

Distributed Systems

Easter term 2009

David Evans (de239@cl.cam.ac.uk)

Introduction

**system, legal, social context
technology-driven evolution
fundamental characteristics
software structure
models, architecture, engineering
domain-structured, large-scale systems**

Time

**event ordering
physical clock synchronisation
process groups
ordering message delivery**

Distributed algorithms and protocols

**strong and weak consistency
replicas of an object, transactions on distributed objects
concurrency control
atomic commitment
election algorithms
distributed mutual exclusion**

Middleware

RPC, OOM, MOM, event-based middleware

Naming

Access Control

**capabilities, ACLs, RBAC and access control policy
OASIS RBAC case study**

Case studies: Event-driven systems, access control

Storage services

distribution issues, outline of Cambridge File Server

Distributed Systems - Introduction

- 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

Costly Failures in Large-Scale Systems

- **UK Stock Exchange** - share trading system
 - abandoned 1993, cost £400M
- **CA automated childcare support**
 - pended 1997, cost \$300M
- **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

What makes things special?

- normal software failure
- errant behaviour not accommodated by other pieces of the system
- cascade of the failure is spectacular

Why high public expectation?

Web experience

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

Things mostly work.

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

Public-Sector Systems Especially

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 (many roles)
- web portal interface, but often not web-service model
- long timescale, high cost
- ubiquitous and mobile computing – still under research
- policy of competition and independent procurement vs. policy requiring interoperation
- 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 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 can be reluctant to cooperate in traffic monitoring, even though monitoring could show that delay is often not their fault.

Data Protection Legislation

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: Jan 2005

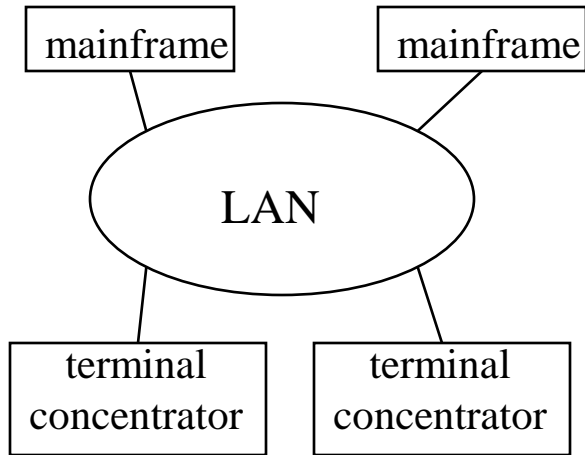
Rapidly Developing and New Technology

- can't ever design a “*second system*”, it's always possible to do more next time
- rapid obsolescence - *incremental growth* not sustainable long-term (unlike e.g. telephone system)
- but *big-bang* deployment is a bad idea
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*

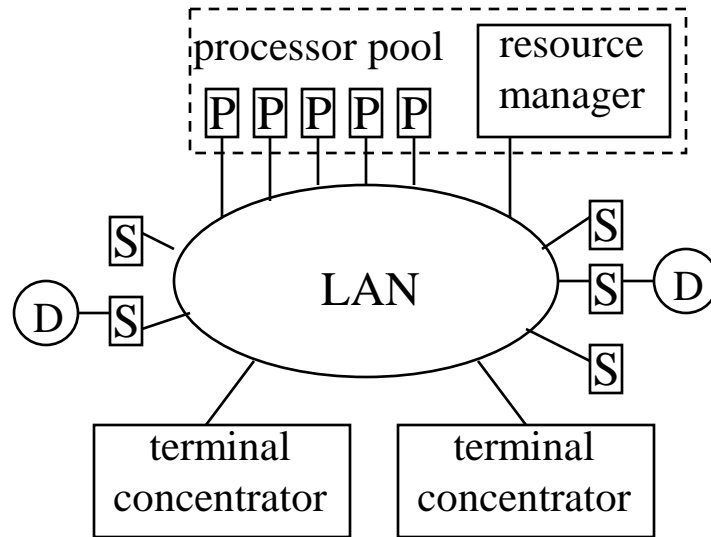
DS history: technology-driven evolution

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

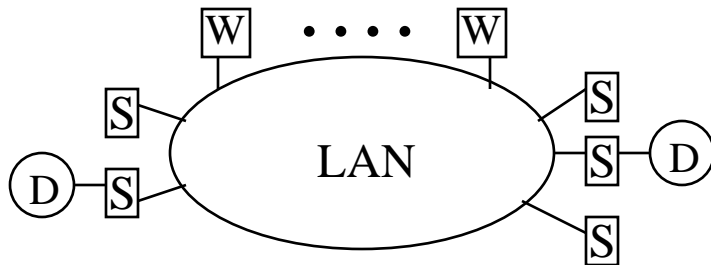
1. LAN as terminal switch to multiaccess systems



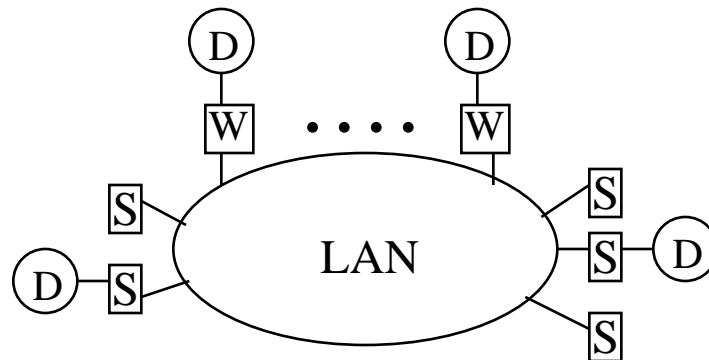
2 terminals + processor pool + servers



3 diskless workstations + servers



4 workstations + servers



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

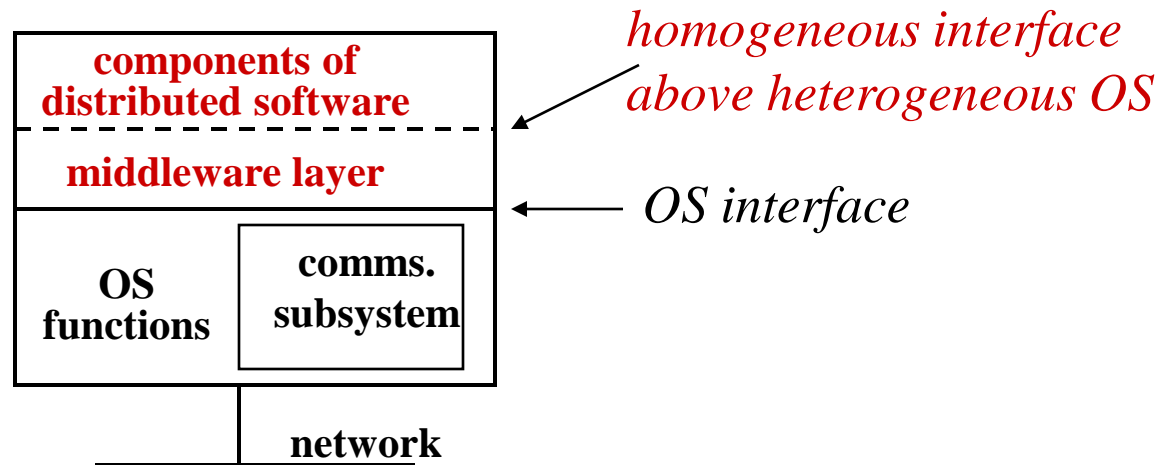
Implications:

- 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

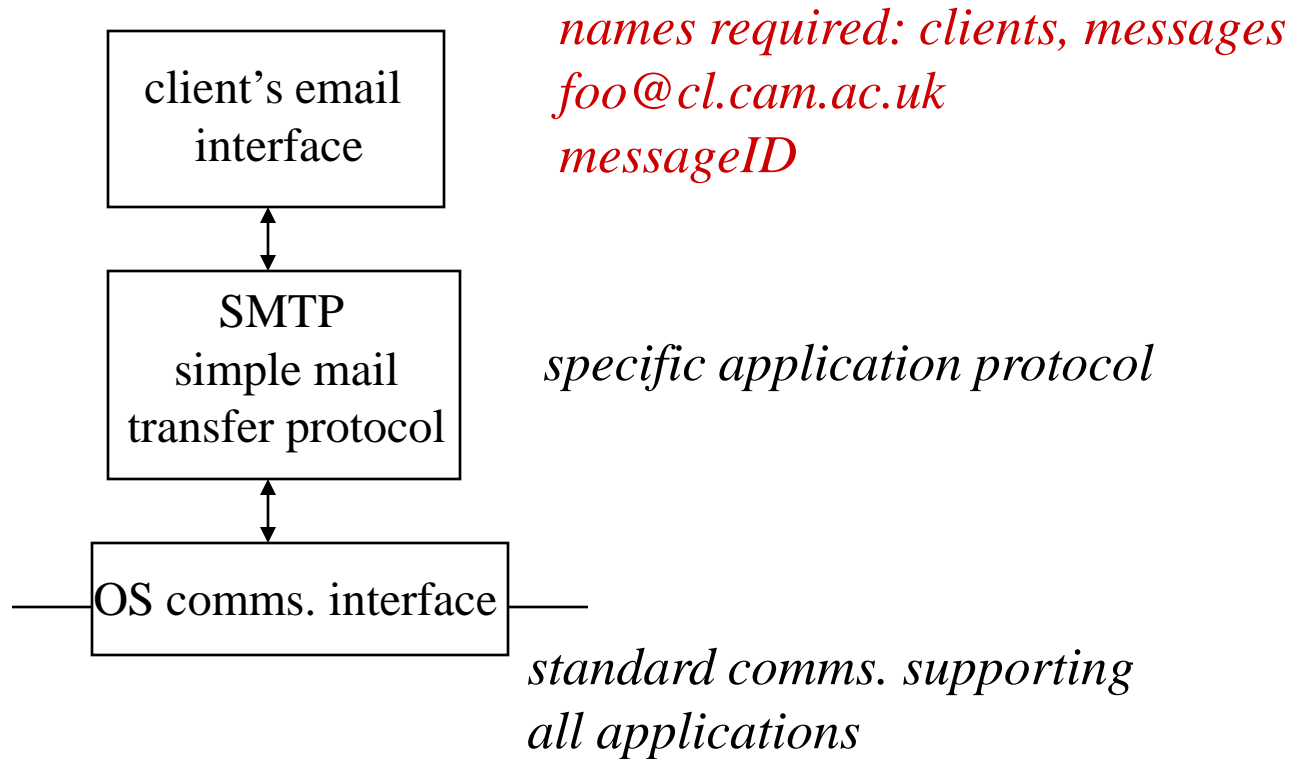
single node - software structure

Support for distributed software may be:

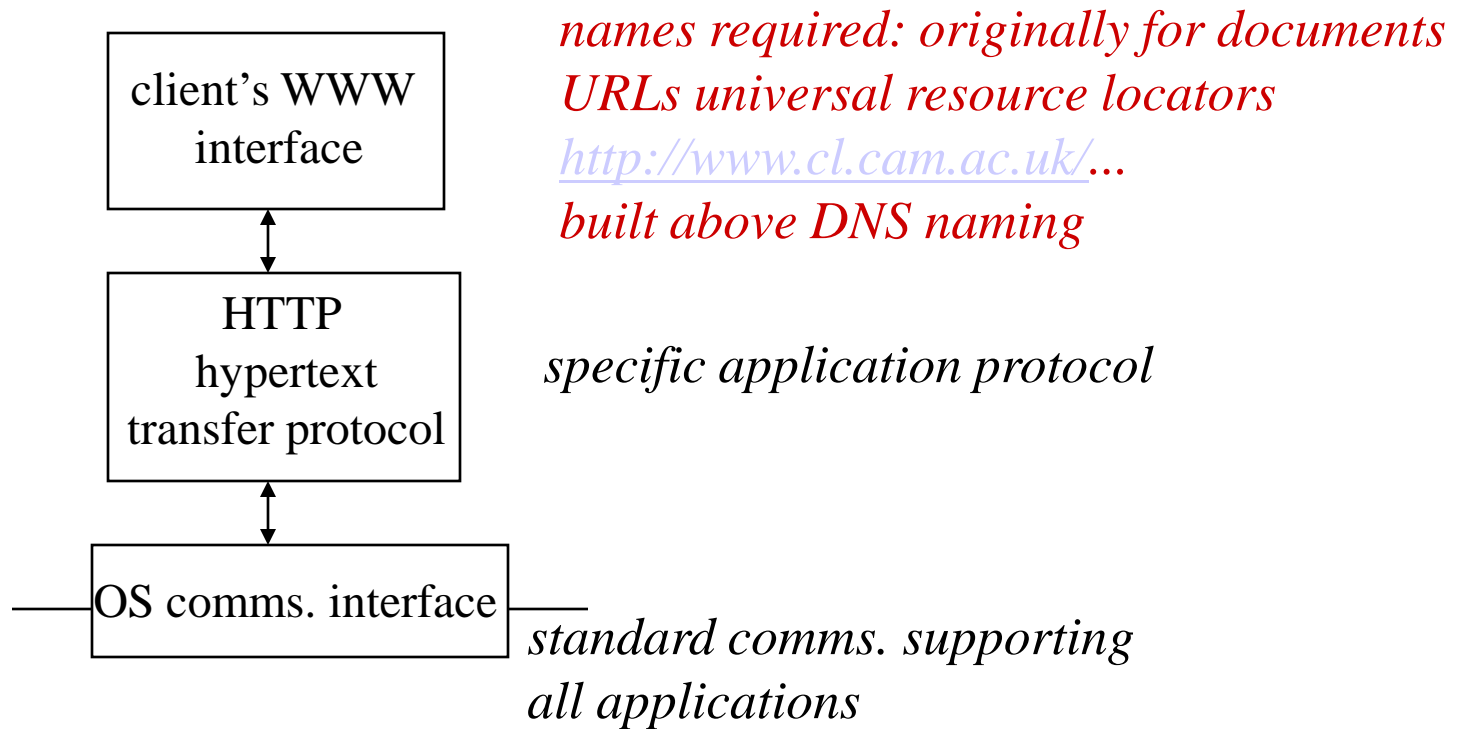
1. directly by OS in a homogeneous cluster (distributed OS design) – not the focus of this course
2. by a software layer (middleware) above potentially heterogeneous OS



Distributed application structure – email, news, ftp

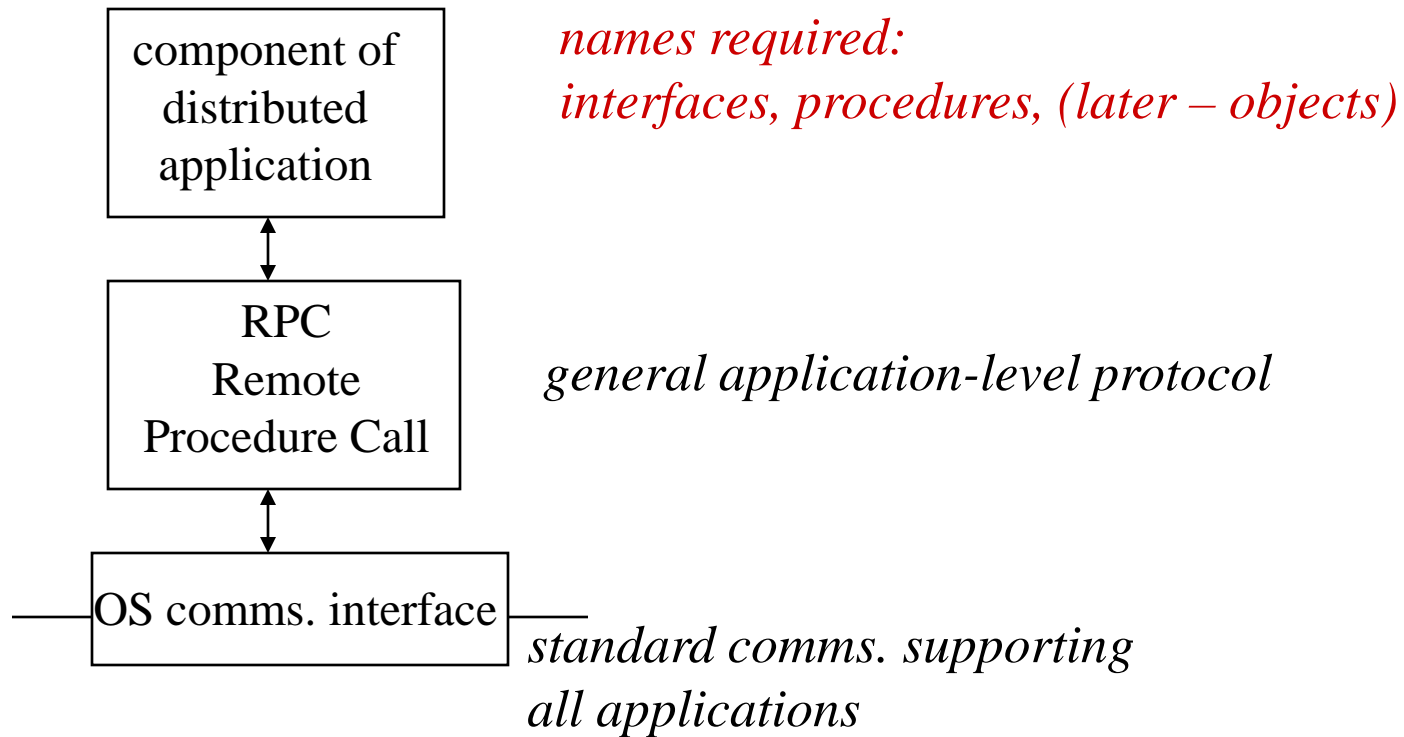


Distributed application structure – web documents



(This scheme has been used for **general** distributed applications;
W3C standards for **Web Services** – later)

Distributed application structure – general support example



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) or consortia (e.g. OMG, W3C). Requests for proposals (RFPs) are issued, draft specifications published with RFCs (requests for comments) or similar.
- **Closed, proprietary:** can be changed by the owner (clients may need to buy a new release). Consistency across versions is not guaranteed. Good for technical extortion.

Related issues:

- **single/multi language:** can components be written in different languages and interoperate?
- **open interoperability:** can your system span multiple middlewares (including different implementations of the same MW)?

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**
- **Authentication** of participants
- **Protection / 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

1. Federated administration domains
 - integration of databases
 - integration of sensor networks
 - small dynamic domains with members grounded in various static administration domains
2. Independent, external services - to be integrated
3. Detached, ad-hoc, anonymous groups;
anonymous principals, issues of trust and risk

1. Federated administration domains: Examples

- **national healthcare services:**
many hospitals, clinics, primary care practices.
- **national police services:**
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 **access control** *intra and inter-domain*, plus some external policies to satisfy government, legal and institutional requirements
- **high trust**, high accountability

small dynamic domains with members grounded in static administration domains

- e.g. assisted home-living (sheltered housing)
“patient” + various carers + technology
- carers have roles in primary care practices, hospitals, social services, out-sourced services
- care programme is specified by contract
 - rights of patient to defined care
 - obligations of carers and patients
 - privacy of patient data
 - need to audit people and technology

dynamic domains - characteristics

- **names:** principals (users, roles): from home domains; services, data-types, messages, sensors: set up for small dynamic domains
- **authentication:** users administered within home domain. Need for credential check back to home domain (as in federated domains).
- **communication:** needed across domains
- **policies:** indicate contractual obligations and privileges (access control)
- **audit** of people, technology
- **trust**, based on observation of audit, (and reputation?)

2. 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* (e.g. Xenoservers), or *storage*
- **e-science** may support “virtual organisations” –
collaborating groups across several domains

Independent, external services - characteristics

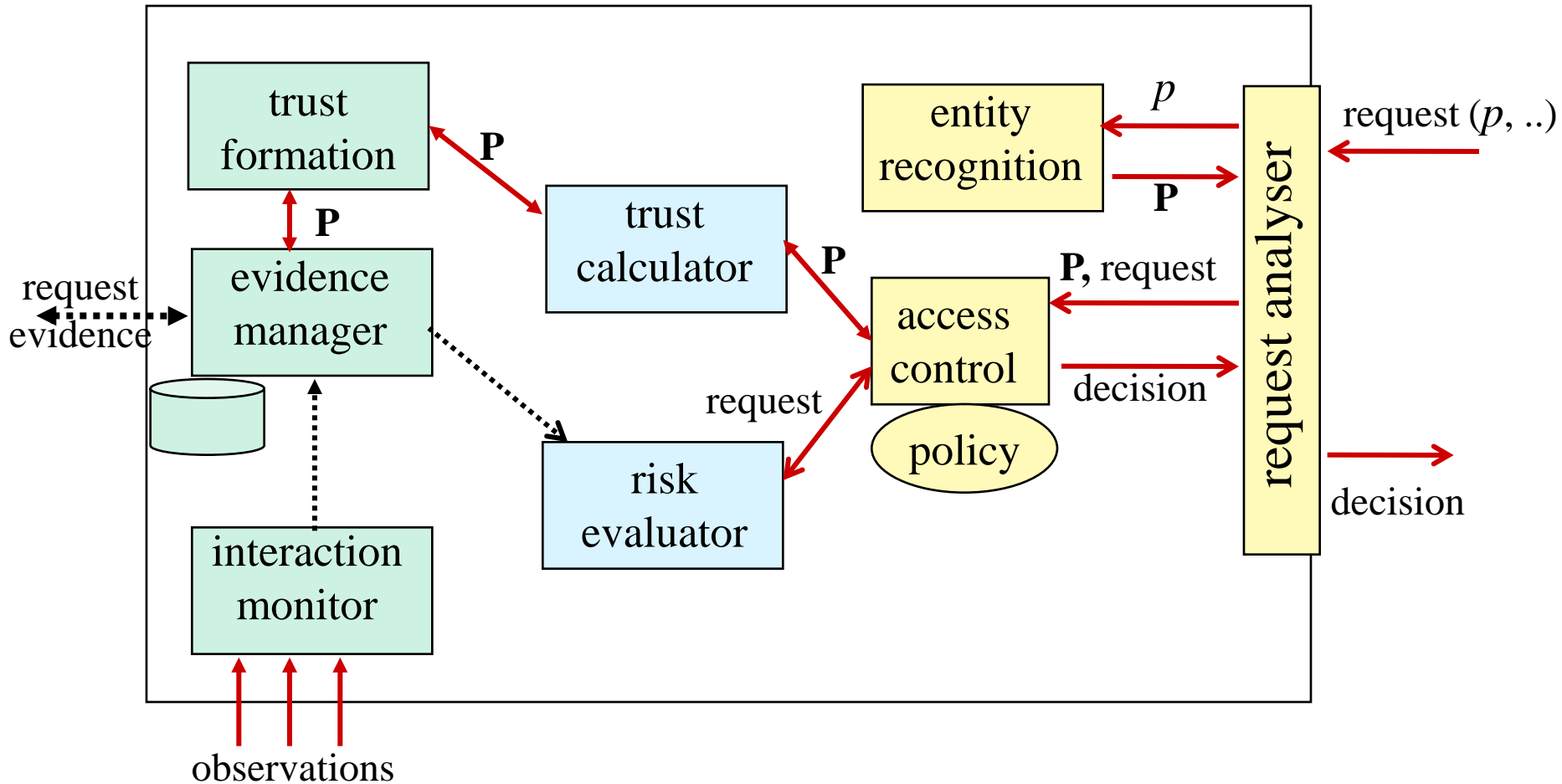
- **naming and authentication**
may be client-domain-related,
and/or of individuals via certification authorities (CAs)
- **access control policies**
related to client roles in domains and/or individual principals
may provide support for “*virtual organisations*”
- need for: **accounting, charging, audit**
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. CAs can authenticate/identify

3. 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

Simplified SECURE Trust Model



Examples of detached ad hoc groups and the need for trust

- Commuters regularly play cards on the train
- E-purse purchases
- Recommendations: of people and e.g. restaurants in a tourist scenario
- Wireless routing via peers
- Routing of messages P2P rather than by dedicated brokers – reliability, confidentiality, altruism
- Trust has a context

Promising Approaches for Large-Scale Systems

- **Roles** for scalability
- **Parametrised roles** for expressiveness
- **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 **A**rchitecture for **S**ecurely **I**nterworking **S**ervices
- **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**)
secure **collaboration** among **ubiquitous** **roaming** **entities**
- **TIME**: Traffic Information Monitoring Environment
TIME-EACM event architecture and context management
- **CareGrid**: dynamic trust domains for healthcare applications
- **SmartFlow**: Extendable event-based middleware

see: www.cl.cam.ac.uk/Research/SRG/opera

for people, projects, publications for download

Time in Distributed Systems

- * There is no common universal time (Einstein)
but the speed of light is constant for all observers irrespective of their velocity

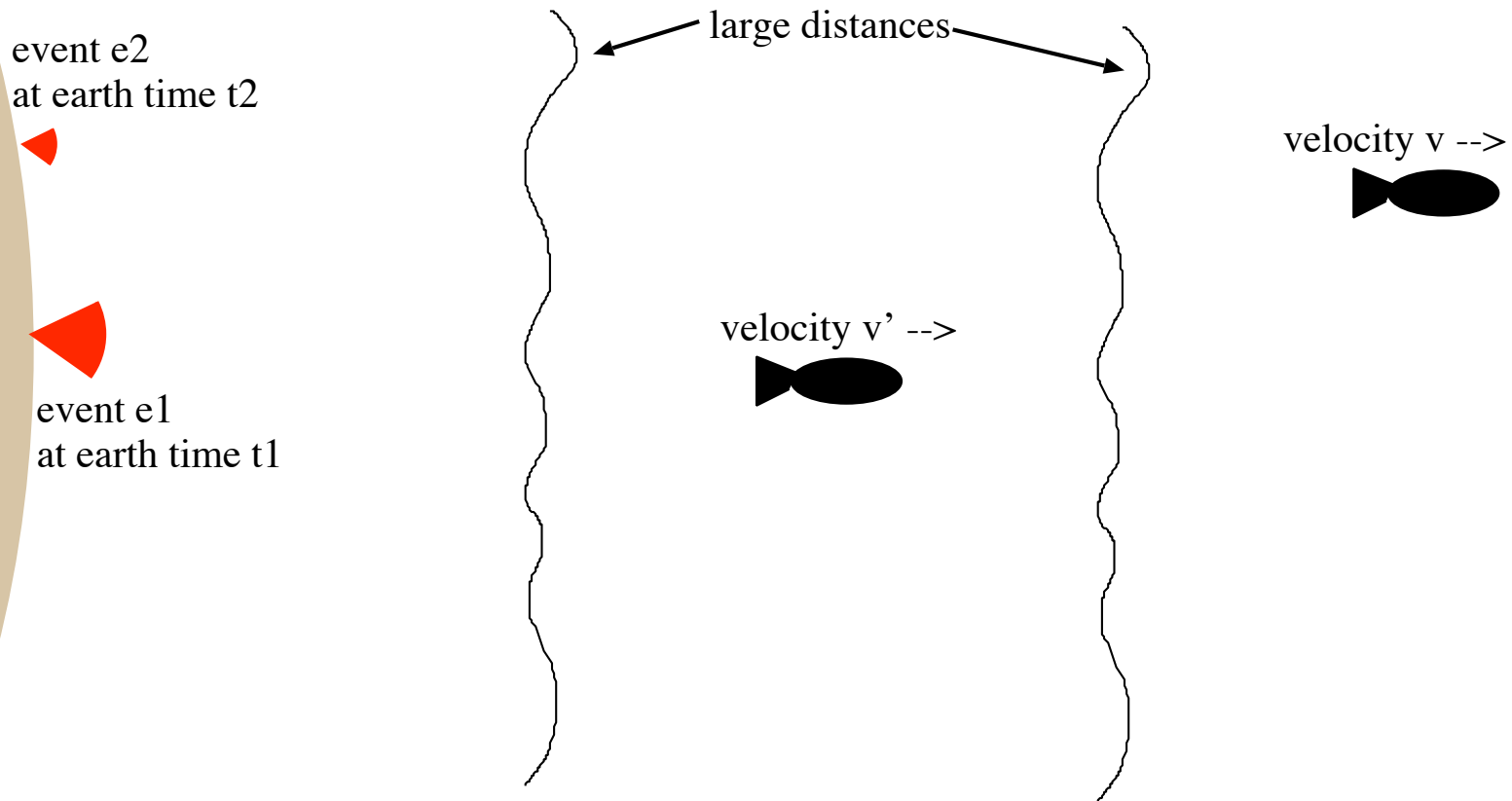
- * Assume our DS is earth-based

- * Even then, time is complex
 - Sunrise/sunset?
 - Radioactive decay?
 - Stars' positions?
 - Seasons?
 - Tides?
 - Slowing of the Earth's rotation?

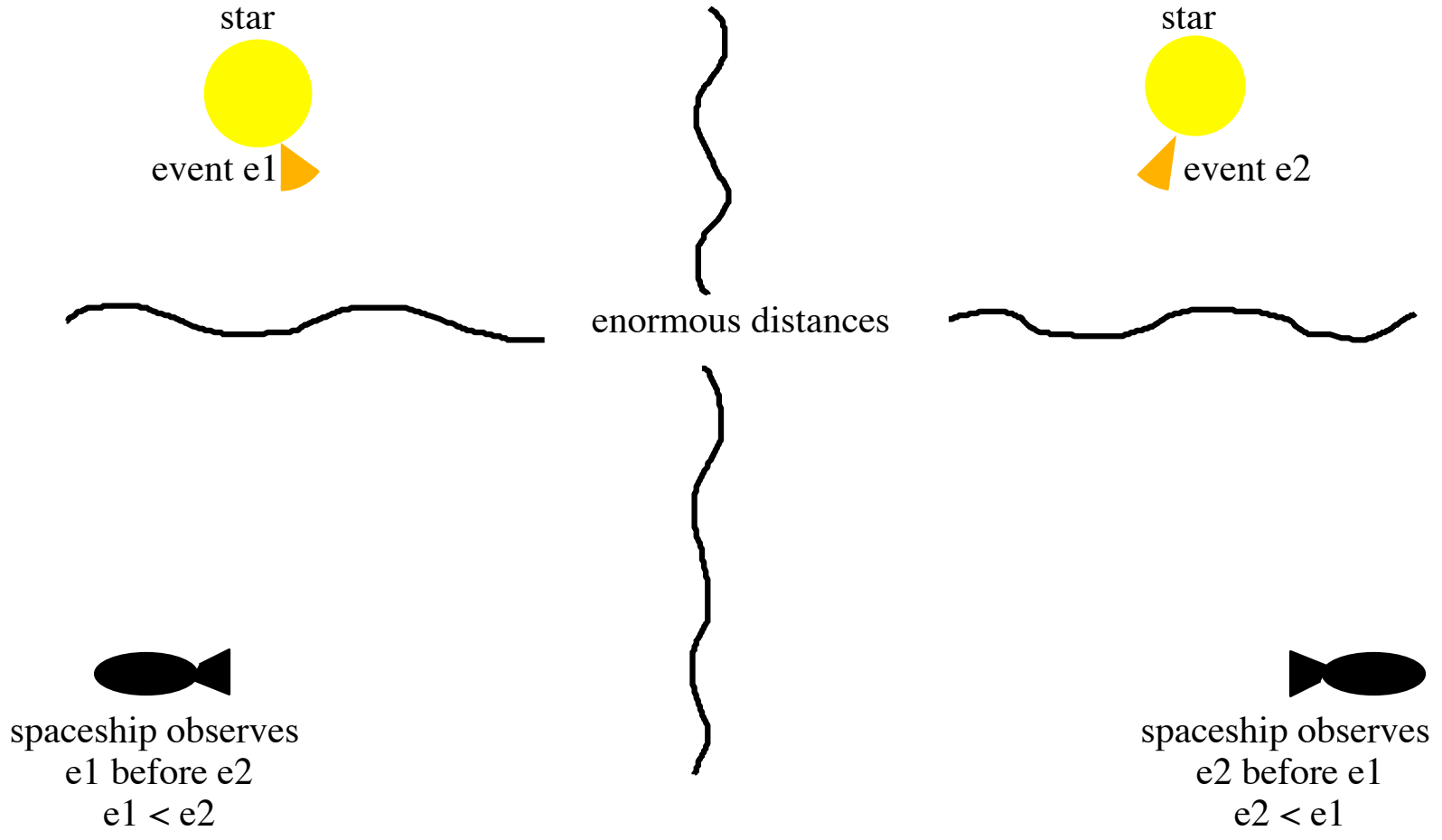
- * **UTC (Coordinated Universal Time)** is in step with TAI but based on UT1

- * UTC services are offered by radio stations and satellites -- receivers are available commercially
Accuracy varies with weather conditions

- * UTC signals take time to propagate -- UTC can't be known exactly
For a given receiver we can estimate a time interval during which an event has happened
w.r.t. UTC, see also T-11 "interval timestamps"



The spaceships observe different times for e1 and e2 and different values for $e2 - e1$ but observe e1 and e2 in the same order



Timers in computers

- * based on frequency of oscillation of a quartz crystal
- * Each computer has a timer which interrupts periodically.
Clock skew: in practice, the number of interrupts per hour varies slightly in the fabricated devices, also with temperature, and clocks may **drift**
- * timers can be set from transmitted UTC
- * We have already seen that we cannot know the time at which an event occurs accurately, but only specify an interval.
We now have to increase that interval to allow for clock drift as well as other sources of inaccuracy.
- * note that computer systems tag events with timestamps, usually a local clock reading strictly, intervals should be used, see T-11

How is time used in distributed systems?

Do we need accurate time?

What does "A happened before B" mean in a distributed system?

If two events have single-value timestamps which differ by less than some value we CAN'T SAY in which order the events occurred.

With interval timestamps, when intervals overlap, we CAN'T SAY in which order the events occurred.

Use of time in distributed systems: Examples

1. Any source of resource contention e.g. airline booking

Policy: if the reservation requests for two transactions may each be satisfied separately but there are not enough seats left for both, then the transaction with the earliest timestamp wins.

Note that there is no causality, the requests are independent. We don't need fine-grained accuracy, we just need a timestamp ordering convention so all agree who won.

On a tie (equal timestamps) use an agreed tie-breaker e.g. IP address/process ID

2. programming environments e.g. UNIX make (compile and link)

suppose a make involves many components which are edited on distributed computers

suppose a component is edited immediately after a make, but on a computer with a slow clock and the edited source is given a timestamp earlier than the make on the next make, this component is not recompiled

- this can be made unlikely to happen, if we ensure that clocks are initialised reasonably accurately e.g. not from the operator's watch

- this is an example of correctness depending on correct event ordering
did the edit take place before or after the last make?

3. Did a credit/debit transaction take place before or after midnight?
This affects the calculation of interest.

4. The value of shares at the time of buying/selling

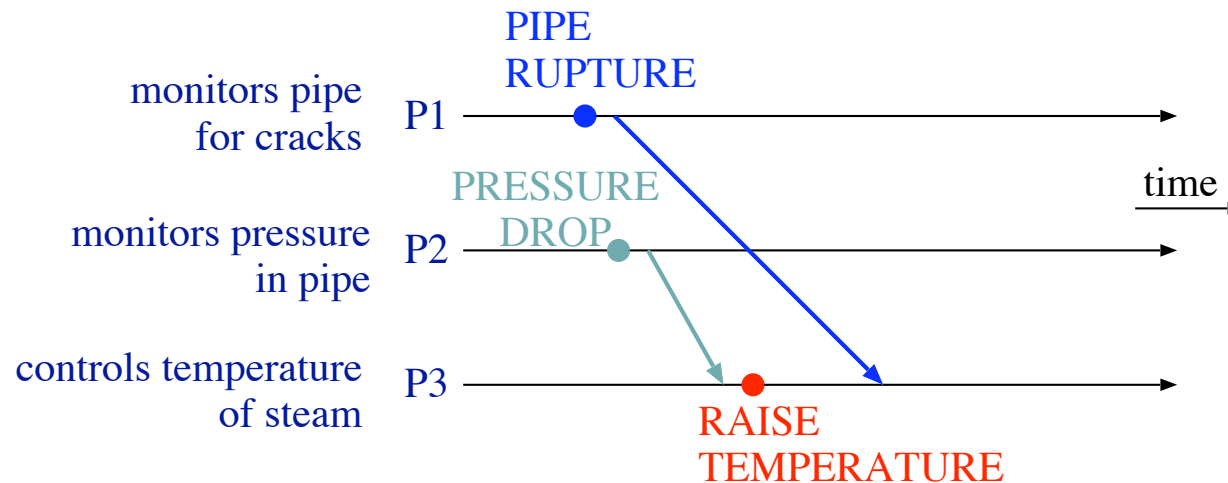
5. Insider dealing? Did X read Y before buying/selling?

Note that some of the above examples require only a means of agreement, so that all participants in the computer system make the same decision.

Others require accurate time or the order of events in the real world when causality is at issue.

But note external channels - and real/physical causality

e.g. monitoring and controlling a pipe along which steam is delivered under pressure



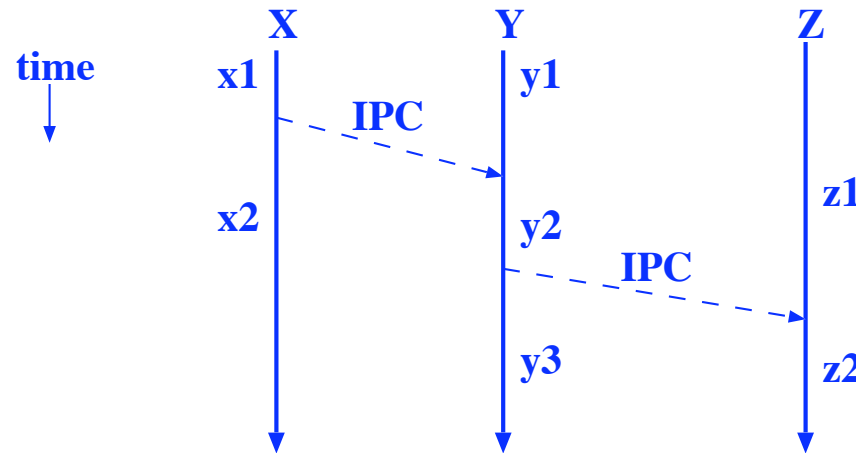
- * The pipe ruptures (which **causes** a drop in pressure)
- * P1 sends message to controller P3 to notify rupture
- * P2 sends message to P3 to notify pressure drop
- * P3 receives P2's message before P1's and increases temperature of steam
- * P3 then receives P1's message and infers (wrongly) that increasing the temperature has caused the pipe to rupture

Here, causality is outside the message transport service.

Controller's algorithm needs to take account of **physical timestamps** which must be accurate

Audit of system fault has to report on cause - maybe "**can't say**" depending on timestamp accuracy

Event ordering in distributed systems



define $<$ to mean "happened before"

suppose inter-process communication (IPC) takes place:

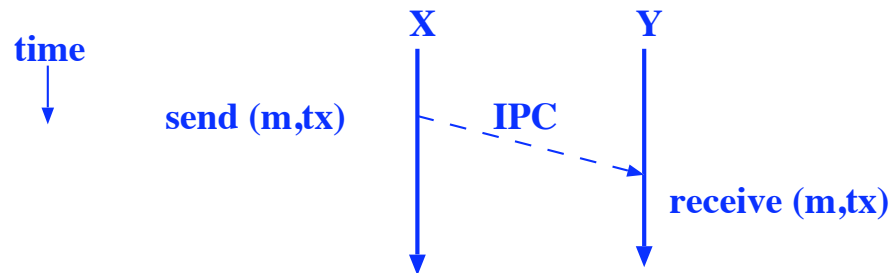
- * events within a single process are ordered
- * events in region x1 $<$ events in regions y2, y3
- * events in region x1 $<$ events in regions z2
- * events in region y1, y2 $<$ events in regions z2
- * for events in other regions we CAN'T SAY, unless we know the precise accuracy of all physical clock values e.g. events in x1 and y1, y1 and z1 etc.

IPC defines a PARTIAL ORDERING on the events in the DS

note that **this ordering is true** whatever the local clocks of X, Y and Z indicate

we must ensure that the values of the local clocks respect this event ordering

suppose a message m is timestamped tx by X on sending



note that X 's send CAUSED Y 's receive

suppose Y 's local clock has reading ty on receive(m,tx)

if $ty > tx$ OK

if $ty \leq tx$ reset ty to $tx + \text{one increment}$

This imposes logical time on the system

(sometimes called Lamport time due to Lamport's classic paper on this topic)

BUT - system time adjusted in this way will drift ahead of UTC

we could use counters rather than timestamps if all we need is event ordering

How can we generate timestamps that are:

- reasonably close to UTC
- preserve causal ordering

Protocols for synchronising physical clocks

Cristian's algorithm 1989

- * assume one machine has a UTC receiver (time server)
- * each machine polls the time server periodically
(period depends on maximum clock drift allowed and accuracy required)
- * server sends back its value of the time
- * client receives this value and may:
 - use it as it is
 - add the known minimum network delay
 - add half the time between this send and receive
- * now consider resetting the receiver's local clock from this value
 - if value \geq local-time OK use it to set the clock
 - or adjust the interrupt rate for a while to speed up the clock (e.g. 10ms \rightarrow 9ms)

NOTE that time can't be put back or event ordering within the local system would become incorrect

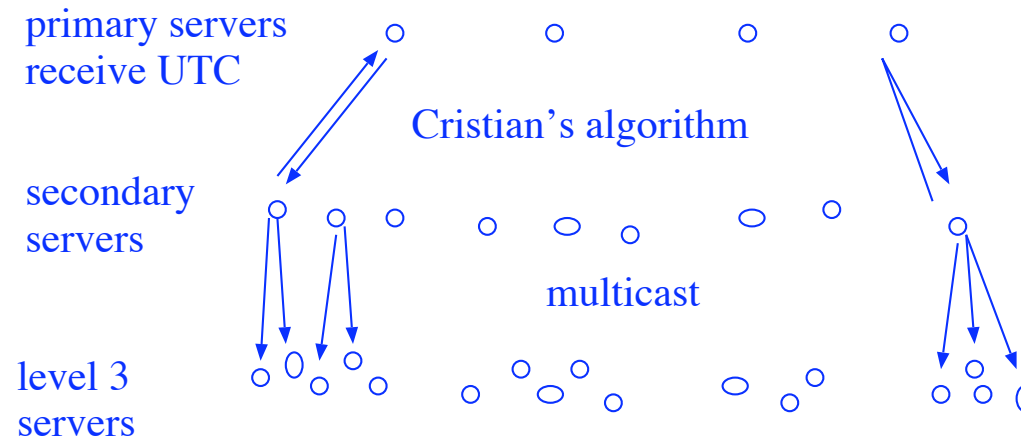
 - if value $<$ local time
 - adjust the interrupt rate to slow down the clock (e.g. 10ms \rightarrow 11ms)
- * a number of time servers can be used (replication increases reliability)

Berkeley UNIX (Gusella & Zatti, 1989)

- * if no machines have receivers . . .
- * a nominated "time-server" asks all machines for their times
- * computes the average value
- * broadcasts to all machines
- * operator may set the time manually from time to time

NTP (Network Time Protocol) (Mills, 1991)

- * for the Internet as a hierarchy of computers



- * uses UDP
- * allow for estimated network delay and adjust clocks as described above
- * accurate to a few tens of ms

Interval timestamps

- * for any computer we can estimate how long UTC takes to reach it, taking into account:
 - atmospheric propagation
 - network(s) transmission
 - software overhead (e.g. local OS)

- * instead of a single-valued timestamp, use an interval in which UTC is estimated to lie

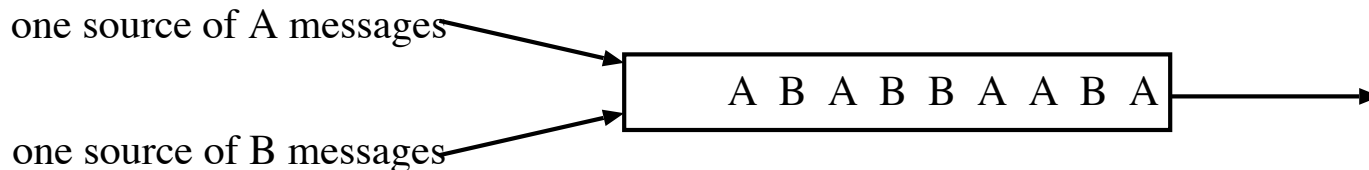
- * use these interval values for ordering events,
for example when arriving messages need to be ordered

- * sometimes we "can't say" - this is the nature of distributed systems.
a weak ordering might be created, based on e.g. the upper interval bound.

Composing events (realised as messages)

- * applications are often interested in **patterns of events**
 - fraud detection
 - fault detection
 - to control the volume of events propagated

- * a composition service receives **streams of events** from distributed sources and creates a stream of composite events. Example with two event types, A, B



- * possible composition operators
 - AND, OR, SEQ (before/after)? UNTIL?, AFTER?, NOT?
 - note fundamental uncertainty of time in distributed systems

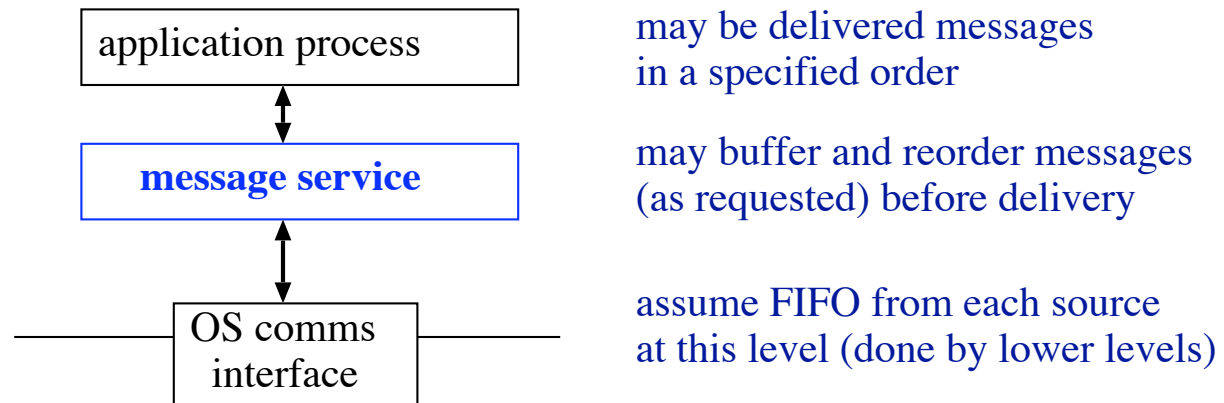
- * ref I-12 for fundamental and engineering issues
 - are all sources of A and B and connections to them operational?
 - have all the As and Bs arrived? (use a heartbeat protocol?)

- * consumption policy for As and Bs? *historical, most recent,*

- * buffer size and garbage collection

ASSUMPTIONS

- messages are multicast to named process groups
- reliable channels: a given message is delivered reliably to all members of the group (no lost messages)
- FIFO from a given source to a given destination
- processes don't crash (failure and restart not considered)
- processes behave as specified and send the same values to all processes
(we are not considering Byzantine behaviour)



* **no order**

messages are delivered to the application process in the order received by the message service

* **total order**

every process receives all messages in the same order

* **causal order**

messages that are potentially causally related are delivered in causal order at all processes


Process groups

- * **membership** create (name, <list of group members>)
 kill (name)
 join (name, process)
 leave (name, process)

- * **internal structure?**

- NO (peer) - failure tolerant, complex protocols
- YES (a single coordinator (and point of failure))
 - simpler protocols
 - e.g. if a join request must be to the coordinator, concurrent joins are avoided

*will need
later for
algorithms*



- * **closed or open?**

- OPEN - a non-member can send to a group, e.g. to a set of servers
- CLOSED - only members may send to the group name e.g. parallel, fault-tolerant algorithms

- * **failures**

a failed process leaves the group without executing leave

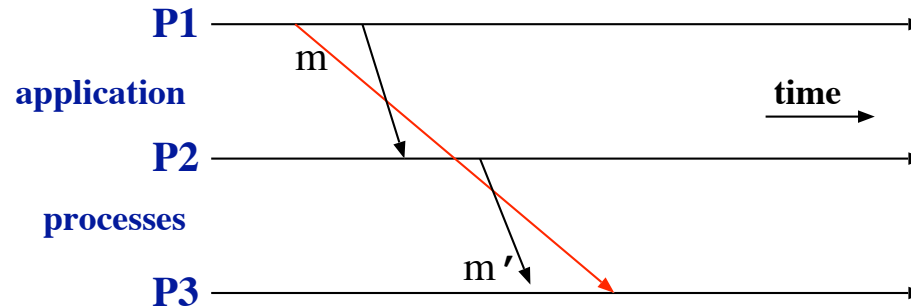
- * **robustness**

leave, join and failures happen during normal operation - algorithms must be robust

Example: ISIS system at Cornell (Birman) -> Horus

Message delivery - causal order

first, define in terms of one-to-one messages; later, multicast



if causal delivery order is required, m should be delivered before m' at P3

we are concerned only with POTENTIAL causality (not the subject of messages)

definition of causal delivery order (where $<$ means "happened before")

$$\text{send}_i(m) < \text{send}_j(m')$$

$$\Rightarrow$$

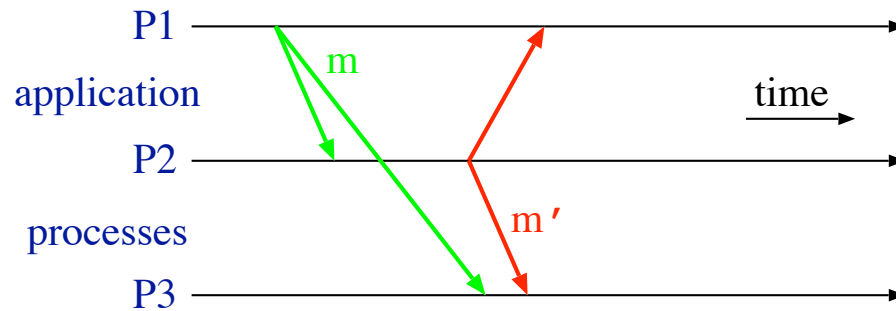
$$\text{deliver}_k(m) < \text{deliver}_k(m')$$

in the above example: P1 sends m before P2 sends m'

$$\Rightarrow$$

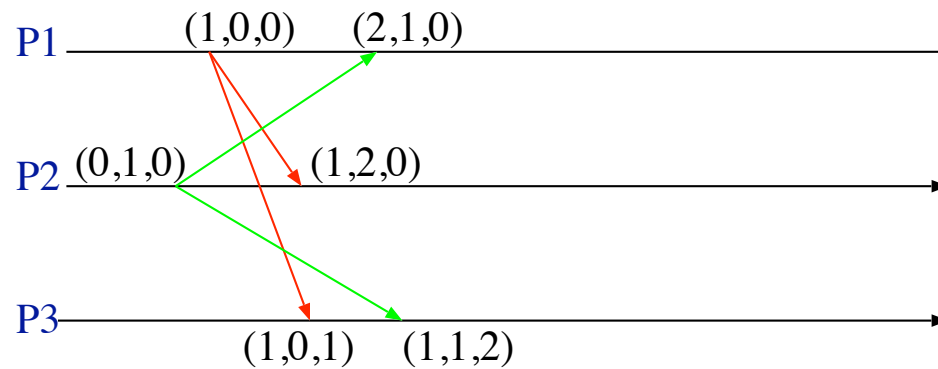
m should be delivered before m' at P3

now assume P1, P2 and P3 are in a process group and all messages are multicast to the group (all processes receive all messages)



In this case, the message delivery system can implement causal delivery order by using vector clocks.
For total order, see T23 - T25

a **VECTOR CLOCK** is maintained by the message service at each node for each process:

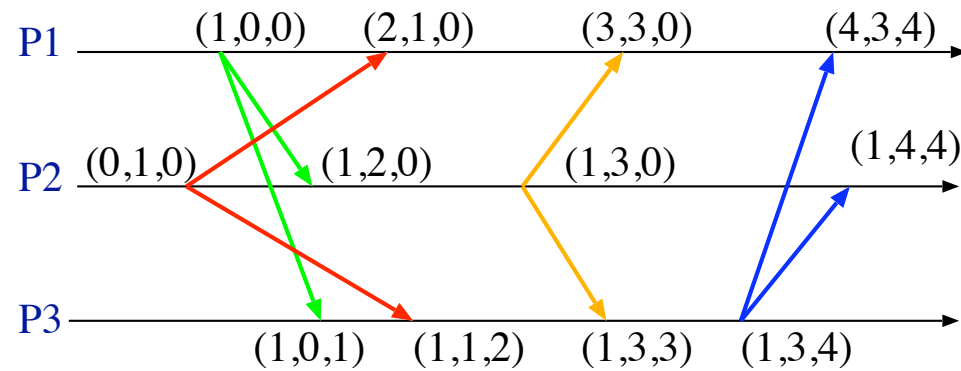


vector notation:

- fixed number of processes, N
- each process's message service keeps a vector of dimension N
- for each process, each entry records the most up-to-date value of the state counter, known to that process, for the process at that position

(set notation would be better for dynamic reconfiguration of groups

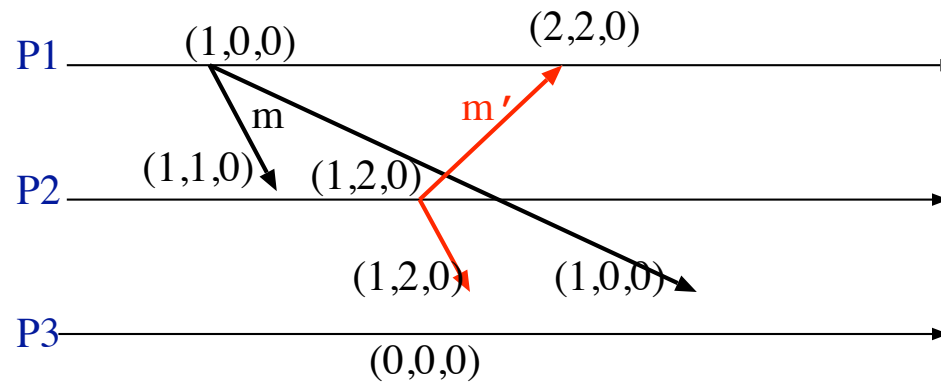
- the literature uses vectors)



message service operation:

- * before **send** increment local process's state-value in local vector
- * on **send**, timestamp message with sending process's local vector
- * on **deliver**, increment receiving process's state-value in its local vector and update the other fields of the vector by comparing its values with the incoming timestamp and recording the higher value in each field
thus updating this process's knowledge of system state

Implementing causal delivery order



at P3's message service:

P3's vector is (0,0,0)

m' comes in from P2 with timestamp (1,2,0)

so P2 received a communication from P1 before sending this message

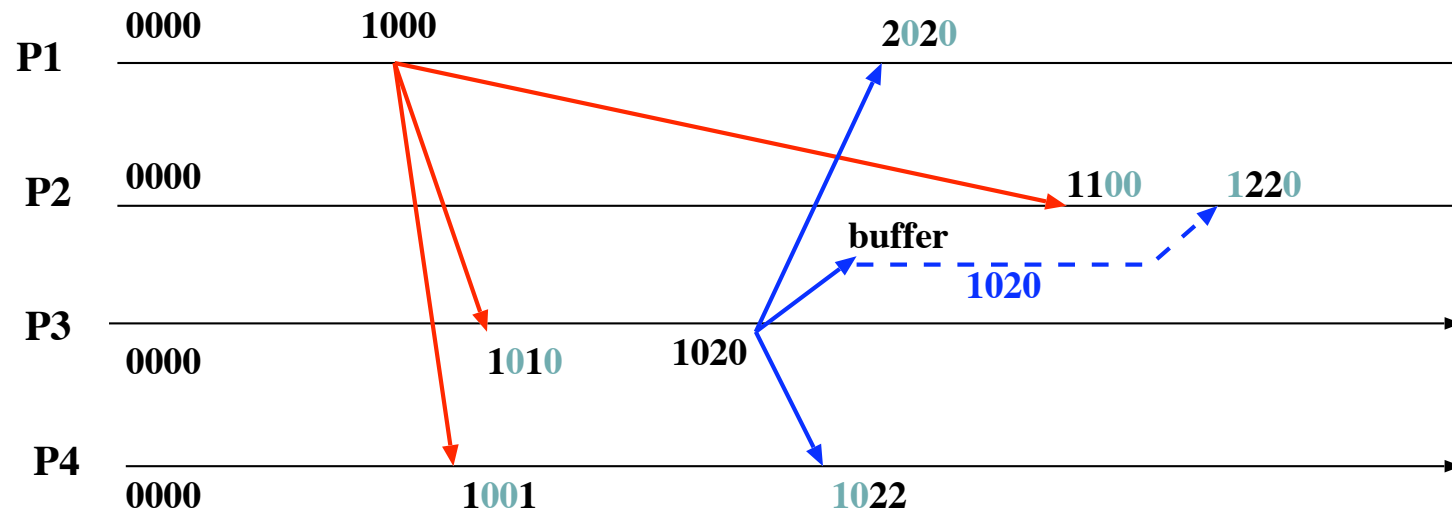
P3's message service buffers m' until a message comes in from P1

the algorithm in more detail (at P3):

receiver vector	sender	sender vector	decision	new receiver vector
(0,0,0)	P2	(1,2,0)	hold in buffer	(0,0,0)
(0,0,0)	P1	(1,0,0)	deliver	(1,0,1)
from buffer: (1,0,1)	P2	(1,2,0)	deliver	(1,2,2)

*in each case - do the sender and receiver agree on the state of all other processes?
if the sender has a higher value, buffer the message.*

CAUSAL ORDER using vector clocks - example



algorithm: check for causal delivery order,

i.e. receiver is not missing any message sender already has
(system state at receiver \geq system state at sender

for processes other than sender and receiver

recall we are assuming FIFO delivery from a given source)

if not causal order, pend message - hold in buffer

if causal order, deliver message to application and update local vector

i.e. increment receiver's count and set all other counts

to greater of sender's and receiver's values,

thus updating receiver's record of system state

P3 (1020) to P1 (1000)

both have state of P2 as 0 and P4 as 0 -> deliver, P1->(2020)

P3 (1020) to P4 (1001)

both have state of P1 as 1 and P2 as 0 -> deliver, P4->(1022)

P3 (1020) to P2 (0000)

both have state of P4 as 0 - OK

P3 has state of P1 as 1, P2 has state of P1 as 0 -> buffer

P1 (1000) to P2 (0000)

both have state of P3 as 0 and P4 as 0 -> deliver, P2->(1100)

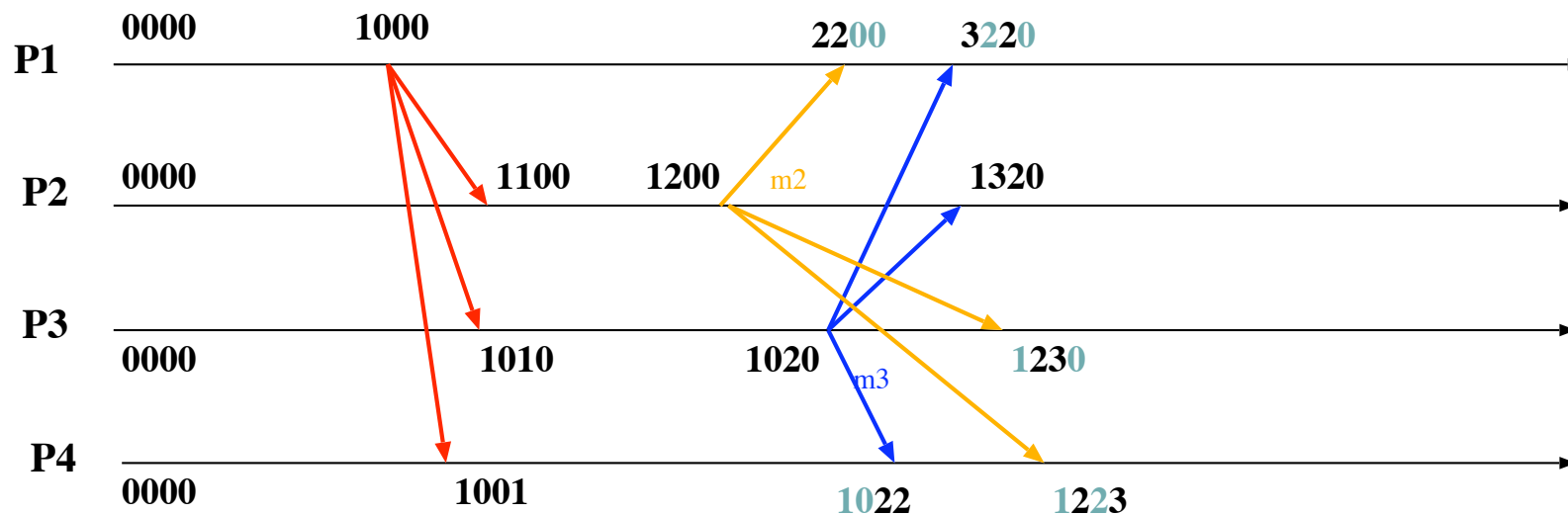
buffered message:

P3 (1020) to P2 (1100)

both have state of P1 as 1 and P4 as 0 -> deliver, P2-> (1220)

note **TOTAL ORDER** is not enforced by this algorithm

T-23



m2 and m3 are not causally related

P1 receives m1, m2, m3

P2 receives m1, m2, m3

P3 receives m1, m3, m2

P4 receives m1, m3, m2

If application requires **TOTAL** order, this can be enforced using vector clocks with extension to include ACKs and delivery to self (see below).

But the vectors can be a large overhead on message transmission.

Totally ordered multicast, can be achieved more simply using a single logical clock value as the timestamp

assumptions again, ref T13:

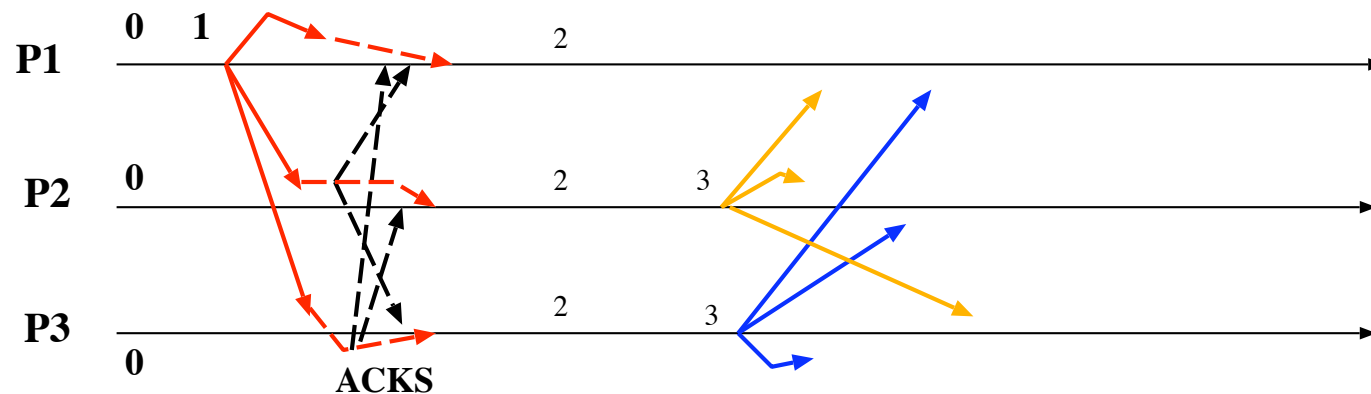
- multicast
- FIFO from each source to each destination
- reliable channels
- processes don't crash
- processes not Byzantine

algorithm:

- sender multicasts to all **including itself**
- all acknowledge receipt as a multicast message
- message is delivered in timestamp order when all ACKs have been received.

If the delivery system must support both, so that applications can choose, vector clocks can achieve both causal and total ordering.

TOTALLY ORDERED MULTICAST - outline of approach



P1 increments its clock to 1 and multicasts a message with timestamp 1
 All delivery systems collect message, send ACK and collect all ACKs
 - no contention - deliver message and increment local clocks to 2

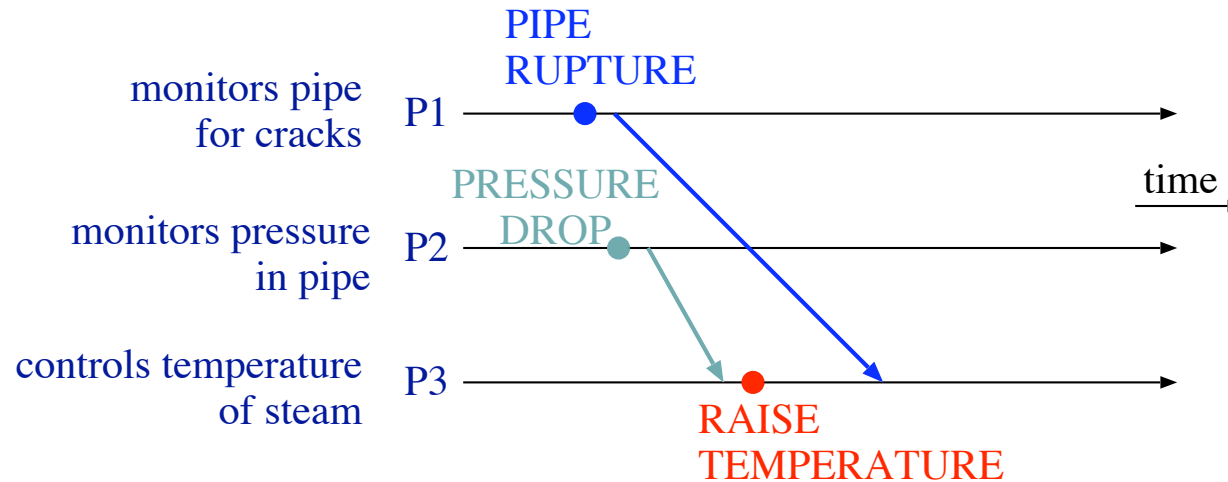
P2 and P3 both multicast messages with timestamp 3

all delivery systems collect messages, send ACKs and collect ACKs
 and use a **tie-breaker** to deliver P2's message before P3's

In practice, timeouts (long delay) due to congestion and/or failure of components and/or communication would have to be considered.

But note external channels - and real/physical causality

e.g. monitoring and controlling a pipe along which steam is delivered under pressure



- * The pipe ruptures (which **causes** a drop in pressure)
- * P1 sends message to controller P3 to notify rupture
- * P2 sends message to P3 to notify pressure drop
- * P3 receives P2's message before P1's and increases temperature of steam
- * P3 then receives P1's message and infers (wrongly) that increasing the temperature has caused the pipe to rupture

Here, causality is outside the message transport service.

Monitors may not be able to participate in a process group and use multicast

Controller's algorithm needs to take account of **physical timestamps** which must be accurate

Audit of system fault has to report on cause - maybe "can't say" depending on timestamp accuracy

Distributed algorithms and protocols

Consistency

Recall the properties of distributed systems:

1. concurrent execution of components
2. independent failure modes
3. transmission delay
4. no global time

DS may be large in scale and widely distributed

Replicated objects

objects may be replicated e.g. naming data (name servers), web pages (mirror sites)

- for reliability
- to avoid a single bottleneck
- to give fast access to local copies

Updates

updates to replicated objects

AND

related updates to different objects

must be managed in the light of 1-4 above

WEAK CONSISTENCY - fast access requirement dominates

update the "local" replica and send update messages to other replicas.
Different replicas may return different values for an item.

STRONG CONSISTENCY - reliability, no single bottleneck

ensure that only consistent state can be seen (e.g. lock-all, update, unlock).
All replicas return the same value for an item.

WEAK CONSISTENCY - system model, architecture and engineering

- * **Simple approach:** have a PRIMARY COPY to which all updates are made and a number of BACKUP copies to which updates are propagated.

Keep a HOT STANDBY for some applications for reliability and accessibility (make update to hot standby synchronously with primary update).

BUT a single primary copy becomes infeasible as systems' scale and distribution increase
- primary copy becomes a bottleneck and (remote) access is slow.

- * **General approach:** we must allow (concurrent) reads and writes to all replicas

Weak Consistency of replicas - continued

Requirement: the system **MUST** be made to converge to a consistent state as the update messages propagate

PROBLEMS (ref. DS properties 1-4)

1. concurrent updates at different replicas + (3) comms. delay

- the updates do not, in general, reach all replicas in the same (total) order? - see T-23
- the order of *conflicting* updates matters

2. failures of replicas

we must ensure, by restart procedures, that every update eventually reaches all replicas

4. no global time

but we need at least a **convention** for arbitrating between conflicting updates

- **timestamps?** - **are clocks synchronised?**

e.g. conflicting values for the same named entry - password or authorisation change

e.g. add/remove item from list - distribution list, access control list, hot list

e.g. tracking a moving object - times must make physical sense

e.g. processing an audit log - times must reflect physical causality

In practice, systems may not rely solely on message propagation but also **compare state** from time to time
e.g. name servers - Grapevine, GNS

Further reading:

Y Saito and M Shapiro, "Optimistic Replication" ACM Computing Surveys 37(1) pp.42-81, March 2005

Strong consistency

concept of TRANSACTION / transactional semantics

- ACID properties (atomicity, consistency, isolation, durability)

start transaction

make the **same update** to all **replicas** of an object

or make **related updates** to a number of **different objects**

end transaction

(either COMMIT - all updates are made, are visible and persist
or ABORT - no changes are made)

implementation - first attempt:

lock all objects

make update(s)

unlock all objects

problem:

- lack of availability (a reason for replication)

because of comms. delays, overload/slowness and failures.

"I can't work because a replica in Peru crashed"

solution for **strong consistency of replicas:**

QUORUM ASSEMBLY

QUORUM ASSEMBLY for replicas

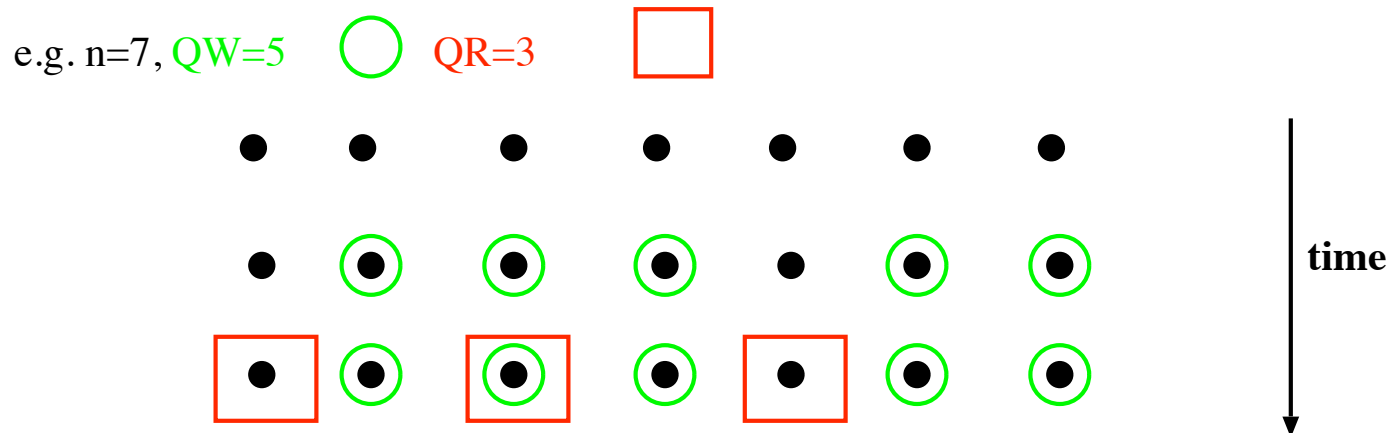
Assume n copies. Define a read quorum QR and a write quorum QW which must be **locked** for reading (QR) and writing (QW).

$$QW > n/2$$

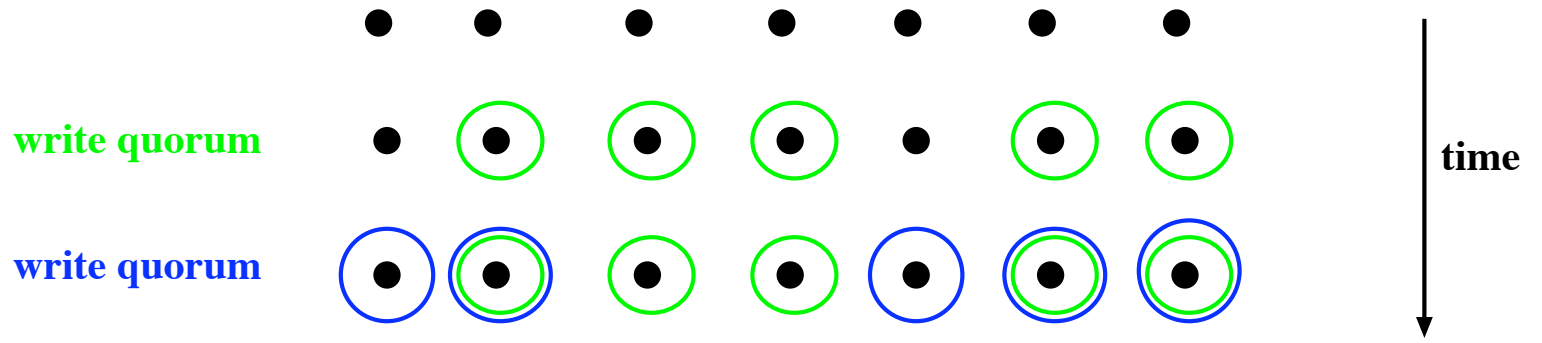
$$QR + QW > n$$

These ensure: only one write quorum at a time can be assembled (deadlock? - see D10)
 every QW and QR contain at least one up-to-date replica.
 After assembling a (write) quorum, make all replicas consistent then do operation.

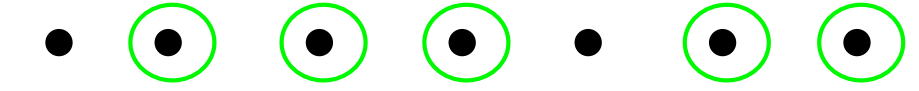
e.g. $QW = n$, $QR = 1$ is lock all copies for writing, read from any



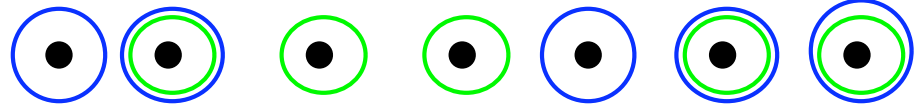
optimisation: after making a write quorum consistent, and performing the update, background-propagate to other replicas not in the quorum



write quorum

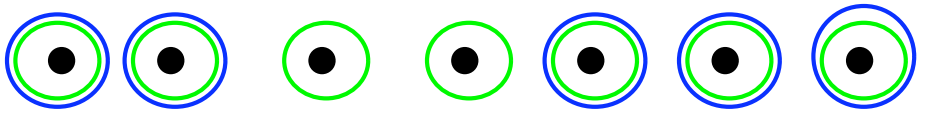


write quorum

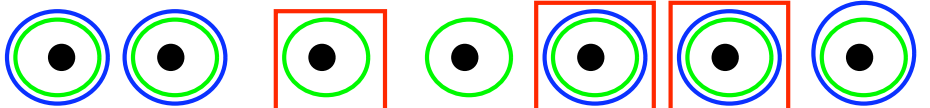


make consistent then apply update

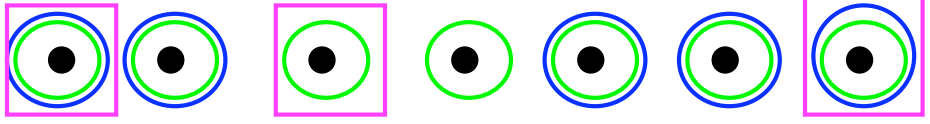
write



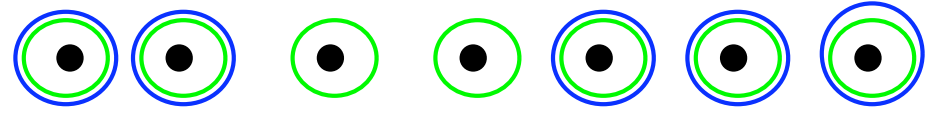
read



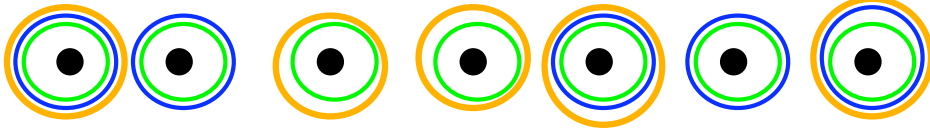
read



note that reads don't change anything, so we still have:

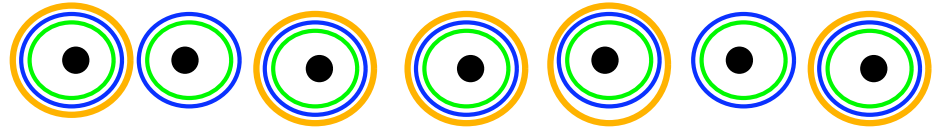


write quorum



make consistent then apply update

write

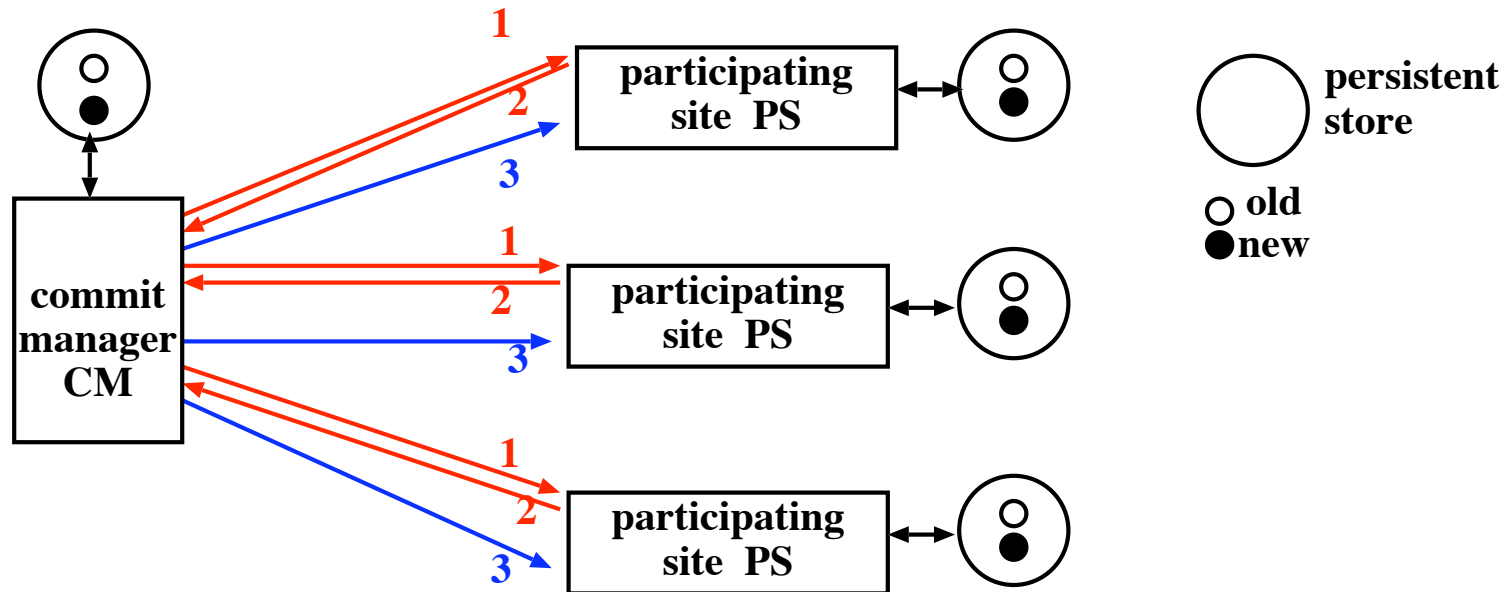


Atomic Update of distributed data

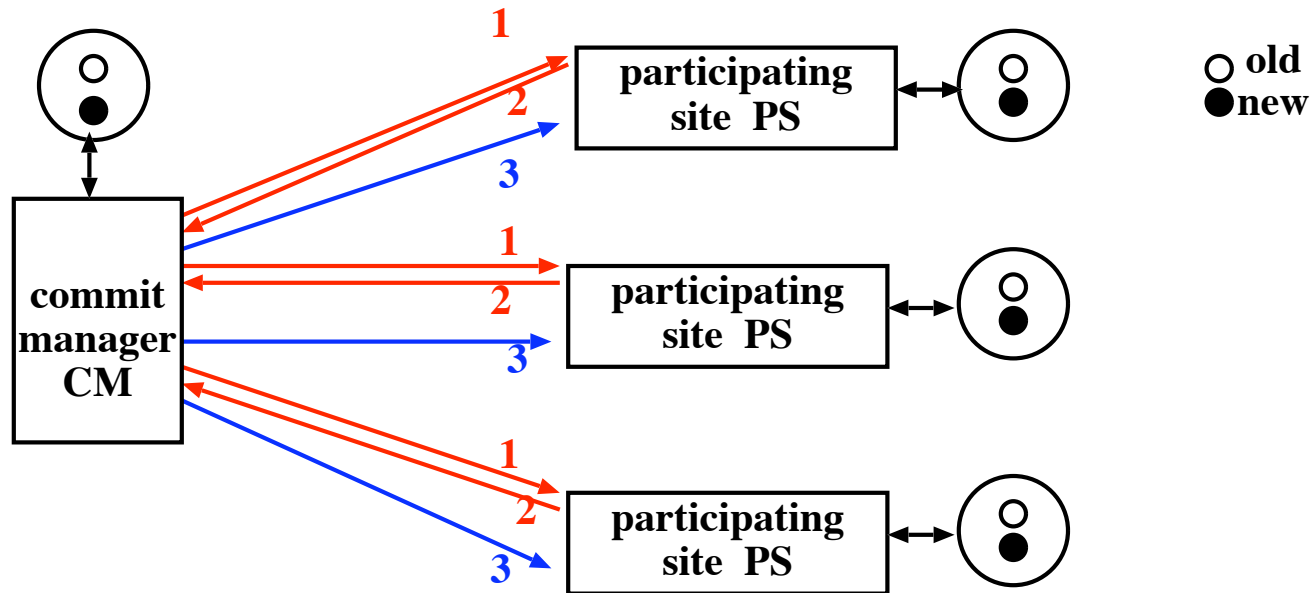
For BOTH quora of **replicas** AND **related objects** being updated under a transaction we need **ATOMIC COMMITMENT** (all make the update(s) or none does)

- achieved by an ATOMIC COMMITMENT PROTOCOL such as two-phase commit (2PC), an ISO standard

Two-phase commit (2PC)



e.g. for a group of four processes:
 one functions as commit manager (CM) (after an external update request)
 the others are participating sites (PS)

**phase 1:**

1. CM requests votes from all (PS and CM) - all secure data and vote
2. CM assembles votes including its own

phase 2:

- CM decides on commit (if all have voted) or abort
- This is the single point of decision - record in persistent store
3. propagate decision

Two-phase commit - notes:

recall - DS - independent failure modes

- * **before voting commit**, each PS must:
 - record update in stable/persistent storage
 - record that 2PC is in progress



crash



on restart, find out what the decision was from CM

- * **before deciding commit**, CM must:
 - get commit votes from all PSs
 - record its own update

on deciding commit, CM must

- record the decision
- then . . . propagate . . . it



crash



on restart, tell the PSs the decision

some detail from the PS algorithm

either: send **abort** vote and exit protocol
or: send **commit** vote and await decision (set timer)

timer expires: consider CM crash which **could have happened**:

either before CM decided commit (perhaps awaiting slow or crashed PSs)

or after deciding commit and
before propagating decision to any PS
after propagating decision to some PSs

optimise - CM propagates PS list so any can be asked for the decision

some detail from the CM algorithm

send vote request to each PS
await replies (set timers)

if any PS does not reply, must abort
if 2PC is for quorum update, CM may contact further replicas after abort

Concurrency issues

consider a process group, each managing an object replica
(ref. T-11, assume open group with no internal structure)

suppose **two (or more)** different updates are requested at different replica managers

two (or more) replica managers attempt to assemble a write quorum
if successful, will run a 2PC protocol as CM

either: one succeeds in assembling a write quorum, the other(s) fail - OK
or: both/all fail to assemble a quorum (e.g. each of two locks half the replicas)
and **DEADLOCK**

must have **deadlock detection or prevention**

assume all quorum assembly requests are multicast to all the replica managers
e.g. quorum assembler's timer expires waiting for enough replicas to join
it releases locked replicas and restarts, after backing off for some time

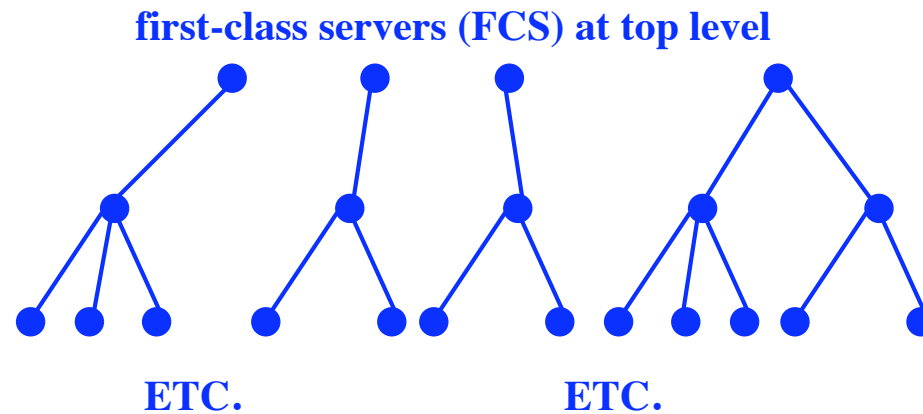
e.g. some replica manager has become part of a quorum (request had a timestamp)
then receives a request from a different quorum assembler (with a timestamp)
All replica managers have an algorithm to agree which quorum wins
and which is aborted, e.g. based on "earliest timestamp wins".

e.g. could use a structured group so update requests are forwarded to the manager

it is difficult to assemble a quorum from a large number of widely distributed replicas

manage scale by hierarchy:

define a small set of first-class servers (FCSs), each above a hierarchy of servers



there are various approaches, depending on application requirements, e.g.

- * update requests must be made to a FCS
- * FCSs use quorum assembly and 2PC among FCSs then propagate the update to all FCSs - each propagates down its subtree(s)
- * **correct read** is from a FCS which assembles a read quorum of FCSs
fast read is from any server - risk missing latest updates

ref lab TR 383 N Adly (PhD) Management of replicated data in large-scale systems (includes a survey of algorithms for both strong and weak consistency) Nov 1995

Concurrency control: general transaction scenario (distributed objects)

(ref: CS/OS books part 3, CSA course, DB course)

- * consider **related updates to different objects**
- * transactions that involve distributed objects, any of which may fail at any time, must ensure atomic commitment
- * concurrent transactions may have objects in common

pessimistic concurrency control

(strict) two-phase locking (2PL)

(strict) timestamp ordering (TSO)

optimistic concurrency control (OCC)

- take shadow copies of objects

- apply updates to shadows

- request commit of a validator which implements commit or abort (do nothing)

- * with pessimistic concurrency control we must use an atomic commitment protocol such as two-phase commit
- * if a fully optimistic approach is taken we do not lock objects for commitment since the validator creates new object versions (ref CS ED1 Ch20, ED2 Ch21, OS Ch 22)

Concurrency control for transactions

strict two-phase locking 2PL

phase 1:

for objects involved in the transaction, attempt to lock object and apply update

- old and new versions are kept
- locks are held while other objects are acquired
- susceptible to DEADLOCK

phase 2:

(for STRICT 2PL, locks are held until commit)

commit update - using e.g. 2PC

strict timestamp ordering

each transaction is given a timestamp

for objects involved in the transaction - attempt to lock object and apply update

- old and new versions are kept

the object compares the timestamp of the requesting transaction with that of its most recent update:

if later - OK

if earlier - REJECT (too late) - that transaction aborts

(for STRICT TSO, locks are held until commit)

commit update - using e.g. 2PC

More algorithms and protocols for Distributed Systems

We have defined process groups as having **peer** or **hierarchical** structure (ref. T-14) and have seen that a **coordinator** may be needed to run e.g. 2PC

With peer structure, an external process may send an update request to any group member which then functions as coordinator

If the group has hierarchical structure, one member is elected as coordinator

That member must manage group protocols and external requests must be directed to it (note that this solves the concurrency control (potential deadlock) problem while creating a single point of failure and a possible bottleneck)

Assume: each process has a **unique ID** known to all members
the process with **highest ID is coordinator**

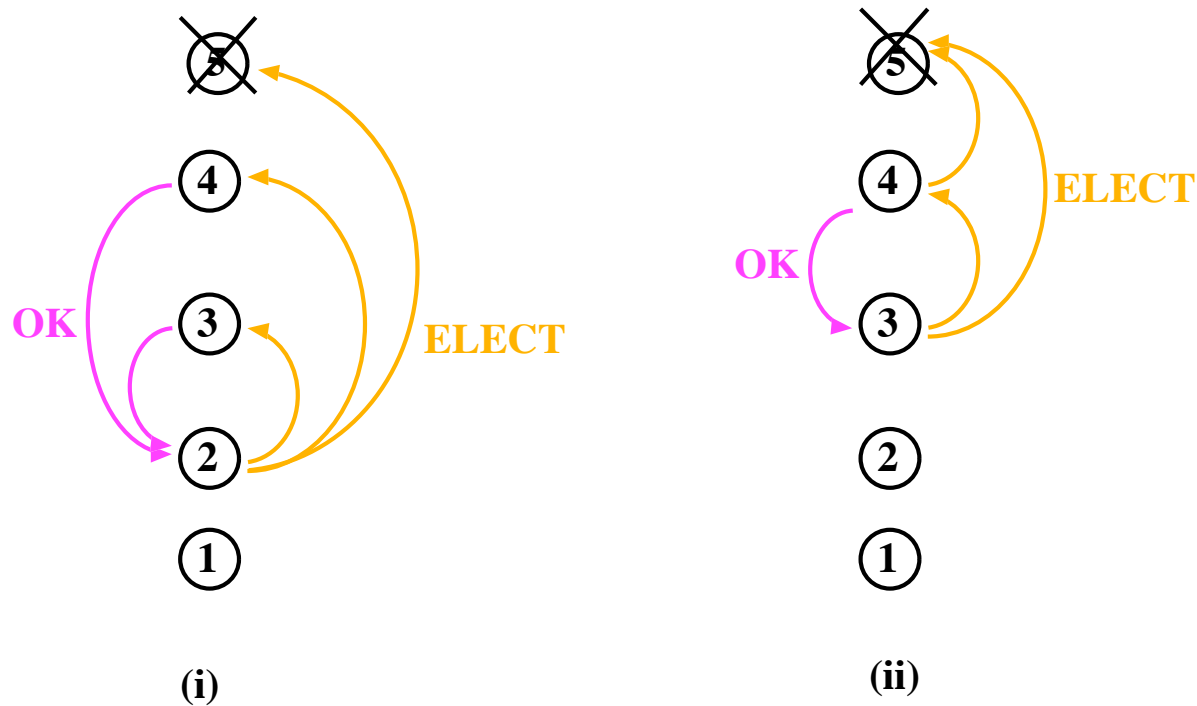
Note: any process may fail at any time

An ELECTION ALGORITHM is run when the coordinator is detected as having failed

Election algorithm - BULLY

- * P notices no reply from coordinator
- * P sends ELECT message to all processes with higher IDs
- * If any reply, P exits
- * if none reply, P wins
 - P gets any state needed from storage
 - P sends COORD message to the group

-
- * on receipt of ELECT message
 - send OK
 - hold an election if not already holding one
-



Election algorithm - RING

Processes are ordered into a ring - known to all

- can bypass a failed process provided algorithm uses acknowledgements

* P notices coordinator not functioning

* P sends ELECT, tagged with its own ID, around the ring

on receipt of ELECT:

without receiver's ID: append ID and pass on

with receiver's ID (has been all round) send (COORD, highest-ID)

many elections may run concurrently
all should agree on the same highest ID

Distributed mutual exclusion

Suppose N processes hold an object replica
and we require that only one at a time may access the object

examples: - ensuring coherence of distributed shared memory
- distributed games
- distributed whiteboard

i.e. for use by simultaneously running, tightly coupled components managing object replicas in main memory. We have already seen the approach of LOCKING persistent object replicas for transactional update.

Assume - the object is of fixed structure,
- processes update-in-place
- then the update is propagated (not part of the algorithm)

Each process executes code of the form:

entry protocol

critical section (access object)

exit protocol

Distributed mutual exclusion:

1. centralised algorithm

one process is elected as coordinator

entry protocol:

send **request** message to coordinator
wait for reply (OK-enter) from coordinator

exit protocol:

send **finished** message to coordinator

- + FCFS or priority or another policy - coordinator reorders
 - + economical (3 messages)
 - single point of failure
 - coordinator is bottleneck
 - what does no reply mean? waiting for region? - OK!
 coordinator has failed?
- solve by using extra messages
- coordinator ack's request
 - send again when process can enter region
 - send periodic heartbeats (at application- or a lower level)

Distributed mutual exclusion:

2. token ring

a token giving permission to enter critical region circulates indefinitely

entry protocol:

wait for token to arrive

exit protocol:

pass token to next process

- inflexible - ring order, not FCFS or priority or ...
- + quite efficient,
but token circulates when no-one wants region
- must handle loss of token
- crashes? use ack - reconfigure - bypass

Distributed mutual exclusion:

3. distributed (peer-to-peer) algorithm

entry protocol:

send a timestamped request to all processes including oneself
(there is a convention for global ordering of TS)

only when all process have replied can the region be entered

on receipt of a message

- defer reply if in CR
- reply immediately if you are not executing a request
- if you are,
 - compare your request message timestamp with that of the message.
 - reply immediately if incoming timestamp is earlier, otherwise, defer reply

exit protocol:

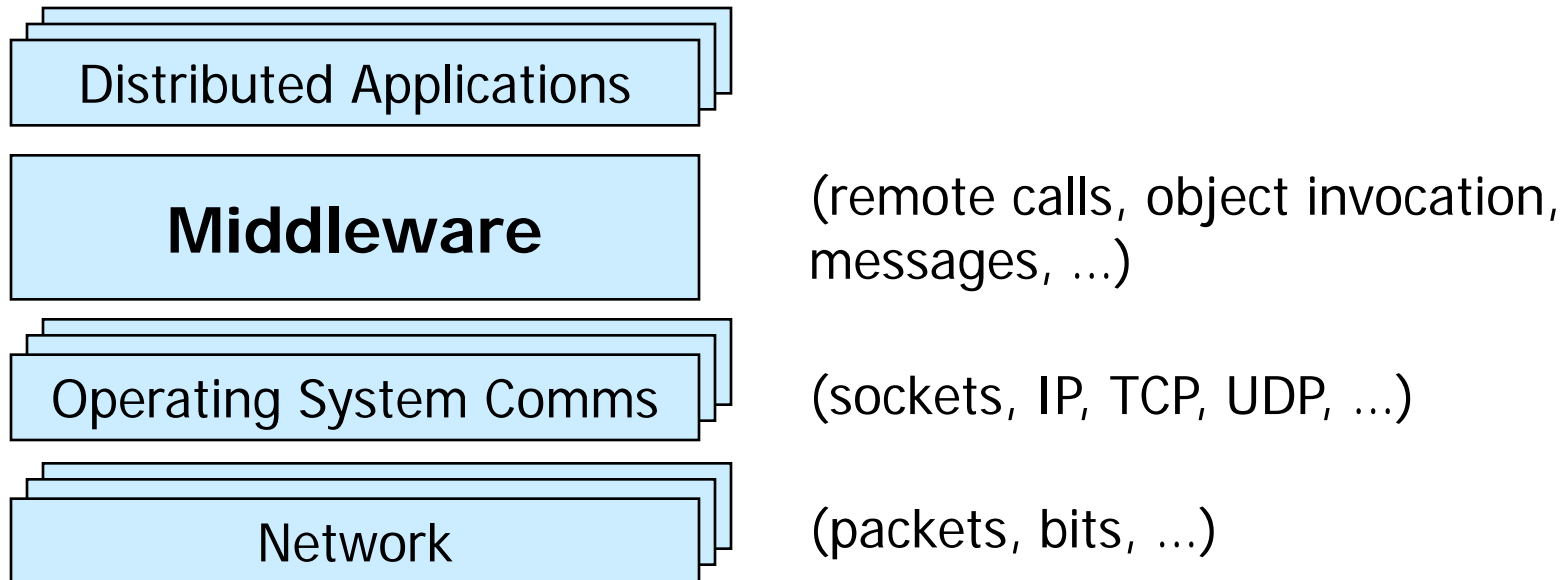
reply to any deferred requests

+ fair - FCFS

- not economical, $2(n-1)$ messages + any acks
- n points of failure
- n bottlenecks
- no reply - failure or deferral?

Introduction to Middleware I

- What is Middleware?
 - Layer between OS and distributed applications
 - Hides complexity and heterogeneity of distributed system
 - Bridges gap between low-level OS comms and programming language abstractions
 - Provides common programming abstraction and infrastructure for distributed applications



Introduction to Middleware II

- Middleware provides support for (some of):
 - Naming, Location, Service discovery, Replication
 - Protocol handling, Communication faults, QoS
 - Synchronisation, Concurrency, Transactions, Storage
 - Access control, Authentication
- Middleware dimensions:

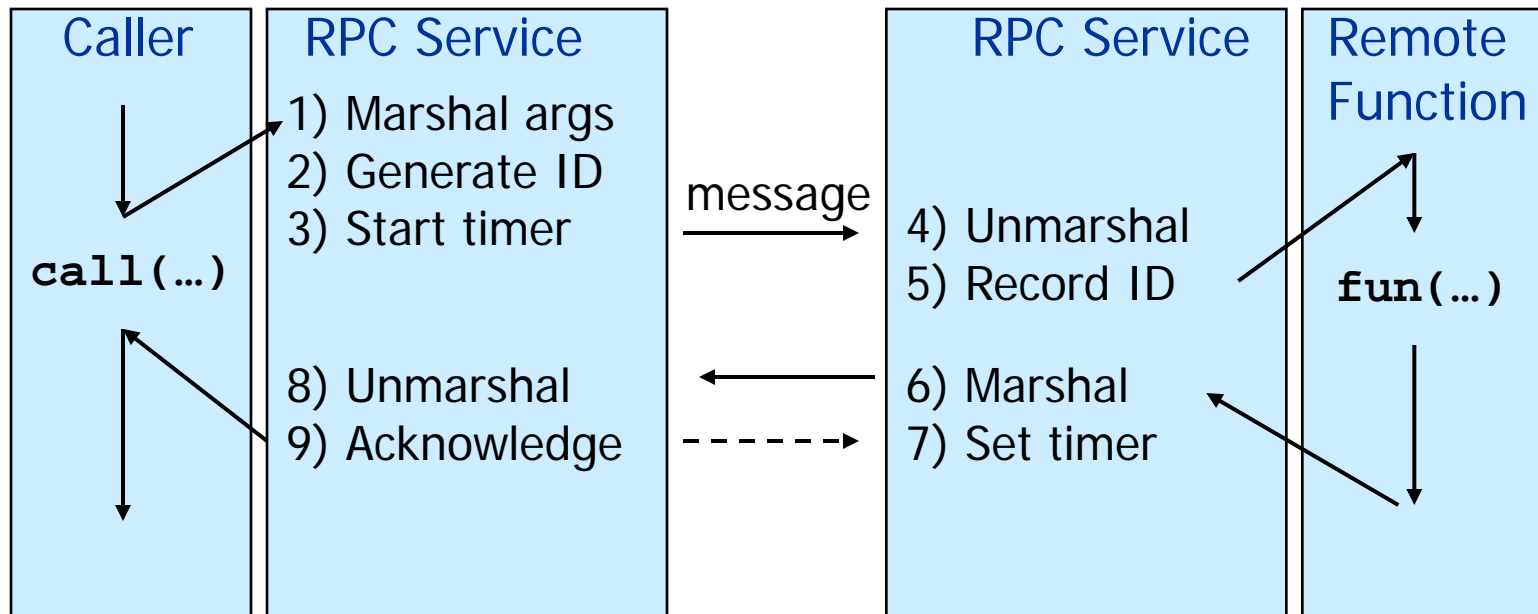
Request/Reply	vs.	Asynchronous Messaging
Language-specific	vs.	Language-independent
Proprietary	vs.	Standards-based
Small-scale	vs.	Large-scale
Tightly-coupled	vs.	Loosely-coupled components

Outline

- **Part I: Remote Procedure Call (RPC)**
 - Historic interest, but can still be very useful
- **Part II: Object-Oriented Middleware (OOM)**
 - Java RMI
 - CORBA
 - Reflective Middleware *research*
- **Part III: Message-Oriented Middleware (MOM)**
 - Java Message Service
 - IBM MQSeries
 - Web Services
- **Part IV: Event-Based Middleware**
 - Cambridge Event Architecture
 - Hermes *research*

Part I: Remote Procedure Call (RPC)

- Makes remote function calls look local
- Client/server model
- Request/reply paradigm usually implemented with message passing in RPC service
- Marshalling of function parameters and return value



Properties of RPC

Language-level pattern of **function call**

- easy to understand for programmer

Synchronous request/reply interaction

- natural from a programming language point of view
- matches replies to requests
- built in synchronisation of requests and replies

Distribution transparency (in the no-failure case)

- hides the complexity of a distributed system

Various **reliability** guarantees

- deals with some distributed systems aspects of failure

Failure Modes of RPC

- Invocation semantics supported by RPC in the light of:
network and/or server congestion,
client, network and/or server failure
note DS independent failure modes
- RPC systems differ, many examples, local was Mayflower

At most once (RPC system tries once)

- Error return – programmer may retry

Exactly once (RPC system retries a few times)

- Hard error return – some failure most likely
note that “exactly once” cannot be guaranteed

Disadvantages of RPC

✘ Synchronous request/reply interaction

- tight coupling between client and server
- may block for a long time
- leads to multi-threaded programming at client and, especially, server

✘ Distribution Transparency

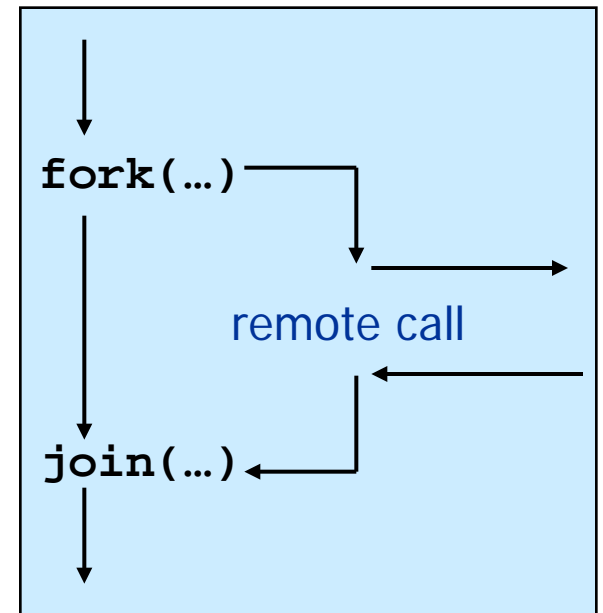
- Not possible to mask all problems

✘ Lacks notion of service

- programmer may not be interested in specific servers

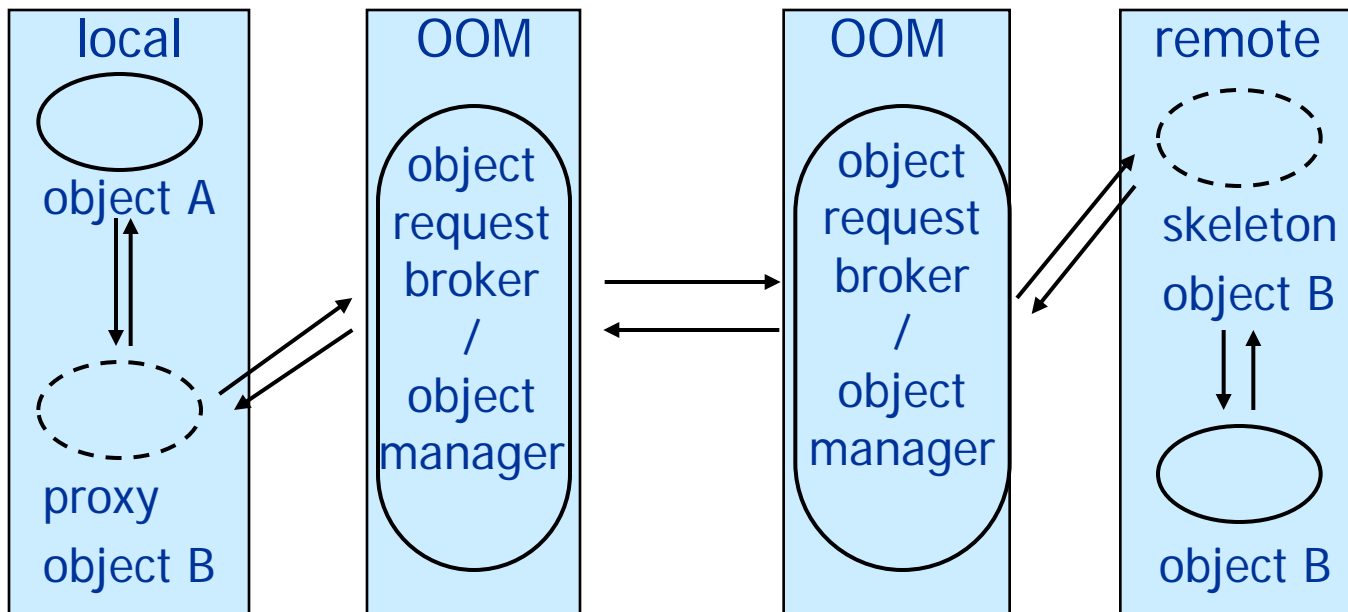
✘ RPC paradigm is not object-oriented

- invoke functions on servers as opposed to methods on objects



Part II: Object-Oriented Middleware (OOM)

- **Objects** can be *local* or *remote*
- **Object references** can be *local* or *remote*
- Remote objects have visible **remote interfaces**
- Makes remote objects look local using **proxy objects**
- **Remote method invocation** looks like this



Properties of OOM

Support for object-oriented programming model

- objects, methods, interfaces, encapsulation, ...
- exceptions (also in some RPC systems e.g. Mayflower)

Location Transparency

- system maps object references to locations

Synchronous request/reply interaction

- same as RPC

Services

- easier to build using object concepts

Java Remote Method Invocation (RMI)

- Distributed objects in Java

```
public interface PrintService extends Remote {  
    int print(Vector printJob) throws RemoteException;  
}
```

- RMI compiler creates proxies and skeletons
- RMI registry used for interface lookup
- Everything in Java, unless you like pain (single-language system)

CORBA

- **Common Object Request Broker Architecture**
 - Open standard by the OMG (Version 3.0)
 - Language and platform independent
- **Object Request Broker (ORB)**
 - General Inter-ORB Protocol (GIOP) for communication
 - Interoperable Object References (IOR) contain object location
 - **CORBA Interface Definition Language (IDL)**
 - Stubs (proxies) and skeletons created by IDL compiler
 - Dynamic remote method invocation
- **Interface Repository**
 - Querying existing remote interfaces
- **Implementation Repository**
 - Activating remote objects on demand

CORBA IDL

- Definition of language-independent remote interfaces
 - **Language mappings** to C++, Java, Smalltalk, ...
 - Translation by IDL compiler
- Type system
 - *basic types*: long (32 bit), long long (64 bit), short, float, char, boolean, octet, any, ...
 - *constructed types*: struct, union, sequence, array, enum
 - *objects* (common super type **Object**)
- Parameter passing
 - **in, out, inout**
 - basic & constructed types passed by value
 - objects passed by reference

```
typedef sequence<string> Files;  
interface PrintService : Server {  
    void print(in Files printJob);  
};
```

CORBA Services (selection)

- Naming Service
 - Names → remote object references
- Trading Service
 - Attributes (properties) → remote object references
- Persistent Object Service
 - Implementation of persistent CORBA objects
- Transaction Service
 - Making object invocation a part of transactions
- Event Service and Notification Service
 - Asynchronous communication based on messaging (cf. MOM); not integrated programming model with general IDL messages

Disadvantages of OOM

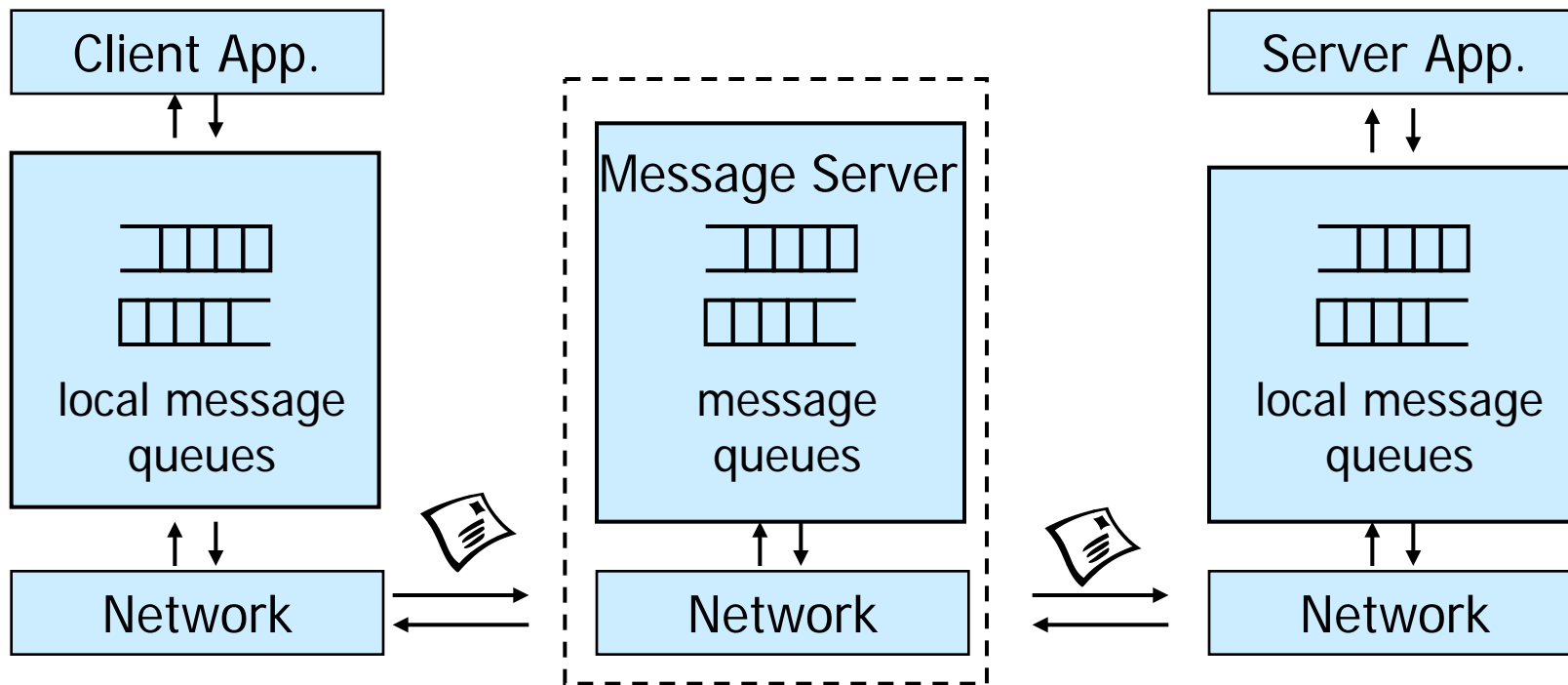
- ✘ Synchronous request/reply interaction only
 - So CORBA **oneway** semantics added
 - Asynchronous Method Invocation (AMI)
 - Can be yucky
 - But *implementations* may not be loosely coupled
- ✘ Distributed garbage collection
 - Releasing memory for unused remote objects
- ✘ OOM rather static and heavy-weight
 - Bad for ubiquitous systems and embedded devices

Reflective Middleware

- Flexible middleware (OOM) for mobile and context-aware applications – adapt to context through *monitoring* and *substitution* of components
- Interfaces for **reflection**
 - Objects can inspect middleware behaviour
- Interfaces for **customisability**
 - Dynamic reconfiguration depending on environment
 - Different protocols, QoS, ...
 - e.g. use different marshalling strategy over unreliable wireless link

Part III: Message-Oriented Middleware (MOM)

- Communication using **messages**
- Messages stored in **message queues**
- Optional **message servers** decouple client and server
- Various assumptions about **message content**



Properties of MOM

Asynchronous interaction

- Client and server are only **loosely coupled**
- Messages are queued
- Good for application integration

Support for **reliable** delivery service

- Keep queues in persistent storage

Processing of messages by intermediate message server

- May do filtering, transforming, logging, ...
- Networks of message servers

Natural for database integration

IBM MQSeries

- One-to-one reliable message passing using queues
 - Persistent and non-persistent messages
 - Message priorities, message notification
- **Queue Managers**
 - Responsible for queues
 - Transfer messages from input to output queues
 - Keep routing tables
- **Message Channels**
 - Reliable connections between queue managers

• Messaging API:	MQopen	Open a queue
	MQclose	Close a queue
	MQput	Put message into opened queue
	MQget	Get message from local queue

Java Message Service (JMS)

- **API specification** to access MOM implementations
- Two modes of operation *specified*:
 - **Point-to-point**
 - One-to-one communication using queues
 - **Publish/Subscribe**
 - cf. Event-Based Middleware
- **JMS Server** implements JMS API
- JMS Clients connect to JMS servers
- Java objects can be serialised to JMS messages
- A JMS interface has been provided for MQ
- pub/sub? - just a specification?

Disadvantages of MOM

- ✘ Poor programming abstraction (but has evolved)
 - Rather low-level (cf. Packets)
 - Request/reply more difficult to achieve
 - Can lead to multi-threaded code
- ✘ Message formats unknown to middleware
 - No type checking (JMS addresses this – implementation?)
- ✘ Queue abstraction only gives one-to-one communication
 - Limits scalability (JMS pub/sub – implementation?)

Web Services

- Use well-known web standards for distributed computing

Communication

- Message content expressed in **XML**
- **Simple Object Access Protocol (SOAP)**
 - Lightweight protocol for sync/async communication

Service Description

- **Web Services Description Language (WSDL)**
 - Interface description for web services

Service Discovery

- **Universal Description Discovery and Integration (UDDI)**
 - Directory with web service descriptions in WSDL

Properties of Web Services

Language-independent and open standard

SOAP offers OOM and MOM-style communication:

- Synchronous request/reply like OOM
- Asynchronous messaging like MOM
- Supports internet transports (http, smtp, ...)
- Uses XML Schema for marshalling types to/from programming language types

WSDL says how to use a web service

<http://api.google.com/GoogleSearch.wsdl>

UDDI helps to find the right web service

- Exports SOAP API for access

Disadvantages of Web Services

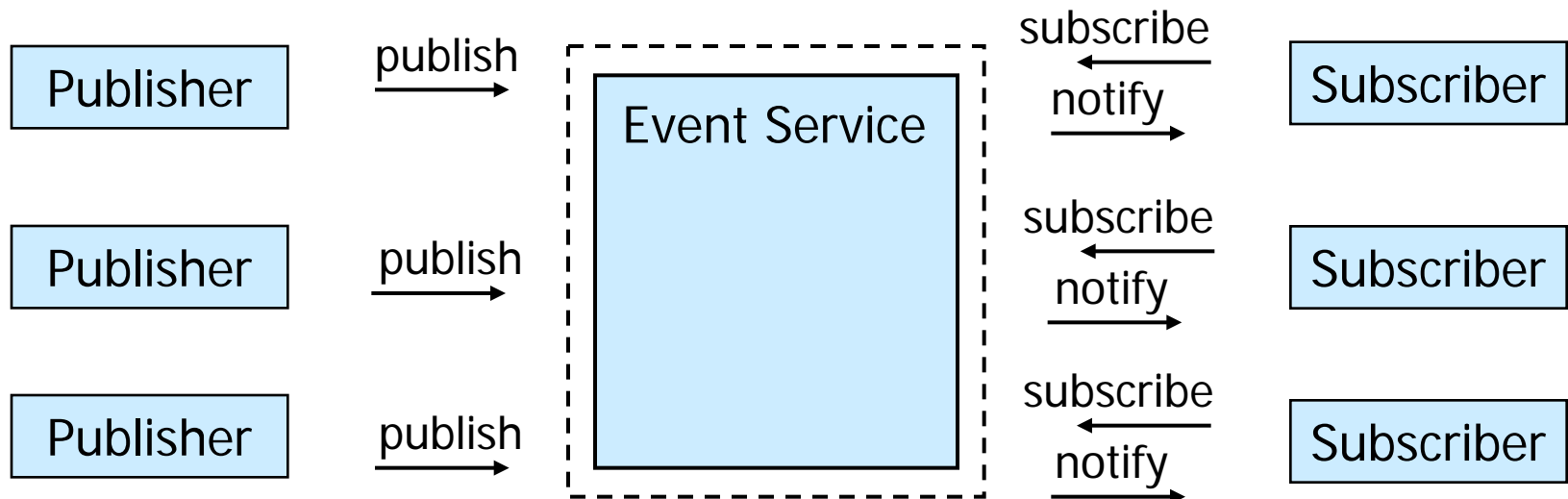
- ✘ Low-level abstraction
 - leaves a lot to be implemented
- ✘ Interaction patterns have to be built
 - one-to-one and request-reply provided
 - one-to-many?
 - still service invocation, rather than notification
 - nested/grouped invocations, transactions, ...
- ✘ Location transparency?

What we lack, so far

- ✘ General interaction patterns
 - we have one-to-one and request-reply
 - one-to-many? many to many?
 - notification?
 - dynamic joining and leaving?
- ✘ Location transparency
 - anonymity of communicating entities
- ✘ Support for pervasive computing
 - data values from sensors

Part IV: Event-Based Middleware aka Publish/Subscribe

- **Publishers** (advertise and) *publish events* (messages)
- **Subscribers** express interest in events with *subscriptions*
- **Event Service** *notifies* interested subscribers of published events
- Events can have arbitrary content (typed) or name/value pairs



Topic-Based and Content-Based Pub/Sub

- Event Service matches events against subscriptions
- What do subscriptions look like?

Topic-Based Publish/Subscribe

- Publishers publish events belonging to **topic** or **subject**
- Subscribers subscribe to **topic**

```
subscribe(PrintJobFinishedTopic, ...)
```

(Topic and) Content-Based Publish/Subscribe

- Publishers publish events belonging to **topics** and
- Subscribers provide a **filter** based on *content* of events

```
subscribe(type=printjobfinshed, printer='aspen', ...)
```

Properties of Publish/Subscribe

Asynchronous communication

- Publishers and subscribers are loosely coupled

Many-to-many interaction between pubs. and subs.

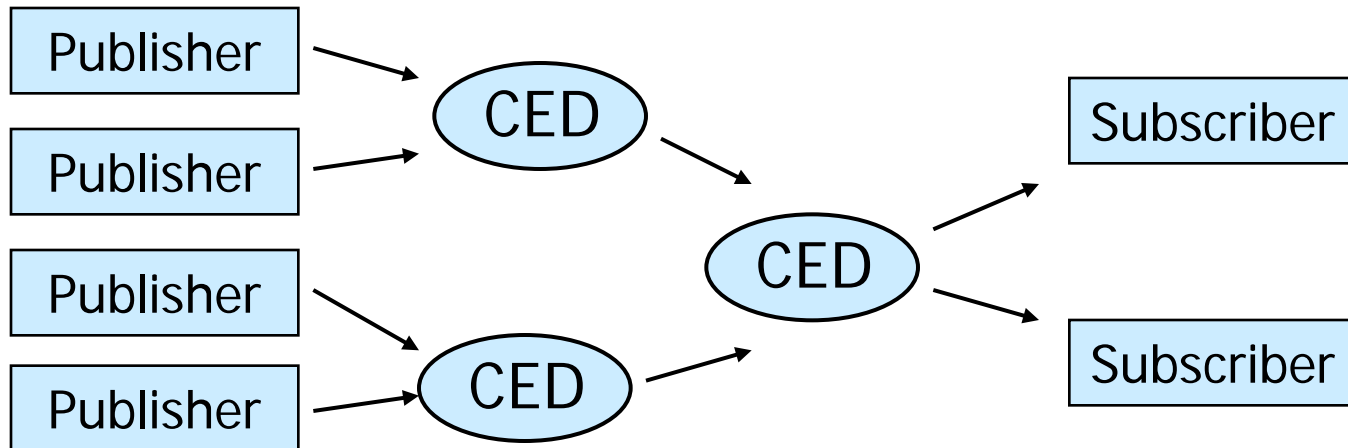
- Scalable scheme for large-scale systems
- Publishers do not need to know subscribers, and vice-versa
- Dynamic join and leave of pubs, subs, (brokers - see later)

(Topic and) Content-based pub/sub very expressive

- Filtered information delivered only to interested parties
- Efficient content-based routing through a broker network

Composite Event Detection (CED)

- Content-based pub/sub may not be expressive enough
 - Potentially thousands of event types (primitive events)
 - Subscribers interest: event *patterns* (define *high-level* events)
- **Event Patterns**
`PrinterOutOfPaperEvent` Or `PrinterOutOfTonerEvent`
- **Composite Event Detectors (CED)**
 - Subscribe to primitive events and publish composite events



Summary

- Middleware is an important abstraction for building distributed systems

1. Remote Procedure Call
2. Object-Oriented Middleware
3. Message-Oriented Middleware
4. Event-Based Middleware

- Synchronous vs. asynchronous communication
- Scalability, many-to-many communication
- Language integration
- Ubiquitous systems, mobile systems

Names in Distributed Systems

N-1

- * **Unique Identifiers (UIDs)** e.g. 128 bits
 - are never reused
 - refer to either: nothing, or: the same thing at all times
 - UIDs (may) achieve location-independence: the named object can be moved

- * **pure and impure names (Needham)**
 - * **pure names**
 - the name itself yields no information e.g. UID
 - contains no location information
 - commits system to nothing
 - can only be used to compare with other similar bit-patterns e.g. in table look-up

 - * **impure names**
 - examples: email addresses**
foo.cl.cam.ac.uk
 - host-ID, object-ID (unless used ONLY to generate UIDs)**
 - disc-pack-ID, object-ID**

- the name yields information
- commits the system to maintaining the context in which the name is to be resolved
e.g. the directory hierarchy uk/ac/cam/cl

Unique names

uniqueness is achievable by using

- a **hierarchical name**: scope of uniqueness is level in hierarchy
- a **bit pattern**: flat, system-wide uniqueness

Issues

* a problem with **pure names**:

- where to look them up?
- how do you know that an object does not exist?
how may a global search be avoided?
- how to engineer uniqueness?

* a problem with **impure names**

- how to restructure the namespace
e.g. when objects move about
when companies restructure

examples of names - note requirement for **unique** identification

Health Service ID - every citizen at birth (but hospitals still use local names)

UK National Insurance - on employment

US Social Security - on employment

Passport

Driving licence

Professional Societies: ACM, IEEE, BCS

Charities: National Trust, RSPB, ...

Services: RAC, AA, AAA (US)

Credit cards

Bank accounts

Utilities: gas/electricity/water/phone customer numbers

Loyalty schemes: Airlines - frequent flyer, hotels, shops

Database key - must be unique. e.g. "David Evans" could be a poor choice

look for structure, explicit or implicit

is allocation centralised or distributed?

what is the resolution context?

Telephone company analogy (wired service)

- * geographically partitioned, distributed naming database
 - electronic is current, paper is an official cache
- * given a name, (Yudel Luke) or (Yudel Luke, 3 Acacia Drive) which directory to use?
 - don't know where to lookup pure names
- * call (#) -> unobtainable, # came from official cache
 - we detect out-of-date values, call directory enquiries
 - cache until new directory comes
- * caching numbers in a personal address book (an unofficial cache)
 - call (#) -> unobtainable, report fault if use often
 - call directory enquiries, or ask another contact (may have moved, or changed provider)
- * frequency of update (some years ago)
 - e.g. Cambridge area: 1,000,000 entries
 - 5,000 updates per week
- * can't find an entry
 - e.g. Phillips company - check spelling: Philips
 - e.g. look under S.S. rather than Social Services

BT offer a web service www.thephonebook.com (name and address -> #)

- only offers exact search e.g. Phillips, Cambridge - doesn't suggest Philips

need higher-level tools - "*do you mean Philips?*" increasingly use search engines to augment directories

in general - provide clients with values of attributes of named objects

name space

the collection of valid names recognised by a name service
a precise specification is required, giving the structure of names

e.g. ISBN-10: 1-234567-89-1 namespace identifier: namespace-specific string
/a/b/c/d filing system, variable length, hierarchical
puccini.cl.cam.ac.uk DNS machine name, see later for DNS
e.g. Mach OS 128-bit port name (system-wide UID)

naming domain

a name space for which there exists a single overall administrative authority
for assigning names within it
this authority may **delegate** name assignment for nested sub-domains (see below for Internet DNS)

name resolution or binding

obtaining a value which allows an object to be used



LATE BINDING is considered GOOD PRACTICE

programs should contain names, not addresses e.g. a machine may fail and a service restarted on another.
Your local agent may cache resolved names for subsequent use + may expire values based on timestamp.
Cached values aren't embedded in programs and are always used at one's own risk.
If they don't work you have (your agent has) to look the name up again.

for a **large-scale system**, name resolution is an iterative process which requires navigation among name servers (N-8)

object type	attribute list
example: user	login name, mailbox host(s)
computer	architecture, OS, network-address, owner
service	network-address, version#, protocol
group	list of names of members
alias	canonical name
directory?	list of hosts holding the directory

directories may be held as a separate structure rather than as just a type of name as here

- * directories are likely to be **replicated** for scalability, fault-tolerance, efficiency
- * directory names will resolve to a list of hosts, as above, with their addresses to avoid further lookup
- * attribute-based (inverse) lookup may be offered - a YELLOW PAGES style of service for object discovery e.g. X.500, LDAP

TOO MUCH information might be dangerous - could reveal structure

A useful structure for names

query:

object type, object-name, attribute-name -> attribute-value

machine, foo.cl.cam.ac.uk, location -> IP address
user, some-user-name, public-key -> PK-bit-pattern
etc

checking:

object type, object-name, attribute-name, attribute-value -> yes/no

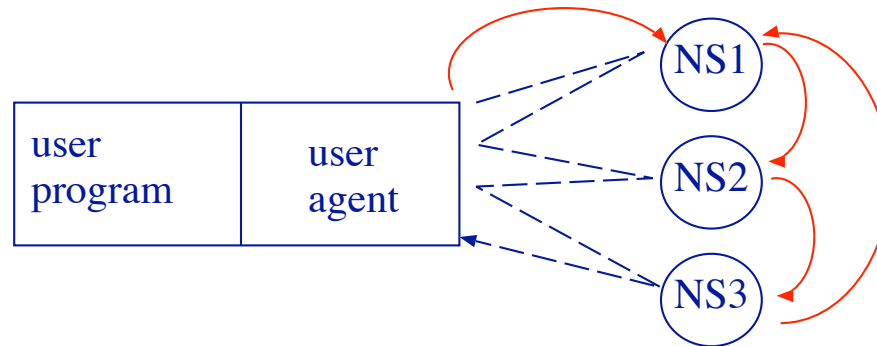
ACL, filename, some-user-name, write-access -> yes/no
(is the user on the ACL with the permissions requested? e.g. Xerox PARC Grapevine)

attribute-based lookup may be offered (yellow-pages style):

object-type, attribute-value -> list of object names having that attribute value

machine, OS version# -> list of machines

Architecture of name resolution



user agent starts off with the root address of the name service or some well-known sub-tree root:

e.g. the location of the uk directory for agents in the UK

to resolve `cl.cam.ac.uk`, the UA can start from the uk directory then ac then cam then cl

the UA (and directories) will cache resolved names as hints for future use

alternative: any name server will take a name,
 resolve it and return the required attribute
 client can sometimes choose e.g. select "recursive" in DNS

in practice, there are engineering optimisations: local caches are tried first, directories may be able to indicate whether they offer recursive evaluation, etc.

Example: DNS - the Internet Domain Name System

Before 1987 the whole naming database was held centrally and copied to selected servers periodically
 The Internet had become too large for this approach and a hierarchical scheme was needed
 (Mockapetris 1987)

What does it name? in practice, the objects named are:

- * computers
- * servers such as mail hosts *named objects* * domains *directories - resolution contexts*
- * servers providing other services

Definition of names

hierarchy of components or labels (total max 255 chars)

highest level of hierarchy is last component

label: max 63 chars, case insensitive, restrictions on characters (but arguments over relaxing these)

final label of a fully qualified name (a TLD) can be :

3+ letter code: type of hosting organisation

edu, gov, mil are still US-based, others e.g. **com, net, org, int, jobs** can be anywhere

2+ letter code: area of registrar, defined by ISO 3166 e.g. **uk, fr, ie, de,**

arpa: for inverse lookup (e.g. 20.0.232.128.in-addr.arpa)

mit.edu

cl.cam.ac.uk examples of domain names:

cs.tcd.ie

tu-darmstadt.de

final 2-letter label doesn't imply country of location of host, just where registered
 e.g. www.yahoo.co.uk has been in Germany

computers using DNS are grouped into *zones*, e.g. uk, cam
within a zone, management of sub-domains is delegated

e.g. **cl** is managed locally by the domain manager - names added to a local file

primary name server (*authority*) for a zone is the computer that holds the master list for the zone
usually there will be secondary servers, holding replicas, for the zone

queries can relate to individual hosts or zones/domains, examples:

A computer name -> IPv4 address
AAAA computer name -> IPv6 address
MX mail-host (domain) -> < host, preference, IP address > list
 includes mail hosts for detached computers
NS DNS servers for a domain -> < host, IP address, ... > list

Unix examples

/etc/hosts holds IP addresses of hosts in local domain

```
$ /usr/bin/nslookup
```

```
> set q=A
```

```
> cosi.cl.cam.ac.uk
```

```
Address: 128.232.8.110
```

```
> www.cl.cam.ac.uk
```

```
Address: 128.232.0.20
```

```
>set q=MX
```

```
>cl.cam.ac.uk
```

```
mail exchanger = 5 mx.cl.cam.ac.uk
```

```
Address: 128.232. ...
```

```
$ /usr/bin/nslookup
```

```
> set q=NS
```

```
> cl.cam.ac.uk
```

```
Server: 128.232.1.2
```

```
Address: 128.232.1.2#53
```

```
cl.cam.ac.uk nameserver=resolver1.cl.cam.ac.uk.
```

```
cl.cam.ac.uk nameserver=resolver6.cl.cam.ac.uk.
```

```
cl.cam.ac.uk nameserver=resolver2.cl.cam.ac.uk.
```

```
cl.cam.ac.uk nameserver=resolver3.cl.cam.ac.uk.
```

```
cl.cam.ac.uk nameserver=resolver0.cl.cam.ac.uk.
```

```
> set q=A
```

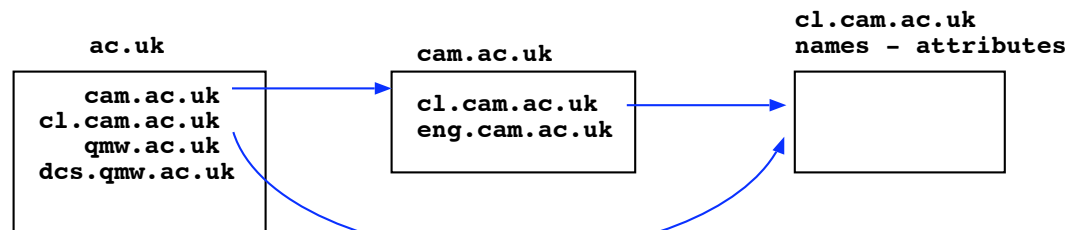
```
> resolver1.cl.cam.ac.uk
```

```
...
```

```
Address = 128.232.1.2
```

DNS name servers (note the large scale)

- * the domain database is partitioned into directories which form a distributed namespace
e.g. **ac.uk** may be held on the computer **ns1.cs.ucl.ac.uk**
- * can lookup DNS directory address for a domain: IP address, well-known port
need a starting point for name resolution



*frequently used so have
a redundant direct link (engineering issue)*

- * directories are replicated for availability and good response
(authoritative name server for domain is distinguished (weak consistency))

optimisations

- * resolved queries are cached (by user agent and at directories) as naming data tends to be stable
if not in directory, consult cache. values returned with a TTL (time to live)
- * queries and responses may be batched into composite query messages

1. mobile/roaming devices attach anywhere worldwide
2. mobile, wireless ad-hoc networks, groups form (MANETS)
 - (i) some node may act as an internet gateway
single-hop or multi-hop connection to it
 - (ii) may be detached from Internet, may be prepared to share services

approaches

first need protocols - mobile IP: mip6 IETF working group

for 1 above: device can contact local DNS server
local and home can cooperate
can you be monitored while roaming? - privacy?

for 2(i) above - any node can connect, as in 1, via the gateway,
provided gateway has appropriate code

for 2(ii) above: any node may broadcast offering to act as DNS server
assuming it has server code and others have client code
the group can then advertise services to clients

directories are replicated for scalability etc....

how should propagation of updates between replicas be managed?

lookup (args)  is the most recent value, known system-wide,
guaranteed to be returned?

if system-wide consistency is guaranteed we have: - delay to update
- delay on lookup

it is essential to have fast access to naming data

- so we must relax the strong consistency requirement

this is justified because:

1. naming data doesn't change very fast, changes propagate quickly, inconsistencies will be rare

YES - info on users and machines

NO - distribution lists

NEW/NO - mobile users and computers

NEW - huge number of things to be named - **does the design rely on low update traffic?**
(like service advertisements)

2. we detect obsolete naming data when it doesn't work

YES - users

NO - distribution lists

3. if it works it doesn't matter that it's out of date

you might have made the request a little earlier - recall uncertainties over time in DS

The crux of this problem

consistency vs availability tradeoff

have to choose availability for name services - they underlie most use of the system

what should be returned when only weak consistency is supported:

lookup (args) \longrightarrow either: value, version# / timestamp
 \longleftarrow or: not known at time (timestamp of last update)

examples:

service on failed machine - restart at new IP address - update directories - rare event

user changes company - coarse time grain

companies merge - coarse time grain

change of password - takes time to propagate - insecurity during propagation?

changes to ACLs and DLs - insecurity during propagation?

revocation of users' credentials - may have been used for authentication/authorisation

hot lists - must PUSH rather than PULL - must propagate fast

so - take care what name services are being used for, and how they are being used.

Perhaps active database triggers could be useful

(register interest in some change - notified of change immediately)

DNS-SD tries to do this

Long-term consistency must be ensured for correctness

requirement:

if updates stopped there would be consistency after all updates had propagated

this cannot be tested

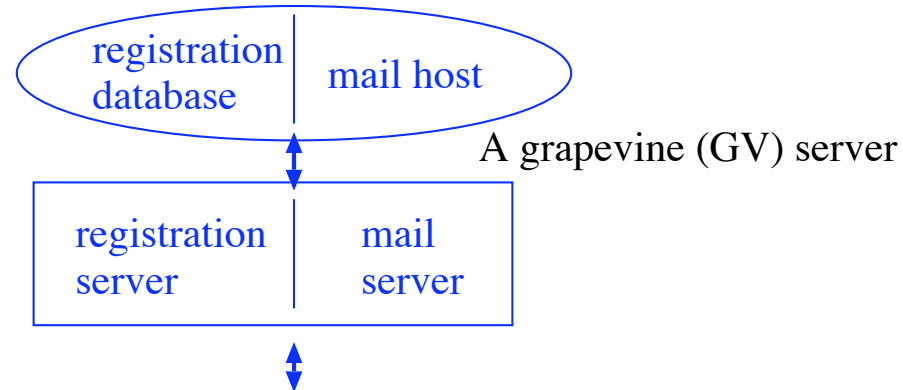
- we cannot guarantee there will be periods with no updates (quiescence)
- we would, in any case, need to specify failure behaviour in detail

* updates are propagated by the message transport system
conflicting updates might arrive out of order
need an arbitration policy e.g. based on timestamps

* typically, transmit whole directories periodically and compare them
tag the directory with a version number after this consistency check
e.g. GNS declares a new "epoch" after such a check

Example: Grapevine - outline

N-17



2D names name@registry

every GV server contains the gv registry which contains

registry name -> list of locations

2 types of name within a registry Note: small scale allows rapid navigation

group-name -> list of members

used for distribution lists,
access control lists

individual-name -> attributes: (password, mail-host list,

problems - soon outgrew its specification:

#servers, #clients

huge distribution lists not foreseen

message transport used for update messages - updates could be held up.

Butler Lampson 1986, Designing a Global Name Service, Proc. 5th ACM PODC, pp1-10

Aims: * long life

many changes in the organisation of the name space

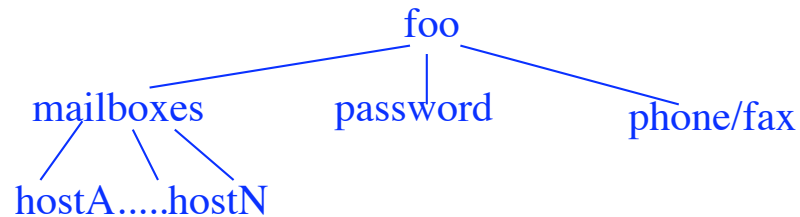
* large size

arbitrary number of names and administrative domains

Names

define two-dimensional (2D) names of the form < directory name, value name >

where value names may be a tree such as



the GNS directory structure is

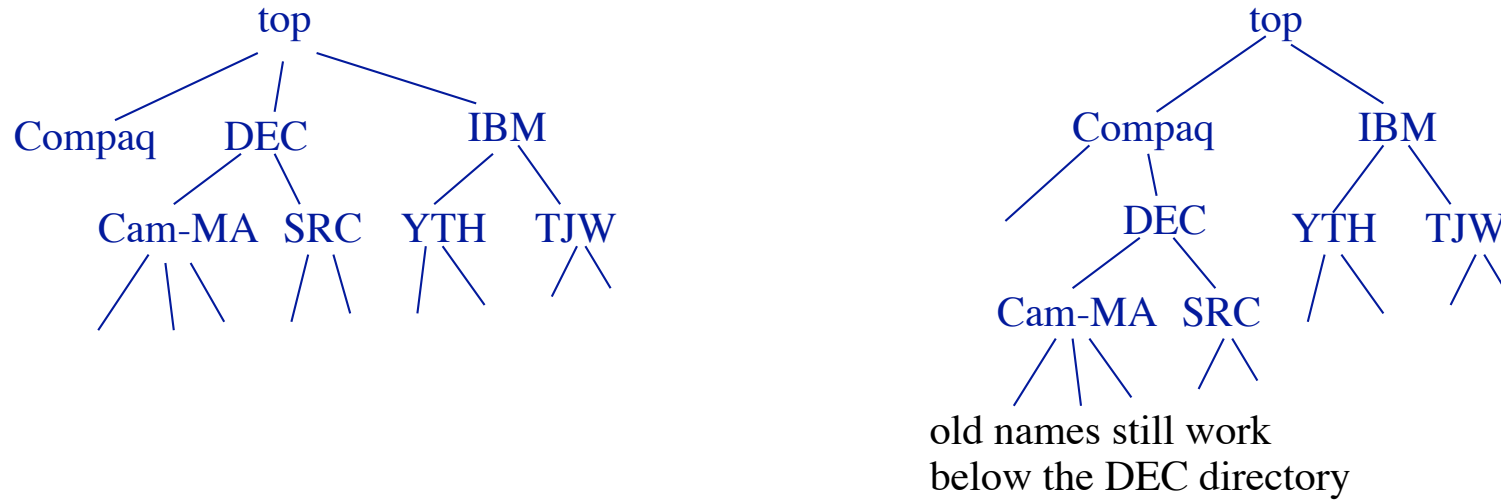
- hierarchical
- **every directory has a UID**, a directory identifier (DI)

A **full name** is any name starting with a DI

- * doesn't require a single root directory
- * doesn't rely on the availability of some root directory

GNS continued

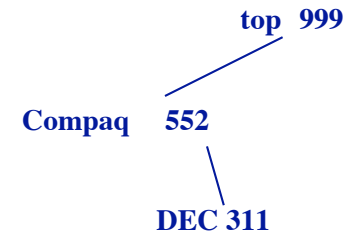
If the directory hierarchy is reconfigured a directory may still be found via its DI
 Names starting with that DI will not change, if the reconfiguration is above that DI



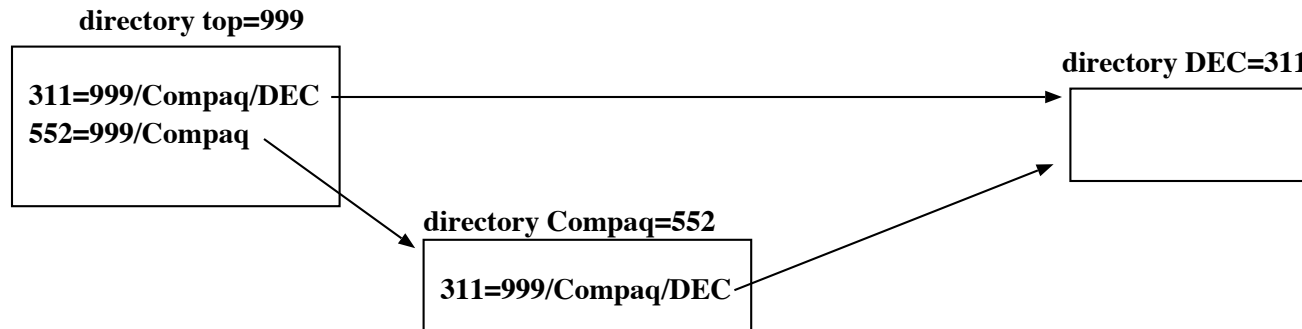
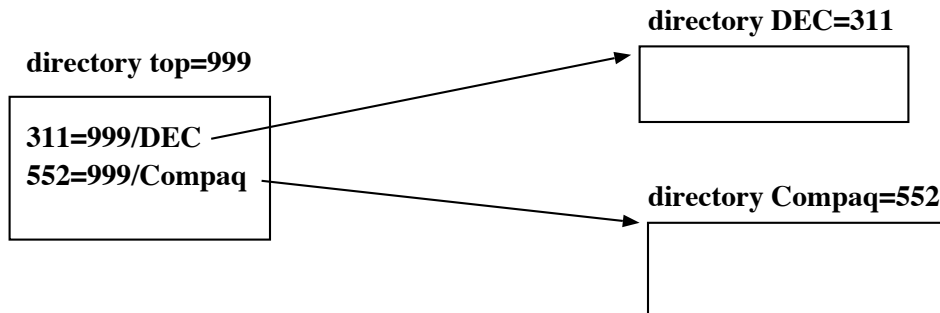
Support is needed to locate a directory from its DI (*a pure name - where do we look it up?*)
 as well as the usual location of directories by pathname lookup.
 Top level directories provide DIs with directory names.



names starting from DEC: **311/SRC, birrell**
 name of DEC directory: **is always 311**
 names starting above DEC
 was: **999/DEC, name**
 now: **999/Compaq/DEC, name**
 was: **999/DEC/SRC, birrell**
 now: **999/Compaq/DEC/SRC, birrell**



directory entries
 - include DIs with directory names
 - include pathnames from root



X.500 Directory Service (White and Yellow pages)

ISO and CCITT standard, above OSI protocol stack.

More general than a name service where names must be known precisely and are resolved to locations.

components:

DIT directory information tree

DSA directory service agent

DUA directory user agent

DAP directory access protocol

resource-consuming and difficult to use

1993 major revision including replication, access control, schema management.

But X.500 was never accepted as a generic name service.

X.509 certificates for authentication and authorisation have been successful.

Lightweight Directory Access protocol (**LDAP**) Howes, Kille, Yeong, Robbins, 1993

IETF accepted. can download free and deploy - widely used

* access protocol built on TCP/IP

* heavy use of strings, instead of ASN.1 data-types

* simplification of server and client

* current status V3

* LDUP duplication and update protocol (but see internet draft draft-zeilenga-ldup-harmful-02.txt)

Naming - summary

Naming for the Internet - see DNS

Naming for companies, world-wide - see Grapevine, GNS

Standard name services - X.500 (CCITT, ISO), X.509 for authentication, LDAP (IETF)

Naming for the web - document names are based on Internet naming:

scheme://host-name:port/pathname at host

scheme = protocol: http, ftp, local file, ...

host name = web server's DNS address, default port 80

pathname in web server's world of file containing web page

e.g. <http://www.cl.cam.ac.uk/research/.....>

Also W3C have defined standards for web services (see Middleware slide 20)

(with message content expressed in XML)

SOAP - simple object access protocol

WSDL - web service description language

UDDI - universal description discovery and integration

(directory with web service description in WSDL)

Name Services in middleware

Object-oriented middleware (ref I-9, I-15, M-8, M-14)

remotely accessible objects are registered with local ORB
a remote object reference is returned which may be entered in a name service
together with an associated name

e.g. CORBA Naming Service

name -> remote object reference

CORBA Trading Service

attributes -> name, remote object reference

JAVA Naming and Directory Interface (JNDI) for services

naming interface: service interface publication

service-name -> remote object reference

directory interface: attribute -> remote object reference

Message-oriented middleware (ref M-16 to 20)

MOM evolved from the packet switching paradigm
 naming and routing may be defined statically
 e.g. IBM MQSeries queue names are assumed known to the application
 and embody fixed routing from client to server
 e.g. JMS (Java Messaging Service) can use JNDI

Event-based middleware (ref M-23 to 26)

names are topics or event types, used by (advertisers) publishers and subscribers

topics may be assumed to be known (TIBCO)
 or may be advertised (research systems: Siena, Hermes, ...)

message routing tables, for publications,
 are set up from (advertisements and) subscriptions

subscription can be either topic/type or attribute/content-value based

e.g. topic hierarchy: `stocks . stock-exchange-name . stock-type . stock-subtype`
 subscription: `stocks . * . utilities . *`

e.g. attribute/content:

subscription: `stocks, stock-exchange-name=NY, stock-type=mining, value>$100, ...`

e.g. for message type: `seen (person, room)`

subscription: `seen (person = *, room = FN34)`

may be integrated with a programming language

Access Control

Motivating example: a national Electronic Health Record (EHR) service. Police and Social Services are similar

- MUST protect **EHRs** from journalists, insurance companies, family members etc.
- access policy defined both nationally and locally
- generic scalable policy => **RBAC**
- **exception of individuals** is allowed by law, (all doctors except my uncle Fred Smith)
“Patients’ Charter” => **parametrised roles**
- may need to express **relationships** between parameters
treating-doctor (doctor-id, patient-id)

Access Control: Requirements / Motivation

- large scale
 - => role based access control (RBAC)**
- potentially widely distributed systems
- heterogeneous components, developed independently but must interoperate
 - => service-level policy agreements (SLAs)**
(which roles authorise their activators to use which services?) negotiated within and between domains
- incremental deployment

OASIS RBAC

- OASIS services name their clients in terms of **roles**
- OASIS services specify **policy** in terms of **roles**
 - for **role entry** (activation)
 - for **service invocation** (authorisation, access control)both in Horn clause form

OASIS model of role activation

a role activation rule is of the form:

condition1, condition2, |- target role

where the conditions can be

- prerequisite role
- appointment credential
- environmental constraint

all are parametrised

OASIS role (continued) **membership** rules

as we have seen, a role activation rule:

cond1*, **cond2**, **cond3***, |- **target role**

role membership rule:

the role activation conditions that must **remain true**, e.g.*
for the principal to remain active in the role

monitored using **event-based middleware**

another contributor to an **active security environment**

OASIS model of authorisation

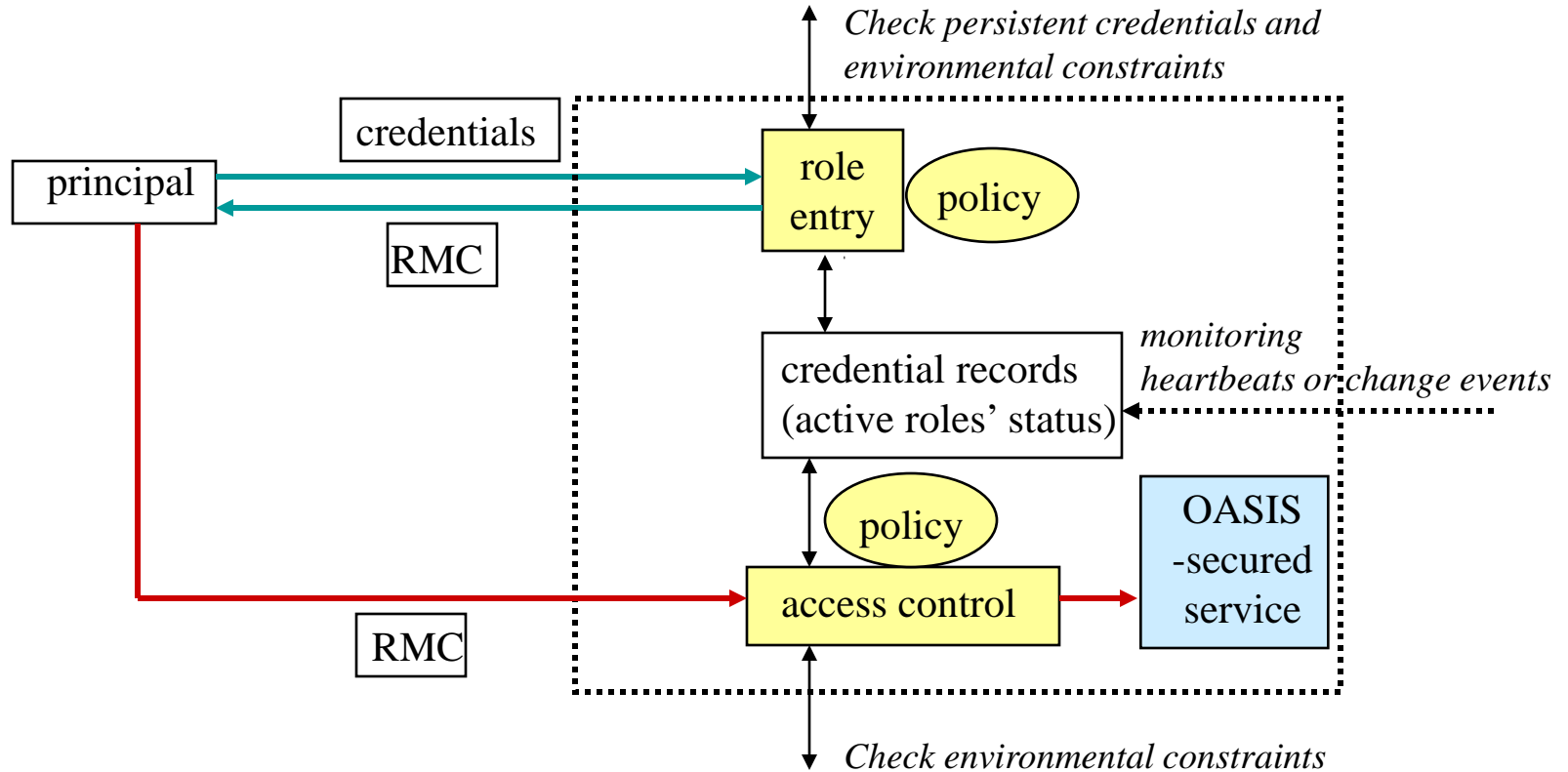
An authorisation rule is of the form:

condition1, condition2, |- access

where the conditions can be

- an active role
 - an environmental constraint
- all are parametrised

A Service Secured by OASIS Access Control



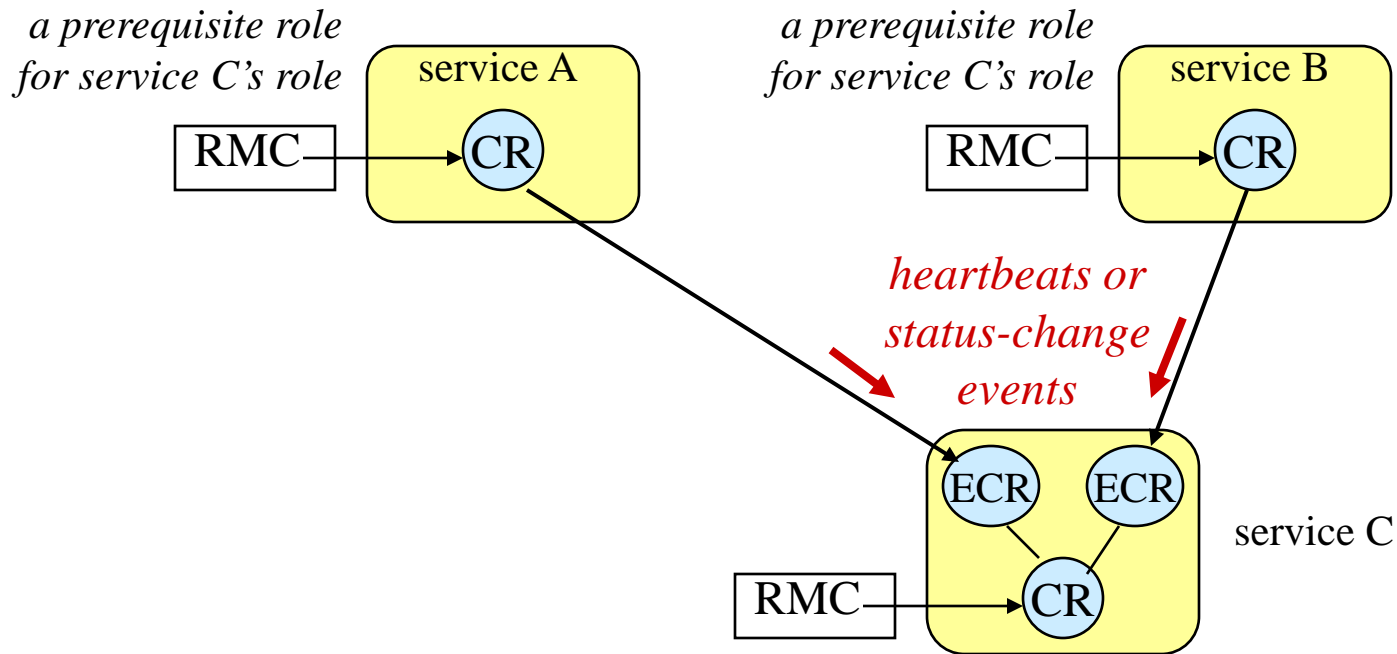
RMC = role membership certificate

→ = role entry

→ = use of service

Active Security Environment

Monitoring membership rules of active roles



RMC = role membership certificate

CR = credential record

ECR = external credential record

Roles and RBAC

- naming of **roles** for scalability, manageability, policy specification *e.g. doctor, sergeant*
- separate administration of people in roles
- **parametrised roles** for expressiveness: exclusions and relationships *e.g. treating-doctor(doctor-ID,patient-ID)*
- **RBAC** for access control policy for services and service-managed objects, (including the communication service)

“all doctors except”

“only the doctor with whom the patient is registered for treatment may prescribe drugs, read the patients’ EHR, ...”

Event-driven communication paradigm

- asynchronous message-passing rather than request-reply
- **advertise**, **subscribe**, **publish/notify** for scalability
e.g. subscribe to and be notified of:
bus-seen-event (busID=uni4., location=*)*
- event-driven paradigm for ubiquitous computing:
sensors generate data, notified as events
- compose/correlate events for higher level semantics
e.g. *traffic congestion, pollution and traffic*
- database integration – how best to achieve it?

Event-Driven Systems (1)

Cambridge Event Architecture (**CEA**), 1992 -

- extension of O-O **middleware**, typed events
 - “**advertise**, **subscribe**, **publish/notify**”, direct or mediated, publishers (or mediators if >1 publisher for a type)
process subscription filters and multicast to relevant subscribers
- federated event systems:
 - gateways/contracts/XML
- applications:
 - multimedia presentation control
 - pervasive environments (active house, active city, active office)
 - tracking mobile entities (active badge technology)
 - telecommunications monitoring and control

Event-Driven Systems (2)

Hermes large-scale event service, 2001-4

work of Peter Pietzuch

- loosely-coupled
- publish/subscribe
- widely distributed event-broker network
- via a P2P overlay network (DHT)
- distributed filtering (optimise use of comms.)
- rendezvous nodes for advertisers/subscribers

Use of P2P/DHT substrate

- Broker IP addresses hashed into 128-bit space
- Event topics hashed into 128-bit space
- Brokers keep tables of nearest neighbours (for different common prefixes) in 128-bit space – see next slide
- Event messages routed to broker nearest to event topic's hash value in $O(\log N)$ hops – called the “rendezvous node” for that topic
- Paths to same destination converge quickly
- Can exploit proximity (latency, bandwidth)
- Resilient to join/leave/failure of nodes
- Scales to millions of nodes

Pastry e.g. node 2030xx...’s routing table starts:

0* Id,a	1* Id,a	2*	3* Id,a
20*	21* Id,a	22* Id,a	23* Id,a
200* Id,a	201* Id,a	202* Id,a	203*
2030*	2031* Id,a	2032* Id,a	2033* Id,a
etc.			

e.g. route to **1**xxxx...




e.g. route to **22**xxx...

e.g. route to **200**xxx...

e.g. route to **2032**xx...

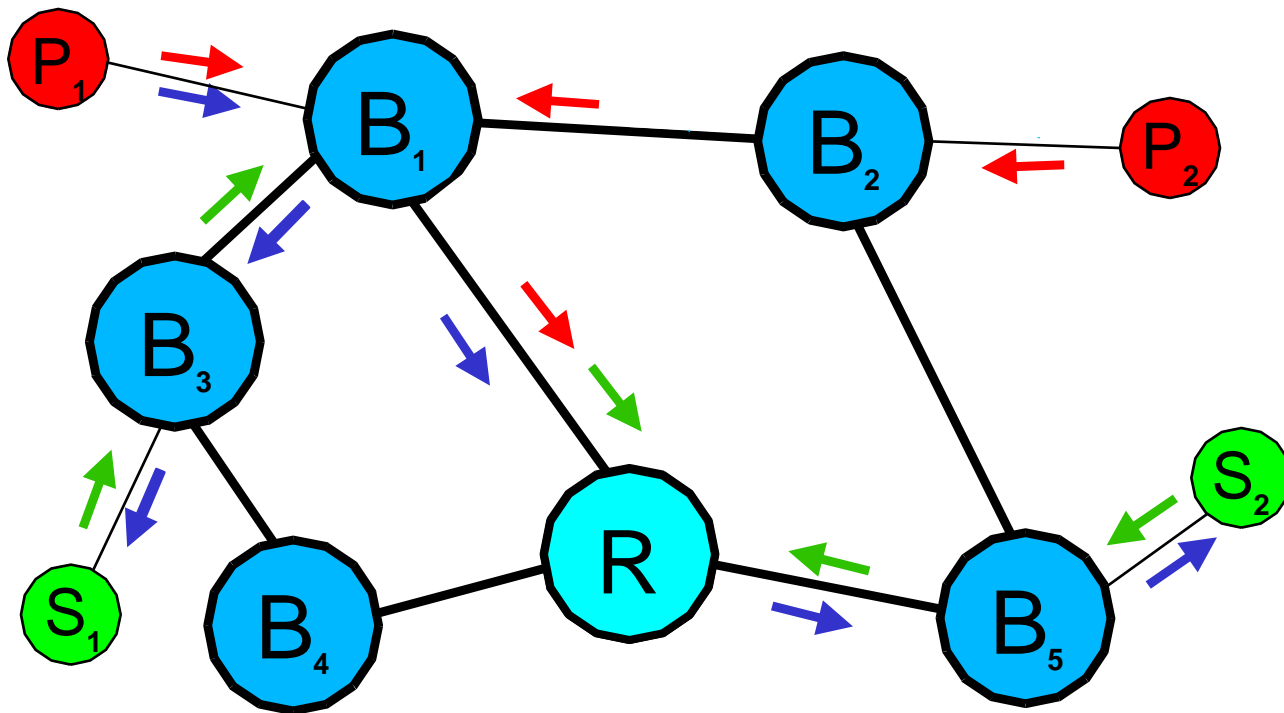
- nodeIds and keys are in some base 2^b (e.g. 4)
- each entry, except those for itself, contains the Id and IP address a of another node

Hermes Pub/Sub Design

- **Event Brokers** 
 - provide middleware functionality
 - logical overlay P2P network with content-based routing and filtering
 - easily extensible
- **Event Clients** (Event Publishers  Event Subscribers )
 - connect to any Event Broker
 - publishers **advertise**,
 - subscribers **subscribe** (brokers set up routing state),
 - publishers **publish**,
 - brokers route messages and **notify** publications to subscribers
 - lightweight, language-independent

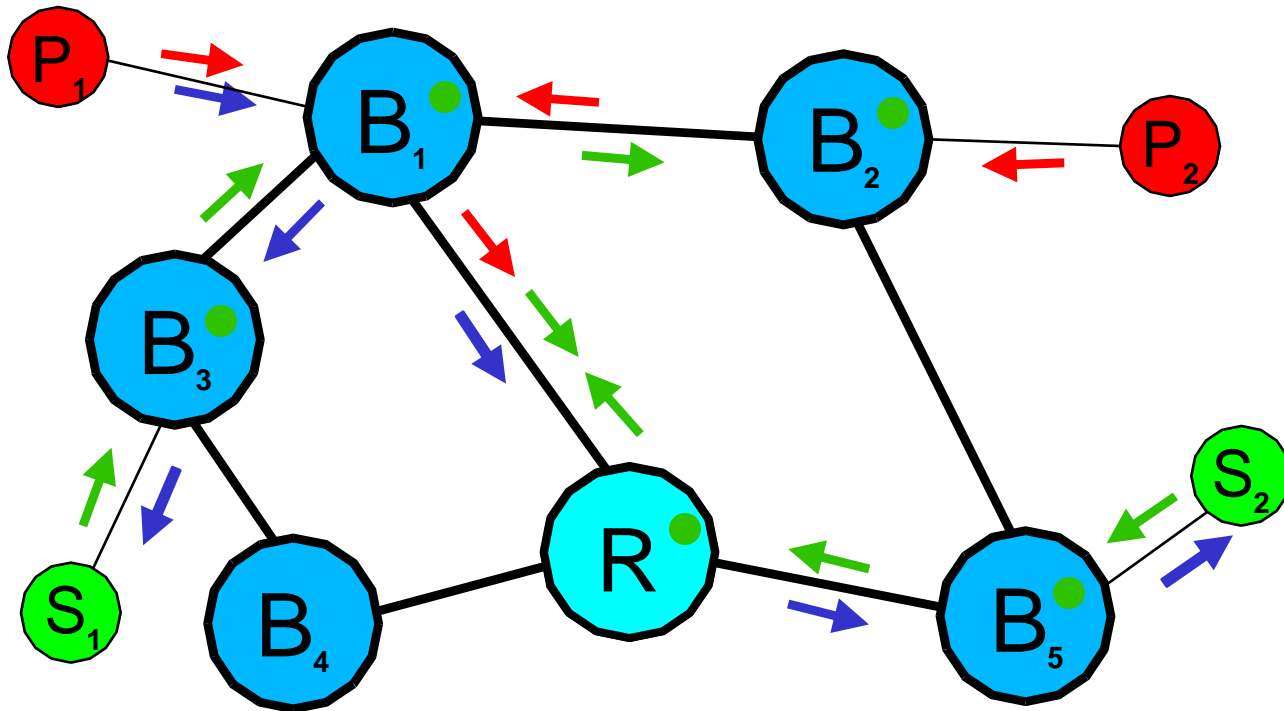
Algorithms I – Topic-Based Pub/Sub

- Type Msg, Advertisements, Subscriptions, Notifications
- Rendezvous Nodes
- Reverse Path Forwarding
 - Notifications follow Advs and then the reverse path of Subs



Algorithms II – Content-Based Pub/Sub

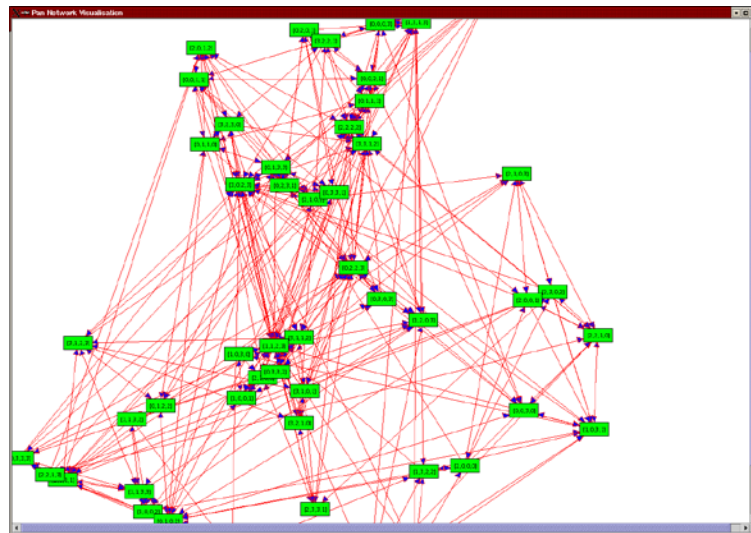
- Filtering State ●
- Notifications follow reverse paths of subscriptions
- Covering and Merging supported



Implementation

- **Actual implementation**

- Java implementation of event broker and event clients
- Event types defined in XML Schema
- Java language binding for events using reflection



- **Implementation within a simulator**

- Large-scale, Internet-like topologies
- Up to 10^4 nodes

But pub/sub is not sufficient for general applications

- decouples publishers and subscribers
 - pubs/subs need not be running at the same time
- publishers are anonymous to subscribers
 - subs need to know topic (attributes), not pubs' names and locations but receivers may **need** to know the sender or sender's role
- only multicast, one-to-many communication
 - may also need one-to-one and request-reply
- can't reply
 - either anonymously, e.g. to vote, or identified
- efficient notification for large-scale systems
 - but one-to-one should also be efficient – optimise

Event-Driven systems (3)

Event composition (correlation)

Pietzuch, Shand, Bacon, Middleware 2003,

IEEE Network, Jan/Feb 2004

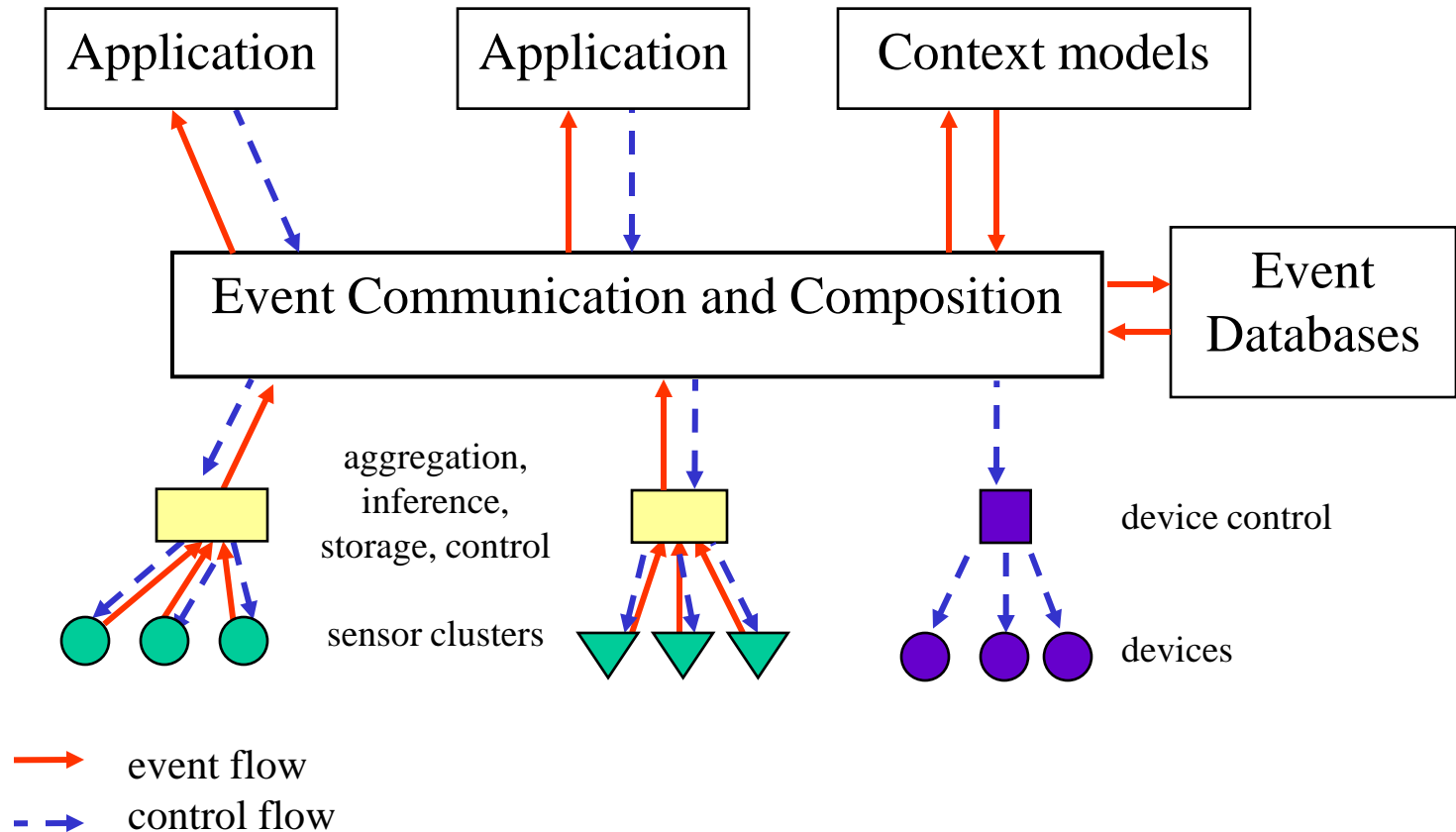
- composite event service above event brokers
- service instances placed to optimise communication
- FSM recognisers – parallel evaluation
- events have source-specific interval timestamps
- simulations of large-scale systems...

Bottom-up and/or Top-Down?

- Can we express all we require by bottom-up composition of primitive events?
- Do we also need **high-level models of context**?
e.g. maps, plans, mathematical models, GIS
- What can users be expected to express?
- How is the *top-down, bottom-up gap* bridged and high-level requirements converted into event subscriptions?
“nearest empty meeting room?”, *“turn off the lights if the room is empty”*, *“quickest way to get to Stansted airport?”*

Work-in-Progress

Integrating sensor networks (1)



Integrating sensor networks (2)

- *data*: sensor-ID, data value, timestamp, location
- value aggregation from densely deployed sensors
- inaccuracies masked – data cleansing
- heterogeneous sensor data correlated (fused)

Information/semantics:

- events defined, to present sensor data to applications including context models
- events correlated, higher-level events generated
- real-time delivery may be required
- level of data logging required?

Traffic monitoring applications

sensors: SCOOT loops for counting, cameras, thermal imaging (infra-red detectors), acoustic detectors

- I subscribe to *bus-seen-event* (*busID=uni4.**, *location=MadingleyP&R*) and my desktop is pinged when the bus is detected.

Traffic monitoring applications (cont'd)

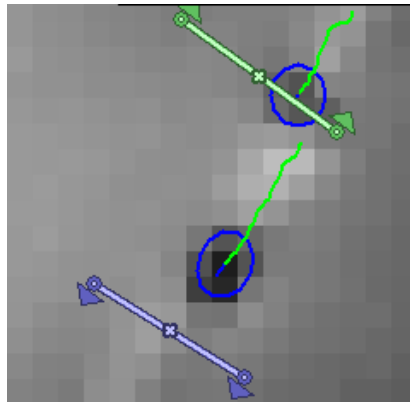
- Route-advice service: on entering my car I indicate my destination on a touch-sensitive map – route is shown, car monitored and route updated dynamically as conditions change.
- Easy to do with bespoke systems and/or coupling applications with sensors

How to allow the application developer builds the service by subscribing to advertised events, including high-level events such as *congestion (degree, location)*

Work-in-Progress: TIME-EACM grant

Irisys infra-red cameras + motion detection

- combined with video for validation (and banal tasks like aiming the thing (
- privacy-preserving
- Testing carried out on Dept. Eng. roof, Fen Causeway, 2006
- ~ 90% accuracy cf. video.
- wired communication via Engineering Dept.



Irisys – mirroring annual manual count

- carried out on Cambridge radial roads annually
- we did Huntingdon Rd, 9th Oct 2006, 8am – 7pm
- using one of DTG's sentient vans
- incoming and outgoing traffic
- validated against video
- over 90% accuracy cf. video if cycles excluded (and our *inexpert* positioning of the sensor)
- county haven't told us how their manual count compares with video



Irisys – ongoing monitoring

- mounted on a lamp post on Madingley Road
- connection to CL via Wifi



Stagecoach/ACIS bus monitoring

- GPS location of buses on some Stagecoach routes
- radio transfers data back to base (some GPRS, some custom)
- bus-stop displays (of timetables and expected arrival times)
- live and historical data from ACIS since Aug 2007, under a NDA
- this data allows **journey times** and **congestion** to be analysed and predicted

Healthcare monitoring application

sensors: **body sensors** for blood-pressure, blood-sugar, etc. **cameras** or thermal imaging (infra-red detectors) in smart homes, **tagging** objects

1. Emergency detection based on sensor values and image analysis – how to decide when to summon help?
2. Smart homes: monitoring for falls, visitors, ...
(guide-dogs - vs - people?) (visitors - vs - burglars?)
3. Tagging objects: “where did I leave ...?”, or to build a world model for navigation avoiding obstacles

Economic model? cost of technology-vs-more people? risks of false positives and false negatives?

Work-in-Progress: CareGrid grant

Integrating databases with pub/sub

- note: continuous queries require recording of individual queries and individual response, one-to-one.
- instead: databases advertise events:

event type (<attribute-type>)

based on virtual relations

- clients **subscribe** and are **notified** of occurrences
- the pub/sub service does the filtering – not the database
- we have used PostgreSQL - active predicate store

“cars-for-sale(maker, model, colour, automatic?,)”

many databases e.g. in Cambridge area

DB Motivating Example – Police IT

George Smiley is suspected of masterminding a nationwide terrorist organisation.

- As well as looking up his past database records, the investigators (special terrorism unit) **subscribe**, in all 43 counties, to **advertised** database update events specifying his name as an attribute.

Note inter-domain naming and access control.

- Triggers are set in the databases so that any future records that are made, relating to his movements and activities, will be **published** and **notified** automatically and immediately to *those authorised* to investigate him.

Securing pub/sub using RBAC

At the event client level – use RBAC

- domain-level authorisation policy indicates, for event types and attributes, the **roles** that can **advertise/publish** and **subscribe**
- inter-domain subscription is negotiated, as for any other service
- note that spamming is prevented – only authenticated roles can use the pub/sub service to advertise/publish

At the event-broker level – use encryption

- are all the event brokers **trusted**?
if not, some may not be allowed to see (decrypt) some (attributes of) some messages.
this affects content-based routing.

Consider various computing environments and scenarios

professional, academic, commercial, home - *based on traditional wired networks*

mobile users with computing devices: internet-connected
and/or using wireless/ad hoc networks - *new - wired and wireless networks*

pervasive/active environments - sensor networks' logs/databases - *new - wired and wireless*

Some scenarios (consider domain architecture, naming, location,
security (authentication, authorisation, communication))
ref: Introduction 21 - 30, Access Control A24 - 27

1. single domain behind firewall - local files served by network-based file service
+ accessing remote files and services
2. Open, internet-based file services: commercial services, cooperative P2P file sharing
3. e-science/GRID: storage for compute-service environments, database services
4. digital libraries, copyright, professional societies, publishers: **scientific archive**

*(high-level issues: persistence of data through technology change
persistence of scientific archive - who **guarantees** persistence?)*

Examples of requirements - 1

Traditional environments

- * program/document storage, development
application/system program load and run-time data access
- * application-level **services** (local and remote)
databases, CAD, email, naming directories,
photo editing, newsgroups, digital libraries

Different media types and file structure

- * integration of various media within a service, as opposed to dedicated servers e.g. VoD
continuous media, audio/video, work best with QoS guarantees
- * composite documents with components of different media types
linking related information across files (copying - vs - dangling references)
*issue: persistence of material linked to
structure helps cooperative work and synchronisation of updates*

Should a storage service provide support for structure representation, indexing and retrieval ?

- * vast amount of material is accumulated:
collections of images,
support for memory loss patients - "my day" images
audits of professional caring activities of NHS and SS
logs of sensor data - traffic, pollution, building projects such as tunnels

Examples of requirements - 2

(consider naming, location, security(authentication, authorisation, communication))

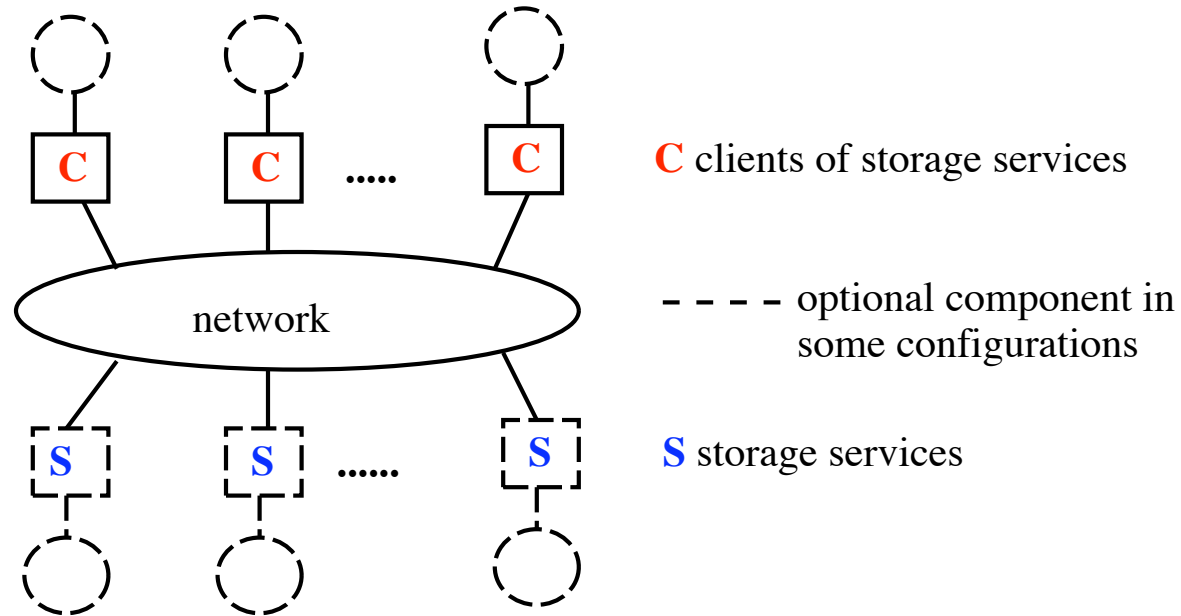
- * applications for download into home/other systems e.g. into thin clients

- * mobile users - access to files from remote locations
 - secure connection to home domain?
 - or use a commercial, internet-based file service?
(to place wanted files close to where they will be used)
 - support for detached operation: copy, disconnect, work on local copy of file, reconnect, **synchronise**

- * peer-to-peer (P2P)
 - using spare capacity across the Internet for file storage, backup/archive
 - issues of privacy, integrity, persistence, trust*
 - cooperative rather than commercial model (e.g. "sharing music with friends")
 - what scarce resource are you saving?

- * grid services
 - e.g. GRID-accessible petabytes of astronomical or genomic data
 - e.g. storage to support e-science computations
 - e.g. data shared by "virtual organisations" - controlled access, non-repudiation
 - e.g. data provenance
 - e.g. public data such as EHRs (security/trust is crucial)

Storage in a single-domain distributed system



clients have no local discs, system provides shared storage servers

early design - V system at Stanford
network computers

clients have local discs, no dedicated storage services

part of shared filing system - Unix mount

clients have local discs, system provides shared storage servers

use of clients discs:

for private system (local desktop separate from shared servers - Xerox, Windows?)

part of shared filing system - Unix mount

system files for bootstrapping

cached files: first-class copy is in shared service

temporary files - not backed up by sys-admin

* open or closed?

is it bound into a single OS file system model
e.g. single pathname format?

* functionality - how to distribute?

- storage and retrieval of data (whole files or parts)
- name resolution (directory service)
- access control
- existence control (garbage collection)
- concurrency control

* level of interface

- remote blocks (some early systems e.g. RVD remote virtual disc; SANs do it now)
client system may do block layout - minimal overhead at server
or server does block layout - interface in terms of blockID
- remote, UID-named files (interactions may involve whole files or parts)
server does block layout - more overhead at server
- remote path-named files (NAS)
bound into a single style of naming

* caching and replication

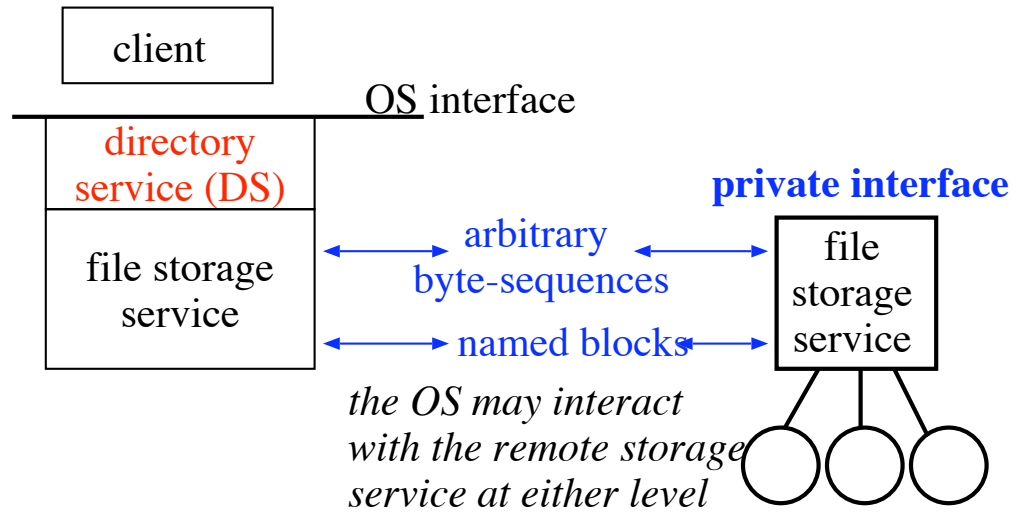
is the service responsible for managing, or assisting with:

- multiple cached copies of a file?
- replicas of a file (replicated on servers for reliability)

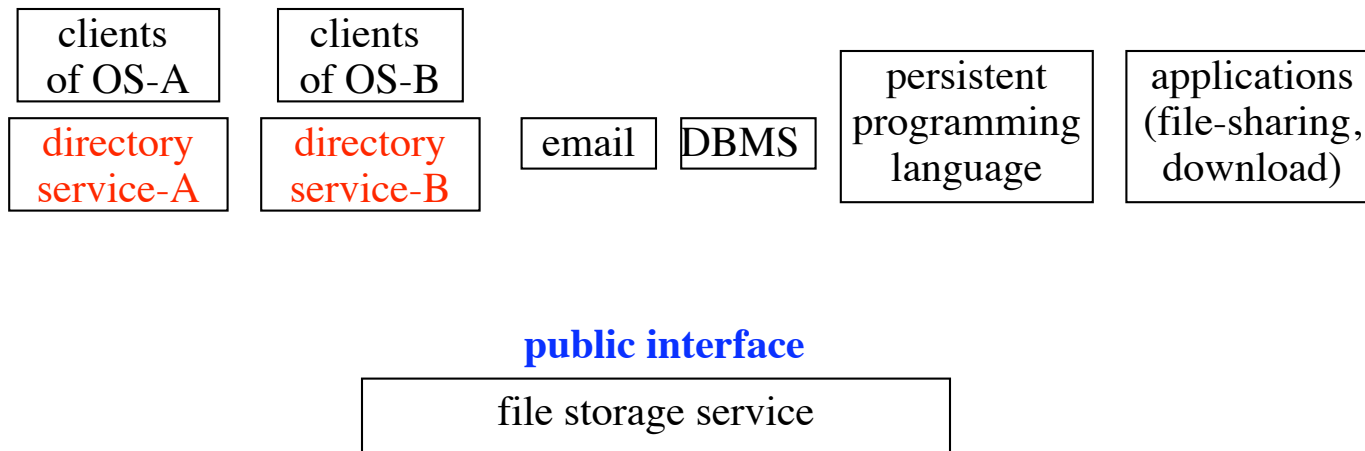
or are these application-level concerns?

Storage service architectures

a) closed storage architecture (single OS accesses SS)



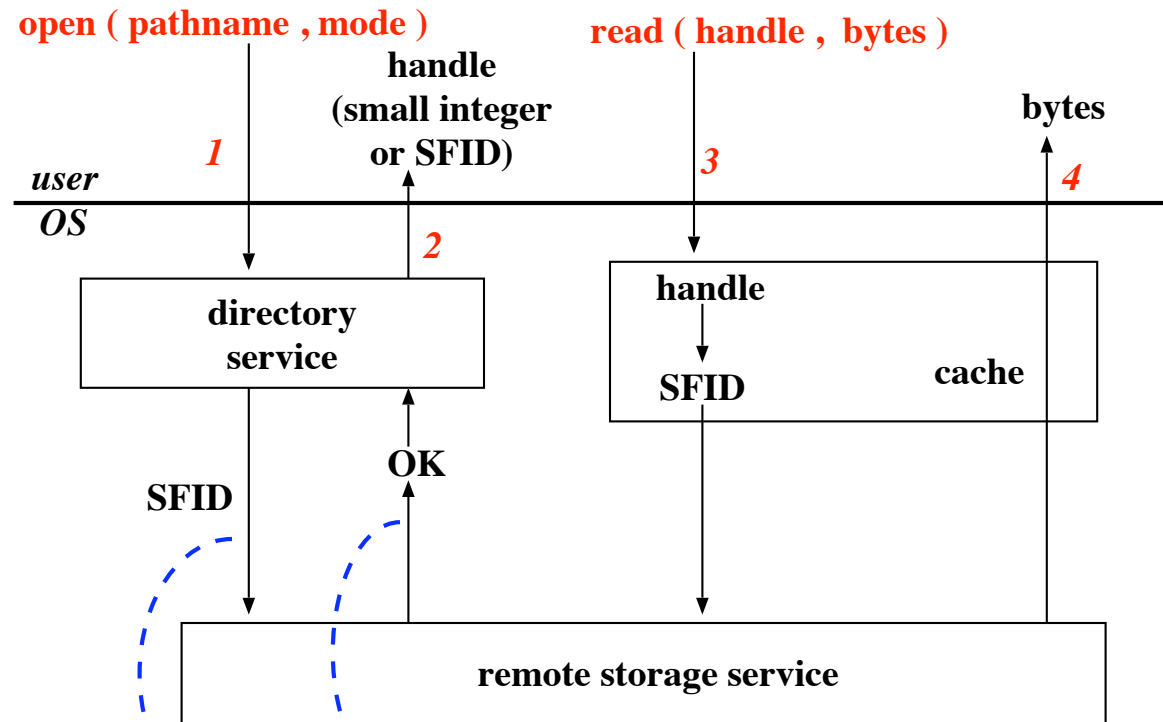
b) open storage architecture



remote interface at file storage level - example

assumes interactions can involve byte sequences rather than only whole-files

SFID = system file-identifier



*if the remote service is stateless
i.e. holds no info on files in use and
does not support an **open** operation
this interaction is just a number of **reads***

operations in remote storage service interface

SFID	←→	create	
		read (SFID, byte-range)	<i>assumes interaction at byte sequence level,</i>
		write (SFID, byte-range)	<i>as in S-7, rather than whole-file</i>
		delete (SFID)	?
		lock (SFID)	?
		open (SFID)	?
		close (SFID)	?

? depend on
design decisions

Does the server hold state?

- * NO specified as stateless e.g. NFS
 - simple crash recovery
 - can't help with concurrency control
 - can't help with cache management (clients have to ask for time-last-modified)

- * YES
 - supports open/close
 - holds who has files open and access mode
 - crash recovery - need to interact with clients to rebuild state
 - concurrency control
 - exclusive or shared locks better than write or read locks
 - cache management
 - can notify holders of copies when a new version is written

a file should stay in existence for as long as it is reachable from the root of the directory naming graph

- * storage service at file level can't help (doesn't see naming graphs)
- * a directory service (multiple instances?) can do existence control for its own objects, ref S-6 b)
OK for a closed architecture and for a single naming scheme within an open architecture
provided sharing is restricted to that scheme's files.
- * what about
 - objects shared by different systems? (e.g. video clip in document)
 - objects not stored in directories?
- * lost object problem **SFID** \longleftrightarrow **create (...)**
server allocates metadata in persistent store
either: - **server crash** -
or: reaches client's main memory only,
 - **client crash** -
on server or client restart, client repeats create (...)
- * Consider a "touch" operation provided by the storage service. All clients (i.e. services, not users) must touch all their files periodically. Untouched files are deleted (archived)

e.g. A Birrell and R Needham "A Universal File Server" IEEE Trans SE 6(5), pp 450-453, May 1980
Cambridge File Server: - open architecture - many OS clients
- minimal support for structure without enforcing path-naming
- some composite operations with transactional semantics

Developed as part of the **Cambridge Distributed Computing System (CDCS)** in the late 1970's.
CDCS was used as the Lab's research environment throughout the 1980's.

<http://www.research.microsoft.com/NeedhamBook/cmds.pdf>

CFS provides:

two primitive types: **byte** and **UID**

two abstractions:

file - an uninterpreted sequence of bytes

named persistently by a **PUID** with a random component

index - a sequence of PUIDs, itself named by a PUID

Indexes are used by CFS's clients to mirror their directory structures
all index operations are failure atomic (all or nothing is done)

*** existence control:**

indexes form a general naming network starting from a specific root index

objects are preserved while they are reachable from the root

- reference counts are used (the number of times a UID is included in an index)
- an asynchronous garbage collector is used for cyclic structures

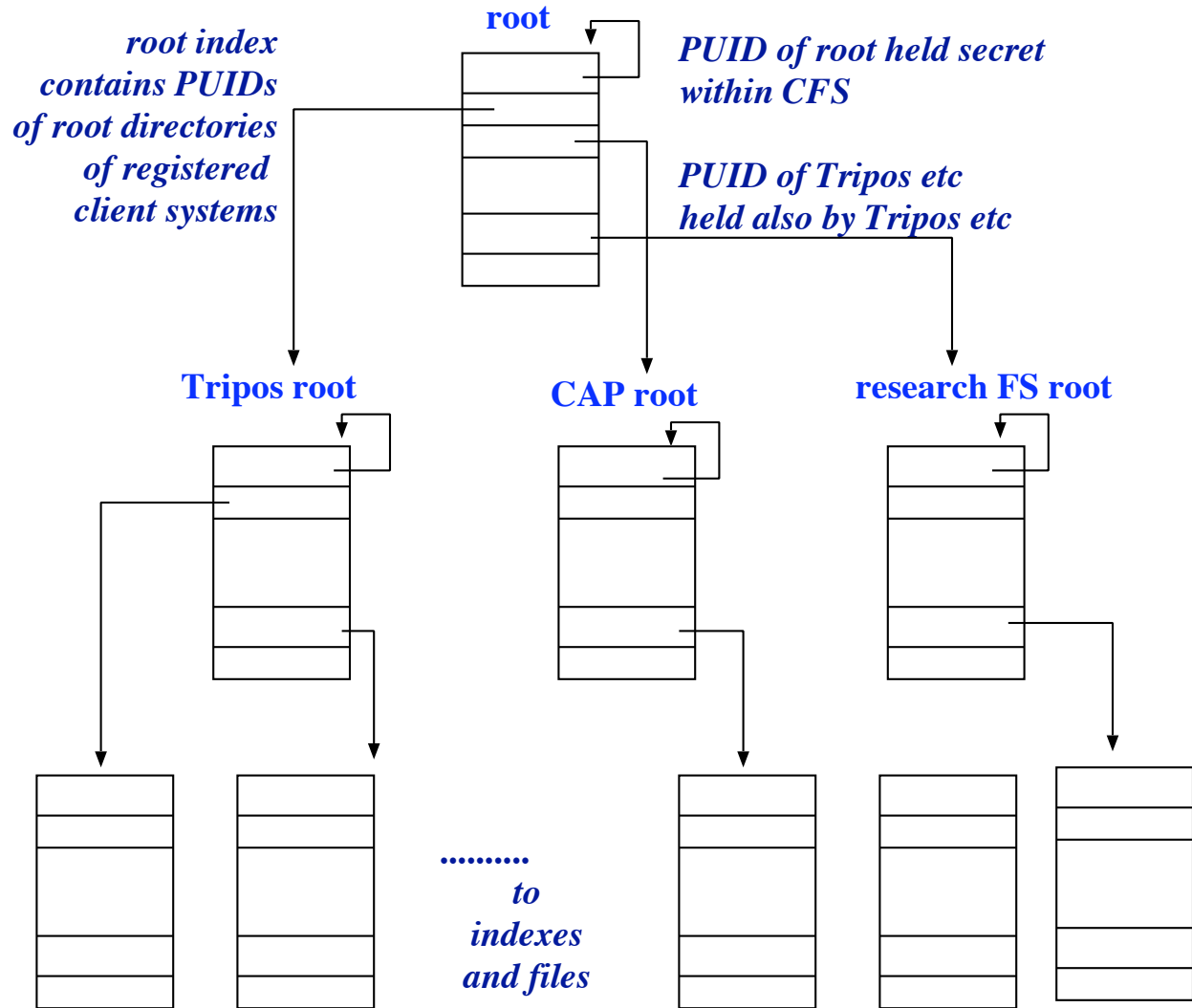
*** concurrency control - just MRSW**

open - a TUID is issued as a handle and the **PUID locked**

TUIDs are timed out (15 mins) reset on access

close - release PUID lock

CFS index structure



some CFS operations

object-ID = file or index ID

PUID = permanent/preserved ID for closed object

TUID = transient ID for open object

open [object-ID, {read/write}] -> TUID

close [TUID, {commit,abort}] -> commit/abort

index operations

create-index [existing index-ID, entry] -> index-ID *note transaction
no lost object problem*

preserve [index-ID, entry, object-ID] -> done

retrieve [index-id, entry] -> object-ID

delete [index-ID, entry] -> done

*NOTE: no **delete-index**, garbage collection instead*

file operations

create-file [index-ID, entry, ...] -> file-ID *note transaction
no lost object problem*

read [file-ID, offset, length] -> data

write [file-ID, offset, length, data] -> done

*NOTE: no **delete-file**, garbage collection instead*

Open, Structured Files, an approach

CFS indexes allow:

- different client operating systems' file services to use CFS
(their filenames and directory specifications can differ) - **openness**
- **existence control** (garbage collection) **across file systems**
via reachability from root of index structure

note: for scalability we would need multiple instances of CFS and distributed garbage collection not addressed in CDCS for a LAN-based file service

Can the CFS index approach be generalised to allow:

- embedded links within files, linking to different file systems
(e.g. to use different media types - video/audio clips)
- still have existence control, so be able to detect these embedded links

Idea: extend the storage type system to specify the storage structure sufficient to locate embedded links

base types: byte-sequence
SFID

generators: sequence
record
union
? other ? set, bag,

e.g. a [directory](#) might be:

a sequence of records with each record containing a byte-sequence and an SFID

e.g. a [thesis](#) might be a loose structure of sequences of variable-length byte-sequences and embedded references.

A typical requirement is to move sections of material around. The structure can be retained.

If the storage structure is stored as **metadata for each stored object** then

- SFIDs can be located for existence control
- objects can be kept in existence while any link to them remains

contrast with:

- typical network-based file systems model a file as a sequence of bytes identified by a SFID.
- relational databases typically specify records with fixed-length fields (and have a higher-level type system cf. programming languages)
- embedded URLs (which typically fail after a short time)
the entire Web cannot be searched for embedded URLs before documents are deleted or moved
the storage service level has no knowledge of structure
- DTDs for web documents - more general info
XML type system - more general info - higher level, not concerned with storage

The idea was explored in the project:

[Multi-Service Storage Architecture \(MSSA\)](#)

theses: Sue Thomson 1990 and Sai lai Lo (1994) Lab TR 326 and DD