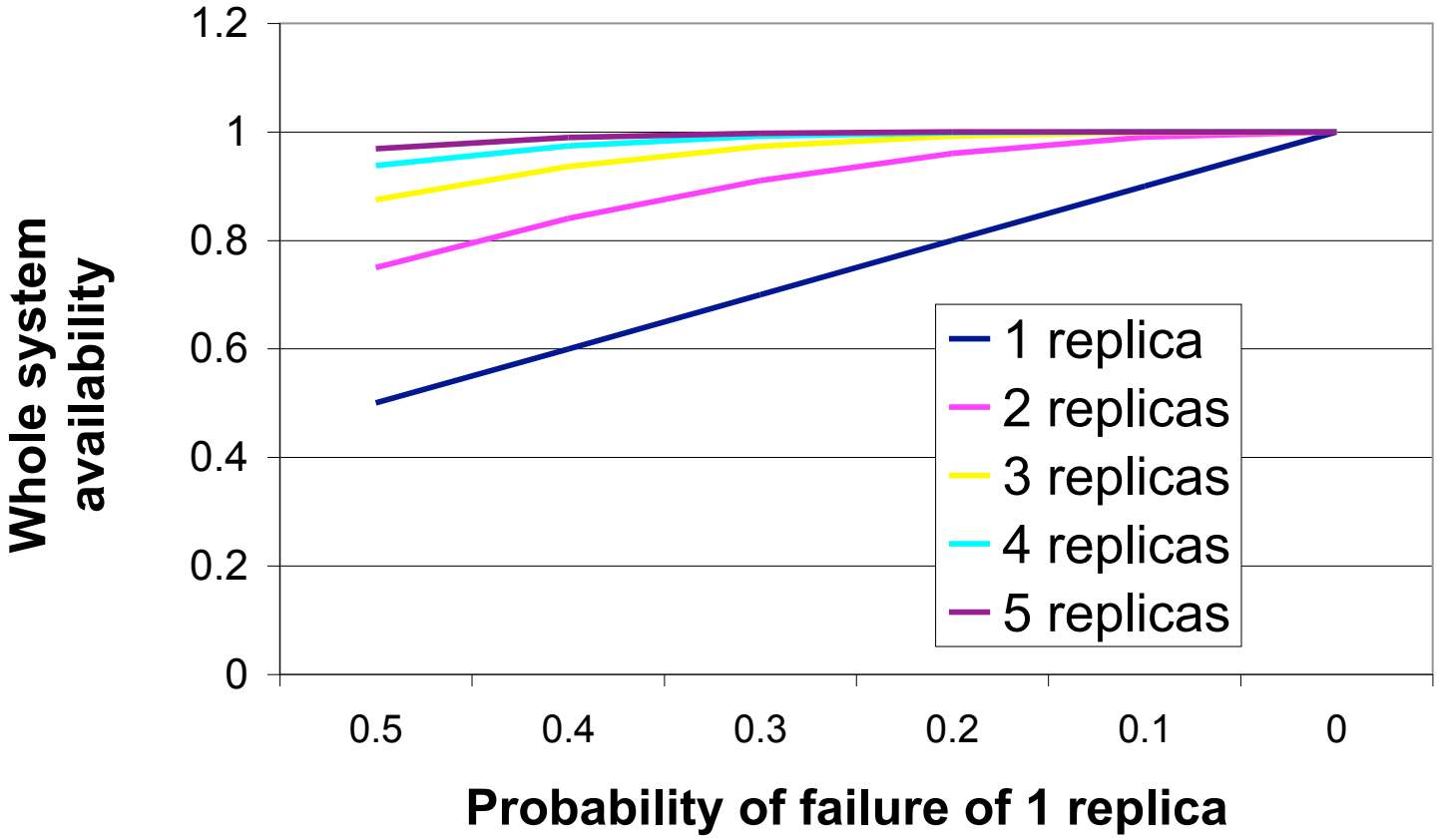
- Example: TelecityGroup (http://www.telecitygroup.fr/)
  - ➔ > 20 data centres across nine cities in Europe.

Recovery by Replication

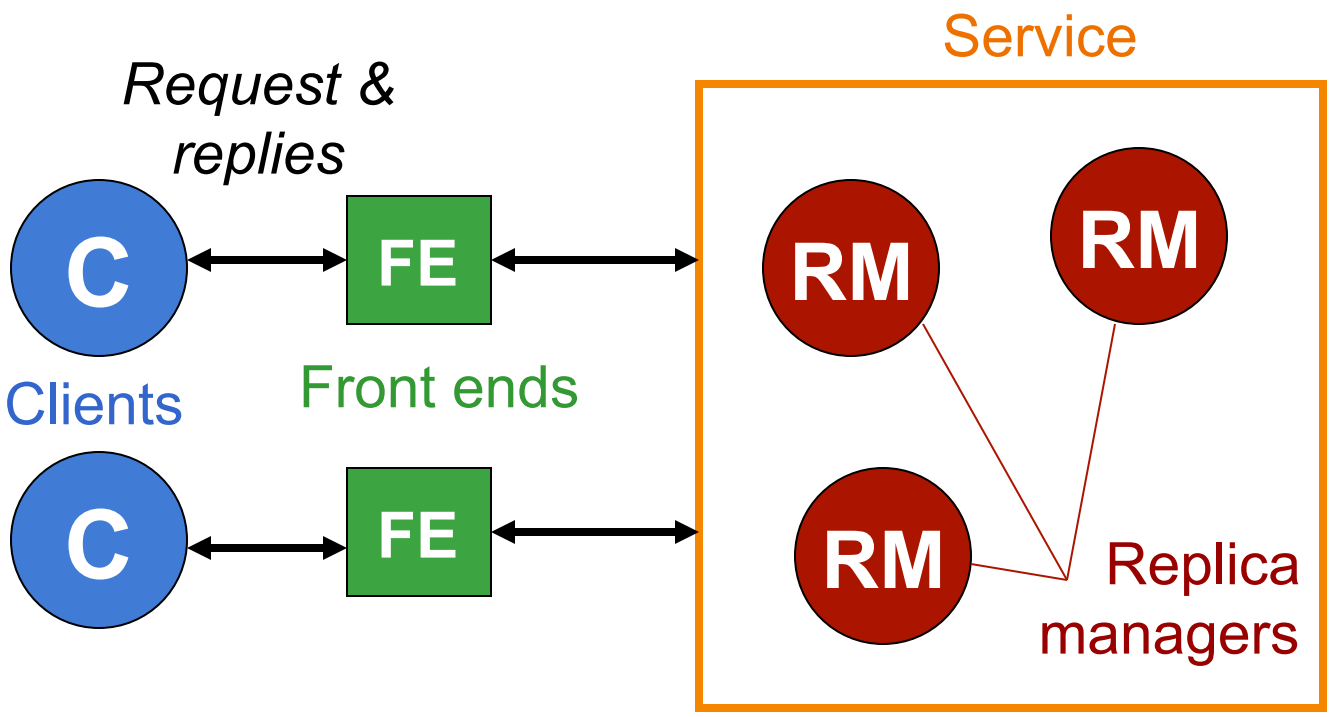
- Why **replicate**?
  - ➔ Performance: e.g. replication of heavily loaded web servers
  - ➔ Allows error detection (when replicas disagree)
  - ➔ Allows backward error recovery
- Importance of **fault-model**
  - ➔ Effect of replication depends on *how* individual replicas fail
  - ➔ Crash faults → availability = $1-p^n$
    (where n = no of replicas, p = probability of individual failure)
- Requirements for a replication service
  - ➔ Replication transparency ...
  - ➔ in spite of network partitions,
    disconnections, etc.

Focus on Availability

- Assumptions (very important!)
  - ➔ Replicas fail independently
  - ➔ Replicas fail silently (crashes, no arbitrary behaviour)

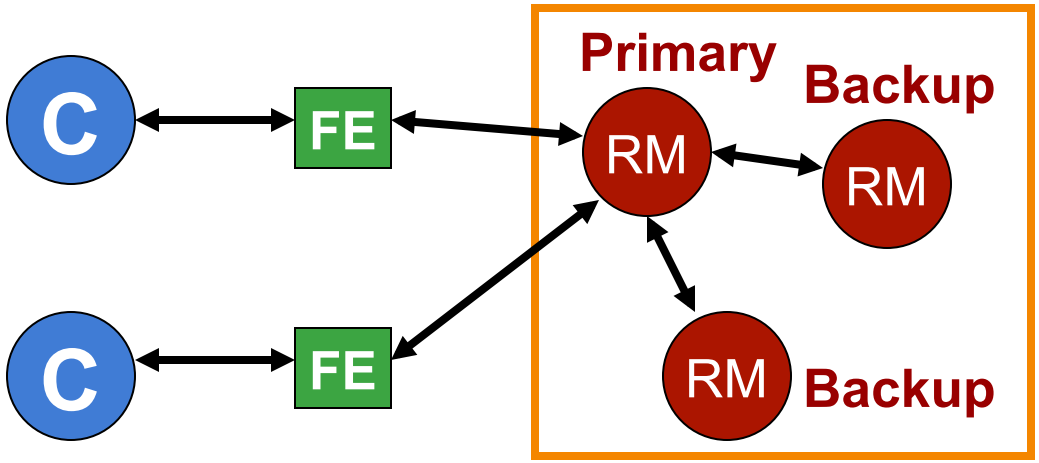


A Generic Replication Architecture



Passive Replication (primary backup)

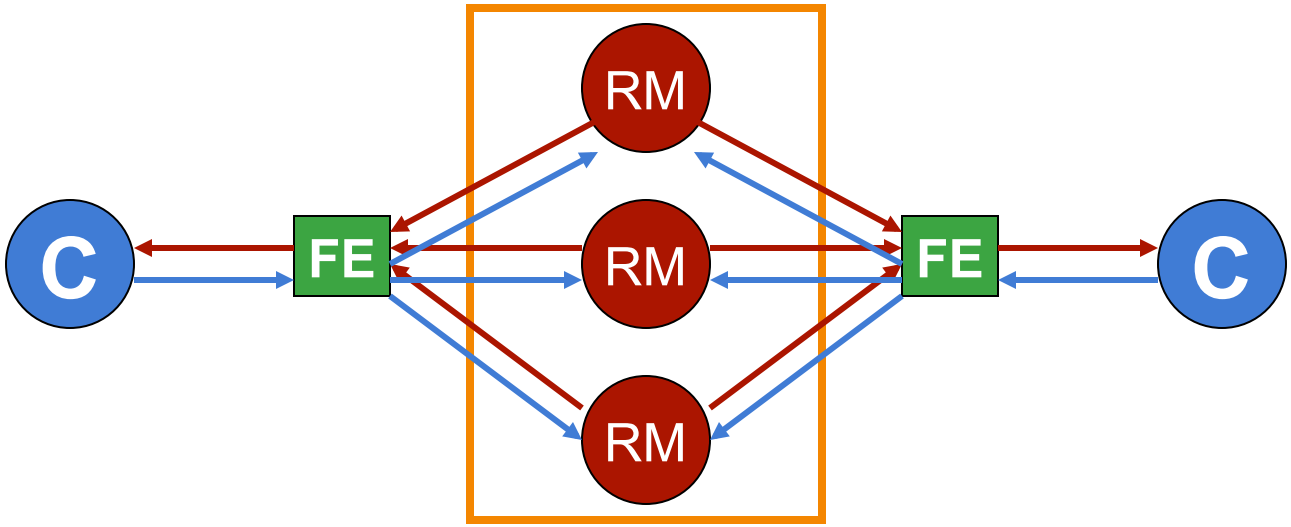
- Primary
  - ➔ processes all requests,
  - ➔ sends the resulting updates to the backup (slaves)
  - ➔ backup do not process requests who don't do any work



  - ➔ Election of new primary in case of primary failure
  - ➔ Only tolerate crash faults (silent failure of replicas)
- Variant: primary's state -> stable storage (cold passive)

Active Replication

- All replicas process requests / return their results to FE
- Appropriate reliable group communications needed
- ordering guarantees crucial



- Tolerate **crash-faults** and **arbitrary faults**

Comparing Active vs. Passive

	Passive	Active
Comm. overhead Process[ing] overhead Recovery overhead Fault model Determinism		

Comparing Active vs. Passive

	Passive	Active
Comm. overhead	**Low**	**High**
Process[ing] overhead	**Low**	**High**
Recovery overhead	**High**	**Low**
Fault model	**Only crash fault**	**Arbitrary faults**
Determinism	**Not required**	**Required**

less expensive
less complex
less powerful

more powerful
more expensive
more complex

What we'll see next week

- Refinement of what we have seen today
- Replication scheme: the problem of consistency
  - ➔Consistent check-pointing for primary backup
  - ➔Fault-tolerant total ordering for active replication
- Contrast replication and distributed transactions
  - ➔Short flash-back to Week 3 lecture

References

- Chapter 7 of Tanenbaum & van Steen (2002 edition)
- Chapter 15 and 18 Coulouris & al (2012 edition)
- On the Titanic
  - ➔ http://www.charlespellegrino.com/time_line.htm
    (Very detailed account)
  - ➔ http://octopus.gma.org/space1/titanic.html
- Fundamental concepts of dependability
  - ➔ by Algirdas Avizenis Jean-Claude Laprie Brian Randell
  - ➔ http://www.cert.org/research/isw/isw2000/papers/56.pdf

Summary

At the end of this Unit:

- You should understand the basic dependability concepts of fault, error, failure
- You should know the basic steps of fault-tolerance
- You should appreciate fundamental fault-techniques like replication and error-detection codes
- You should be able to explain and compare active and passive replication style

SR (Systèmes Répartis)

Unit 9: Fault Tolerance II

François Taïani

Overview of the Session

Investigate advanced issues of fault-tolerance

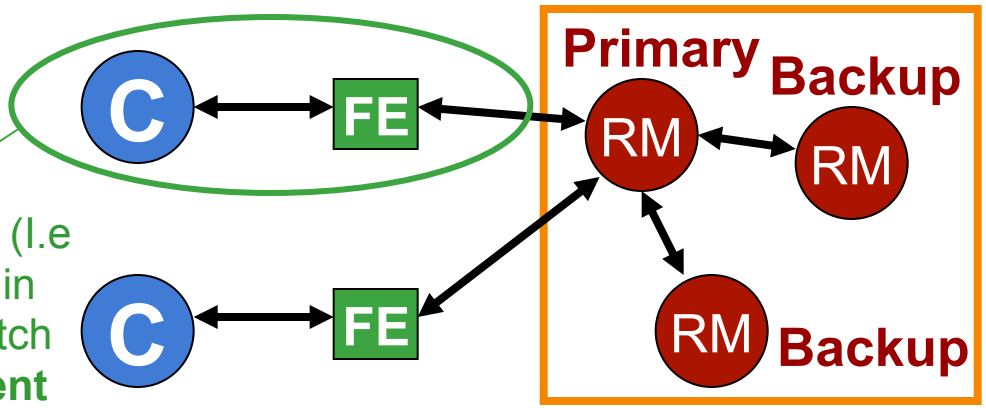
- Passive replication
  - ➔output commit problem
  - ➔how to provide exactly once semantics
- Active replication
  - ➔Importance of total order multicast
  - ➔Link total-order and consensus

Associated Reading: Section 7.5 (Distributed Commit) and 7.6 (Recovery) of Tanenbaum & van Steen; 15.4 (Hierarchical and group masking of faults) of Coulouris & al

Replication & Consistency

- Reminder: **passive replication** (aka primary backup)
  - ➔ FEs communicate with a single **primary** Replication Managers (RM), which must then communicate with secondary RMs

Note: for CW, OK to merge client and FE (I.e replicas hard-coded in client). However switch should be **transparent** to user.



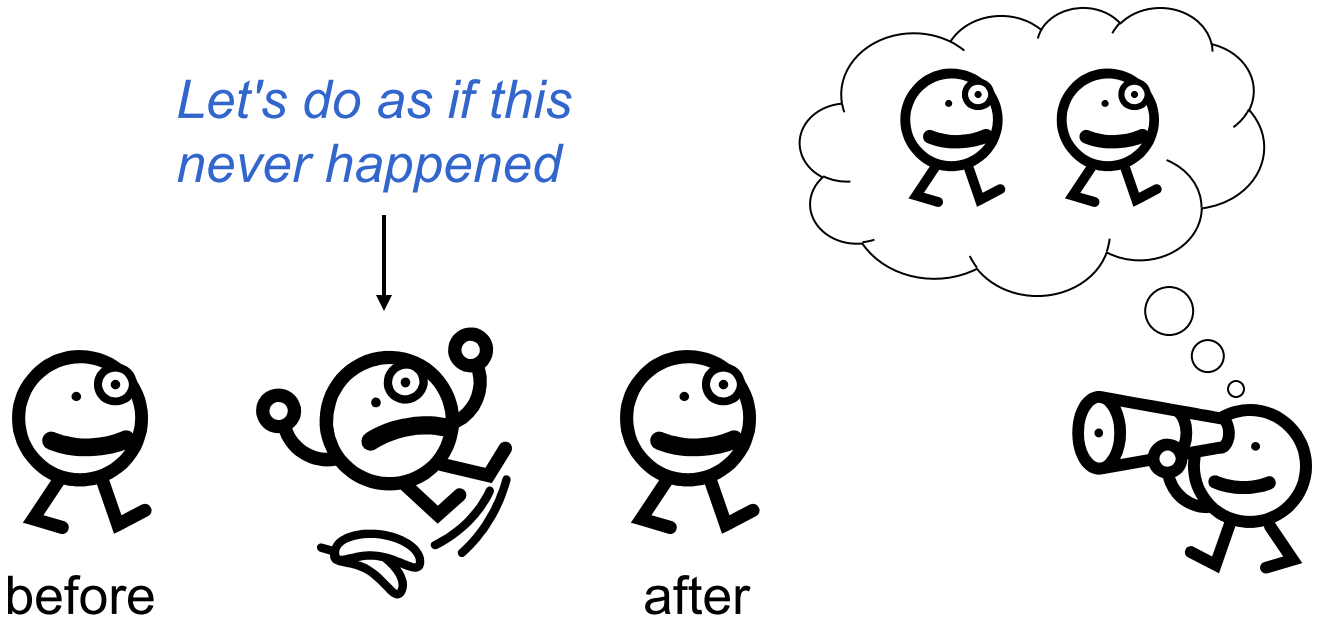
FT I

  - ➔ New primary must be **elected** in case of primary failure
  - ➔ Only tolerates **crash faults** (silent failure of replicas)

Consistency & Recovery

- "A system recovers correctly if its internal state is consistent with the observable behaviour of the system before the failure"

[Strom and Yemini 1985]

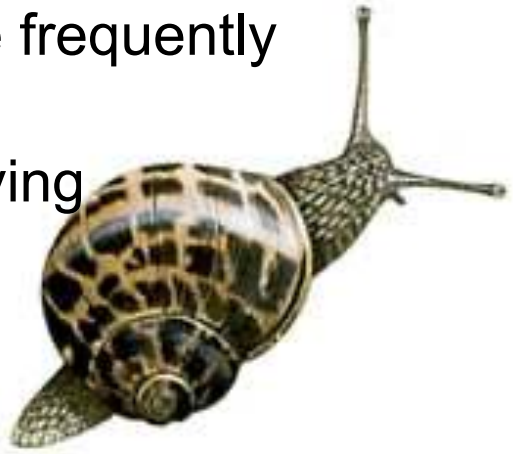


Replication & Consistency

The problem with Passive Replication

- Primary / backup hand-over ➔ **consistency** issue
  - ➔ when primary crashes, backup might be **lagging** behind
  - ➔ backup may not resume exactly were primary left
  - ➔ risk of **inconsistency** from the client point of view

- How to avoid this? *(aka checkpointing)*
  - ➔ synchronise backup with primary more frequently
  - ➔ but: too frequent ➔ high overhead
  - ➔ but: still no guarantee if wrong interleaving
  - ➔ "enough synchronisation but not more"

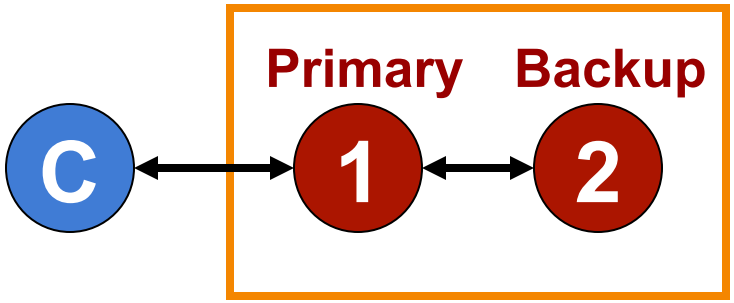


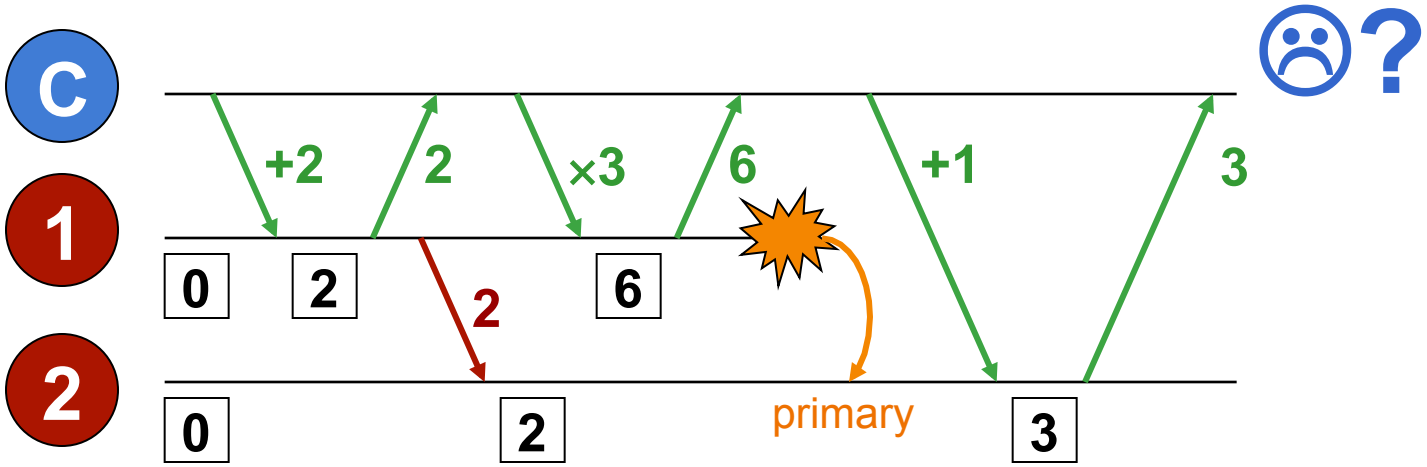
Our Assumptions

- In the following **we assume** the following
  - ➔ messages that are sent do arrive (**FIFO reliable** comm.)
  - ➔ switch from primary to backup is transparent to the client
  - ➔ client will **replay** requests for which it does not get replies
- **Our goal**
  - ➔ "smooth" hand-over to backup on primary crash
- We **don't consider** the following cases
  - ➔ backup crashes
  - ➔ client crashes
  - ➔ any arbitrary failure ("wrong" messages)

Nicolaus Copernicus (1473-1543)

Replication & Consistency

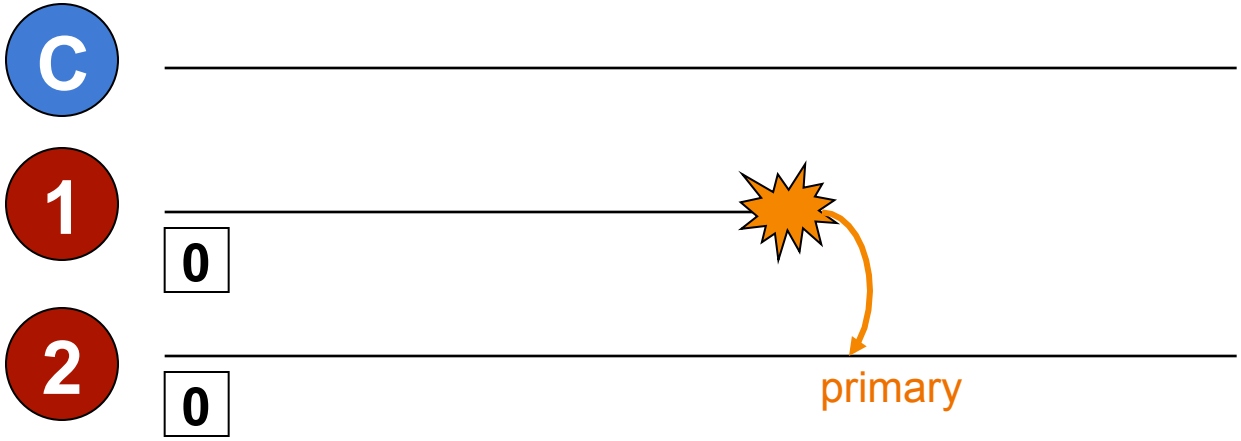




- How to avoid this?

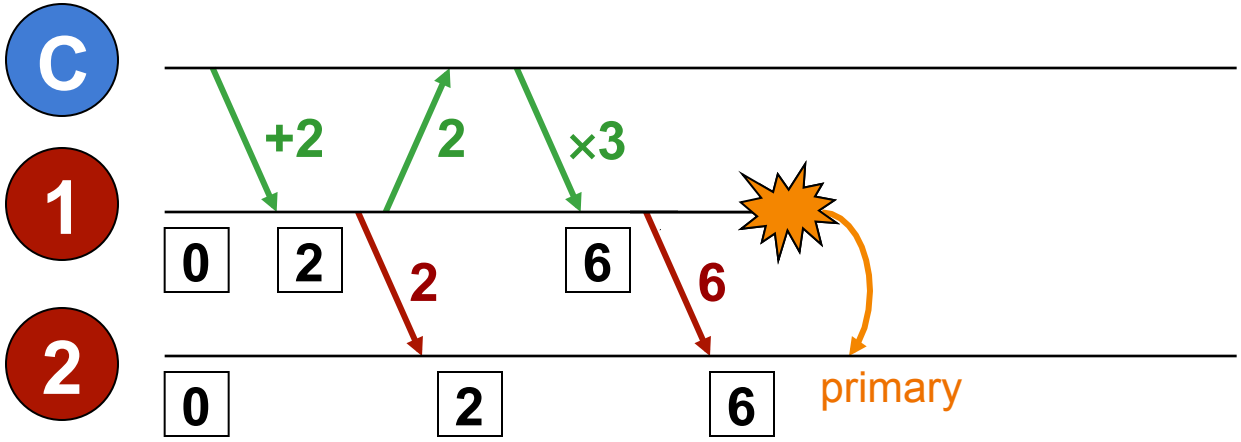
Output Commit Problem

- Algorithm

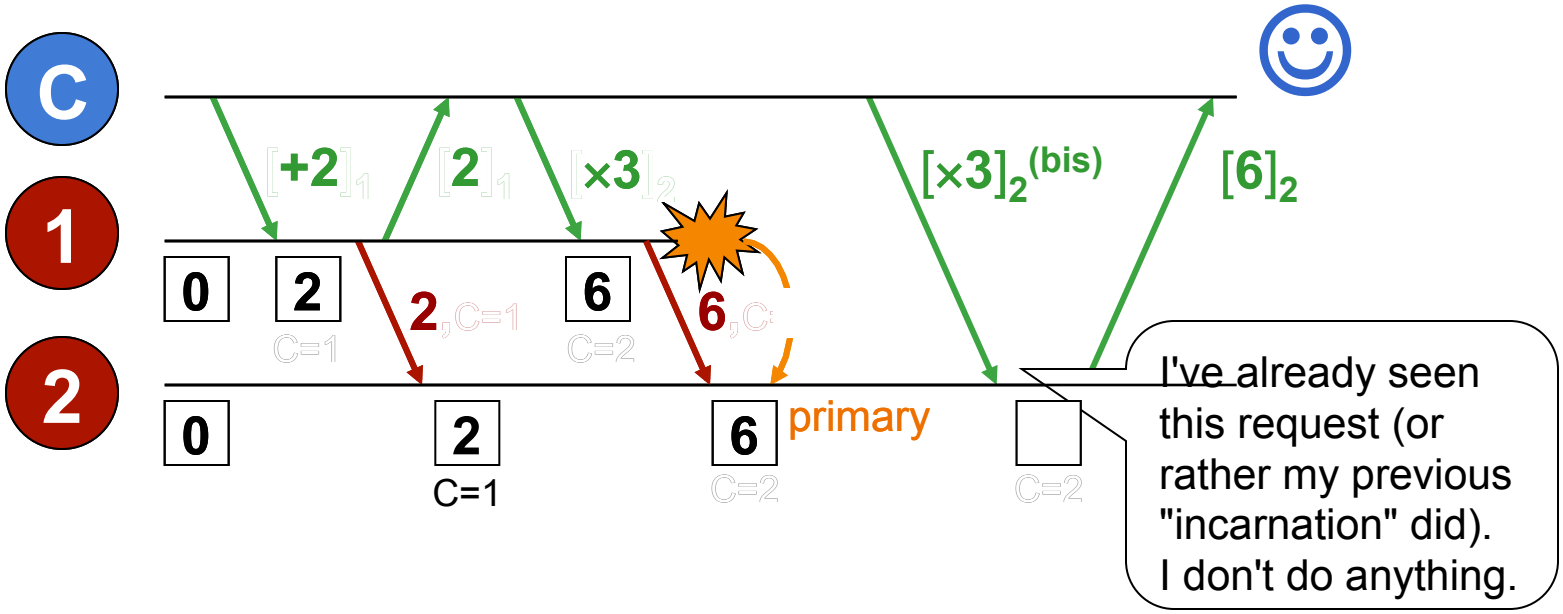


- There is still a problem with this new algorithm. Which one?

More-than-Once Problem



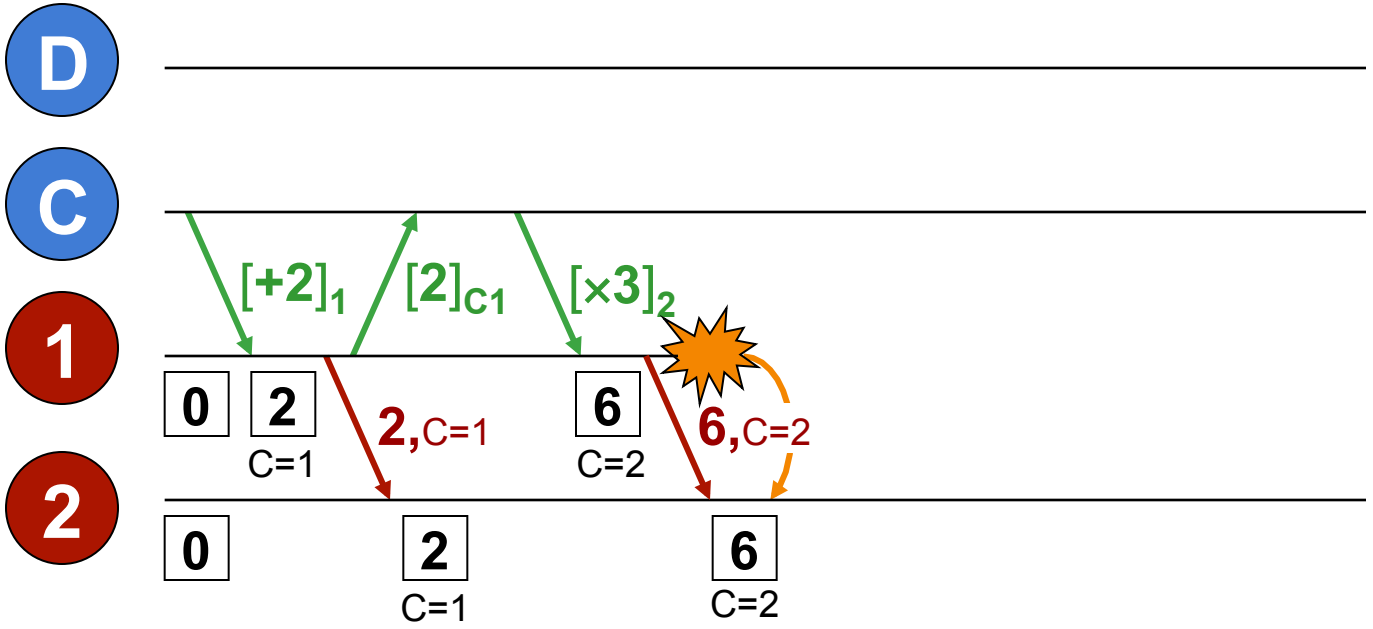
Exactly-Once: Solution 1



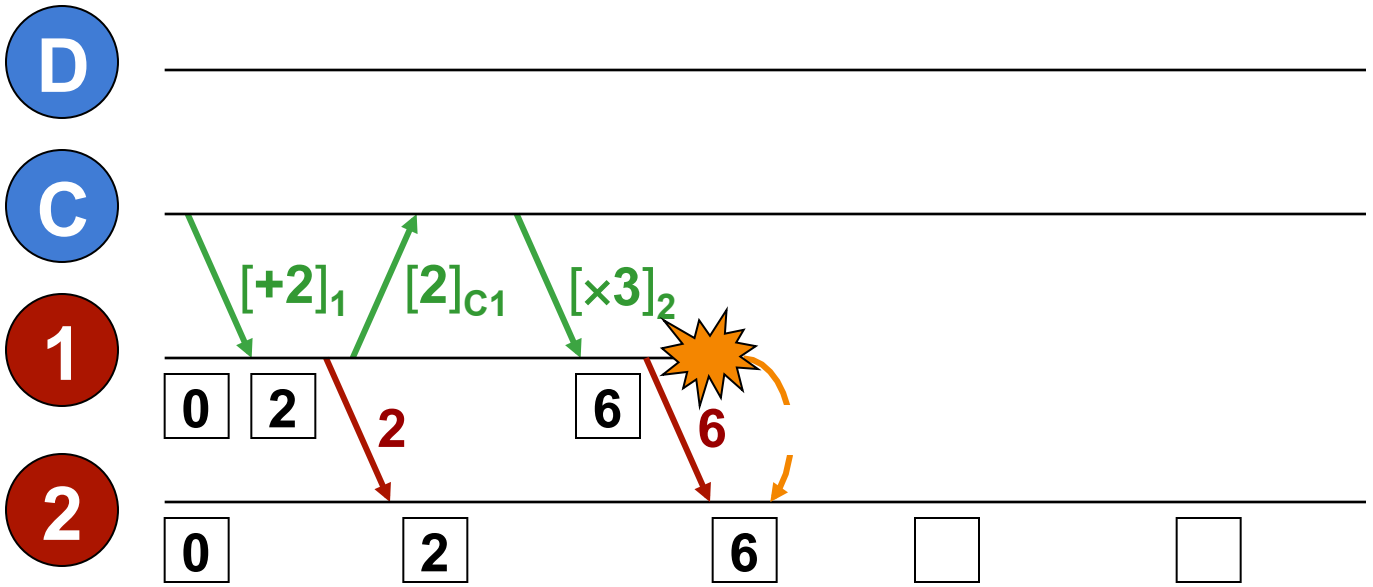
- This solution might break w/ multiple clients. Do you see how?

Multi-Client Problem

- Problem is **not** that C sees the effect of D's operation
  - ➔ This can happen in a failure-free execution. Valid result.
- Problem is that **no** failure-free execution could ever return 8



Exactly-Once: Solution 2
(multi-client safe)



Notes on Solution 2

- *Smooth* hand-over from primary to backup on crash
  - ➔all operations are **executed exactly once**
- Primary failure is **masked** but not entirely transparent
  - ➔reply received much later than in failure-free case

➔ Major disturbance (server crash) replaced by minor annoyance (network delay)

➔**Graceful degradation**:
lesser quality of service but still running

FT I

Further Comments

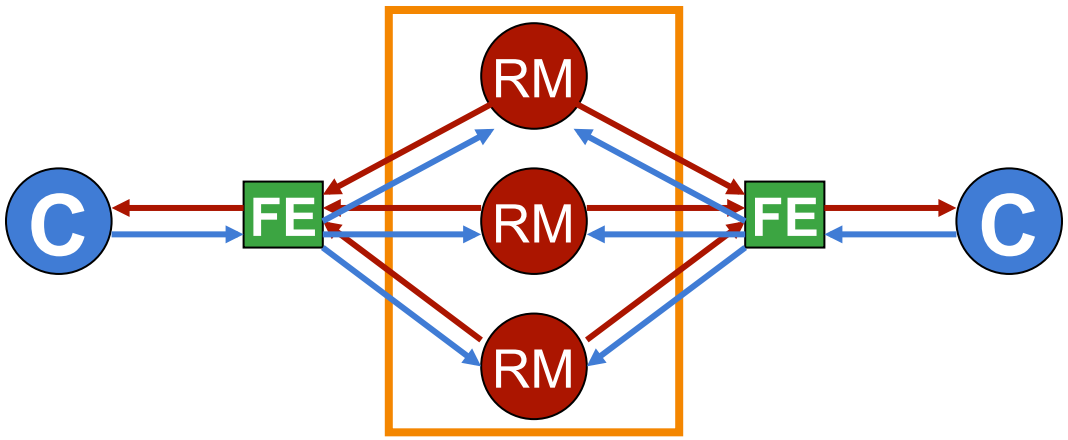
- Assumes bounded network delays (for switch): bad for WAN
- Previous algo **does not scale** to many clients / large state
  - ➔ If millions of clients and big database: intractable
- Solution to large state problem: use **logging**
  - ➔ “save” log of operations performed on primary
  - ➔ regularly "flush" log by checkpointing whole server state
  - ➔ on recovery: latest checkpoint + reapply current log
- All the above assume **sequential** server (i.e. monothreaded)
  - ➔ state saved when no “request in progress” (quiet state)
- Multithreading usually requires
  - ➔ "hot" checkpointing/ backup ability
  - ➔ (Not as good) alternative: Wait/create for ‘quiet’ state

What to remember from this?

- The exact algorithms are not important
  - ➔ Although you should be able to re-analyse them
- What is important are the issues that were raised
- **Output commit problem**
  - ➔ Clients should not see changes that hasn't be made permanent
- **Duplicate requests and exactly once semantics**
  - ➔ No counter and retry ➔ at least-once-semantics
  - ➔ Counter and retry ➔ at most-once semantics
  - ➔ Exactly once requires some atomicity mechanism
  - ➔ What is atomic = "checkpointing" message to the backup
  - ➔ Either the backup receives it or it does not

Active replication

- Reminder
  - ➔ appropriate **fault-tolerant** group communications needed
  - ➔ **ordering** guarantees crucial (see Group Communication Session)



- Even more complex than passive replication
  - ➔ Some of the complexity encapsulated by group comm.
  - ➔ Pb: choosing the **right group comm.** for application & faults

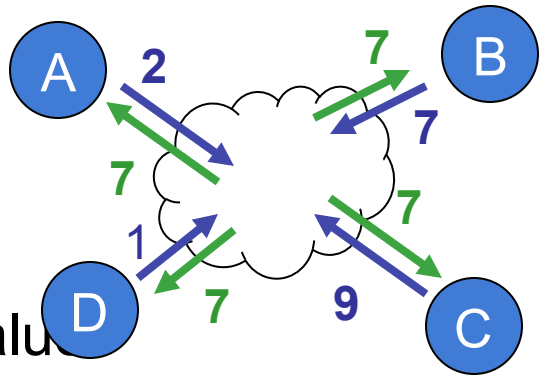
Realising Active Replication

- Group communication needed, with **total ordering** property
  - ➔ See example for what happens without total ordering
- In course for totally ordered multicast presented
  - ➔ centralised sequencer
  - ➔ based on time-stamping with logical clocks

- Problem: none of them is **fault-tolerant**
  - ➔ the centralised sequencer is single point of failure
  - ➔ time-stamping: crash of any participant blocks the algorithm
- We need a fault-tolerant (atomic) totally ordered multicast
  - ➔ tolerating crash-fault if active replication used against them
  - ➔ tolerating arbitrary fault if active replication used against them

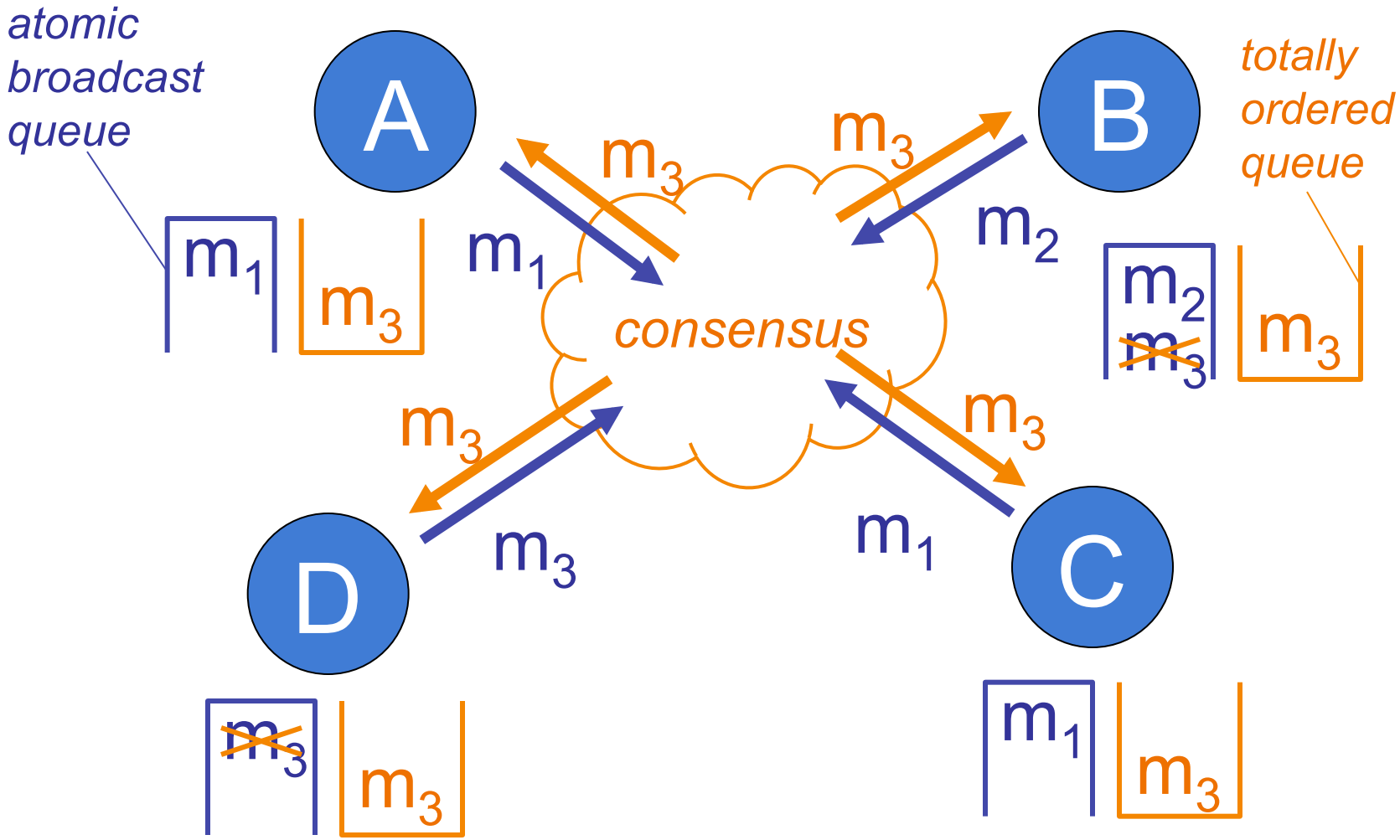
Total Ordering and Consensus

- Realising **total-ordering** is equivalent to realising **distributed consensus**



- **Distributed consensus**
  - ➔ All participants start by proposing a valu
  - ➔ At the end of the algorithm one of the proposed value has been picked, and everybody agrees on this choice
- From distributed consensus to total ordering
  - ➔ Each participant proposes the next message it would like to accept
  - ➔ Using consensus everybody agrees on next message
  - ➔ This message is the next delivered
- **Fault-Tolerant consensus ➔ fault-tolerant total ordering**

Total Ordering and Consensus



Fault-Tolerant Consensus

- **Main idea 1:** not be blocked by crashed processes
  - ➔ use failure detection to stop waiting for crashed processes
- **Main idea 2**: propagate influence of crashed processes
  - ➔ before crashing a process might have comm with others
  - ➔ these messages must be share with all non-crashed processes (or with none of them). They might influence consensus outcome.
- The properties of the **failure detector** is essential
  - ➔ in reality **false positive** happen (time-out and slow network)
  - ➔ different algo for different classes of **imperfect failure detectors**
  - ➔ the more imperfect the less crashes can be tolerated
- FT Consensus algorithms even exist for **arbitrary failure**
  - ➔ Even more redundancy required

FT Consensus (strong failure detector, crash faults)

procedure *propose*(v_p)

$V_p \leftarrow \langle \perp, \perp, \ldots, \perp \rangle$
$V_p[p] \leftarrow v_p$
$\Delta_p \leftarrow V_p$

{*p's estimate of the proposed values*}

Phase 1: {*asynchronous rounds* r_p, $1 \leq r_p \leq n-1$}
for $r_p \leftarrow 1$ to $n-1$
send (r_p, Δ_p, p) to all
wait until $[\forall q :$ received (r_p, Δ_q, q) **or** $q \in \mathcal{D}_p]$ {*query the failure detector*}
$msgs_p[r_p] \leftarrow \{(r_p, \Delta_q, q) \mid$ received $(r_p, \Delta_q, q)\}$
$\Delta_p \leftarrow \langle \perp, \perp, \ldots, \perp \rangle$
for $k \leftarrow 1$ to n
if $V_p[k] = \perp$ **and** $\exists (r_p, \Delta_q, q) \in msgs_p[r_p]$ with $\Delta_q[k] \neq \perp$ **then**
$V_p[k] \leftarrow \Delta_q[k]$
$\Delta_p[k] \leftarrow \Delta_q[k]$

Phase 2: send V_p to all
wait until $[\forall q :$ received V_q **or** $q \in \mathcal{D}_p]$ {*query the failure detector*}
$lastmsgs_p \leftarrow \{V_q \mid$ received $V_q\}$
for $k \leftarrow 1$ to n
if $\exists V_q \in lastmsgs_p$ with $V_q[k] = \perp$ **then** $V_p[k] \leftarrow \perp$

Phase 3: *decide*(first non-$\perp$ component of V_p)

Fig. 5. Solving Consensus using any $\mathcal{D} \in \mathcal{S}$. *[Chandra & Toueg, 1996]*

For information only. Not Exam material

Expected Learning Outcomes

At the end of this unit:

- You should understand what the output commit problem is about.
- You should appreciate the mechanisms involved in realising exactly once semantics for passive replications.
- You should be able to explain the role played by total ordering in active replication.

References

- A survey of rollback-recovery protocols in message-passing systems
  - ➔ E. N. Elnozahy, Lorenzo Alvisi, Yi-Min Wang, David B. Johnson, ACM Computing Surveys, Volume 34 , Issue 3 (September 2002), Pages: 375 - 408
  - ➔ Very good and extensive survey on algorithmic issues
- Unreliable failure detectors for reliable distributed systems
  - ➔ Tushar Deepak Chandra, Sam Toueg, Journal of the ACM (JACM), Volume 43 , Issue 2 (March 1996), Pages: 225 - 267
  - ➔ Fundamental consensus algorithms under various assumptions

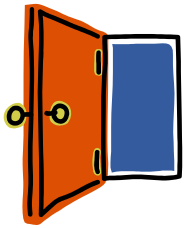
ESIR SR

Unit 10a: JMS

François Taïani

Overview of the Session

- What is JMS
- Messages vs. RPC
- Interaction Styles
- Main JMS Classes
- Advanced Features



See lecture on indirect communication

What is JMS?

- "Java Message Service"

http://java.sun.com/products/jms/

- An **API** that is part of the J2EE standard (since Java 1.3)
  - ➔ Relevant classes and interface in javax.jms
  - ➔ Provided my most J2EE implementations (I.e. JBoss, Sun's J2EE SDK)
- **JMS** = Message Oriented Middleware (MOM) + Publish Subscribe (Pub/Sub)
  - ➔ Clients communicate via "queues" and "topics"
  - ➔ degrees of guarantees: persistence, atomicity, blocking or not
  - ➔ A central broker: possibly distributed / replicated
- Other MOM products
  - ➔ Websphere MQ (IBM), Microsoft's MSMQ and Oracle's Streams Advanced Queuing AQ, Apache ActiveMQ

Messages vs. RPC

Why uses Messages?

- Messages provides loose time & space coupling
  - In JMS "Messages" used both for MOM and Pub/Sub
  - Large scale systems: RPC often too tightly coupled
  - Asynchronous interaction possible
    - Sender up while receiver down (or disconnected) and vice-versa
- Messages are highly flexible
  - Anything can be a message in JMS: string, object, XML
  - Arbitrary interaction patterns possible
    - 1-1, n-n, with replies, with no replies
- Messages don't hide distribution
  - Sometimes this is needed to control side effects of distribution
  - Particularly true for large scale entreprise-wide systems

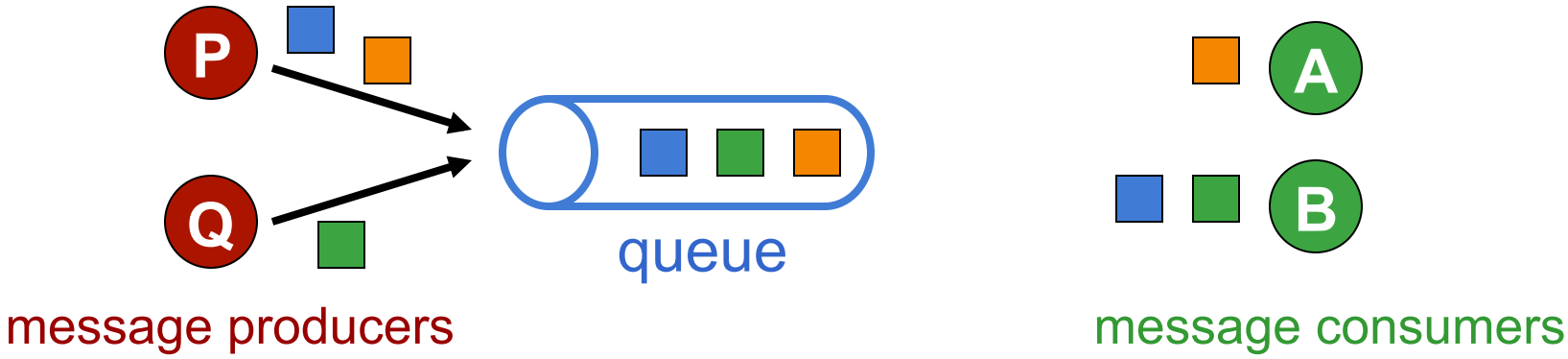
Interaction Styles in JMS

Covered in lecture on indirect communication

- Two main **modes of communication**
  - ➔ Message queuing (aka 1-to-1 communication)
  - ➔ Publish-Subscribe (aka 1-to-many communication)
- Two main **mode of message consumption** on the receiver
  - ➔ Blocking (aka synchronous, aka pull mode) with **MessageConsumer.receive( )**
  - ➔ Non-blocking (aka asynchronous, aka push mode) with **MessageListener.onMessage(..)**
- Both dimensions can be combined
  - ➔ Four possibilities
  - ➔ Usually blocking reception used with message queuing
  - ➔ And non-blocking reception used with publish-subscribe

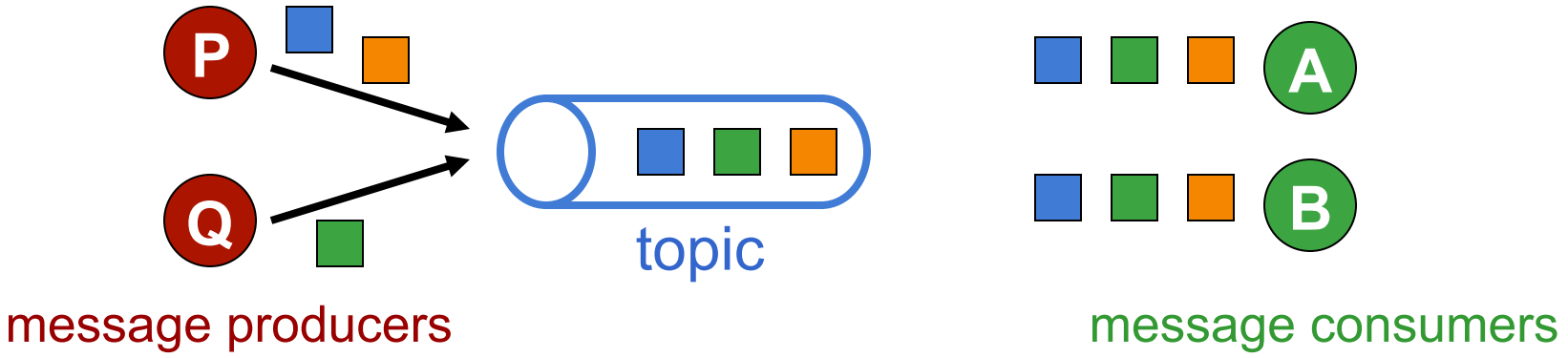
1-to-1 communication

- **Queues**:
  - Messages are sent to a queue object on the server
  - They are received from the queue by message consumers (clients)
  - **One** message can only be received by **one** clients
  - But several clients can be writing to / reading from the same queue concurrently
    - How message are dispatched is implementation dependent



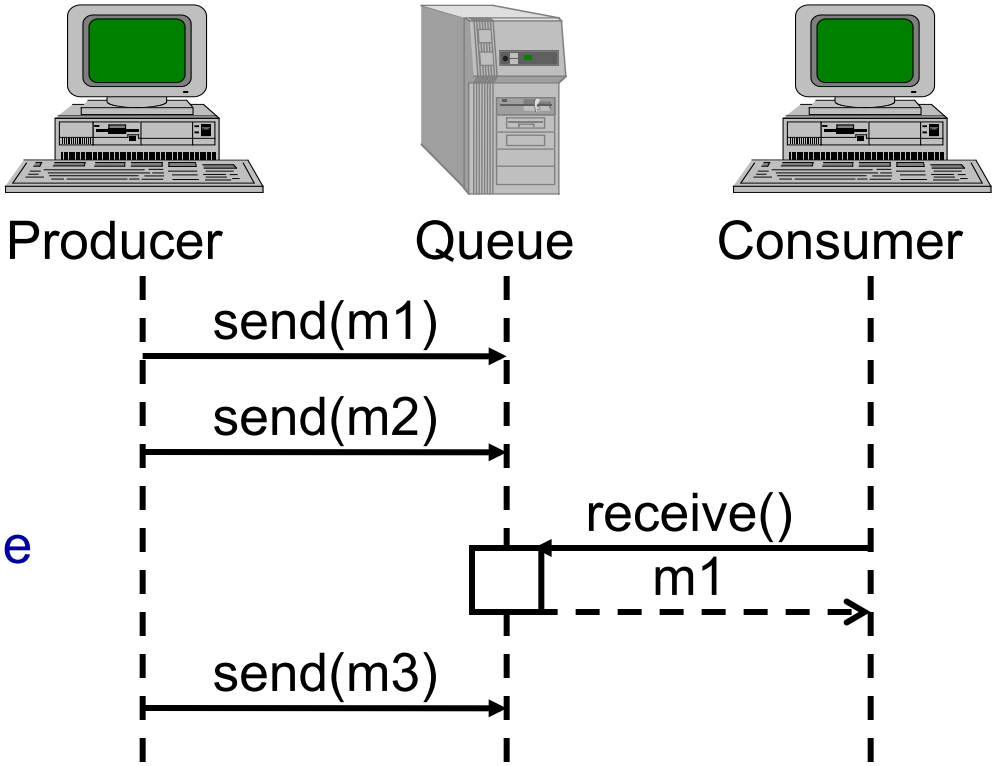
Publish / Subscribe (1-many)

- **Topics**:
  - Messages are “published” relative to a topic object on the server
  - They are received by message consumers (clients) that have subscribed to the topic
  - **One** message is received by **all subscribers**
  - If clients subscribe/ unsubscribe while messages are sent, results are undefined (they may or may not get the messages)



Blocking Reception (Pull)

- Aka Active Reception
- Typically used with point-to-point queues



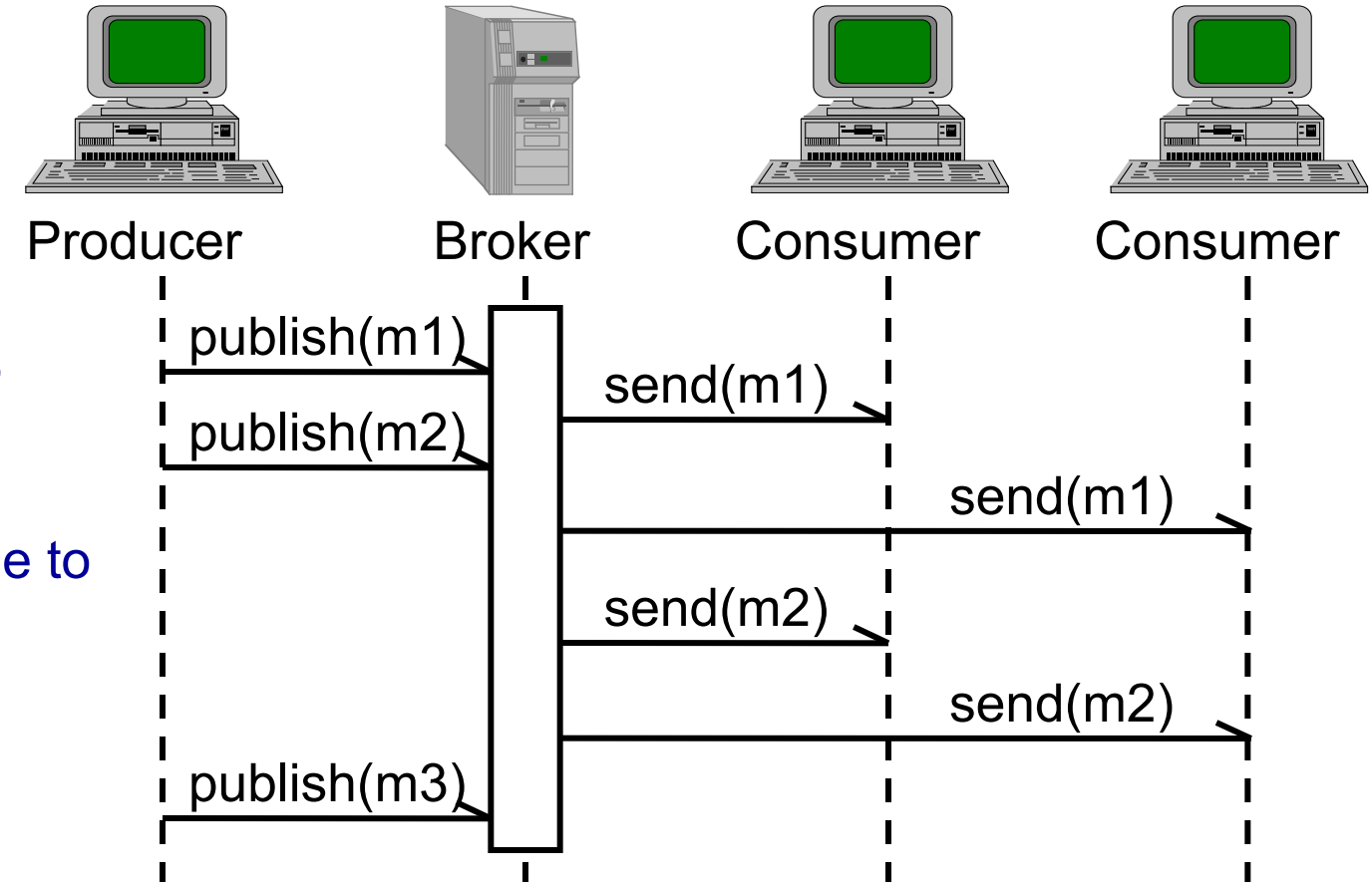
Put message into queue

Consume message

Source: Kelly & Maffeis

Non-Blocking Reception (Push)

- Aka passive reception
- Typically used with 1-n communication (Publish/Subscribe)



Pass message to broker

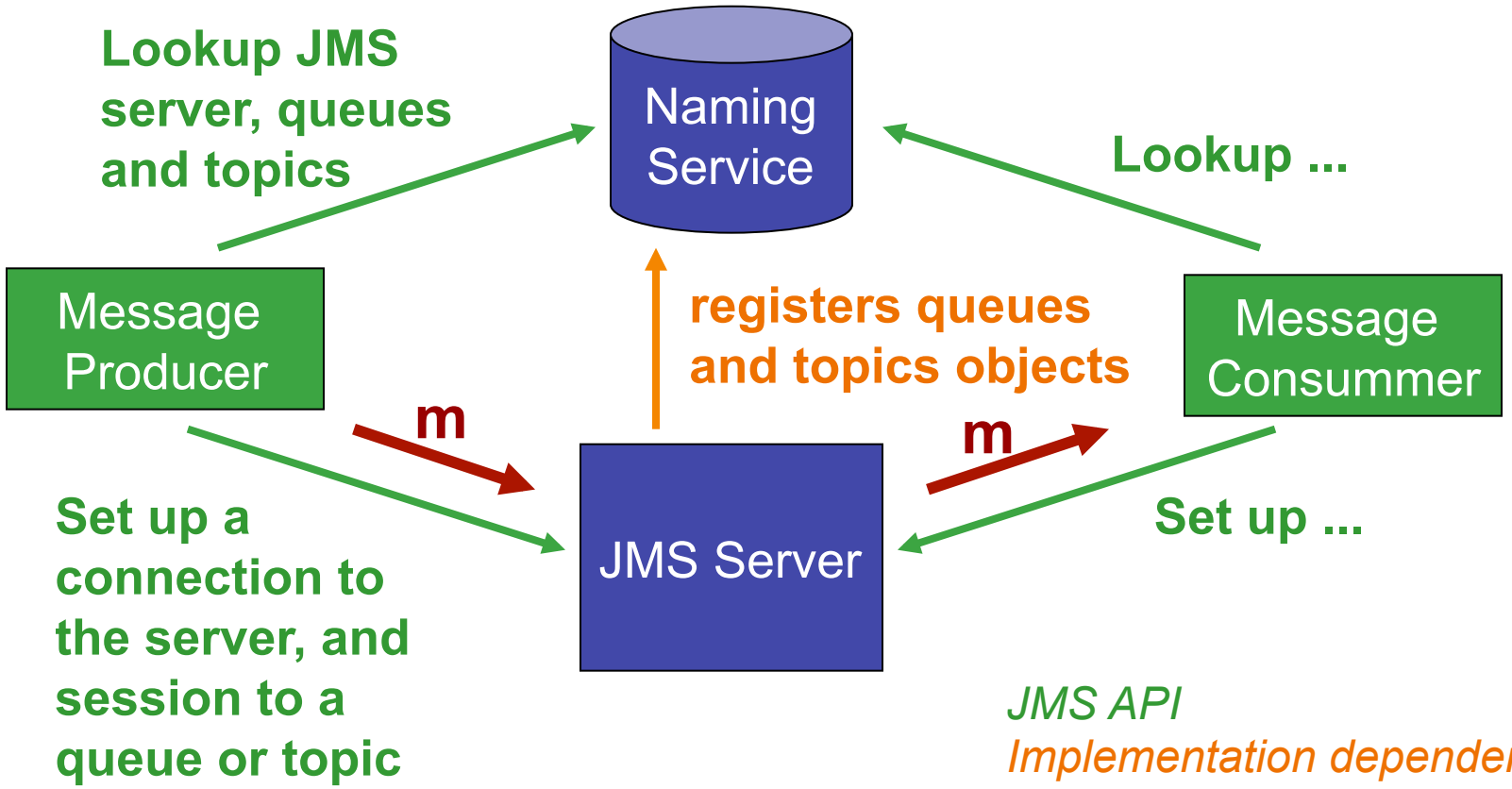
Dispatch message to all consumers

Source: Kelly & Maffeis

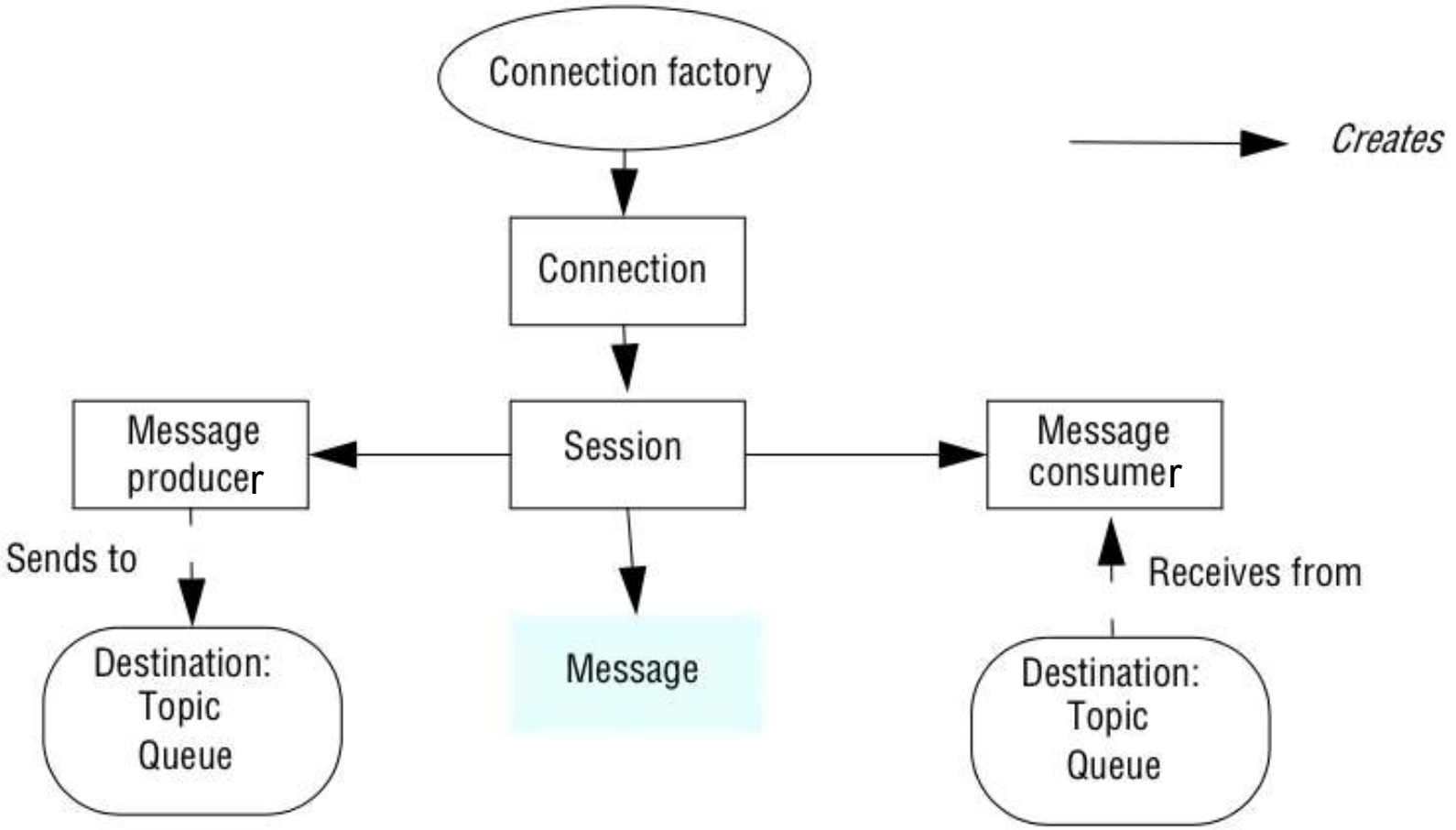
Important Note

- The JMS API **only** provide interfaces to program **the clients** of a messaging system
  - ➔ It provides means to retrieve references to Queue and Topic objects through a naming service (usually JNDI)
  - ➔ It does not permit the creation of Queues and Topics ('Destinations') on the server
- The **creation** of Queues and Topic on the server ("JMS Provider") is implementation dependant
  - ➔ In SUN's implementation done using the J2EE admin console
- **Goal**: The same client code can be used with different server implementations.

Typical JMS Architecture



The Main JMS Classes



The Main JMS Classes

- Actually they are not classes but interfaces
  - As a user you don't get to see the implementation classes
- Typical set up of a JMS client process
  - create a **Connection**
  - one or more **Sessions**
  - a number of **MessageProducers** and **MessageConsumers**
- A **Connection** object:
  - Encapsulates an open connection with a JMS provider
    - typically an open TCP/IP socket between a client and the JMS server
  - Its creation is where client authentication takes place
  - It is created from a **ConnectionFactory**
    - the connection factory is typically retrieved from the naming service
    - ConnectionFactory cf = NamingService.lookup("someName");

Session Objects

- A **Session** object is a single-threaded context for producing and consuming messages
  - ➔ It is created using **Connection.createSession(..)**
- A **Session** object serves several purposes
  - ➔ It is a factory for its message producers and consumers.
  - ➔ It is a scoping unit to perform atomic transactions that span its producers and consumers
  - ➔ It enforces a serial order for the messages it consumes and the messages it produces across all its producers and consumers
  - ➔ It retains messages until they have been acknowledged

Queue and Topic Objects

- **Queue** and **Topic** are both sub-interface of Destination
- Not created by clients but retrieved from a Naming Service
  - Actually what is retrieved is a *reference* to a topic or a queue
  - Same mechanisms as ConnectionFactories
    - Queue myQueue = NamingService.lookup("myQueue")
  - Queues, Topic + Connection Factories = "administered object"
    - They Must be set up in an implementation dependent manner
- They are needed to create **Message Consumer** and **Message Producer** from a Session object
  - MessageConsumer Session.createConsumer (Destination)
  - MessageProducer Session.createProducer (Destination)

Message Consumer and Producers

- They provide methods to send and receive messages either to/from a queue or about a topic
- **Message production** by Producer
  - ➔ send(Message message) + variant with fine tuning
- **Message reception** by Consumer
  - ➔ Synchronous: receive(), receive(long timeout), receiveNoWait()
  - ➔ Asynchronous: Listener mechanism setMessageListener(..)

Example: Fire Alarm (Publisher)

```
import javax.jms.*;
import javax.naming.*;

public class FireAlarmJMS {
  public void raise() {
    try {
      Context ctx = new InitialContext();
      TopicConnectionFactory topicFactory =
        (TopicConnectionFactory)ctx.lookup("TopicConnectionFactory");
      Topic topic =       (Topic)ctx.lookup("Alarms");
      TopicConnection topicConn =
        topicFactory.createTopicConnection();
      TopicSession    topicSess  =
        topicConn.createTopicSession(false, Session.AUTO_ACKNOWLEDGE);
      TopicPublisher  topicPub =
        topicSess.createPublisher(topic);
      TextMessage     msg =
        topicSess.createTextMessage();
      msg.setText("Fire!");
      topicPub.publish(message);
    } catch (Exception e) {
    }
  } // EndMethod raise()
} // EndClass FireAlarmJMS
```

Jini Lookup

Connection, Session, and Publisher

message published

Example: Fire Alarm (Consumer)

```
import javax.jms.*;
import javax.naming.*;

public class FireAlarmConsumerJMS {
  public String await() {
    try {
      Context ctx = new InitialContext();
      TopicConnectionFactory topicFactory =
        (TopicConnectionFactory)ctx.lookup("TopicConnectionFactory");
      Topic topic =      (Topic)ctx.lookup("Alarms");
      TopicConnection topicConn =
        topicFactory.createTopicConnection();
      TopicSession    topicSess   =
        topicConn.createTopicSession(false, Session.AUTO_ACKNOWLEDGE);
      TopicSubscriber topicSub =
        topicSess.createSubscriber(topic);
      topicSub.start();
      TextMessage msg
        = (TextMessage) topicSub.receive();
      return msg.getText();
    } catch (Exception e) {
    }
  } // EndMethod await()
} // EndClass FireAlarmConsumerJMS
```

Jini Lookup

Connection, Session, and Publisher

message received (blocking)

Using the code

- On the consumer: waiting for an alarm

```
FireAlarmConsumerJMS
  alarmCall = new FireAlarmConsumerJMS();
String msg = alarmCall.await();
```

- On the publisher's side: raising an alarm

```
FireAlarmJMS alarm = new FireAlarmJMS();
alarm.raise();
```

Advanced Aspects of JMS

- **Reliability**
  - By default message are sent in `PERSISTENT` mode
  - JMS server takes extra care to prevent message loss
  - In particular messages sent in this mode this are logged to stable storage when sent
  - Possible to switch this off to gain performance
- **Durability**
  - Default: consumers only receive messages sent while active
  - Possible to create "durable subscription": like asking a neighbour to record your favourite TV program while you're on holyday

Advanced Topics (cont.)

- **Message Expiration**
  - By default messages never expire
  - Possible to set expiration time
  - Messages not received after this time are destroyed

- **Transaction**
  - Grouping of a sequence of client operations (sending, receiving) into one atomic unit of work
  - If anything goes wrong, work done is rolled back and transaction can be started all over again

Summing Up

At the end of this Session:

- You should understand what **Message Oriented middleware** is about
- You should know what the **Java Messaging Service** is
- You should know the difference between **point-to-point** and **publish-subscribe** message communication, and between blocking and non-blocking reception
- You should have some idea of what the main classes of the JMS are and what they do

References

- J2EE API Documentation on JMS
  - ➔ http://docs.oracle.com/javaee/6/api/javax/jms/package-summary.html
- Java Message Service - What and Why?
  - ➔ Bill Kelly, Silvano Maffeis
  - ➔ www.jug.ch/events/slides/000508_jmsintro-english.pdf
- Chapter 45 of the J2EE Tutorial on Oracle's web site
  - ➔ http://docs.oracle.com/javaee/6/tutorial/doc/bncdq.html

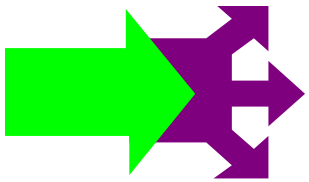
ESIR SR

Unit 10b: JGroups

François Taïani

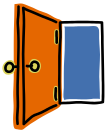
Overview of the Session

- Group Communication
- What is JGroups?
- JGroups' Architecture
  - ➔The Channel Class
  - ➔The Protocol Stack Infrastructure
  - ➔The Building Blocks
- Comparison with JMS

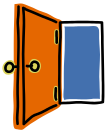


Introducing Group Communication

- What is group communication?
  - Enables the multicasting of a message to a group of processes as a single action
  - Sender is unaware of the destinations for the message
- Why group communication?
  - Support for replication: fault tolerance & scalability
  - Support the efficient dissemination of data Service discovery, publish/subscribe



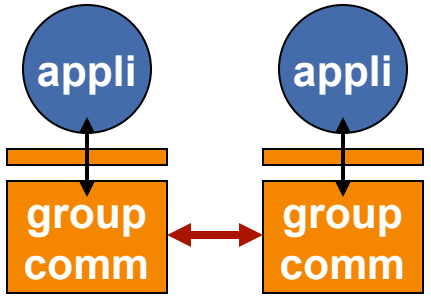
Indirect Communication
Fault Tolerance



JMS

Associated reading: section 7.4 of Tanenbaum and Van Steen (2002 ed); Section 4.4 and 11.5 of Coulouris & al. (2nd ed)

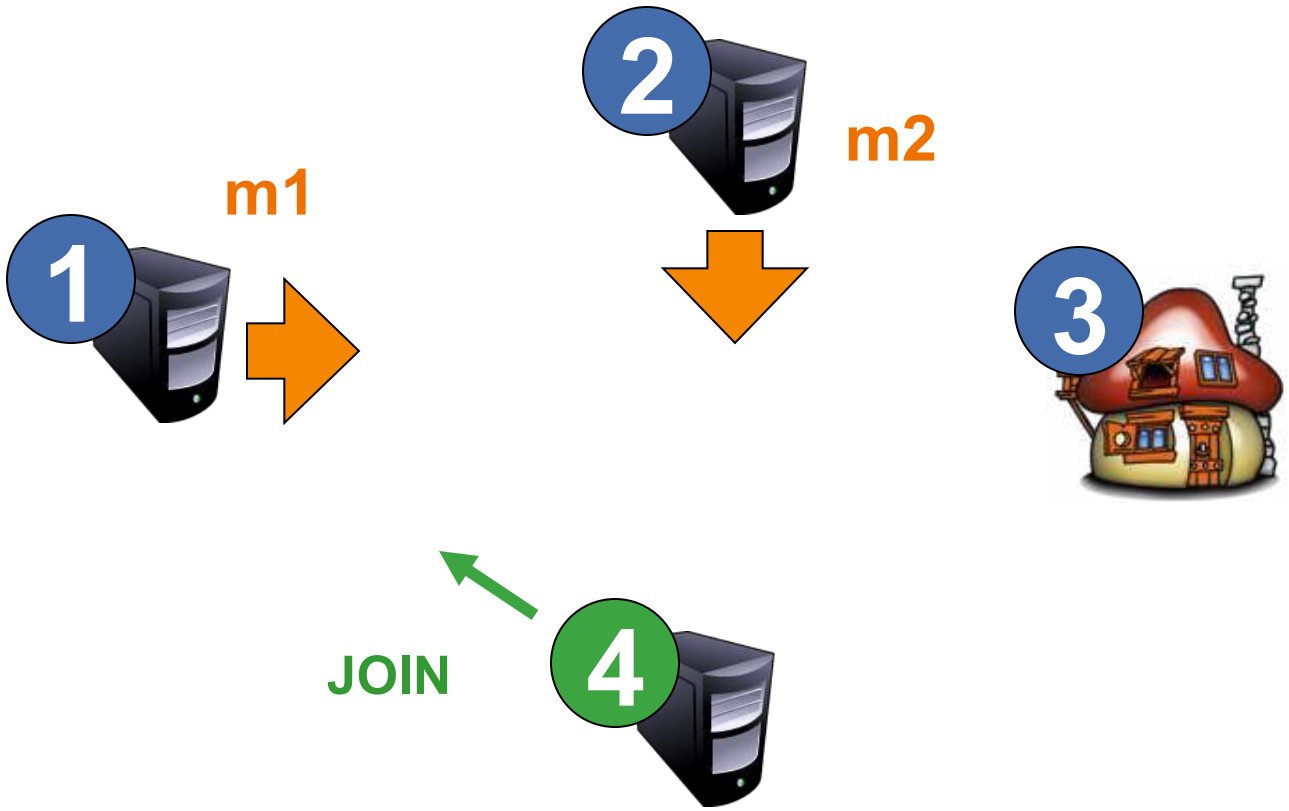
A Typical Group Service



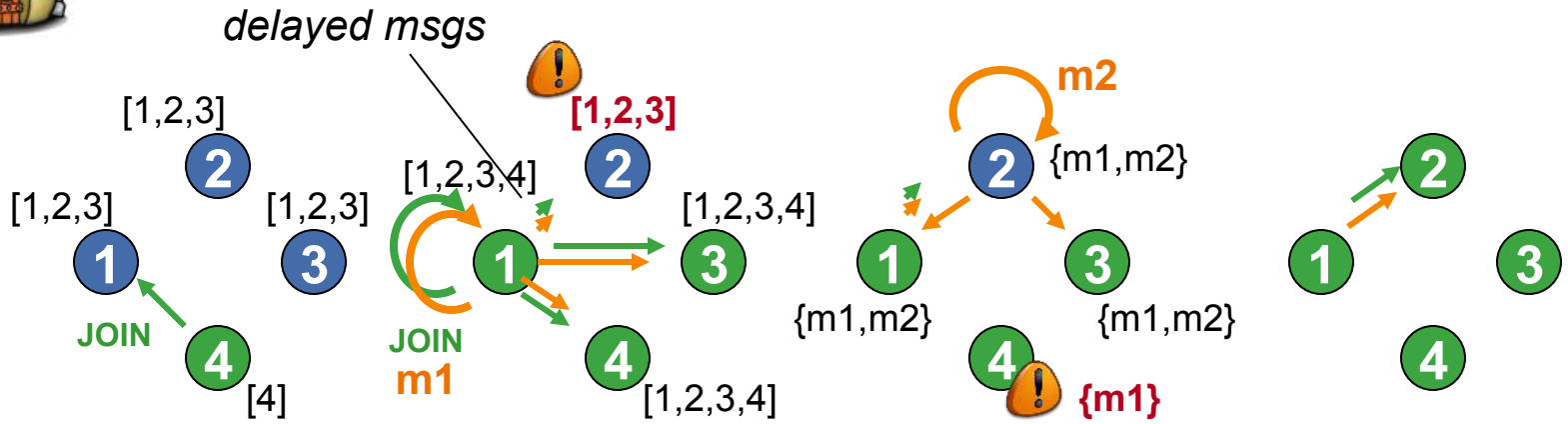

```
interface GroupCommunicationService {

    // creates a new group and returns the groups ID

    public GroupID      groupCreate();

    // Adding & Removing a member to/from a group

    public void         groupJoin   (GroupID group, Participant member);
    public void         groupLeave  (GroupID group, Participant member);

    // multicasts a message to the named group with
    // the specified delivery semantics, and
    // optionally collects a number of replies

    public Messages[]   multicast   (GroupID group, OrderType order,
                                     Messages message, int nbReplies)

}
```

More Ordering Woes

- What happens if a process joins/ leaves during multicast op?
  - ➔ Which messages will 4 get?



The Join/Leave Problem



- Once '4' has started receiving messages, it should get all the subsequent ones (I.e. again, all-or-nothing)
  - ➔ Participants need to agree on **who's in the group**!
- But agreeing requires a new multicast operation with **ordering** guarantees → **virtual synchrony**
  - ➔ The 'local view' of the sender piggybacked on sent message
  - ➔ Protocol must make sure message received in same local view as it was sent

Example: JGroups

- A toolkit for reliable multicast communication
  - In Java
  - Open source (LGPL license from Free Software Foundation)
  - Used in commercial products like **JBoss** to implement JMS

JMS

- Key Feature: Flexible Protocol Stack
  - Can be **tailored** to application needs and network characteristics
  - Can be **extended**
  - Provides a wide spectrum of properties. Focuses on **reliability**.
- JGroups created by Belan Ban at Cornell Uni (US)
  - Incidentally the home of Isis
  - Where Werner Vogel (Amazon's CTO) worked for a long time

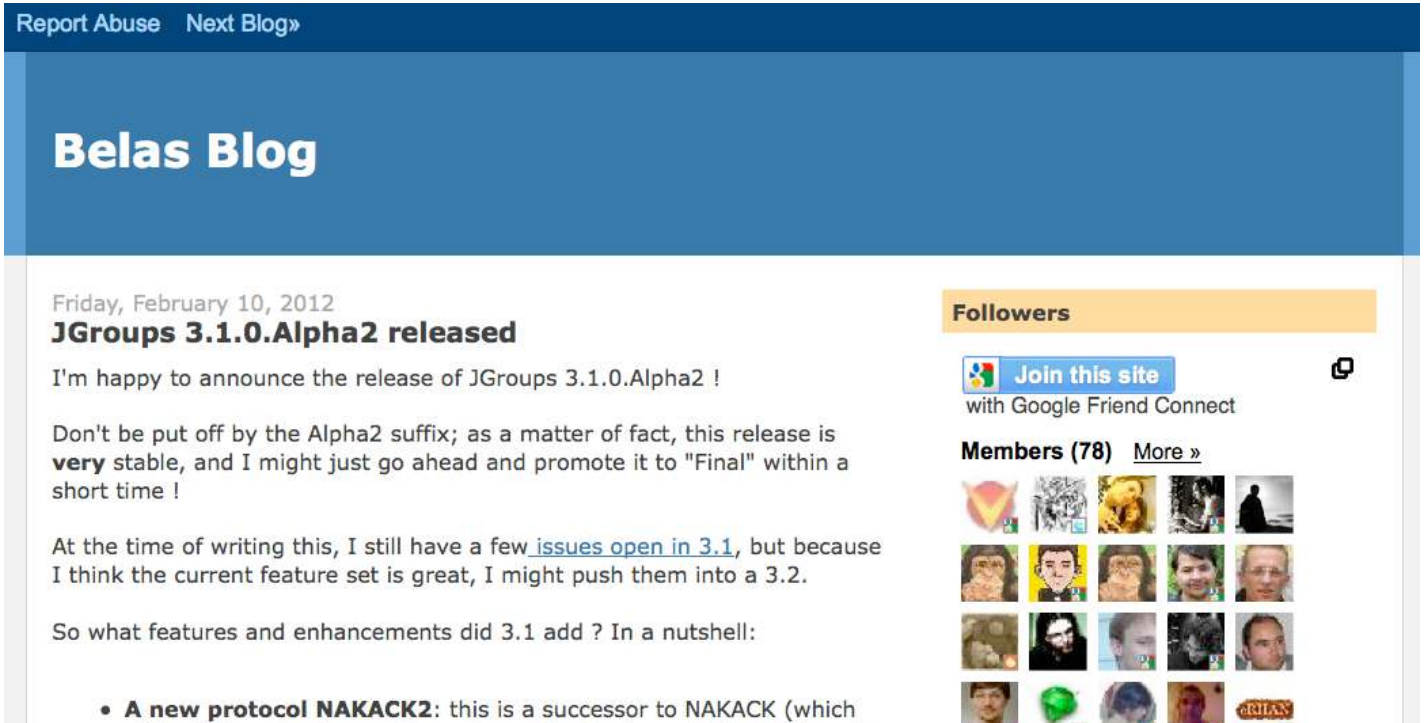
amazon.com

Where do I get it?

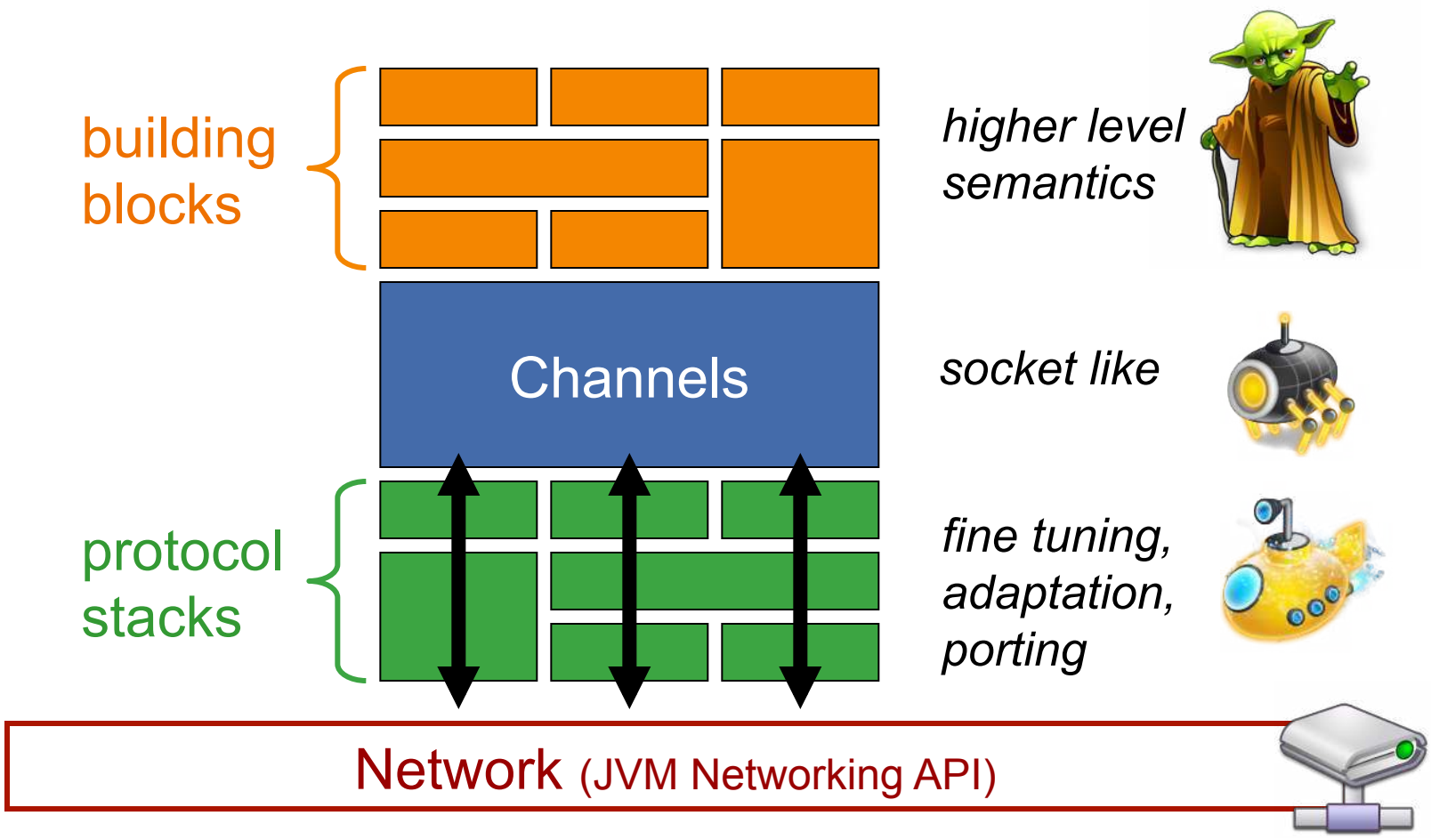
- From the JGroups Web site
  - ➔http://www.jgroups.org
- Refs:
  - ➔Doc on the site (javadoc, manuals, tutorials)
  - ➔Virtual Synchrony see http://www.theserverside.com/news/1363871/New-Features-in-JGroups-25

A developer's view

- JGroups still being actively developed by Bela Ban (2012)
  - ➔ http://belaban.blogspot.com/
  - ➔ Bela works for RedHat, who own JBoss (J2EE FOSS server)
- 1st hand experience of a distributed system developer
  - ➔ Insight into design decision, everyday problems, etc.



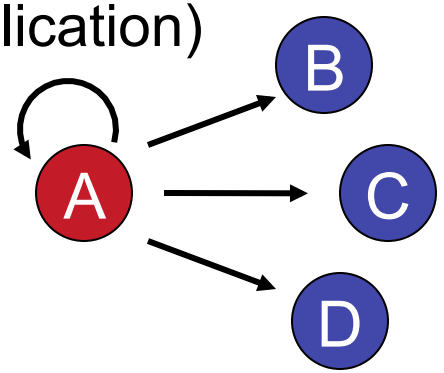
JGroups Architecture



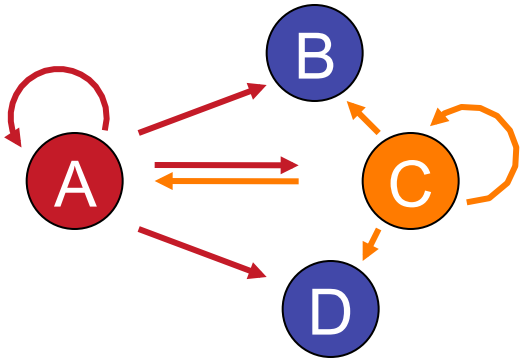
Note on reception

In JGroup any sending nodes receives its own message

- encourages symmetric code (e.g. active replication)



- sender can observe reception context (order, stabilisation)
  - ➔ E.g. total order
  - ➔ A and C needs to receive their own msg to know which one arrived first



JGroups API: Channels

- Design by **Simplicity**
  - ➔ One core class: **org.jgroups.Channel**

properties of the channel (see later)

```
String props="UDP:PING:FD:STABLE:NAKACK:UNICAST:" +
             "FRAG:FLUSH:GMS:VIEW_ENFORCER:" +
             "STATE_TRANSFER:QUEUE";

Message send_msg;
Object  recv_msg;
Channel channel=new JChannel(props);

channel.connect("MyGroup");

  send_msg=new Message(null, null, "Hello world");
  channel.send(send_msg);

  recv_msg=channel.receive(0);
  System.out.println("Received " + recv_msg);

channel.disconnect();
channel.close();
```

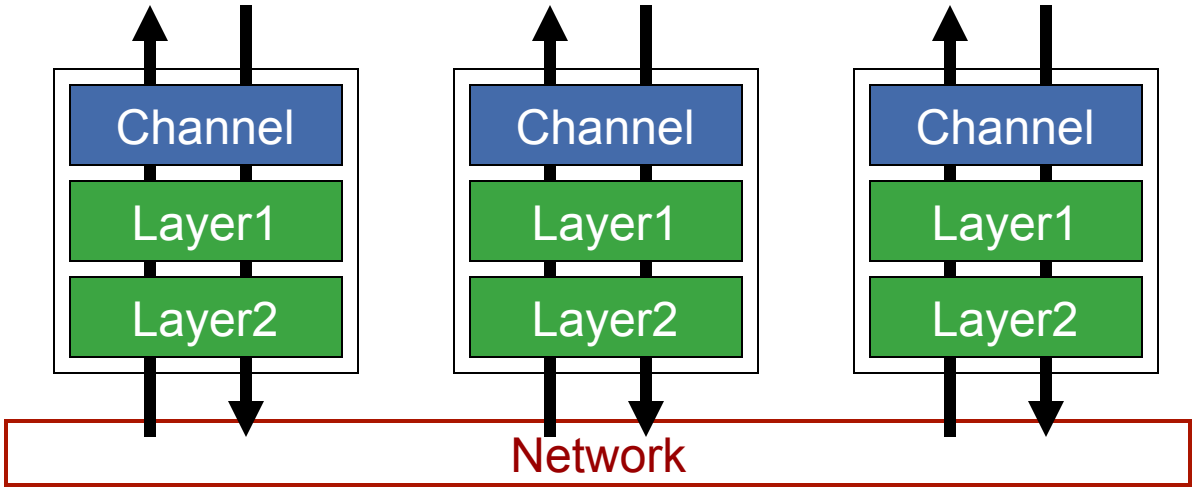
sender's address (filled up by protocol stack)

all group members

I receive my own message

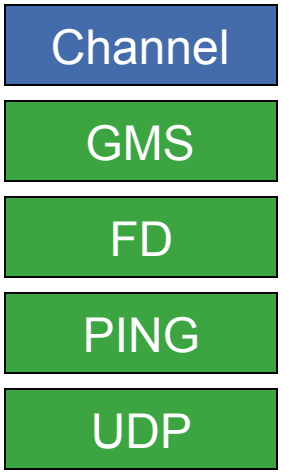
Protocol Stacks

- Each Channel instance sits on top of a protocol stack
  - ➔ Stack content defined by the property string of the Channel:
  - ➔ `Channel myChannel = new JChannel("LAYER1:LAYER2")`
- Protocol stack composed out of "layers"
  - ➔ Messages go up and down the layer stack
  - ➔ Each layer can modify, reorder, pass, drop or add a header to messages



Protocol Layers (1)

- `Channel myChan = new JChannel("UDP:PING:FD:GMS");`
  - Stack contains layers UDP, PING, FD, and GMS (bottom-up)
  - Corresponds to classes:
    ```
    org.jgroups.protocols.UDP
    org.jgroups.protocols.PING
    org.jgroups.protocols.FD
    org.jgroups.protocols.GMS
    ```
  - **UDP**: IP multicast transport based on UDP
  - **PING**: initial membership (used by GMS)
  - **FD**: Failure detection (heartbeat protocol)
  - **GMS**: Group membership protocol.



- Some expertise needed
  - Syntactically any combination possible
  - Does not necessarily make sense (e.g. GMS requires PING)

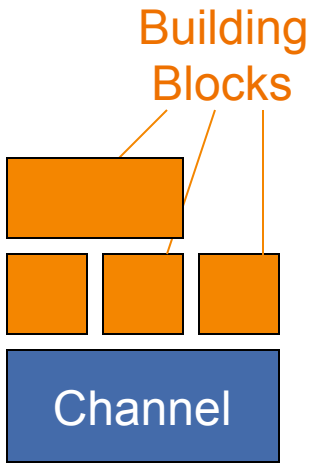
Protocol Layers (2)

Changes from 2.x to 3.x

- Example of other protocol layers
  - ~~**CAUSAL**: causal ordering layer using vector clocks~~
  - **SEQUENCER**: total ordering layer using a message sequencer
  - **NAKACK**: negative ACKs (NAKs), paired with positive ACKs
  - **STABLE**: computes the broadcast messages that are stable
  - ~~**PERF**: measures time taken by layers to process a message~~
  - **COMPRESS**: compresses the payload of a message
  - **pbcast.FLUSH**: virtual synchrony
- Constrains must be obeyed
  - Usually same layers needed in all group members
  - Dependencies: e.g. GMS needs PING, STABLE needs NAKACK

Building Blocks

- Channel class very simple
  - Similar to sockets, but group behaviour
  - Message based. No reply/request concept.
  - Active (aka blocking, pull-style) message reception
    - Explicit threading needed on top of channel for passive reception
- Building Blocks (org.jgroups.blocks package)
  - **Higher level** classes on top of Channel
  - Provides higher level programming abstractions
- Examples
  - **ReceiverAdapter**: passive reception
  - **RpcDispatcher**: remote invocation



ReceiverTest

```
public class ReceiverTest extends ReceiverAdapter {



  public static void main(String args[]) throws Exception {









  }
}
```

ReceiverTest

```
public class ReceiverTest extends ReceiverAdapter {
  public void viewAccepted(View new_view)
  { System.out.println("view: " + new_view); }
  public void receive(Message msg)
  { System.out.println("Received msg: " + msg.getObject()); }
  public static void main(String args[]) throws Exception {
    Channel chan=new JChannel();
    chan.setReceiver(new ReceiverTest());
    chan.connect("ReceiverTest");
    for(int i=0; i < 10; i++) {
      System.out.println("Sending msg #" + i);
      chan.send(new Message(null,null,"Hello "+ i));
      Thread.currentThread().sleep(1000);
    }
    chan.close();
  }
}
```

Small Quizz

- Imagine 5 processes execute the previous code
  - ➔ What's the maximum number of "`Hello i`" being printed?
  - ➔ What would be the minimum number? (no crash)
  - ➔ How many would this be with ***n*** processes?

RpcDispatcher

```
public class RpcDispatcherTest {

   public static void main(String[] args) throws Exception {

   }
}
```

RpcDispatcher

```
public class RpcDispatcherTest {
  public static int print(int number) throws Exception
  { return number * 2; }
  public static void main(String[] args) throws Exception {
    JChannel        channel =new JChannel();
    RpcDispatcher   disp    =
       new RpcDispatcher(channel, new RpcDispatcherTest());
    RequestOptions opts     =
       new RequestOptions(ResponseMode.GET_ALL, 5000);
    channel.connect("RpcDispatcherTestGroup");
    for(int i=0; i < 10; i++) {
      Thread.sleep(100);
      RspList rsp_list=disp.callRemoteMethods
      (null,"print",new Object[]{i},new Class[]{int.class},opts);
      System.out.println("Responses: "+rsp_list); }
    channel.close();
    disp.stop();
  }
}
```

RpcDispatcher: Quizz

- Assuming 3 processes execute the previous program
  - ➔ What's the max number of time `print` will be invoked?

RpcDispatcher (2)

- Provides "**Group**" Remote Procedure Call behaviour
  - Looks up the invoked method (here "print")
  - In example collects answers from all group members
  - Each group member is potentially both a client and a server!
- Not completely **transparent**
  - No stub or skeleton to hide remote invocation
  - Arguments passed as an array of Object
- **Return behaviour** can be adapted
  - GET_ABS_MAJORITY: return majority (of all members, may block)
  - GET_ALL: return all responses
  - GET_FIRST: return only first response
  - GET_MAJORITY: return majority (of all non-faulty members)
  - GET_N: return n responses (may block)

Comparison with JMS

- **JMS** is a standardised API
  - ➔ Various implementations
- **JGroups** is a library
  - ➔ Has its own API (not JMS compliant)
  - ➔ Only one implementation
  - ➔ Can be (and is) used to implement the JMS API (in JBoss)
- **JMS** is a Message Oriented Middleware
  - ➔ Higher level that plain group communication as in JGroups
  - ➔ E.g. persistence, durability, transactions
- **JMS** assumes a server based architecture
  - ➔ JGroups can be used in fully decentralised manner (or not)

Expected Learning Outcomes

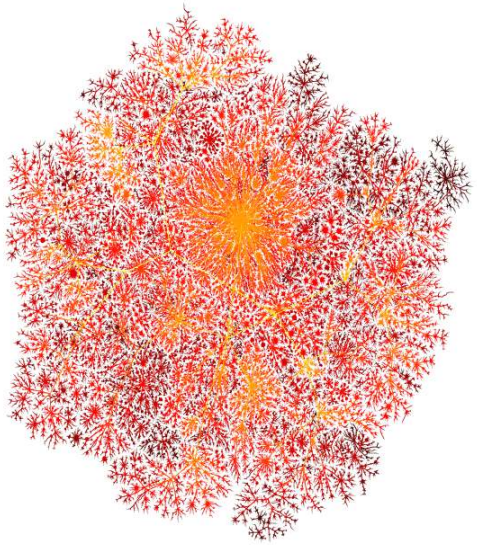
At the end of this 5th Technical Session:

- You should know what JGroups is about
- You should appreciate the basic working of the Channel class
- You should understand JGroups' protocol stack mechanism
- You should be able to compare JGroups to JMS
- You should be able to solve some of the small quizzes shown in the slides (and explain your solution)

ESIR SR

Unit 11: Introduction to Cloud Computing

François Taïani



[http://www.cheswick.com/ches/map/]

Session Outline

- Defining cloud computing
- Related technologies and precursors
  - ➔ Grid
  - ➔ Virtualisation
- Cloud architecture and cloud technologies
- An example: AWS
- Challenges (for users and providers)

Associated Reading: Cloud computing: state-of-the-art and research challenges" Qi Zhang, Lu Cheng, Raouf Boutaba. JISA May 2010

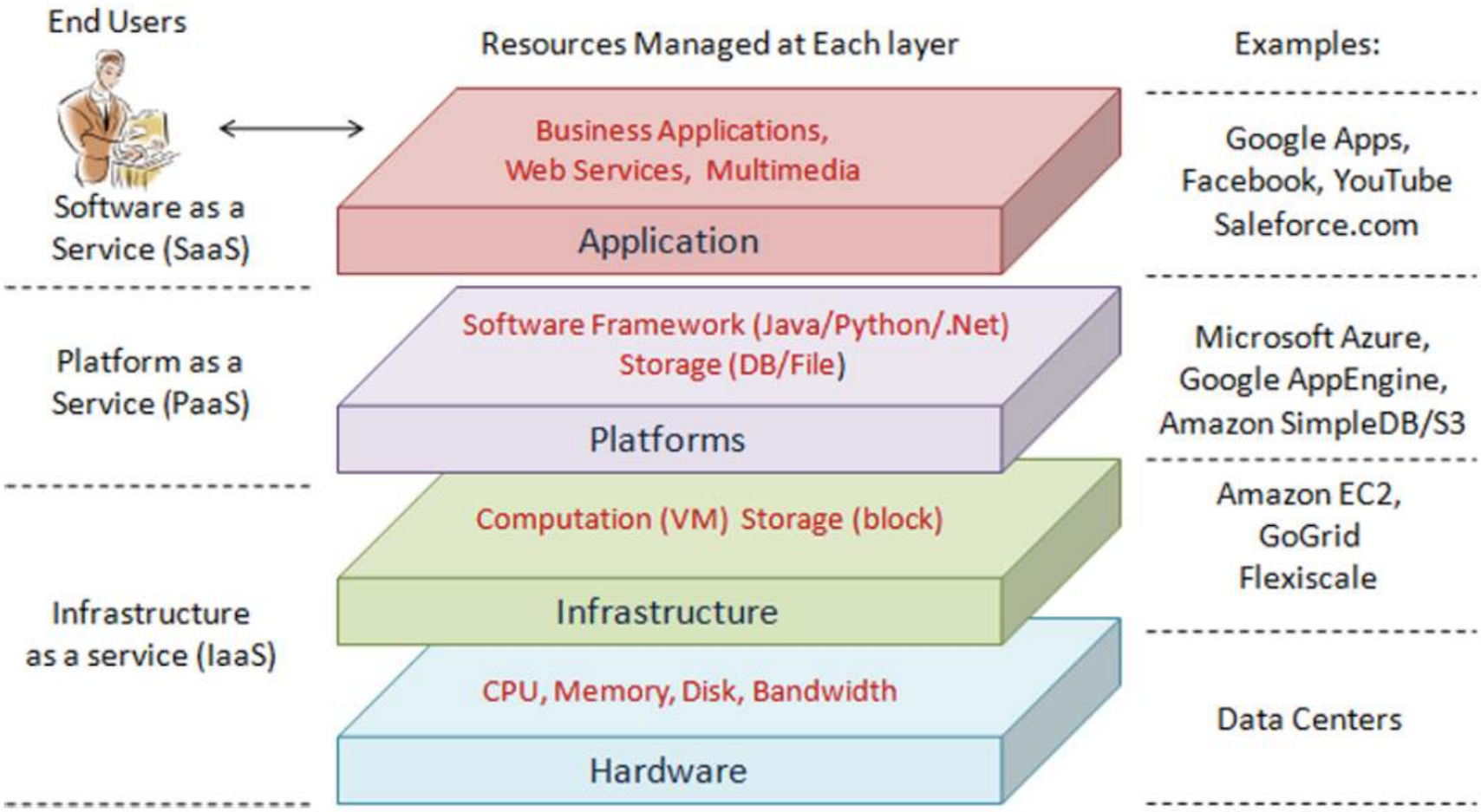
Definitions

- NIST definition
  - Cloud computing is a model for enabling convenient, **on-demand network access** to a **shared pool** of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be **rapidly provisioned** and released with **minimal management effort** or service provider interaction.
- Notes
  - underlying technologies (vritualisation, web, etc.) not really new
  - what is new is the operational model

Different Styles of Cloud Comp

- **SaaS**: Software as service
  - Usually through a Browser-based rich client
  - Linked to rich web/browser technology (AJAX, Flash)
  - E.g. Google Mail, SalesForce.com
- **PaaS**: Platform as a service
  - Library stack for on-line application development
  - At the moment each provider offers its own set of APIs
  - E.g. Windows Azure, Google App Engine, Hadoop
- **IaaS**: Infrastructure as a service
  - Based on virtualisation
  - Most flexible: users can choose which OS image to run
  - E.g. AWS, Eucalyptus

Styles of Cloud (cont.)



[source: Fu, Tian, Zhang, Cheng, and Boutaba, 2010]

Why use cloud computing?

- No up-front investment
  - ➔ pay as you go, utility-based pricing
- Lowering operating cost
  - ➔ easy to rapidly add and remove resources
  - ➔ no need to dimension for peak load
- Elastic
  - ➔ highly scalable (linked to previous point)
  - ➔ “surge computing”
- Easy to acess
  - ➔ often web-based (for SaaS, and management for I/PaaS)
- Reduce risks and contracts out expertise
  - ➔ lower staff training / HW maintenance costs

Forerunners


- **Grid** computing
  - ➔ sharing resources, principally for scientific applications
  - ➔ but does not heavily use virtualisation
  - ➔ as a result, less control to users, less flexible, more complex
- **Utility** computing / **on-demand** computing
  - ➔ idea around for a long time (60's)
  - ➔ cloud is a realisation

- **Virtualisation** (Xen, VMware, VirtualBox, …)
  - ➔ key building block, in particular for IaaS

- **Autonomic** computing (IBM, 2001)
  - ➔ cloud 'autonomic' to some extent, but not main goal

world community grid.
technology solving problems

Help us find ways to produce cleaner water.

Can one person make a difference?

Yes you can! Donate your unused computer time to World Community Grid and the Computing for Clean Water project to find more efficient and lower-cost methods for producing clean water. > **Learn More**

Join Today!

It's simple, secure, and you can contribute for free.

Computing for Clean Water

HOME | ABOUT US | RESEARCH | FORUMS | STATISTICS | MY GRID | DOWNLOAD | SELECT LANGUAGE

HELP

Who We Are

World Community Grid brings people together from across the globe to create the largest non-profit computing grid benefiting humanity. It does this by pooling surplus computer processing power. We believe that innovation combined with visionary scientific research and large-scale volunteerism can help make the planet smarter. Our success depends on like-minded individuals - like you.

How You Can Help

Download and install secure, free software that captures your computer's spare power when it is on, but idle. You will then be a World Community Grid volunteer. It's that simple! Just click the "Join Today" button below.

Join Today!

What's New

People for a **Smarter Planet**

We are now partnering with **People for a Smarter Planet**, a collective of communities that let you make a personal difference in solving some of the world's toughest challenges. Please show your support by clicking the Like button on their

Tell A Friend

Help accelerate humanitarian research even further by telling your friends about World Community Grid.

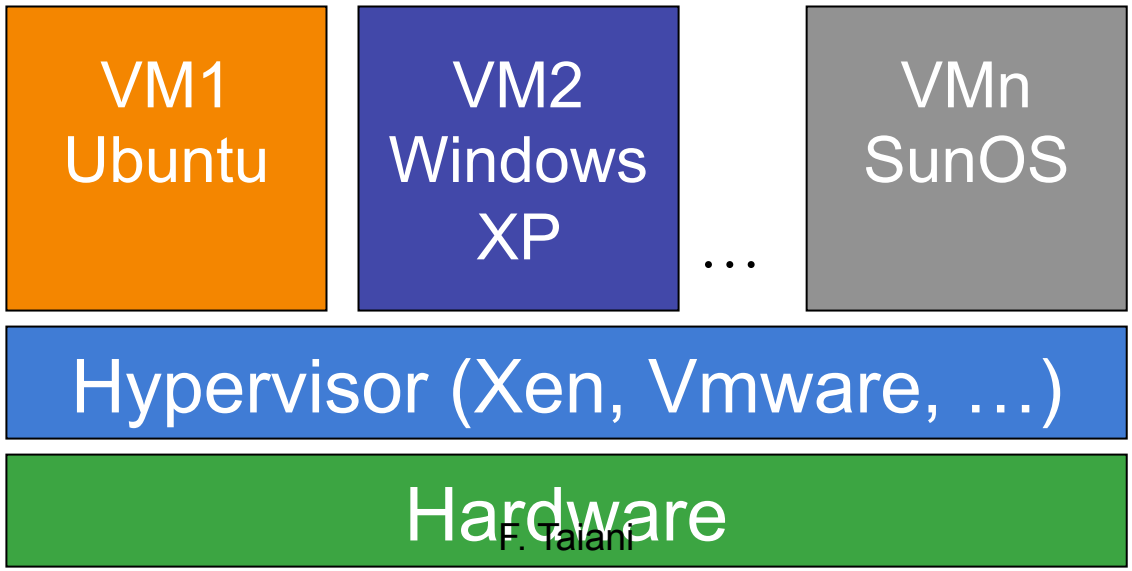
F. Taiani

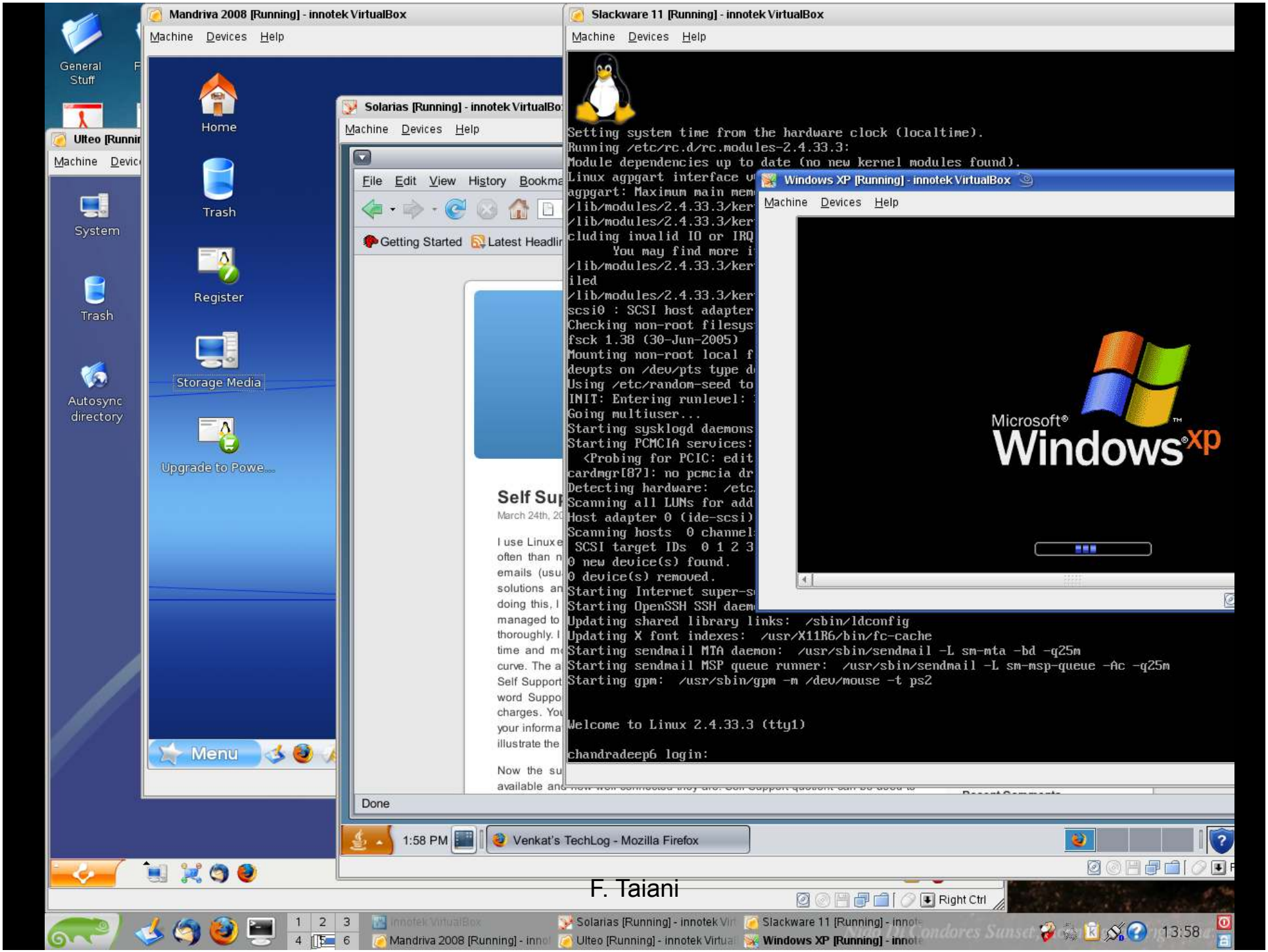
Spotlight on: Grid Computing

- Grid computing too tries to realise utility computing
- Example: BOINC (e.g. used by World Community Grid)
  - ➔ desktop grid solution, ‘volunteer computing’
  - ➔ only perfectly parallelisable tasks (no interaction)
  - ➔ e.g. molecule computation, optimisation
  - ➔ would not be able to execute a web app or MMOG
- Globus middleware
  - ➔ essentially tasks scheduler: find best fit for tasks
  - ➔ machines ‘as they are’: dependencies can be complex to tackle
  - ➔ lesser isolation than virtualisation: security a big issue
- Move to the cloud model, e.g. Nimbus
  - ➔ http://www.nimbusproject.org/

Spotlight on: Virtualisation

- Benefits
  - uncouples OS from actual hardware
  - allows hardware consolidation
  - strong isolation (more than between OS processes)
  - solves OS heterogeneity problem
  - highly flexible (dynamic creation, migration, checkpoints)

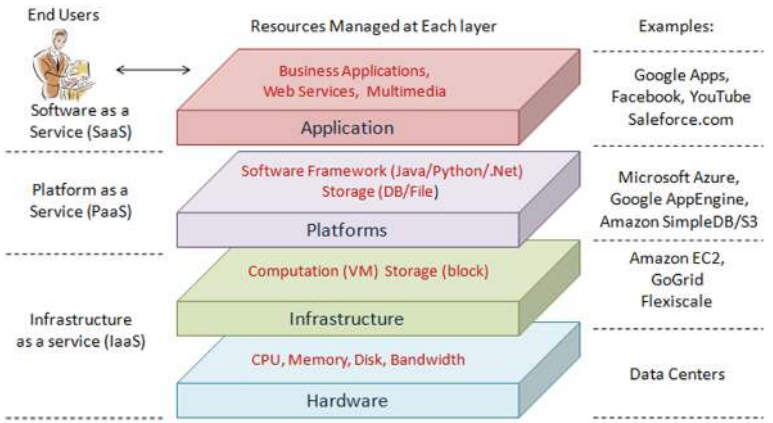



Mandriva 2008 [Running] - innotek VirtualBox
Machine Devices Help
Slackware 11 [Running] - innotek VirtualBox
Machine Devices Help
Solarias [Running] - innotek VirtualBox
Machine Devices Help
Windows XP [Running] - innotek VirtualBox
Machine Devices Help
Microsoft® Windows® xp
General Stuff
Home
Trash
Register
Storage Media
Upgrade to Powe...
System
Trash
Autosync directory
Menu
File Edit View History Bookma
Getting Started
Latest Headlir
Self Sup
March 24th, 20
Done
Setting system time from the hardware clock (localtime).
Running /etc/rc.d/rc.modules-2.4.33.3:
Module dependencies up to date (no new kernel modules found).
Welcome to Linux 2.4.33.3 (tty1)
chandradeep6 login:
1:58 PM
Venkat's TechLog - Mozilla Firefox
Right Ctrl
13:58

F. Taiani

Cloud architecture

- The cloud uses itself!
  - ➔ SaaS providers typically rely on a PaaS cloud
  - ➔ PaaS providers on an inderlying IaaS cloud
- Explain why some players in the market
  - ➔ Amazon uses its cloud for itself before opening it up
  - ➔ Google was first a SaaS provider before opening up
- Some clouds remain private
  - ➔ Some internal Google service not available externally (see 3.1)
- Characteristic
  - ➔ modular
  - ➔ loose coupling between layers



[source: Zhang, Cheng, and Boutaba, 2010]

Types of cloud

- **Public clouds**: available to general public
  - ➔ +: no initial investment, minimal risks
  - ➔ -: lack of control, problematic for many businesses
- **Private clouds**: internal consolidation / provisioning
  - ➔ many of today's public cloud arose from private clouds
  - ➔ but what's the difference to traditional server farms?
- **Hybrid clouds**: trying to the combine the above two
  - ➔ but where to split? how to coordinate?
- **Virtual Private Clouds**
  - ➔ adds VPN to public cloud
  - ➔ more security, smoother transition path

Cloud Characteristics

- **Multitenancy**
  - ➔ multiple users form multiple organisations
- **Elasticity / Dynamic resource provisioning**
  - ➔ receive resources / QoS on-demand
- **Resource Sharing**
  - ➔ spare cloud resources transparently applied to maintain QoS
- **Horizontal scaling (for cloud providers, private clouds)**
  - ➔ cloud capacity extensible in small increments
- **Metering**
- **Security**
- **Availability**
- **Operability**

[sources: Ramakrishnan 2009 / Zhang et al 2010]

Cloud technologies (1)

Beneath it all: The data-center

- Data Centre Architecture
  - racks, blades, switches, routers
  - hierarchical and (partially) redundant design
  - high networking capability (10Gbps and more)
  - virtualisation + migration used for reconfiguration
- Trend: Modular Data Centre
  - One module: size of a shipping container, highly integrated
  - Geo-dispersed to form a full data-centre

Cloud technologies (2)

Supporting services

- cloud file systems
  - GFS, Google File System
  - HFS, Hadoop File System (similar to GFS)
  - quite different properties from traditional file systems!
  - More on GFS tomorrow!
- software framework
  - MapReduce from Google
  - Hadoop from Apache / Yahoo (simialr to MapReduce)
  - More on these tomorrow
- coordination services
  - e.g. chubby distributed locking service (Google)

Spotlight on: AWS

- Set of multiple services
  - Computing: EC2
  - Storage: S3, SimpleDB, EBS
  - Access through HTTP, with SOAP or REST
- EC2 + EBS: IaaS
  - based on Xen virtualization engine
  - any Xen image (any OS, retrieved from S3)
  - users have full control (++) of software stack
  - but means automatic scaling difficult (--)
  - placements based on Regions and Availability Zones
- S3: PaaS
  - ‘objects’ (up to 5 GB) grouped in ‘buckets’

Example: REST on S3

- deletes the puppy.jpg file from the mybucket bucket
- HTTP request (note DELETE rather than GET)

```
DELETE /puppy.jpg HTTP/1.1
User-Agent: dotnet
Host: mybucket.s3.amazonaws.com
Date: Tue, 15 Jan 2008 21:20:27 +0000
x-amz-date: Tue, 15 Jan 2008 21:20:27 +0000
Authorization: AWS 0PN5J17HBGZHT7JJ3X82:k3nL7gH3+PadhTEVn5EXAMPLE
```

[source: AWS documentation]

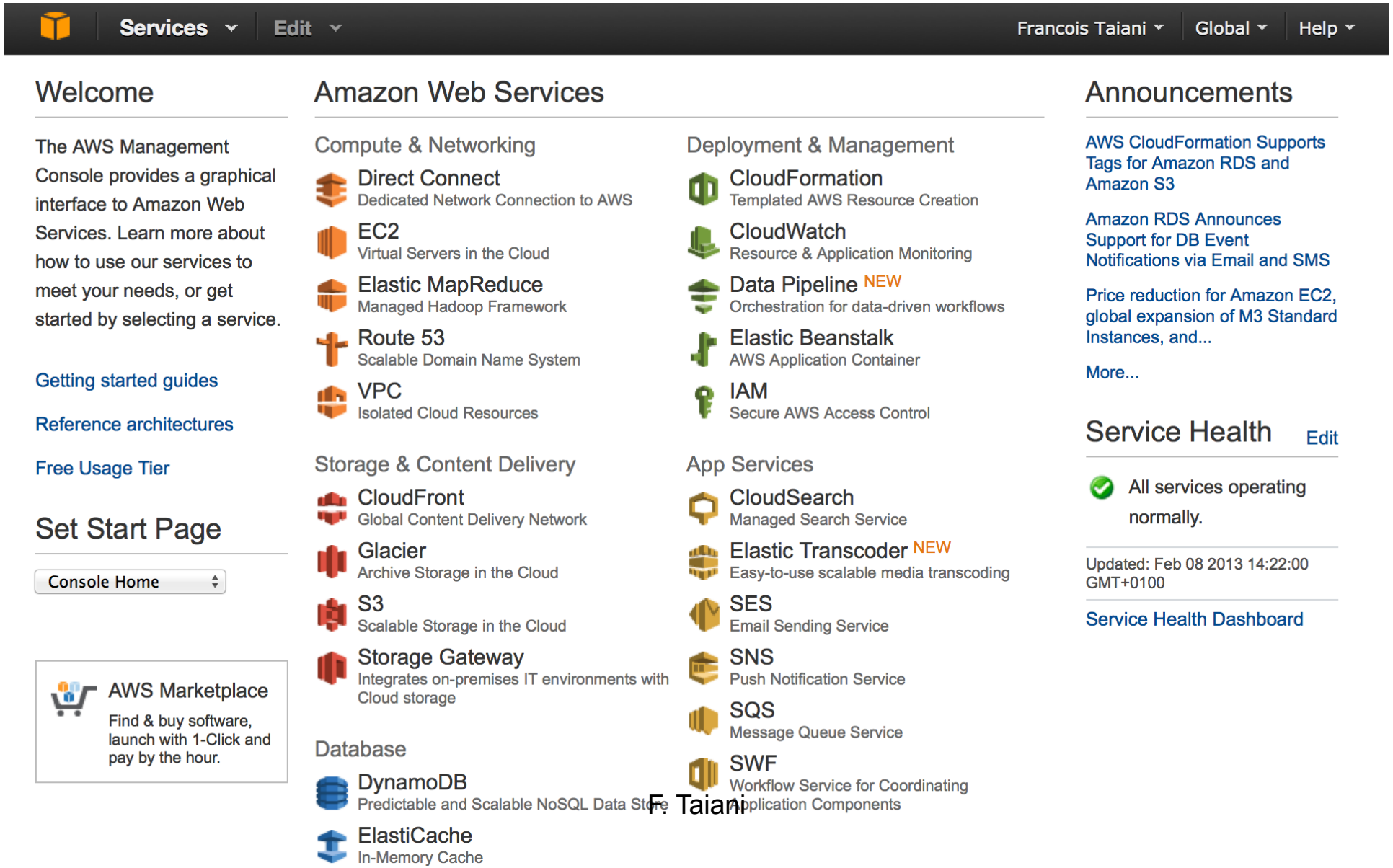
Example: SOAP on EC2

- This example starts the i-10a64379 instance
  - ➔ would be encapsulated in SOAP message

```
<StartInstances xmlns="http://ec2.amazonaws.com/doc/2010-08-31/">
 <instancesSet>
  <item>
   <instanceId>i-10a64379</instanceId>
  </item>
 </instancesSet>
</StartInstances>
```

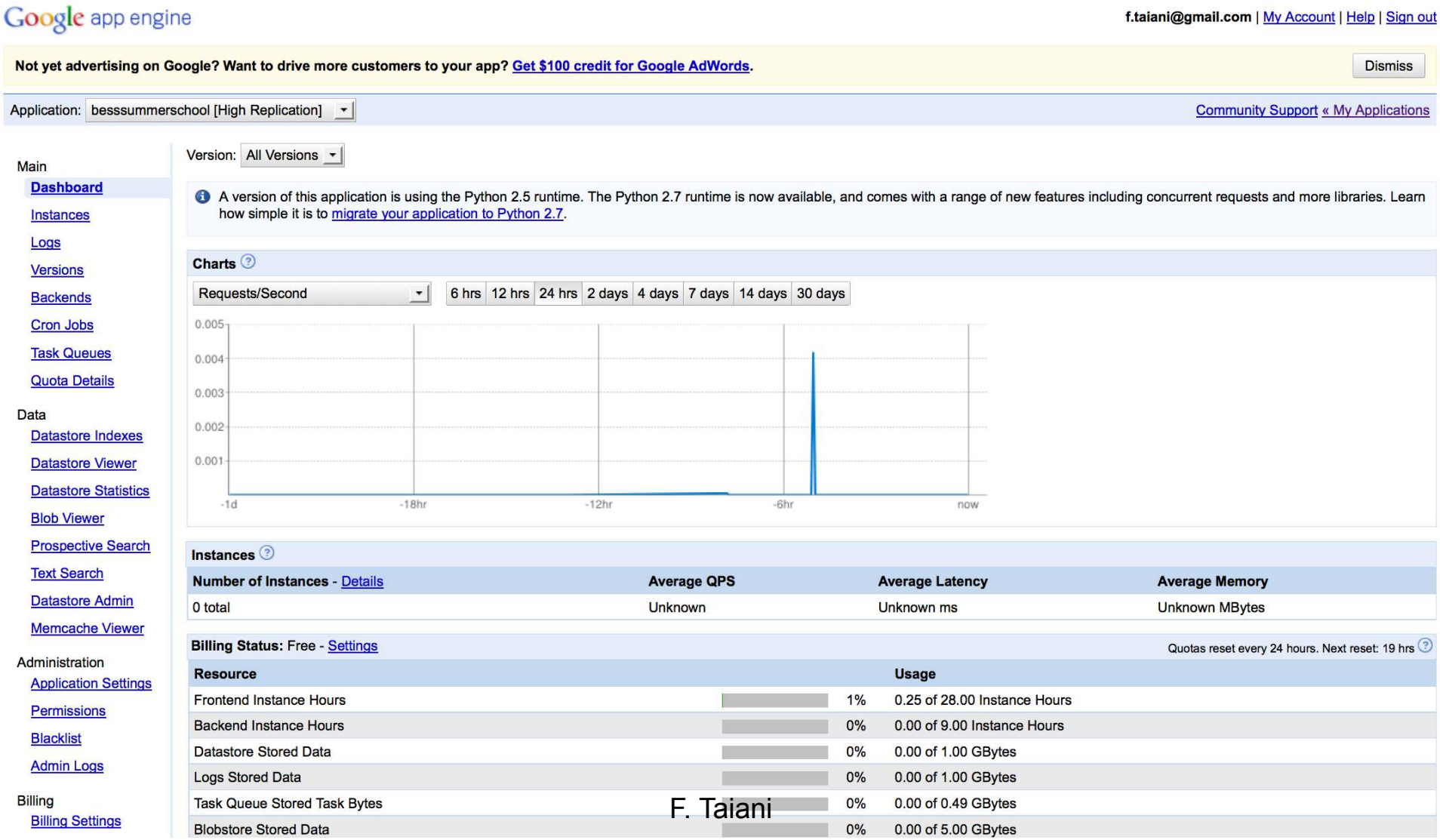
[source: AWS documentation]

AWS Web Console



F. Taiani

Similar at Google App Engine



F. Taiani

Cloud Challenges (1)

For users:

- Standards (in particular for PaaS, and partially for SaaS)
  - ➔ Key for portability and to avoid lock-in
- Monitoring and SLA
  - ➔ Key for informed choice and trust
- Security
  - ➔ confidentiality
  - ➔ auditability (any tampering?)
- Systemic risks
  - ➔ E.g. S3 failure

Bits

Business ▪ Innovation ▪ Technology ▪ Society

February 15, 2008, 1:32 PM

Amazon's S3 Cloud Has a Dark Lining for Start-Ups

By BRAD STONE

UPDATE: *see updated Amazon statement below.*

A few days after the BlackBerry e-mail system's latest downtime, Amazon is giving companies another reason to worry about outsourcing company-critical functions.

Amazon's S3 service, which offers cheap, accessible Web storage for hundreds of thousands of companies, went down this morning at around 7:30 a.m. Eastern time and is only now slowly creeping back up, delivering high error rates, according to various bloggers and posters on Web forums.

S3, one of Amazon's stable of Web services, lets businesses store their data "in the cloud," avoiding the need to operate their own servers. It is part of the same online infrastructure that Amazon uses to run its own business. Over 330,000 developers have registered to use Amazon Web Services, up more than 30,000 from last quarter, according to Amazon's recent quarterly earnings announcement.

An Amazon spokesman said: "We've resolved this issue and performance is returning to normal levels for all Amazon Web Services that were impacted. We understand the critical importance of our services to our customers, which is why operational excellence is our highest priority."

F. Taiani

source: http://bits.blogs.nytimes.com/

S3 Failure

- 15 February 2010, during night
  - elevated levels of authenticated requests from multiple users
  - not been monitoring the proportion of authenticated requests
  - cryptographic requests consume more resources
  - cascading effect on more users
  - authentication service over its maximum capacity
  - authentication service also performs account validation for S3
  - S3 unavailable by 8:40pm
  - Not fully restored until 5pm

S3 Failure

- Diagnosis
  - ➔ gossip used internally by Amazon authentication server
  - ➔ during failure: many servers almost all of their time gossiping
  - ➔ a large amount of servers that had failed while gossiping
- Explanation
  - ➔ handful of messages had a single bit corrupted
  - ➔ message was still intelligible
  - ➔ but the system state information was incorrect
  - ➔ caused authentication server to run amok
- Solution: kill all communications and restart
  - ➔ but took a while to understand what was wrong
- Source: http://status.aws.amazon.com/s3-20080720.html

BBC Mobile

News | Sport | Weather | iPlayer | TV | R

NEWS TECHNOLOGY

Home | World | UK | England | N. Ireland | Scotland | Wales | Business | Politics | Health | Education | Entertainment & Arts

28 October 2010 Last updated at 10:53

Big name firms form alliance to drive cloud standards

By Maggie Shiels
Technology reporter, BBC News, Silicon Valley

Some of the world's biggest companies are using their market clout to demand that computer equipment makers change the way they make their machines.

The 70 firms, which includes BMW, Shell and Marriott Hotels, said systems that do not work together are holding back the spread of cloud computing.

Intel referred to the launch of the Alliance as "Cloud Independence Day"

The companies have formed the Open Data Alliance Centre to push for unified standards for technology.

The businesses involved account for more than $50bn (£32bn) in IT spending.

Cloud challenges (2)

For providers:

- Automatic scaling
  - → dynamic reorganising resource to meet agreed QoS not trivial
  - → need to understand link between low-level decision and QoS
  - → long running problem, complex (see R-Capriccio on Friday)
- VM migration
  - → a lot of progress already (downtime < 1s)
  - → but avoiding ‘hotspots’ still a challenge
  - → being efficient and consistent challenging
- Server consolidation
  - → how to pick the VMs to put together?
  - → must take multiple resources into account
  - → bin-packing problem with a time dimension!

Cloud challenges (3)

For providers (cont.)

- Energy consumption and cooling
  - ➔ > 50% of operational expenditure
  - ➔ energy efficient networks, board, etc. needed
  - ➔ integration with application needs (should not hurt perf.)
- Improved frameworks
  - ➔ how to make them better adapt to application characteristic
  - ➔ e.g. placement, scheduling, replication
- Continent scale engineering
- Novel architectures
  - ➔ nano-clouds, ‘volunteer’ clouds

Expected learning outcomes

At the end of this session, you should be able to

- Provide a general definition of cloud computing
- Explain the reasons why cloud computing might be attractive
- Explain how cloud computing relate to other related technologies
- Present the different styles and types of cloud
- Discuss a representative examples of cloud infrastructure
- Discuss current challenges facing cloud computing

References

- "An Overview of Cloud Computing @ Yahoo!"
  by Raghu Ramakrishnan
  - ➔ http://www.eu.apachecon.com/page_attachments/0000/0194/ApacheCon-09cloud.ppt
- "Cloud computing: state-of-the-art and research challenges"
  Qi Zhang, Lu Cheng, Raouf Boutaba. Journal of Internet Services and Applications, Vol. 1, No. 1. (2010), pp. 7-18.
- "Cloud Comedy, Cloud Tragedy"
  by William Vambenepe (practice oriented)
  - ➔ http://stage.vambenepe.com/