# Distributed Systems - Overview

- some systems background/context
- some legal/social context
- development of technology – DS evolution
- ** DS fundamental characteristics **
- software structure for a node
- model/architecture/engineering for a DS
- architectures for large-scale DS
  - federated administration domains
  - integrated domain-independent services
  - detached, ad-hoc groups

# Life is grim.

# Costly Failures in Large-Scale Systems

- UK Stock Exchange - share trading system
  - abandoned 1993, cost £400M
- US tax system modernisation
  - scrapped 1997, cost $4B
- UK ASSIST, statistics on welfare benefits
  - terminated 1994, cost £3.5M
- London Ambulance Service Computer Aided Despatching (LASCAD) scrapped 1992, cost £7.5M, 20 lives lost in 2 days

This may seem very 1990s but things haven't improved

- NHS National Programme for IT, “the world's biggest civil information technology programme”

# Why high public expectation?

## Web applications work OK

e.g. information services: trains, postcodes, phone numbers

e.g. online banking

e.g. airline reservation

e.g. conference management

e.g. online shopping and auction

e.g. Facebook, Twitter, flickr, ...

Properties: often read mostly, server model, client-server paradigm, closely coupled, synchronous interaction (request-reply), single-purpose, (often) private sector

# Public-Sector Systems

## healthcare, police, social services, immigration, passports, DVLA (driver + vehicle licensing), court-case workflow, tax, independent living for the aged and disabled, …

- bespoke and complex
- large scale
- many types of client, meaning many roles
- web portal interface, but often not web-service model
- long timescale, high cost
- ubiquitous and mobile computing – still under research (!)
- former policy of competition and independent procurement
- current policy requiring interoperation...which changes, of course
- legislation and government policy

# Some Legal/Policy Requirements - 1

"*patients may specify who may see, and not see, their electronic health records (EHRs)*" - exclusions

"*only the doctor with whom the patient is registered (for treatment) may e.g. prescribe drugs, read the patient's EHR, etc.*" - relationships

"*the existence of certain sensitive components of EHRs must be invisible, except to explicitly authorised roles*"

# Some Legal/Policy Requirements - 2

"*buses should run to time and bus operators will be punished if published timetables are not met*."

so bus operators may refuse to cooperate in traffic monitoring, even though monitoring could show that delay is often not their fault.

# Data Protection Legislation et al.

*Gathered data that identifies individuals must not be stored:*

CCTV cameras: software must not *recognise* people and store identities with images

(*thermal imaging (infra-red) - just monitor/count*)

Vehicle number plate recognition: must not be associated with people then stored with identities

(*only police allowed to look up*)

Police records: accusations that are not upheld?

*Sally Geeson murder - previous army records of LC Atkinson*

*Soham murders – previous police records of Huntley;*

Govt. now require interaction between counties

UK Freedom of Information Act

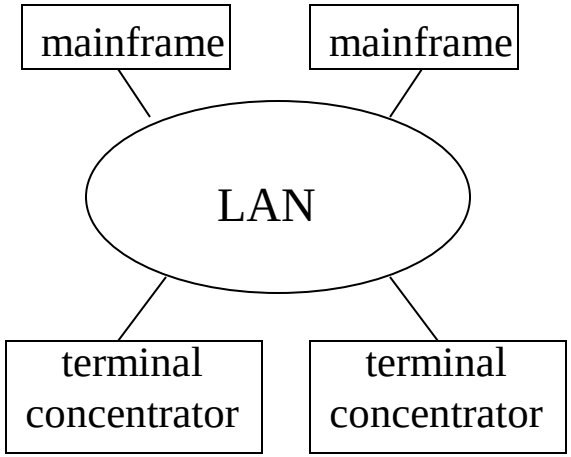
# Rapidly Developing and New Technology

- can't ever design a "*second system*", it's always possible to do more next time so you face dangerously-shifting goalposts
- rapid obsolescence - incremental *growth* usually not sustainable long-term (unlike e.g. telephone system)
  - a current software engineering research area
- but *big-bang* deployment is a bad idea, so have to design for incremental *deployment*
- mobile workers in healthcare, police, utilities etc.: integration of wired and wireless networks
- ubiquitous computing: integration of camera and sensor data

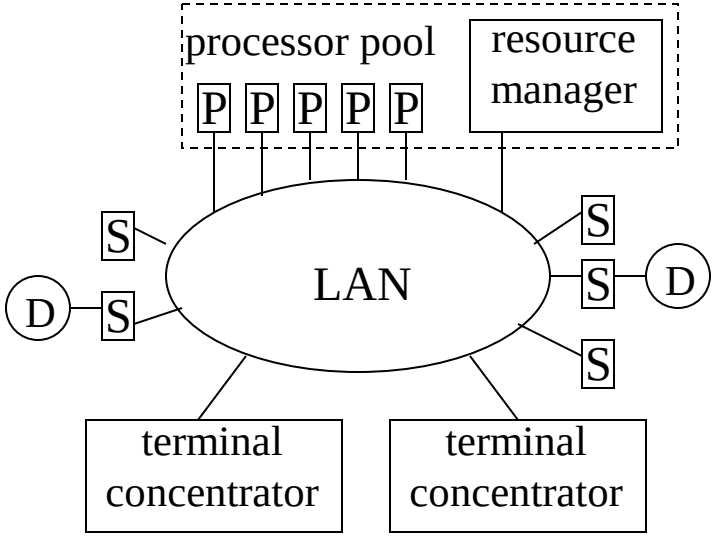
# DS history: technology-driven evolution

- Fast, reliable (interconnected) LANs (e.g. Ethernet, Cambridge Ring) made DS possible in 1980s
- Early research was on distribution of OS functionality
  1. terminals + multiaccess systems
  2. terminals + pool of processors + dedicated servers (Cambridge CDCS)
  3. Diskless workstations + servers (Stanford)
  4. Workstations + servers (Xerox PARC)

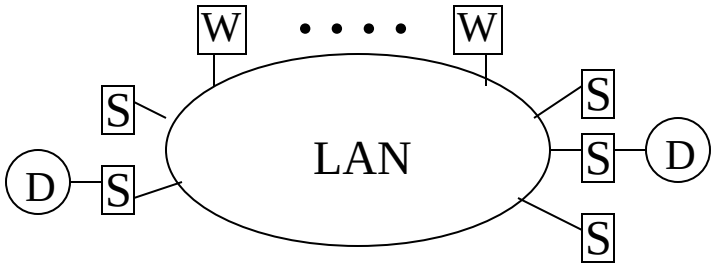
1. LAN as terminal switch to multiaccess systems



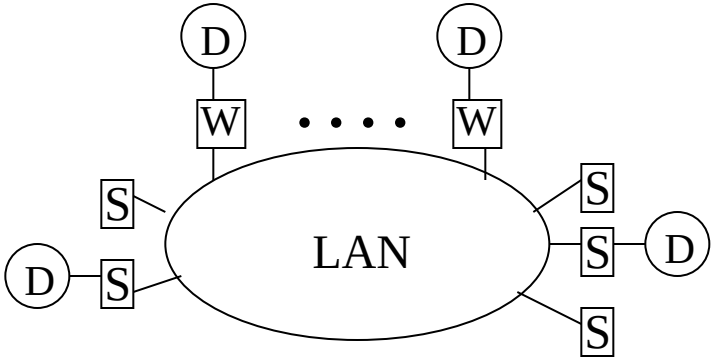

2 terminals + processor pool + servers




3 diskless workstations + servers




4 workstations + servers



# Technology-driven evolution – comms.

- WANs *quickly* became as high bandwidth and reliable as LANs
- Distributed database research such as data-shipping vs. query-shipping became obsolete in the 1990s
- Web services created new problems such as flash crowds
- Bandwidth had become high but latency was and remains a problem, due to end-system processing time for huge numbers of clients
- We shall return to this...

# How to think about Distributed Systems

- fundamental characteristics
- software structure for a node
- model/architecture/engineering for a system

# DS fundamental characteristics

1. Concurrent execution of components
2. Independent failure modes
3. Transmission delay
4. No global time

Implications:

2, 3 - can't know why there's no reply – node/comms. failure and/or node/comms. congestion

4 - can't use locally generated timestamps for ordering distributed events

1, 3 - inconsistent views of state/data when it's distributed

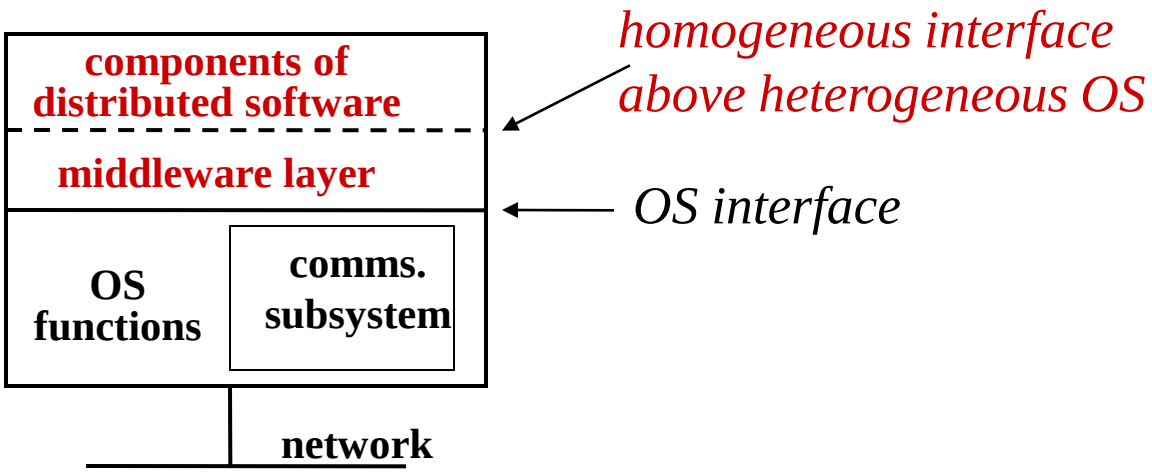
1 - can't wait for quiescence to resolve inconsistencies

# single node - software structure

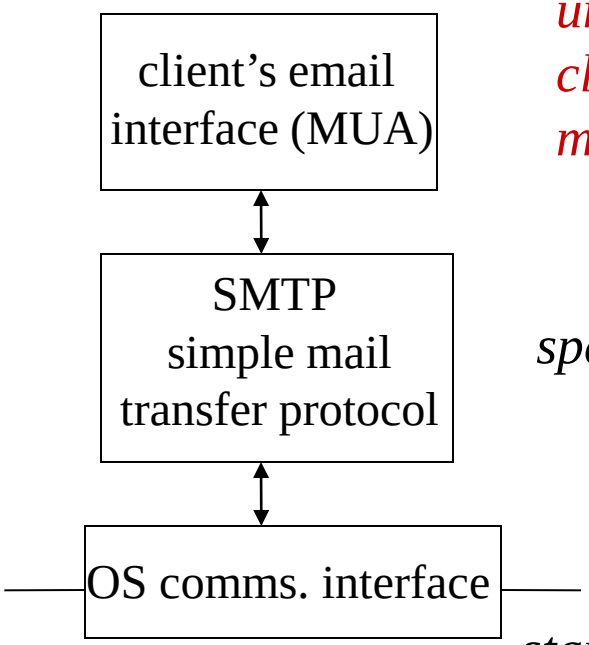
Support for distributed software may be:

directly by OS in a cluster (distributed OS design) – not the focus of this course

by a software layer (middleware) above potentially heterogeneous OS



# Distributed application structure – email, news, ftp

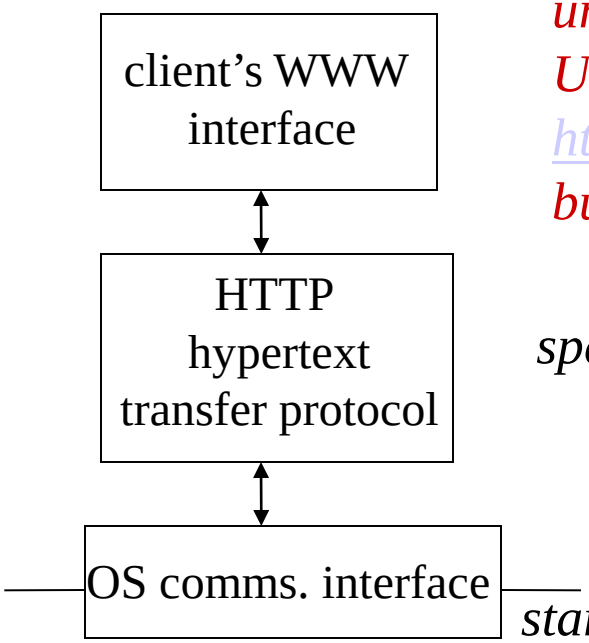



*unique names required: see lecture DS-5*
*clients: jmb25@cl.cam.ac.uk*
*messages: messageID*

*specific application protocol*

*standard comms. supporting all applications*

# Distributed application structure – web documents




*unique names required: originally for documents*
*URLs*
*http://www.cl.cam.ac.uk/...*
*built above DNS naming, see lecture DS-5*

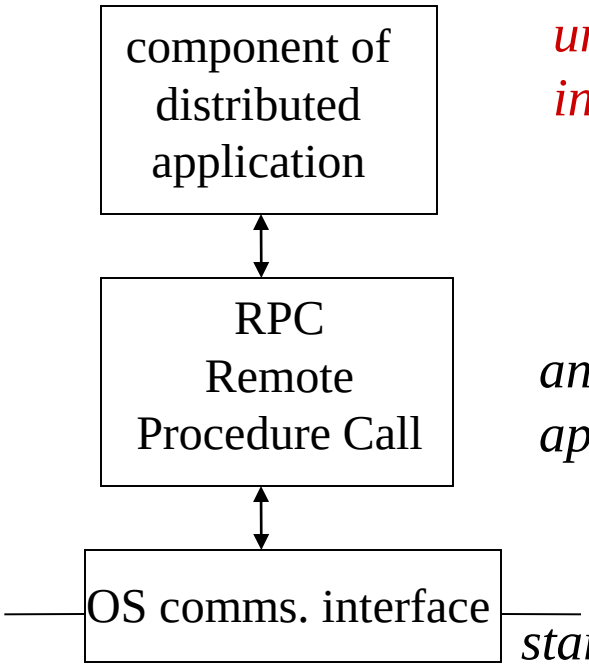
*specific application protocol*

*standard comms. supporting all applications*

A browser interface came to be used for **general** distributed applications

W3C standards for Web Services – see lectures DS-4, DS-7

# Distributed application structure – general support example




*unique names required:*
*interfaces, procedures, (later – objects)*

*an early example of a general application-level protocol*

*standard comms. supporting all applications*

RPC is an early example of a protocol above which distributed applications may be developed. RPC examples: ISO-ODP, OSF-DCE
A middleware also includes services above the RPC layer

# Open and proprietary middleware

- Open: evolution is controlled by standards bodies (e.g. ISO, NIST) or consortia (e.g. OMG, W3C). Requests for proposals (RFPs) are issued, draft specifications published with RFCs (requests for comments). Compromise is common.
- Closed, proprietary: can be changed by the owner (clients need to buy a new release). Consistency across versions is not guaranteed. Plus *"embrace, enhance, extinguish"*; yachtware.

## Related issues:

- single/multi language: can components be written in different languages and interoperate?
- open interoperability: is desirable across middlewares (including different implementations of the same middleware)

# DS Design: model, architecture, engineering

Programming model of distributed computation:

- What are the named entities? objects, components, services,..

- How is communication achieved?
  - synchronous/blocking (request-response) invocation
    e.g. client-server model
  - asynchronous messages e.g. event notification model
  - one-to-one, one-to-many?

- Are the communicating entities closely or loosely coupled?
  - must they share a programming context?
  - must they be running at the same time?

System architecture: the framework within which the entities in the model interoperate

- Naming
- Location of named objects
- Security of communication, as required by applications
- Authentication of participants
- Access control / authorisation
- Replication to meet requirements for reliability, availability

May be defined within *administration domains*

Need to consider *multi-domain systems* and interoperation within and between domains

# System engineering: implementation decisions

- Placement of functionality: client libraries, user agents, servers, wrappers/interception
- Replication for failure tolerance, performance, load balancing –> consistency issues
- Optimisations e.g. caching, batching
- Selection of standards e.g. XML, X.509
- What *"transparencies"* to provide at what level:

  (transparent = hidden from application developer: needn't be programmed for, can't be detected when running).

  distribution transparency: location? failure? migration?

  may not be achievable or may be too costly

# Architectures for Large-Scale, Networked Systems

Individual user using globally available service

Single administration domain

Federated administration domains

Independent, external services - to be integrated

Detached, ad-hoc, anonymous groups;
anonymous principals, issues of trust and risk

# Federated administration domains: Examples

- national healthcare services:
  many hospitals, clinics, primary care practices
- national police services:
  many county police forces
- global company:
  branches in London, Tokyo, New York, Berlin, Paris...
- transport
  County Councils responsible for cities, some roads
- active city:
  fire, police, ambulance, healthcare services.
  mobile workers
  sensor networks e.g. for traffic/pollution monitoring

# Federated domains - characteristics

- names: administered per domain (users, roles, services, data-types, messages, sensors, ...)
- authentication: users administered within a domain
- communication: needed *within* and *between* domains
- security: per-domain firewall protection
- policies: specified per domain e.g. for communication, access control *intra and inter-domain*, plus some external policies to satisfy government, legal, and institutional requirements
- high trust and accountability within a domain,
  known trust between domains

# Independent, External Services - Examples

- commercial web-based services
  e.g. online banking, airline booking etc.
- national services used by police and others
  e.g. DVLA, court-case workflow
- national health services
  e.g. national Electronic Health Record (EHR) service
- e-science (grid) databases and generic services
  e.g. astronomical, transport, medical *databases*
  for *computation* or *storage*
- e-science may support “virtual organisations” – collaborating groups across several domains

# Independent, external services - characteristics

- naming and authentication

  may be of individuals via trusted third parties (TTPs)

  and/or via home domain of client

- access control policies

  related to client roles in domains and/or individuals

  support for *"virtual organisations"* spanning domains

- need for: accounting, charging, audit

  these may be done by trusted third parties

  a basis for mutual trust (service done, client paid)

- trust

  based on evidence of behaviour

  clients exchange experiences, services monitor and record

  assume full connectivity, e.g. TTPs can authenticate/identify

# Examples of detached ad hoc groups and the need for trust

- Commuters regularly play cards on the train
- Auctions – build up trust of an ID through small honoured purchases, then default on a big one
- E-purse purchases – trust in system
- Recommendations: e.g. in a tourist scenario - restaurants, places to visit. Recommendations of people and their skills.
- Wireless routing via peers:

  routing of messages P2P rather than by dedicated brokers – reliability, confidentiality, altruism
- Trust has a context – skills may not transfer

  e.g. drivers of cars, trains, planes ...

# Detached, ad-hoc, anonymous groups

- e.g. connected by wireless
- can't assume trusted third-parties (CAs) accessible
- can't assume knowledge of names and roles, identity likely to be by key/pseudonym
- new identities can be generated (by detected villains)

- parties need to decide whether to interact
- each has a trust policy and a trust engine
- each computes whether to proceed – policy is based on:

  - accumulated trust information

    (from recommendations and evidence from monitoring)

  - risk (resource-cost) and likelihood of possible outcomes

# Promising Approaches for Large-Scale Systems

- Roles for scalability
- Parametrised roles for expressiveness, scalability, simplicity
- RBAC for services, service-managed objects, including the communication service
- Policy specification and change management
- Policy-driven system management

- Asynchronous, loosely-coupled communication
  publish/subscribe for scalability
  event-driven paradigm for ubiquitous computing
- Database integration – how best to achieve it?

And don't forget:

- Mobile users
- Sensor network integration

# Opera Group – research themes

## (**objects policy events roles access control**)

- Access Control (OASIS RBAC)
  Open Architecture for Securely Interworking Services
- Policy expression and management
- Event-driven systems (CEA, Hermes)
  EDSAC21: event-driven, secure application control for the 21st Century
- Trust and risk in global computing (EU SECURE)
  **se**cure **c**ollaboration among **u**biquitous **r**oaming **e**ntities
- TIME: Traffic Information Monitoring Environment
  TIME-EACM event architecture and context management
- CareGrid: dynamic trust domains for healthcare applications
- SmartFlow: Extendable event-based middleware
- PAL: personal and social communications services for health and lifestyle monitoring.

see: *www.cl.cam.ac.uk/research/srg/opera*

for people, projects, publications for download

# Mini Erlang review (from last term)

Erlang is a functional, declarative language with the following properties:

1. single assignment – a value can be assigned to a variable only once, after which the variable is immutable

2. Erlang processes are lightweight (language-level, not OS) and share no common resources. New processes can be forked (***spawn***ed), and execute in parallel with the creator:
   ***Pid = spawn ( Module, FunctionName, ArgumentList )***
   returns immediately – doesn't wait for function to be evaluated
   process terminates when function evaluation completes
   ***Pid*** returned is known only to calling process (basis of security)
   ***Pid*** is a first class value that can be put into data structures and passed in messages

3. asynchronous message passing is the only supported communication between processes.
   ***Pid ! Message***
   ***!*** means send
   ***Pid*** is the identifier of the destination process
   ***Message*** can be any valid Erlang term

Erlang came from Ericsson and was developed for telecommunications applications.
It is becoming increasingly popular and more widely used (e.g., ejabberd, RabbitMQ).

# Erlang – 2: receiving messages

The syntax for receiving messages is:

```
receive
    Message1 ( when Guard1) ->
        Actions1 ;
     Message2 ( when Guard2 ) ->
        Actions2 ;
     .....
end
```

Each process has a mailbox – messages are stored in it in arrival order.

***Message1*** and ***Message2*** above are patterns that are matched against messages in the process mailbox. A process executing ***receive*** is blocked until a message is matched.

When a matching ***MessageN*** is found and the corresponding ***GuardN*** succeeds, the message is removed from the mailbox, the corresponding ***ActionsN*** are evaluated and ***receive*** returns the value of the last expression evaluated in ***ActionsN***.

Programmers are responsible for making sure that the system does not fill up with unmatched messages.

Messages can be received from a specific process if the sender includes its ***Pid*** in the pattern to be matched: ***Pid ! {self( ), abc}***

***receive {Pid, Msg}***

# Erlang – 3: example fragment

Client:

***PidBufferManager ! { self ( ), put, <data> }***

***PidBufferManager ! { self ( ), get, <pointer for returned data> }***

Buffer Manager:

***receive {PidClient, put, <data> }*** (buffer not full)

insert item into buffer and return

***{PidClient, get, <pointer for returned data> }*** (buffer not empty)

remove item from buffer and return it to client

# Erlang - 4: further information and examples

Part 1 of Concurrent Programming in Erlang is available for download from *http://erlang.org/download/erlang-book-part1.pdf*

The first part develops the language and includes many small programs, including distributed programs, e.g. page 89 (page 100 in pdf) has the server and client code, with discussion, for an ATM machine.

The second part contains worked examples of applications, not available free.

A free version of Erlang is easy to find.

Erlang, most importantly, is distributed. This course won't teach you to be an Erlang master, but regularly it will be used in examples.