

1 Characterization of distributed systems

1.1 Introduction

What is a Distributed System?

A **distributed system** is one in which components located at networked computers communicate and coordinate their actions only by passing messages

A **distributed system** consists of a **collection of autonomous computers linked by a computer network** and equipped with **distributed system software**. This software enables computers to **coordinate** their activities and to **share the resources of the system hardware, software, and data**.

How to characterize a distributed system?

- concurrency of components
- lack of global clock
- independent failures of components

Leslie Lamport :-)

You know you have a distributed system when the crash of a computer you've never heard of stops you from getting any work done!

Prime motivation: **to share resources**

What are the challenges?

- heterogeneity of their components
- openness
- security
- scalability – the ability to work well when the load or the number of users increases
- failure handling
- concurrency of components
- transparency
- providing quality of service

1.2 Examples of distributed systems

Distributed Systems application domains connected with networking:

12 Characterization of Distributed Systems 1.2 Examples of distributed systems

Finance and commerce	eCommerce e.g. Amazon and eBay, PayPal, online banking and trading
The information society	Web information and search engines, ebooks, Wikipedia; social networking: Facebook and MySpace
Creative industries and entertainment	online gaming, music and film in the home, user-generated content, e.g. YouTube, Flickr
Healthcare	health informatics, on online patient records, monitoring patients
Education	e-learning, virtual learning environments; distance learning
Transport and logistics	GPS in route finding systems, map services: Google Maps, Google Earth
Science	The Grid as an enabling technology for collaboration between scientists
Environmental management	sensor technology to monitor earthquakes, floods or tsunamis

1.2.1 Web search

An example: Google

Highlights of this infrastructure:

- physical infrastructure
- distributed file system
- structured distributed storage system
- lock service
- programming model

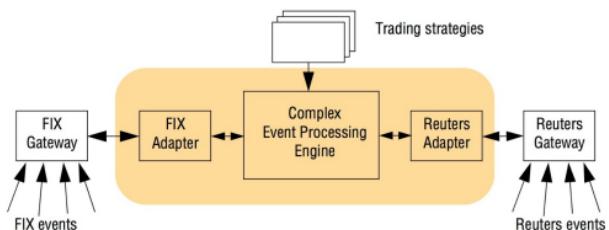
1.2.2 Massively multiplayer online games (MMOGs)

Examples

- EVE online – *client-server architecture!*
- EverQuest – more distributed architecture
- Research on completely decentralized approaches based on *peer-to-peer (P2P) technology*

1.2.3 Financial trading

- distributed even-based systems*



- **Reuters market data events**
- **FIX events** (events following the specific format of the Financial Information eXchange protocol)

CEP (Complex Event Processing): composing events into logical, temporal and spatial patterns.

```

WHEN
    MSFT price moves outside 2% of MSFT Moving Average
    FOLLOWED-BY (
        MyBasket moves up by 0.5%
        AND (
            HPQ-s price moves up by 5%
            OR
            MSFT-s price moves down by 2%
        )
    )
    ALL WITHIN
        any 2 minute time period
    THEN
        BUY MSFT
        SELL HPQ
  
```

(Apama - www.progress.com)

Algorithmic trading systems

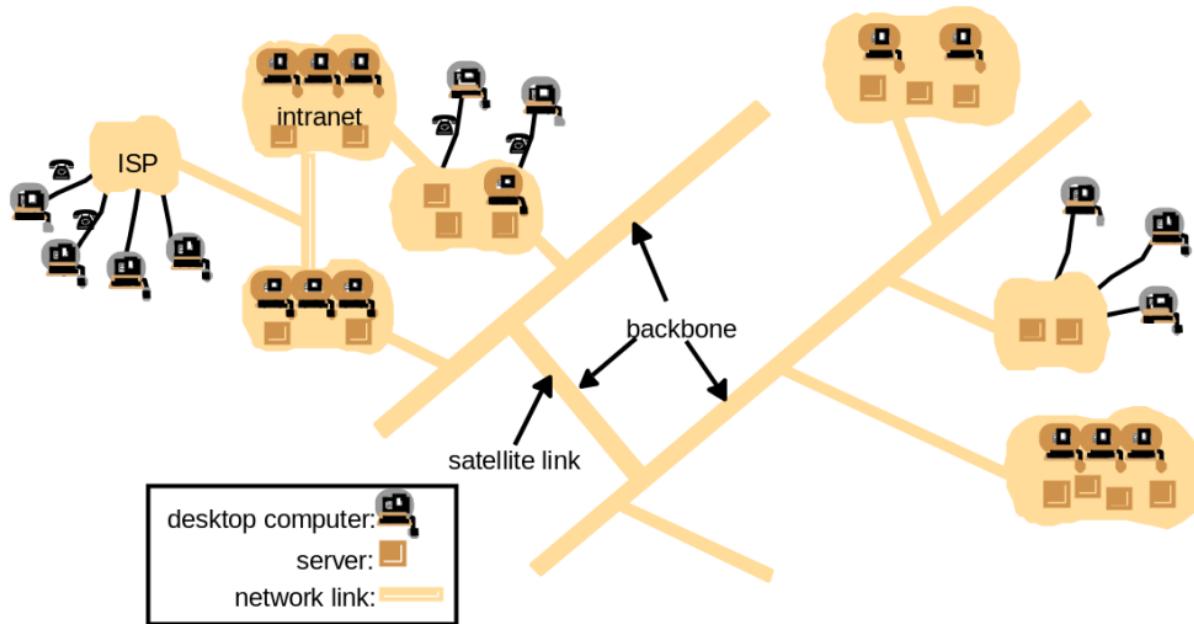
1.3 Trends in distributed systems

- emergence of pervasive networking technology
- emergence of ubiquitous computing coupled with the desire to support user mobility
- multimedia services
- distributed systems as utility

1.3.1 Pervasive networking and the modern Internet

networking has become a pervasive resource and devices can be connected at any time and any place

A typical portion of the Internet:

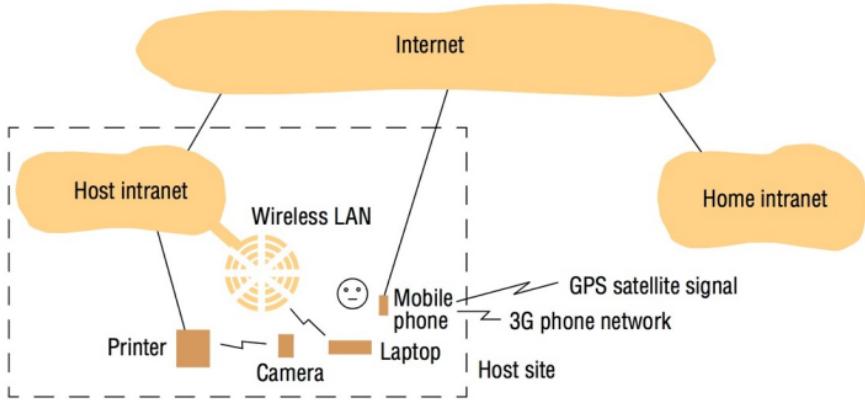


1.3.2 Mobile and ubiquitous computing

- laptop computers
- handheld devices (mobile phones, smart phones, tablets, GPS-enabled devices, PDAs, video and digital cameras)
- wearable devices (smart watches, glasses, etc.)
- devices embedded in appliances (washing machines, refrigerators, cars, etc.)

Portable and handheld devices in a distributed system

- mobile computing
- location/context-aware computing
- ubiquitous computing
- spontaneous interoperation
- service discovery

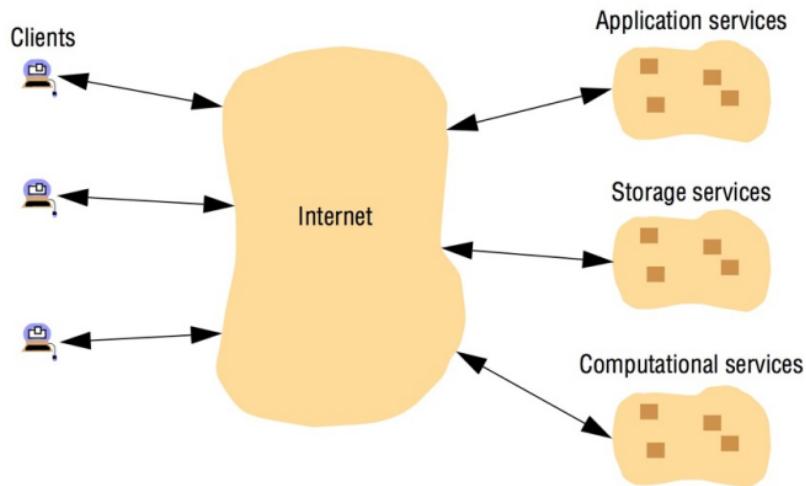


1.3.3 Distributed multimedia systems

- live or pre-ordered television broadcasts
- video-on-demand
- music libraries
- audio and video conferencing

1.3.4 Distributed computing as a utility

- Cluster computing
- Grid computing
- Cloud computing



1.4 Sharing resources

What are the resources?

- Hardware
 - Not every single resource is for sharing
- Data
 - Databases
 - Proprietary software
 - Software production
 - Collaboration

Sharing Resources

- Different resources are handled in different ways, there are however some generic requirements:
 - Namespace for identification
 - Name translation to network address
 - Synchronization of multiple access

1.5 Challenges

1.5.1 Heterogeneity

Heterogeneity – variety and difference in:

- networks
- computer hardware
- OS
- programming languages
- implementations by different developers

Middleware

- *middleware* – software layer providing:
 - programming abstraction
 - masking heterogeneity of:
 - * underlying networks
 - * hardware
 - * operating systems

Heterogeneity and mobile code

Mobile code – programming code that can be transferred from one computer to another and run at the destination (Example: think Java applets)

Virtual machine approach – way of making code executable on a variety of host computers – the compiler for a particular language generates code for a virtual machine instead of a particular hardware order code.

1.5.2 Openness

OPENNESS of a:

computer system - can the system be extended and reimplemented in various ways?

distributed system - can new resource-sharing services be added and made available for use by variety of client programs?

An open system – key interfaces need to be published!

An open distributed system has:

- uniform communication mechanism
- published interfaces to shared resources

Open DS - heterogeneous hardware and software, possibly from different vendors, but conformance of each component to published standard must be tested and verified for the system to work correctly

1.5.3 Security

1. ***Confidentiality*** – protection against disclosure to unauthorized individuals
2. ***Integrity*** – protection against alteration or corruption
3. ***Availability*** – protection against interference with the means to access the resources

Security challenges not yet fully met:

- *denial of service attacks*
- *security of mobile code*

1.5.4 Scalability

- the ability to work well when the system load or the number of users increases

Challenges with building scalable distributed systems:

- Controlling the cost of physical resources
- Controlling the performance loss
- Preventing software resources running out (like 32-bit internet addresses, which are being replaced by 128 bits)
- Avoiding performance bottlenecks
 - Example: some web-pages accessed very frequently – remedy: *caching* and *replication*

1.5.5 Failure handling

Techniques for dealing with failures

- Detecting failures
- Masking failures
 1. messages can be retransmitted
 2. disks can be replicated in a synchronous action
- Tolerating failures
- Recovery from failures

- Redundancy
 - redundant components
 1. at least two different routes
 2. like in DNS every name table replicated in at least two different servers
 3. database can be replicated in several servers

Main goal: **High availability** – measure of the proportion of time that it is available for use

1.5.6 Concurrency

Example: Several clients trying to access shared resource at the same time

Any object with shared resources in a DS must be responsible that it operates correctly in a concurrent environment

Discussed in Chapters 7 and 17 in the book

1.5.7 Transparency

Transparency – concealment from the user and the application programmer of the separation of components in a Distributed System for the system to be perceived as a whole rather than a collection of independent components

- **Access transparency** – access to local and remote resources identical
- **Location transparency** – resources accessed without knowing their physical or network location
- Concurrency transparency – concurrent operation of processes using shared resources without interference between them
- Replication transparency – multiple instances seem like one
- Failure transparency – fault concealment
- Mobility transparency – movement of resources/clients within a system without affecting the operation of users or programs

Access and Location transparency – together called also Network transparency

1.5.8 Quality of service

Main nonfunctional properties of systems that affect *Quality of Service (QoS)*:

- **reliability**
- **security**
- **performance**

Time-critical data transfers

Additional property to meet changing system configuration and resource availability:

- **adaptability**

1.6 Case study: The World Wide Web

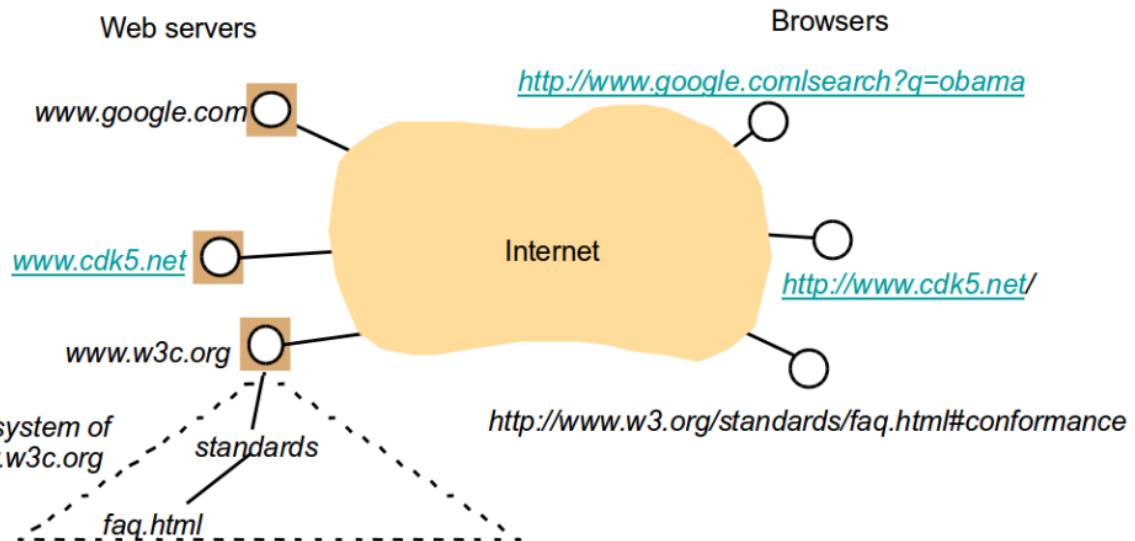
CERN 1989

hypertext structure, hyperlinks

- Web is an open system
- content standards freely published and widely implemented
- Web is open with respect to types

Figure 1.7 Web servers and web browsers

35 Characterization of Distributed Systems 1.6 Case study: The World Wide Web



HTML

HyperText Markup Language www.w3.org

URL-s

Uniform Resource Locators (also known as URI-s - Uniform Resource Identifiers)

`http://servername[:port][/pathName][?query][#fragment]`

HTTP

- Request-reply interactions
- Content types
- One resource per request
- Simple access control
- Dynamic pages

Web services

HTML – limited – not extensible to applications beyond information browsing

The Extensible Markup language (XML) designed to represent data in standard, structured, application-specific way

XML data can be transmitted by POST and GET operations

- Semantic web – web of linked metadata resources

Web as a system – main problem – the problem of scale

End of week 1

2 System models

2.1 Outline

What are the three basic ways to describe Distributed systems? –

- Physical models – consider DS in terms of hardware – computers and devices that constitute a system and their interconnectivity, without details of specific technologies
- Architectural models – describe a system in terms of the computational and communication tasks performed by its computational elements. Client-server and peer-to-peer most commonly used
- Fundamental models – take an abstract perspective in order to describe solutions to individual issues faced by most distributed systems
 - interaction models
 - failure models
 - security models

Difficulties and threats for distributed systems:

- Widely varying modes of use
- Wide range of system environments
- Internal problems
- External threats

2.2 Physical models

- Baseline physical model – minimal physical model of a distributed system as an extensible set of computer nodes interconnected by a computer network for the required passing of messages.

Three generations of distributed systems

- Early distributed systems
 - 10 and 100 nodes interconnected by a local area network
 - limited Internet connectivity
 - supported a small range of services e.g.
 - * shared local printers
 - * file servers
 - * email

- * file transfer across the Internet
- Internet-scale distributed systems
 - extensible set of nodes interconnected by a network of networks (the Internet)
- Contemporary DS with hundreds of thousands nodes + emergence of:
 - mobile computing
 - * laptops or smart phones may move from location to location – need for added capabilities (service discovery; support for spontaneous interoperation)
 - ubiquitous computing
 - * computers are embedded everywhere
 - cloud computing

- * pools of nodes that together provide a given service
- Distributed systems of systems (ultra-large-scale (ULS) distributed systems)

- significant challenges associated with contemporary DS:

Figure 2.1 Generations of distributed systems

<i>Distributed systems:</i>	<i>Early</i>	<i>Internet-scale</i>	<i>Contemporary</i>
<i>Scale</i>	Small	Large	Ultra-large
<i>Heterogeneity</i>	Limited (typically relatively homogenous configurations)	Significant in terms of platforms, languages and middleware	Added dimensions introduced including radically different styles of architecture
<i>Openness</i>	Not a priority	Significant priority with range of standards introduced	Major research challenge with existing standards not yet able to embrace complex systems
<i>Quality of service</i>	In its infancy	Significant priority with range of services introduced	Major research challenge with existing services not yet able to embrace complex systems

2.3 Architectural Models

Major concerns: make the system *reliable*, *manageable*, *adaptable* and *cost-effective*

2.3.1 Architectural elements

- What are the entities that are communicating in the distributed system?
- How do they communicate, or, more specifically, what communication paradigm is used?
- What (potentially changing) roles and responsibilities do they have in the overall architecture?
- How are they mapped on to the physical distributed infrastructure (what is their placement)?

Communicating entities

- From system perspective: **processes**
 - in some cases we can say that:
 - * **nodes** (sensors)
 - * **threads** (endpoints of communication)
- From programming perspective
 - *objects*
 - * computation consists of a number of interacting objects representing natural units of decomposition for the given problem domain
 - * Objects are accessed via interfaces, with an associated interface definition language (or IDL)

- ***components*** – emerged due to some weaknesses with distributed objects
 - * offer problem-oriented abstractions for building distributed systems
 - * accessed through interfaces
 - + assumptions to components/interfaces that must be present (i.e. making all dependencies explicit and providing a more complete contract for system construction.)
- ***web services***
 - * closely related to objects and components
 - * intrinsically integrated into the World Wide Web
 - using web standards to represent and discover services

The World Wide Web consortium (W3C):

Web service is a software application identified by a URI, whose interfaces and bindings are capable of being defined, described and discovered as XML artefacts. A Web service supports direct interactions with other software agents using XML-based message exchanges via Internet-based protocols.

- objects and components are often used within an organization to develop tightly coupled applications
- web services are generally viewed as complete services in their own right

Communication paradigms

What is:

- interprocess communication?
- remote invocation?
- indirect communication?

Interprocess communication – low-level support for communication between processes in distributed systems, including *message-passing* primitives, direct access to the API offered by Internet protocols (socket programming) and support for *multicast communication*

Remote invocation – calling of a remote operation, procedure or method

Request-reply protocols – a pattern with message-passing service to support client-server computing

Remote procedure call (RPC)

- procedures in processes on remote computers can be called as if they are procedures in the local address space
- supports client-server computing with servers offering a set of operations through a service interface and clients calling these operations directly as if they were available locally
 - RPC systems offer (at a minimum) access and location transparency

Remote method invocation (RMI)

- strongly resemble RPC but in a world of distributed objects
- tighter integration into object-orientation framework

In RPC and RMI –

- senders-receivers of messages
 - coexist at the same time
 - are aware of each other's identities

Indirect communication

- Senders do not need to know who they are sending to (*space uncoupling*)
- Senders and receivers do not need to exist at the same time (*time uncoupling*)

Key techniques in indirect communication:

- Group communication
- Publish-subscribe systems:

- (sometimes also called distributed event-based systems)
 - publishers distribute information items of interest (events) to a similarly large number of consumers (or subscribers)
- Message queues:
 - (publish-subscribe systems offer a one-to-many style of communication), message queues offer a point-to-point service
 - producer processes can send messages to a specified queue
 - consumer processes can
 - * receive messages from the queue or
 - * be notified
 - Tuple spaces (also known as generative communication):
 - processes can place arbitrary items of structured data, called tuples, in a persistent tuple space

- other processes can either read or remove such tuples from the tuple space by specifying patterns of interest
 - readers and writers do not need to exist at the same time (Since the tuple space is persistent)
- Distributed shared memory (DSM):
 - abstraction for sharing data between processes that do not share physical memory

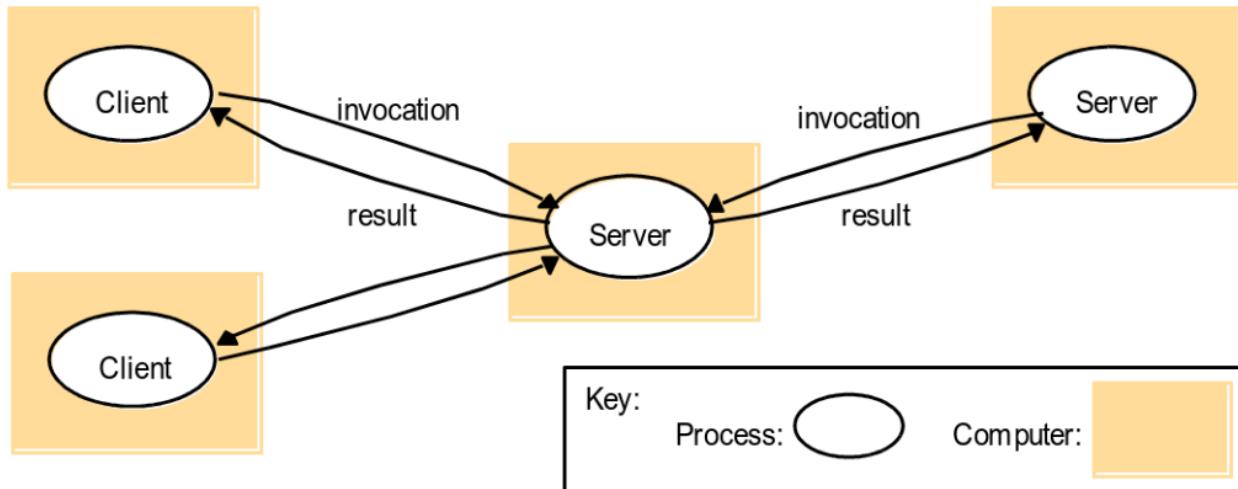
Figure 2.2 Communication entities and communication paradigms

<i>Communicating entities (what is communicating)</i>		<i>Communication paradigms (how they communicate)</i>		
<i>System-oriented entities</i>	<i>Problem-oriented entities</i>	<i>Interprocess communication</i>	<i>Remote invocation</i>	<i>Indirect communication</i>
Nodes	Objects	Message passing	Request-reply	Group communication
Processes	Components	Sockets	RPC	Publish-subscribe
	Web services	Multicast	RMI	Message queues
				Tuple spaces
				DSM

Roles and responsibilities

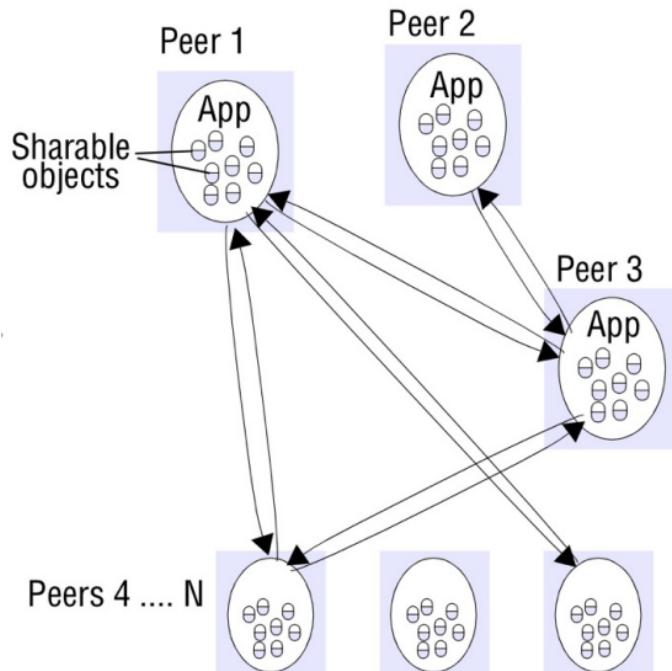
- Client-server

Figure 2.3 Clients invoke individual servers



- Peer-to-peer

Figure 2.4a Peer-to-peer architecture



- same set of interfaces to each other

Placement

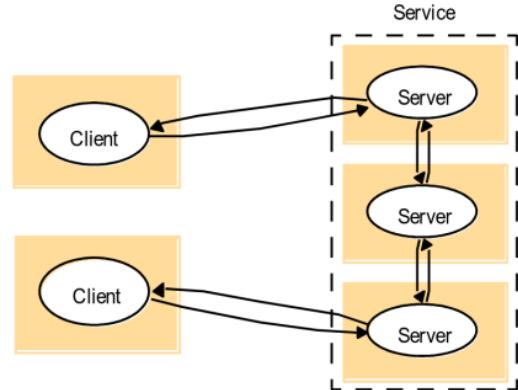
- crucial in terms of determining the DS properties:
 - performance
 - reliability
 - security

Possible placement strategies:

- mapping of services to multiple servers

- mapping distributed objects between servers, or
- replicating copies on several hosts
- more closely coupled multiple-servers – cluster

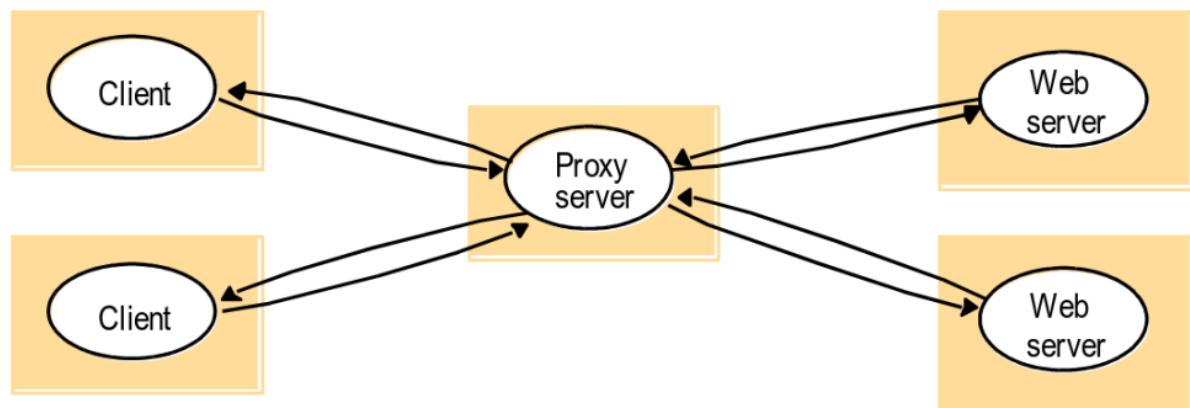
Figure 2.4b A service provided by multiple servers



- caching

- A cache is a store of recently used data objects that is closer to one client or a particular set of clients than the objects themselves

Figure 2.5 Web proxy server

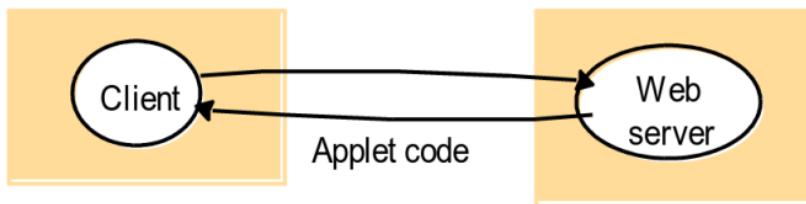


- mobile code

- Applets are an example of mobile code

Figure 2.6 Web Applets

- a) client request results in the downloading of applet code



- b) client interacts with the applet



- yet another possibility – *push* model: server initiates interaction (e.g. on information updates on it)

- mobile agents

- Mobile agent – running program (including both code and data) that travels from one computer to another in a network carrying out a task on someone's behalf (e.g. collecting information), and eventually returning with the results.
- could be used for
 - * software maintenance
 - * collecting information from different vendors' databases of prices

Possible security threats with mobile code and mobile agents...

2.3.2 Architectural patterns

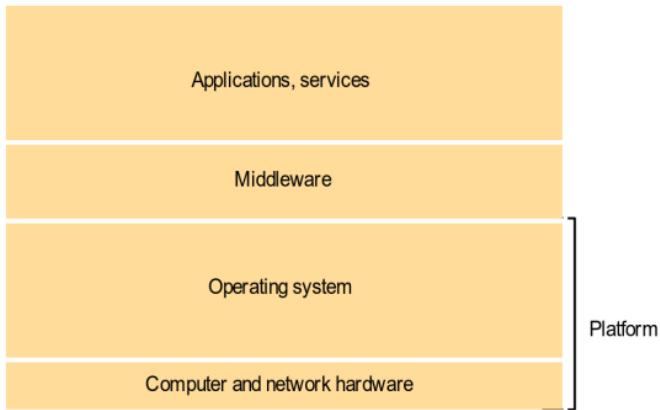
Layering

Layered approach – complex system partitioned into a number of layers:

- vertical organisation of services
- given layer making use of the services offered by the layer below
- software abstraction
- higher layers unaware of implementation details, or any other layers beneath them

Platform and Middleware

Figure 2.7 Software and hardware service layers in distributed systems



- A platform for distributed systems and applications consists of the lowest-level hardware and software layers.

- Middleware – a layer of software whose purpose is to mask heterogeneity and to provide a convenient programming model to application programmers.

Tiered architecture

Tiering is a technique to organize functionality of a given layer and place this functionality into appropriate servers and, as a secondary consideration, on to physical nodes

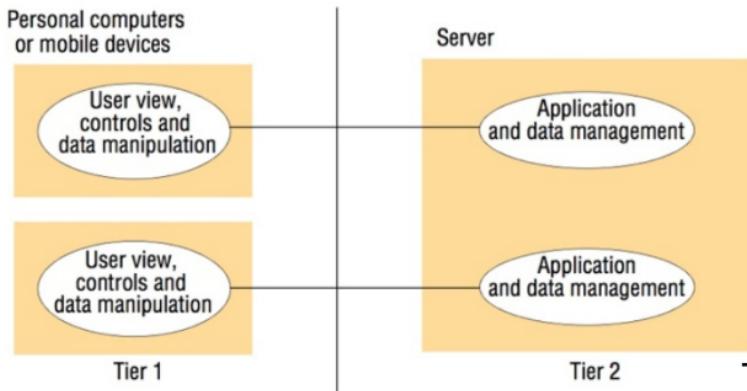
Example: two-tier and three-tier architecture

functional decomposition of a given application, as follows:

- presentation logic
- application logic
- data logic

Figure 2.8 Two-tier and three-tier architectures

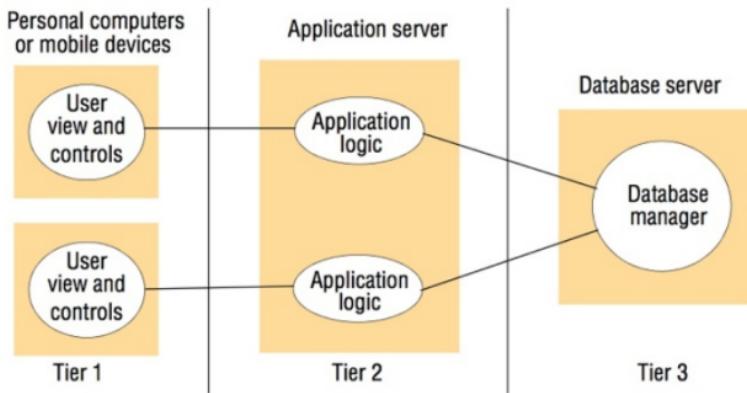
a)



- three aspects partitioned into two processes

- (+) low latency
- (-) splitting application logic

b)



- (+) one-to-one mapping from logical elements to physical servers

- (-) added complexity, network traffic and latency

AJAX (Asynchronous Javascript And XML) – a way to create interactive, partially/selectively-updatable webpages

- extension to the standard client-server style of interaction in WWW
 - Javascript frontend and server-based backend

Figure 2.9 AJAX example: soccer score updates

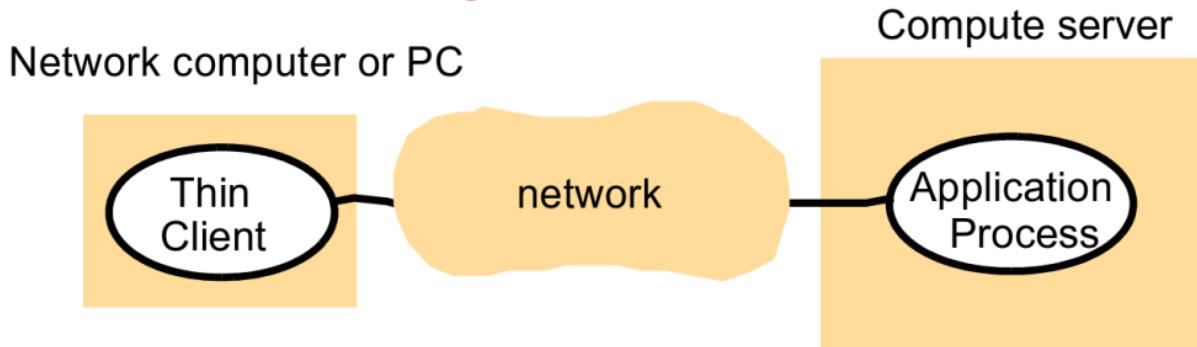
```
new Ajax.Request('scores.php?game=Arsenal:Liverpool',
{onSuccess: updateScore});
function updateScore(request) {
    ...
    ( request contains the state of the Ajax request including the returned result.
    The result is parsed to obtain some text giving the score, which is used
    to update the relevant portion of the current page.)
    ...
}
```

(two-tier architecture)

Thin clients

- enabling access to sophisticated networked services (e.g. cloud services) with few assumptions to client device
- software layer that supports a window-based user interface (local) for executing remote application programs or accessing services on remote computer

Figure 2.10 Thin clients and computer servers

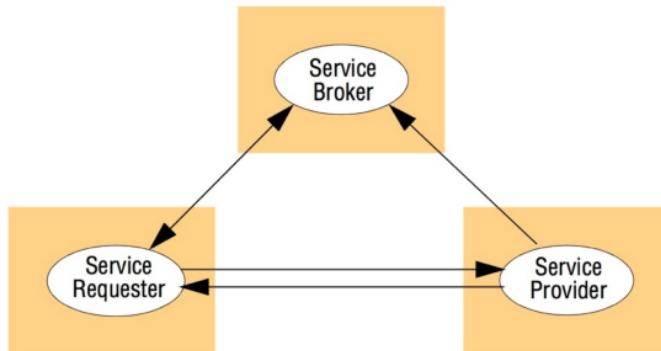


Concept led to Virtual Network Computing (VNC) – VNC clients accessing VNC servers using VNC protocol

Other commonly occurring patterns

- ***proxy pattern***
 - designed to support location transparency in RPC or RMI
 - proxy created in local address space, with same interface as the remote object
- ***brokerage in web services***
 - supporting interoperability in potentially complex distributed infrastructures
 - service provider, service requestor and service broker
 - brokerage reflected e.g. in registry in Java RMI and naming service in CORBA

Figure 2.11 The web service architectural pattern



- ***Reflection* pattern**
 - a means of supporting both:
 - * introspection (the dynamic discovery of properties of the system)
 - * intercession (the ability to dynamically modify structure or behaviour)
 - used e.g. in Java RMI for generic dispatching
 - ability to intercept incoming messages or invocations

- dynamically discover interface offered by a given object
- discover and adapt the underlying architecture of the system

2.3.3 Associated middleware solutions

The task of middleware is to provide a higher-level programming abstraction for the development of distributed systems and, through layering, to abstract over heterogeneity in the underlying infrastructure to promote interoperability and portability.

Categories of middleware

Figure 2.12 Categories of middleware

Major categories:	Subcategory	Example systems
<i>Distributed objects (Chapters 5, 8)</i>	Standard	RM-ODP
	Platform	CORBA
	Platform	Java RMI
<i>Distributed components (Chapter 8)</i>	Lightweight components	Fractal
	Lightweight components	OpenCOM
	Application servers	SUN EJB
	Application servers	CORBA Component Model
	Application servers	JBoss
<i>Publish-subscribe systems (Chapter 6)</i>	-	CORBA Event Service
	-	Scribe
	-	JMS
<i>Message queues (Chapter 6)</i>	-	Websphere MQ
	-	JMS
<i>Web services (Chapter 9)</i>	Web services	Apache Axis
	Grid services	The Globus Toolkit
<i>Peer-to-peer (Chapter 10)</i>	Routing overlays	Pastry
	Routing overlays	Tapestry
	Application-specific	Squirrel
	Application-specific	OceanStore
	Application-specific	Ivy
	Application-specific	Gnutella

Limitations of middleware

Some communication-related functions can be completely and reliably implemented only with the knowledge and help of the application standing at the end points of the communication system.

Example: e-mail transfer need another layer of fault-tolerance that even TCP cannot offer

2.4 Fundamental models

What is:

- Interaction model?
- Failure model?
- Security model?

2.4.1 Interaction model

- processes interact by passing messages –
 - communication (information flow) and
 - coordination (synchronization and ordering of activities) between processes

- communication takes place with delays of considerable duration
 - accuracy with which independent processes can be coordinated is limited by these delays
 - and by difficulty of maintaining the same notion of time across all the computers in a distributed system

Behaviour and state of DS can be described by a *distributed algorithm*:

- steps to be taken by each interacting process
- + transmission of messages between them

State belonging to each process is completely private

Performance of communication channels

- *latency* – delay between the start of message's transmission from one process and the beginning of receipt by another
- *bandwidth* of a computer network – the total amount of information that can be transmitted over it in a given time
- *Jitter* – the variation in the time taken to deliver a series of messages

Computer clocks and timing events

- *clock drift rate* – rate at which a computer clock deviates from a perfect reference clock

*Two variants of the interaction model**Synchronous distributed systems:*

- The time to execute each step of a process has known lower and upper bounds
- Each message transmitted over a channel is received within a known bounded time
- Each process has a local clock whose drift rate from real time has a known bound

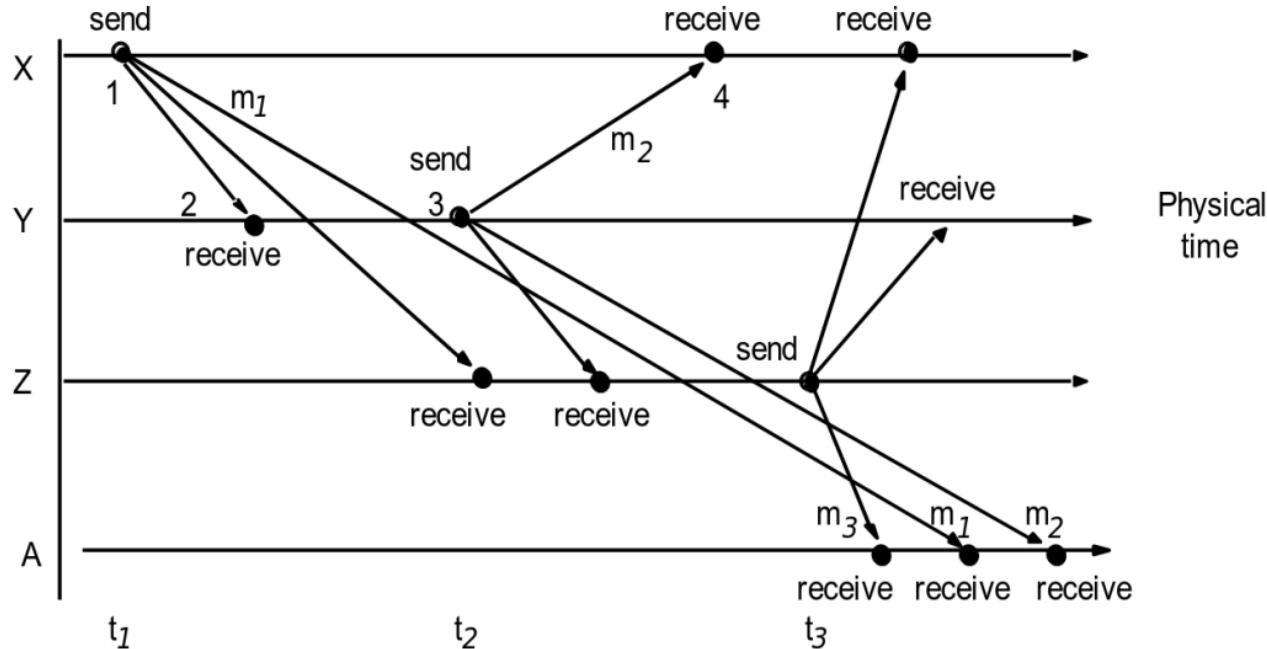
Asynchronous distributed systems:

No bounds on:

- Process execution speeds
- Message transmission delays
- Clock drift rates

Event ordering

Figure 2.13 Real-time ordering of events



- *Logical time* – based on event ordering

2.4.2 Failure model

- faults occur in:
 - any of the computers (including software faults)
 - or in the network
- Failure model defines and classifies the faults

Omission failures

- process or communication channel fails to perform actions it is supposed to do

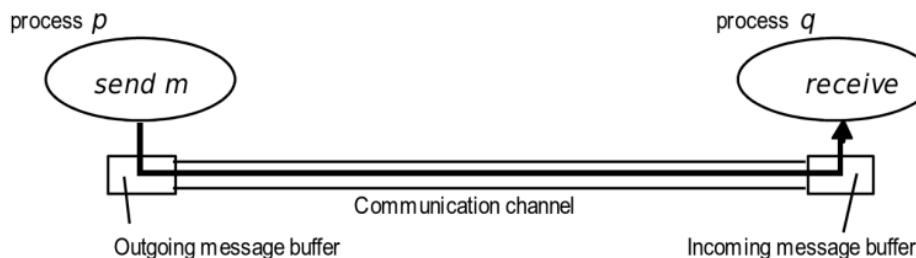
Process omission failures

- cheap omission failure of a process is to crash
 - crash is called *fail-stop* if other processes can detect certainly that the process has crashed

Communication omission failures

- communication channel does not transport a message from p 's outgoing message buffer to q 's incoming message buffer
 - known as dropping messages
 - * send-omission failures
 - * receive-omission failures
 - * channel-omission failures

Figure 2.14 Processes and channels



All failures so far: **benign failures**

Arbitrary failures

arbitrary or *Byzantine failure* is used to describe the worst possible failure semantics, in which any type of error may occur

Figure 2.15 Omission and arbitrary failures

<i>Class of failure</i>	<i>Affects</i>	<i>Description</i>
Fail-stop	Process	Process halts and remains halted. Other processes may detect this state.
Crash	Process	Process halts and remains halted. Other processes may not be able to detect this state.
Omission	Channel	A message inserted in an outgoing message buffer never arrives at the other end's incoming message buffer.
Send-omission	Process	A process completes a <i>send</i> , but the message is not put in its outgoing message buffer.
Receive-omission	Process	A message is put in a process's incoming message buffer, but that process does not receive it.
Arbitrary (Byzantine)	Process or channel	Process/channel exhibits arbitrary behaviour: it may send/transmit arbitrary messages at arbitrary times, commit omissions; a process may stop or take an incorrect step.

Timing failures

- applicable in synchronous distributed systems

Figure 2.16 Timing failures

<i>Class of Failure</i>	<i>Affects</i>	<i>Description</i>
Clock	Process	Process's local clock exceeds the bounds on its rate of drift from real time.
Performance	Process	Process exceeds the bounds on the interval between two steps.
Performance	Channel	A message's transmission takes longer than the stated bound.

Masking failures

- knowledge of the failure can enable a new service to be designed to mask the failure of the components on which it depends

Reliability of one-to-one communication

- reliable communication:
 - *Validity*: Any message in the outgoing message buffer is eventually delivered to the incoming message buffer
 - *Integrity*: The message received is identical to one sent, and no messages are delivered twice

2.4.3 Security model

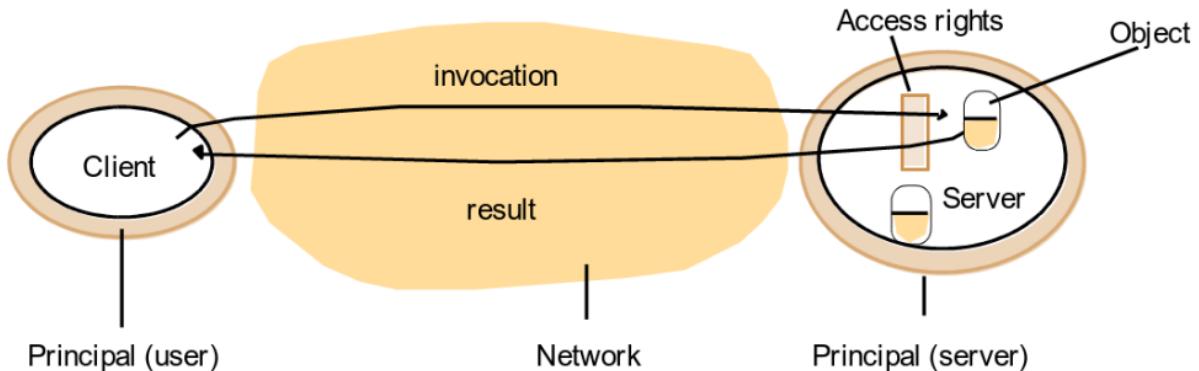
- modular nature of distributed systems and their openness exposes them to attack by
 - both external and internal agents
- Security model defines and classifies attack forms,
 - providing a basis for the analysis of threats
 - basis for design of systems that are able to resist them

the security of a distributed system can be achieved by securing the processes and the channels used for their interactions and by protecting the objects that they encapsulate against unauthorized access.

Protecting objects

- Users with access rights
- association of each invocation and each result with the authority on which it is issued
 - such an authority is called *a principal*
 - * principal may be a user or a process

Figure 2.17 Objects and principals



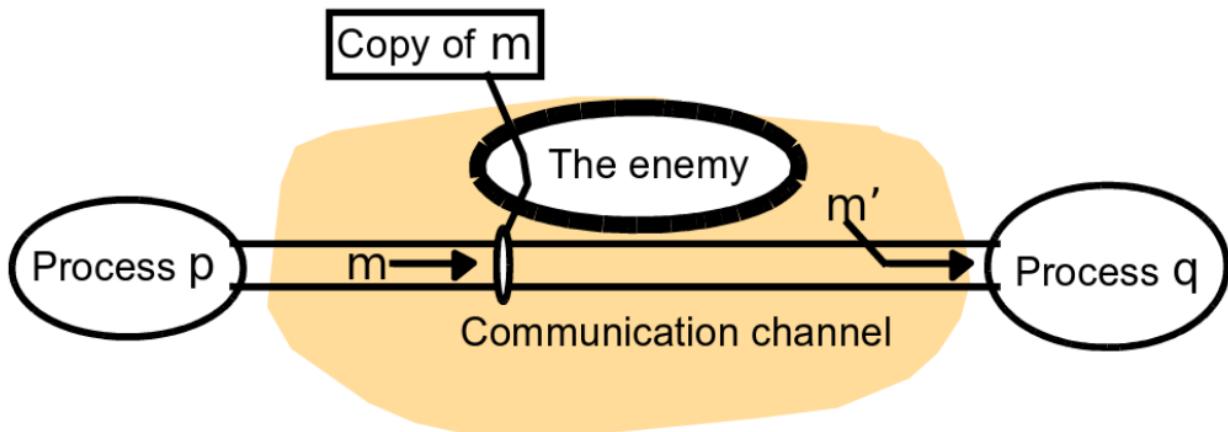
Securing processes and their interactions

- securing communications over open channels
- open service interfaces

The enemy

or also: *adversary*

Figure 2.18 The enemy



Threats to processes

- lack of knowledge of true source of a message
 - problem both to server and client side
 - example: spoofing a mail server

Threats to communication channels

- threat to the privacy and integrity of messages
- can be defeated using *secure channels*

Defeating security threats

Cryptography and shared secrets

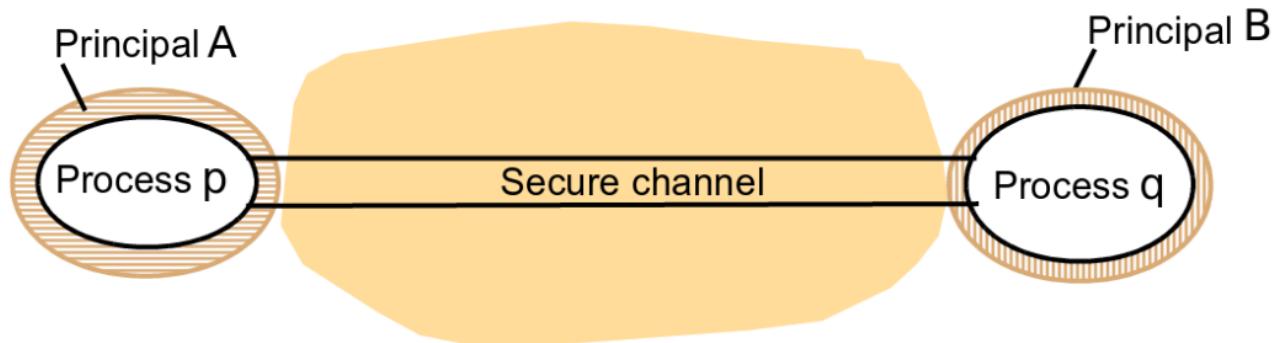
- Cryptography is the science of keeping messages secure
- Encryption is the process of scrambling a message in such a way as to hide its contents

Authentication

- based on shared secrets authentication of messages – proving the identities supplied by their senders

Secure channels

Figure 2.19 Secure channels



Properties of a secure channel:

- Each of the processes knows reliably the identity of the principal on whose behalf the other process is executing
- A secure channel ensures the privacy and integrity (protection against tampering) of the data transmitted across it

- Each message includes a physical or logical timestamp to prevent messages from being replayed or reordered

Other possible threats from an enemy

- Denial of service:
 - the enemy interferes with the activities of authorized users by making excessive and pointless invocations on services or message transmissions in a network, resulting in overloading of physical resources (network bandwidth, server processing capacity)
- Mobile code:
 - execution of program code from elsewhere, such as the email attachment etc.

The uses of security models

Security analysis involves

- the construction of a threat model:
 - listing all the forms of attack to which the system is exposed
 - an evaluation of the risks and consequences of each

End of week 2

3 Networking and internetworking

Distributed systems use local area networks, wide area networks and internetworks for communication

Changes in user requirements have resulted in the emergence of wireless networks and of high-performance networks with QoS guarantees

3.1 Introduction

<i>transmission media</i>	hardware devices	software components
<ul style="list-style-type: none">• wire• cable• fibre• wireless channels	<ul style="list-style-type: none">• routers• switches• bridges• hubs• repeaters	<ul style="list-style-type: none">• protocol stacks• communication handlers• drivers

communication subsystem – hardware and software components that provide the communication facilities for a distributed system

hosts – computers and other devices that use the network for communication purposes

node – any computer or switching device attached to a network.

Internet constructed from any subnets

subnet – unit of routing (delivering data from one part of the Internet to another); collection of nodes that can all be reached on the same physical network

History

1961 – theoretical basis for packet switching (Leonhard Kleinrock)

1962 – discussion on potential for interactive computing and wide area networking (J.C.R. Licklider and W. Clark)

1964 – outline of a practical design for reliable and effective wide area networks
(Paul Baran)

3.1.1 Networking issues for distributed systems

Performance

- Message transmission time = latency + length / data transfer rate
- longer messages segmented – transmission time is the sum of transferring the segments
- total system bandwidth of a network is a measure of throughput – the total volume of traffic that can be transferred across the network in a given time
- in most wide area networks messages can be transferred on several different channels simultaneously

- The performance of networks deteriorates in conditions of overload
- time required to access shared resources on a local network remains about a 1000x greater than that required to access resources that are resident in local memory

Scalability

- Difficult to estimate the real size nowadays
- The potential future size of the Internet
 - will be of order of the population of the planet
 - several billion nodes and hundreds of millions of active hosts

Reliability

- Many applications are able to recover from communication failures and hence do not require guaranteed error-free communication
- usually errors due to
 - software errors in sender or receiver
 - * (for example, failure by the receiving computer to accept a packet)
 - * buffer overflow
 - network errors (not that often though)

Security

- firewall technology
- chryptographic technology
- *virtual private network (VPN)* techniques

Mobility

The Internet's mechanisms have been adapted and extended to support mobility, but the expected future growth in the use of mobile devices will demand further development

Quality of service

- multimedia data transmission

Multicasting

- simultaneous transmission of messages to several recipients

3.2 Types of network

types of networks are confusing because they seem to refer to the physical extent (local area, wide area), but they also identify physical transmission technologies and low-level protocols

Figure 3.1 Network performance

	Example	Range	Bandwidth (Mbps)	Latency (ms)
<i>Wired:</i>				
LAN	Ethernet	1–2 kms	10–10,000	1–10
WAN	IP routing	worldwide	0.010–600	100–500
MAN	ATM	2–50 kms	1–600	10
Internetwork	Internet	worldwide	0.5–600	100–500
<i>Wireless:</i>				
WPAN	Bluetooth (IEEE 802.15.1)	10–30m	0.5–2	5–20
WLAN	WiFi (IEEE 802.11)	0.15–1.5 km	11–108	5–20
WMAN	WiMAX (IEEE 802.16)	5–50 km	1.5–20	5–20
WWAN	3G phone	cell: 1–5	348–14.4	100–500

- Personal area networks (PANs)

- Local area networks (LANs)

- segment – section of cable that serves a department or a floor of a building and may have many computers attached.
 - * No routing of messages is required within a segment, since the medium provides direct connections between all of the computers connected to it
 - * The total system bandwidth is shared between the computers connected to a segment
- **Ethernet** emerged as the dominant technology for wired local area networks. It was originally produced in the early 1970s with a bandwidth of 10 Mbps (million bits per second) and extended to 100 Mbps, 1000 Mbps (1 gigabit per second) and 10 Gbps versions more recently

- Ethernet

- * lacks the latency and bandwidth guarantees needed by many multimedia applications
- * high-speed Ethernets have been deployed in a switched mode that overcomes these drawbacks to a significant degree, though not as effectively as ATM

- **Wide area networks (WANs)**

- communication medium linking a set of dedicated computers – **routers**
- latencies can be as high as 0.1 ... 0.5 seconds
 - * Example: Europe-Australia
 - via terrestrial link 0.13 seconds
 - satellite 0.20 seconds

- **Metropolitan area networks (MANs)**

- based on the high-bandwidth copper and fibre optic cabling
- distances upto 50 km
- technology ranging from Ethernet to ATM (Asynchronous Transfer Mode)

- **Wireless local area networks (WLANs)**

- IEEE 802.11 standard (WiFi)

- **Wireless metropolitan area networks (WMANs)**

- IEEE 802.16 WiMAX standard

- **Wireless wide area networks (WWANs)**

- GSM (3G, 4G); UMTS (Universal Mobile Telecommunication System)
 - rates upto 100 Mbps

- **Internetworks**

- several networks linked together
 - routers, gateways

3.3 Network principles

Basis – packet switching technique (developed in 1960s)

- packets addressed to different destinations to share a single communications link
- Packets are queued in a buffer and transmitted when the link is available
- Communication is asynchronous – messages arrive with a delay depending upon properties and utilization of the network

3.3.1 Packet transmission

- messages – sequences of data items of arbitrary length
 - subdivided into packets
 - * packets have a restricted length

3.3.2 Data streaming

streaming – transmission and display of audio and video in real time

- video stream requires
 - 1.5 Mbps if data compressed
 - 120 Mbps if uncompressed
- channel from source to destination of a multimedia stream
 - predefined route
 - reserved set of resources
 - buffering where appropriate for smoothness
- ATM
- IPv6 includes features for real-time separate IP stream treatment

3.3.3 Switching schemes

Broadcast

- involves no switching
- some LAN technologies (including Ethernet) based on broadcasting
- wireless networking with nodes grouped in cells

Circuit switching

- plain old telephone system (or POTS) – typical switching network

Packet switching

- store-and-forward network

Frame relay

- switching small packets called frames on the fly
- switching nodes route frames based on the examination of their first few bits
- frames as a whole are not stored at nodes but pass through them as short streams of bits

3.3.4 Protocols

protocol – well-known set of rules and formats to be used for communication between processes in order to perform a given task

two important parts to it:

- a specification of the sequence of messages that must be exchanged
- a specification of the format of the data in the messages

The existence of well-known protocols enables the separate software components of distributed systems to be developed independently and implemented in different programming languages on computers that may have different order codes and data representations

- protocol is implemented by a pair of software modules located in the sending and receiving computers.

Protocol layers

Network software arranged in

- hierarchy of layers
- each layer presents an interface (service) to the layer(s) above it

Figure 3.2 Conceptual layering of protocol software

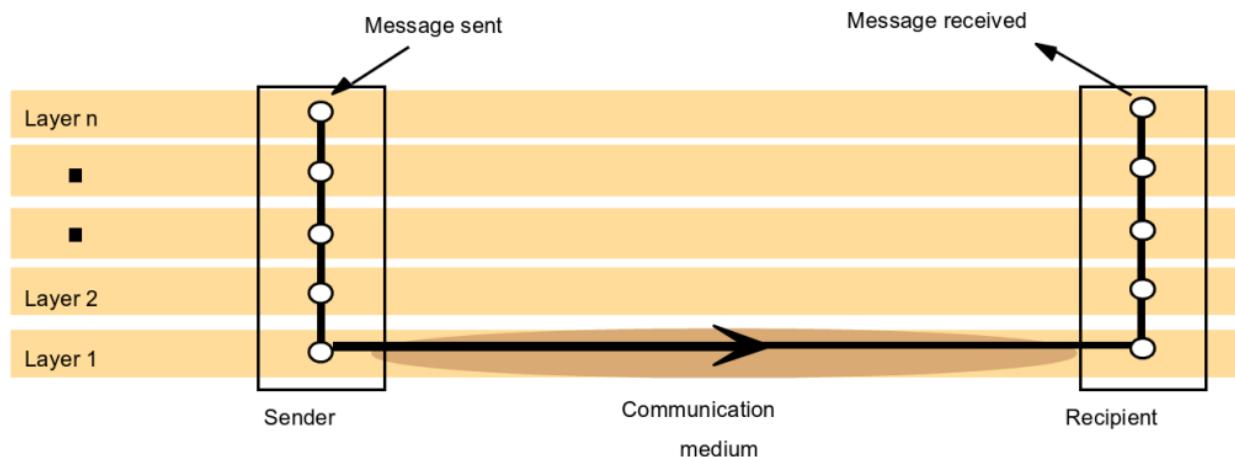
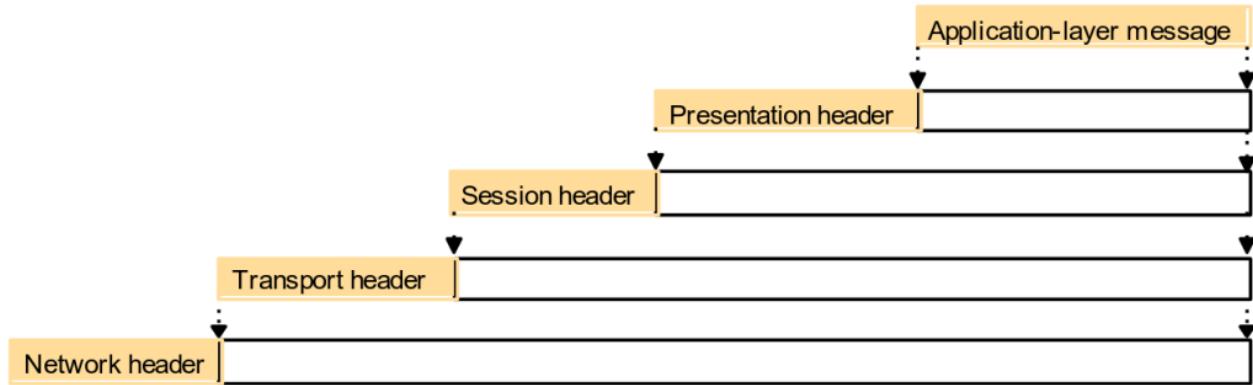


Figure 3.3 Encapsulation as it is applied in layered protocols



Protocol suites

protocol suite (or protocol stack) – complete set of protocol layers

Seven-layer Reference Model for *Open Systems Interconnection* (OSI)

Figure 3.4 Protocol layers in the ISO Open Systems Interconnection (OSI) model

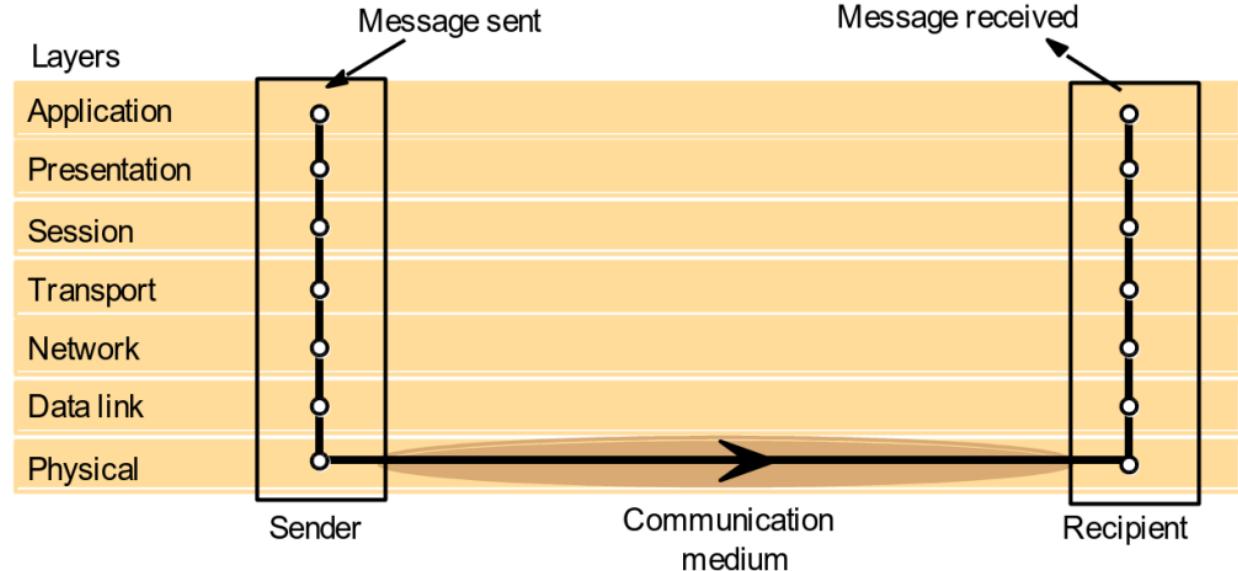
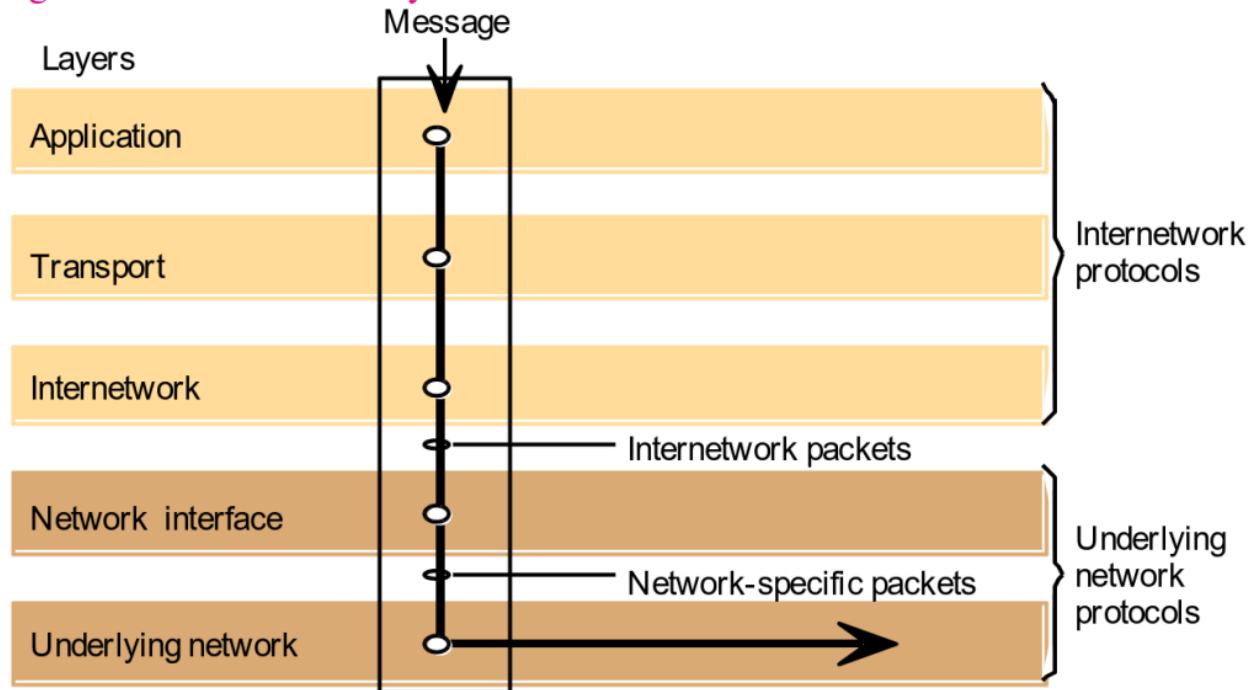


Figure 3.5 OSI protocol summary

<i>Layer</i>	<i>Description</i>	<i>Examples</i>
Application	Protocols that are designed to meet the communication requirements of specific applications, often defining the interface to a service.	HTTP, FTP , SMTP, CORBA IIOP
Presentation	Protocols at this level transmit data in a network representation that is independent of the representations used in individual computers, which may differ. Encryption is also performed in this layer, if required.	Secure Sockets (SSL),CORBA Data Rep.
Session	At this level reliability and adaptation are performed, such as detection of failures and automatic recovery.	
Transport	This is the lowest level at which messages (rather than packets) are handled. Messages are addressed to communication ports attached to processes, Protocols in this layer may be connection-oriented or connectionless.	TCP, UDP
Network	Transfers data packets between computers in a specific network. In a WAN or an internetwork this involves the generation of a route passing through routers. In a single LAN no routing is required.	IP, ATM virtual circuits
Data link	Responsible for transmission of packets between nodes that are directly connected by a physical link. In a WAN transmission is between pairs of routers or between routers and hosts. In a LAN it is between any pair of hosts.	Ethernet MAC, ATM cell transfer, PPP
Physical	The circuits and hardware that drive the network. It transmits sequences of binary data by analogue signalling, using amplitude or frequency modulation of electrical signals (on cable circuits), light signals (on fibre optic circuits) or other electromagnetic signals (on radio and microwave circuits).	Ethernet base- band signalling, ISDN

Figure 3.6 Internetwork layers



Packet assembly

- in transport layer
- header field
- data field
- *maximum transfer unit* (MTU)

Ports

- software-defined destination points at a host computer

Addressing

Internet Assigned Numbers Authority (IANA) [www.iana.org I]).

FTP : 21 ; HTTP :80

- ≤ 1023 – well-known ports – restricted to privileged processes
- 1024 ... 49151 – registered ports for which IANA holds service descriptions
- remaining ports upto 65535 available for private use

Packet delivery

Datagram packet delivery

- ‘*datagram*’ refers to the similarity of this delivery mode to the way in which letters and telegrams are delivered

Virtual circuit packet delivery:

- analogous to a telephone network
- A virtual circuit must be set up before packets can pass from a source host A to destination host B
- Virtual circuit currently in use: ATM (*Asynchronous Transfer Mode*)

3.3.5 Routing

in large networks – *adaptive routing* – the best route for communication between two points in the network is re-evaluated periodically

Two parts of routing algorithm:

1. make decisions that determine the route taken by each packet
2. dynamically update the knowledge of the network

Figure 3.7 Routing in a wide area network

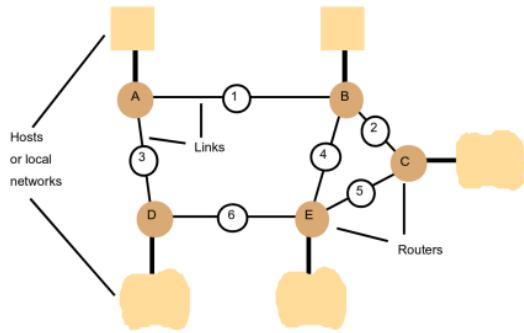


Figure 3.8 Routing tables for the network in Figure 3.7

Routings from A			Routings from B			Routings from C		
To	Link	Cost	To	Link	Cost	To	Link	Cost
A	local	0	A	1	1	A	2	2
B	1	1	B	local	0	B	2	1
C	1	2	C	2	1	C	local	0
D	3	1	D	1	2	D	5	2
E	1	2	E	4	1	E	5	1

Routings from D			Routings from E		
To	Link	Cost	To	Link	Cost
A	3	1	A	4	2
B	3	2	B	4	1
C	6	2	C	5	1
D	local	0	D	6	1
E	6	1	E	local	0

A simple routing algorithm

- 'distance vector' algorithm, instance of Bellman's shortest path algorithm (Bellman 1957)

Router information protocol (RIP)

1. Periodically, and whenever the local routing table changes, send the table (in a summary form) to all accessible neighbours (– send an RIP packet containing a copy of the table on each non-faulty outgoing link)
2. When a table is received from a neighbouring router
 - if the received table shows a route to a new destination, or
 - a better (lower-cost) route to an existing destination
 - update the local table with the new route

If the table was received on link n and it gives a different cost than the local table for a route that begins with link n :

replace the cost in the local table with the new cost.

Figure 3.9 Pseudo-code for RIP routing algorithm

```
Send: Each t seconds or when Tl changes, send Tl on each non-faulty
      outgoing link.  
Receive: Whenever a routing table Tr is received on link n:  
for all rows Rr in Tr {  
    if (Rr.link == n) {  
        Rr.cost = Rr.cost + 1;  
        Rr.link = n;  
        if (Rr.destination is not in Tl) add Rr to Tl; // add new  
            destination to Tl  
        else for all rows Rl in Tl {  
            if (Rr.destination == Rl.destination and  
                (Rr.cost < Rl.cost or Rl.link == n)) Rl = Rr;  
                // Rr.cost < Rl.cost : remote node has better route  
                // Rl.link = n : remote node is more authoritative  
            }  
        }  
    }  
}
```

Note: the value for t adopted throughout the Internet – 30 seconds

When a faulty link n is detected, set $cost := \infty$ for all entries in the local table that refer to the faulty link and perform the *Send* action

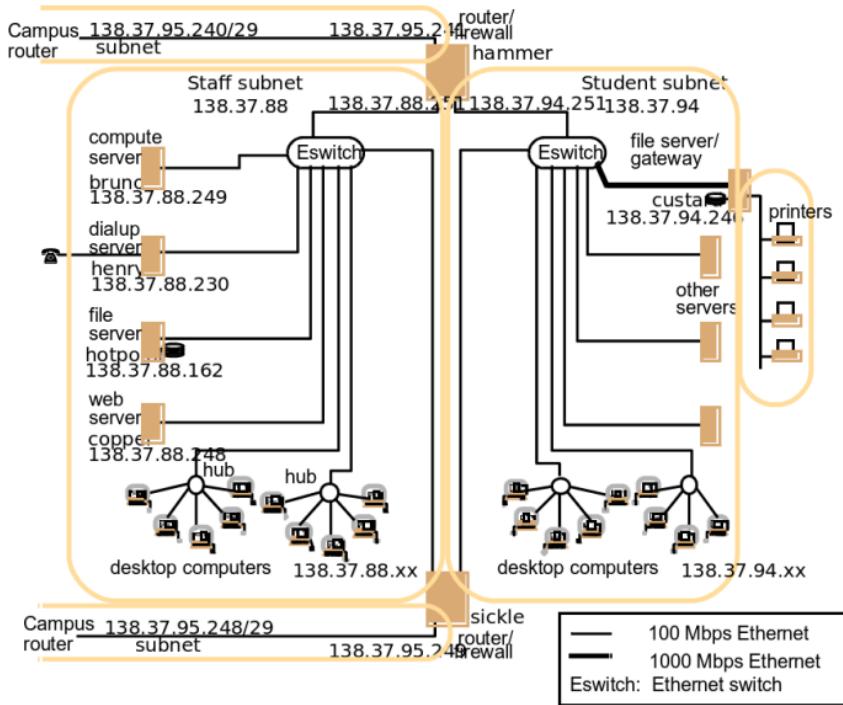
3.3.6 Congestion control

As a rule of thumb, when the load $> 80\% \Rightarrow$ total throughput drops due to packet losses

- before packet reaches congested node – better hold it back at earlier nodes!
- IP and Ethernets – end-to-end control of traffic
 - sending node must reduce the rate at which it transmits packets based only on information that it receives from the receiver
 - Congestion information may be supplied to the sending node by explicit transmission of special messages (called choke packets) requesting a reduction in transmission rate

3.3.7 Internetworking

Figure 3.10 Simplified view of part of a university campus network



Routers

- routing required in all networks except Ethernets and wireless

Bridges

- link networks of different types

Hubs

- convenient means of connecting and extending segments of Ethernet and other broadcast local network technologies

Switches

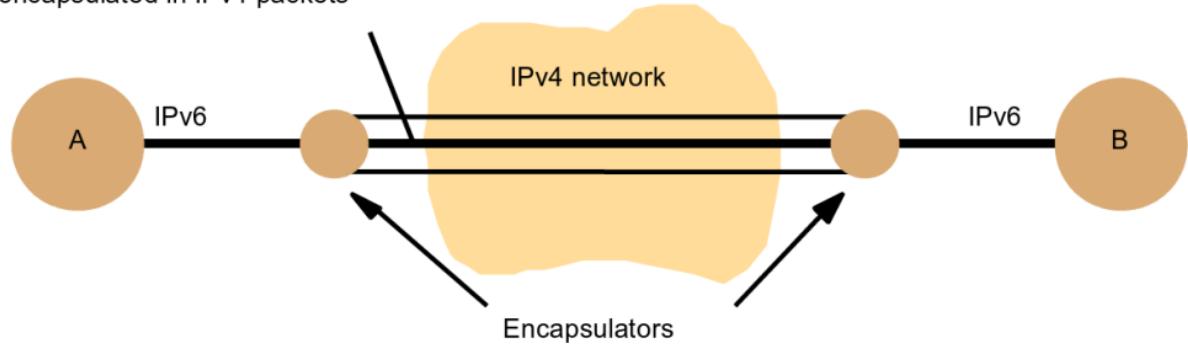
- perform similar function to routers, but for LANs (normally Ethernets)

Tunnelling

protocol tunnel – software layer to transmit packets through alien network

Figure 3.11 Tunnelling for IPv6 migration

IPv6 encapsulated in IPv4 packets



3.4 Internet protocols

ARPANET (1970, USA) – the first large-scale computer network

Figure 3.12 TCP/IP layers

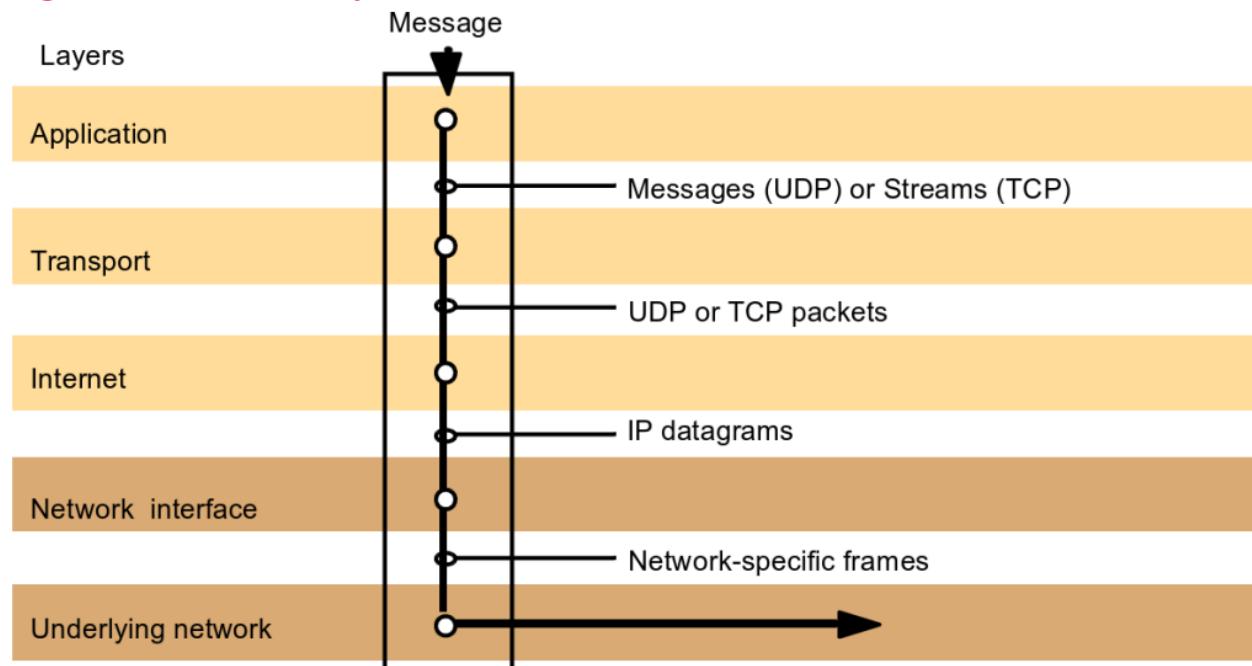


Figure 3.13 Encapsulation as it occurs when a message is transmitted via TCP over an Ethernet

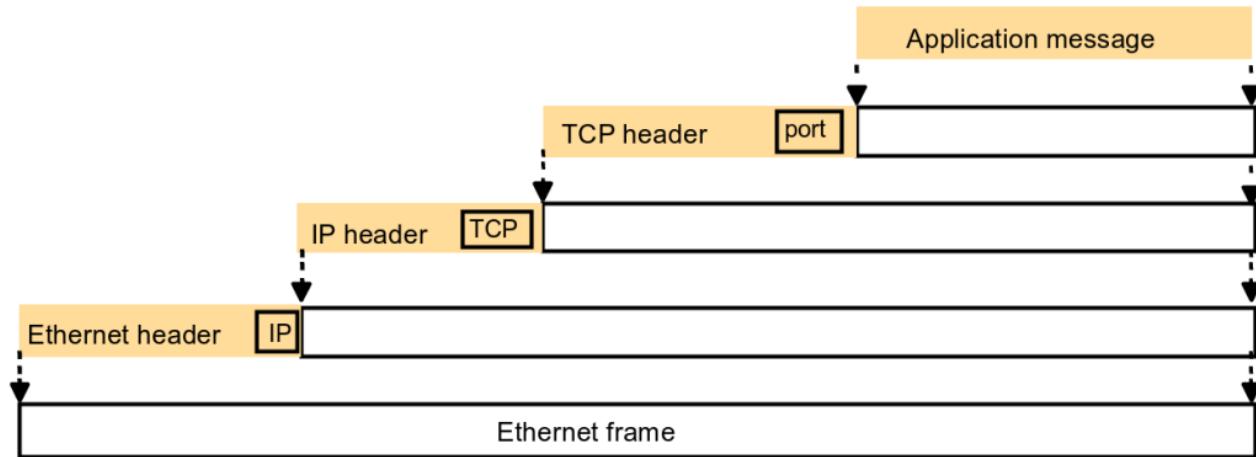
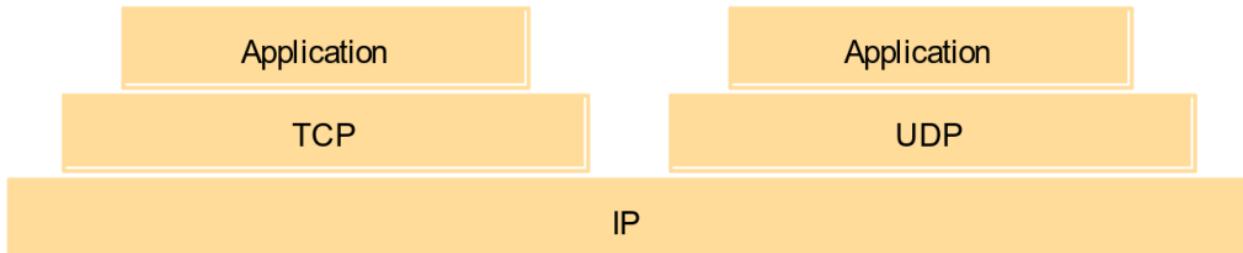


Figure 3.14 The programmer's conceptual view of a TCP/IP Internet



3.4.1 IP addressing

Should be

- universal
- efficient its use of the address space
 - TCP/IP – $2^{32} \approx 4$ billion addressable hosts
- flexible and efficient routing scheme
 - addresses themselves cannot contain very much of the information needed to route a packet

Figure 3.15 Internet address structure, showing field sizes in bits

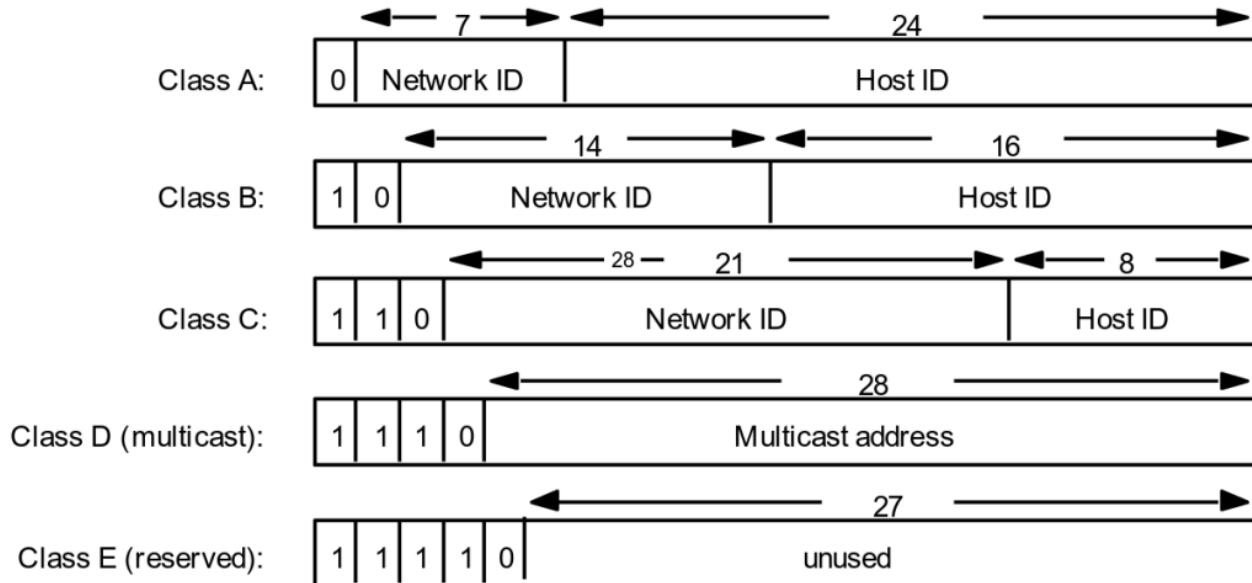


Figure 3.16 Decimal representation of Internet addresses

	octet 1	octet 2	octet 3	Range of addresses
Class A:	Network ID 1 to 127	0 to 255	Host ID 0 to 255	0 to 255 1.0.0.0 to 127.255.255.255
Class B:	128 to 191	Network ID 0 to 255	Host ID 0 to 255	128.0.0.0 to 191.255.255.255
Class C:	192 to 223	0 to 255	0 to 255	192.0.0.0 to 223.255.255.255
Class D (multicast):	224 to 239	0 to 255	0 to 255	224.0.0.0 to 239.255.255.255
Class E (reserved):	240 to 255	0 to 255	0 to 255	240.0.0.0 to 255.255.255.255

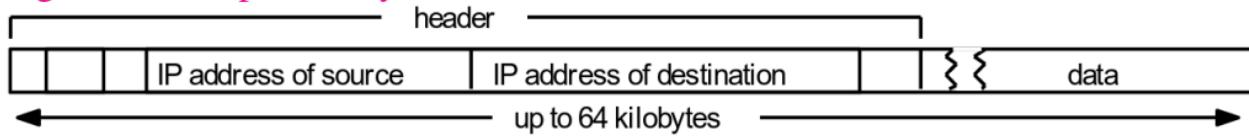
- host identifier
 - 0 – this host
 - all 1s – broadcast message

Around 1990: – IP addresses likely to run out around 1996

- Specification of IPv6
- Classless InterDomain Routing (CIDR)
- Network Address Transition (NAT) scheme to enable unregistered computers to access the Internet

3.4.2 The IP protocol

Figure 3.17 IP packet layout



- *unreliable* or *best-effort* delivery semantics (due to no guarantee of delivery)
- the only checksum in IP – a header checksum

Address resolution

- converting Internet addresses to network addresses
 - address resolution protocol (ARP)

IP spoofing

- IP packets include a source address + port address encapsulated in the data field
 - this information easily substitutable by attacks...!

3.4.3 IP routing

Backbones

Topological map of the Internet partitioned into *autonomous systems* (ASs), which are subdivided into *areas*

- Every AS in the topological map has a *backbone* area
- The collection of routers that connect non-backbone areas to the backbone and the links that interconnect those routers are called the backbone of the network

Routing protocols

- RIP-1 and RIP-2
- *open shortest path first* (OSPF)

Default routes

- key routers
- default destination

Routing on local subnet

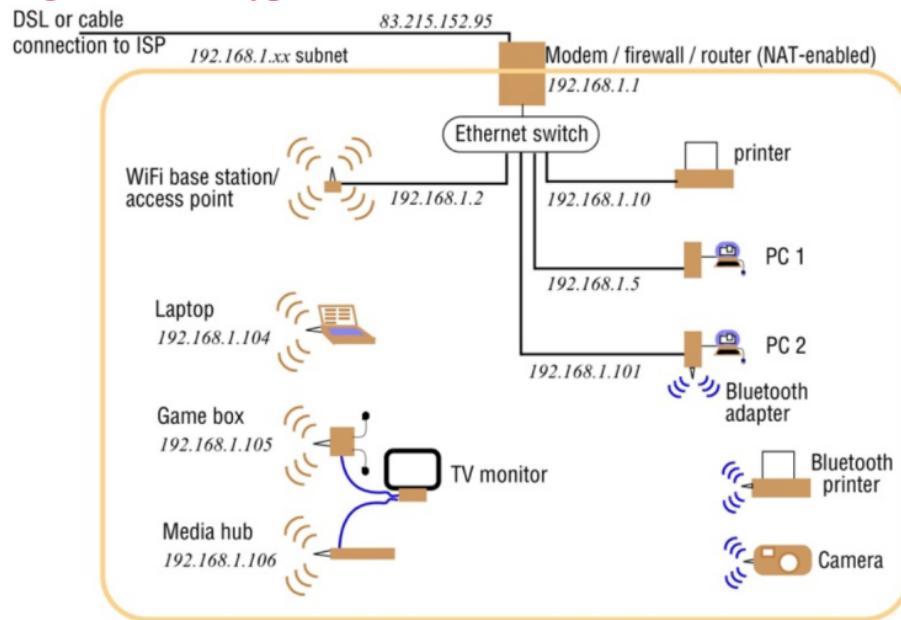
- using underlying network to transmit the packets

Classless interdomain routing (CIDR)

- add mask field to the routing tables

Unregistered addresses and Network Address Translation (NAT)

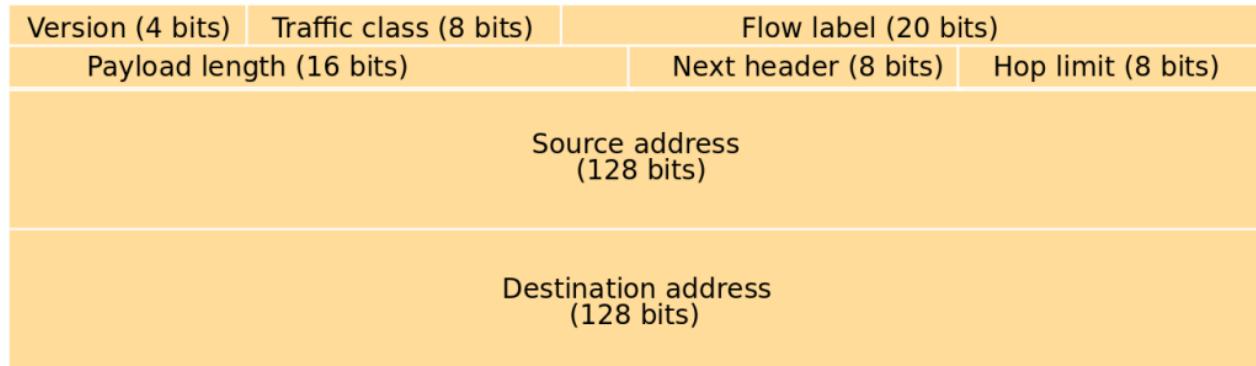
Figure 3.18 A typical NAT-based home network



- Dynamic Host Configuration Protocol (DHCP)
- three blocks of addresses (10.z.y.x, 172.16.y.x or 192.168.y.x) that IANA has reserved for private internets

3.4.4 IP version 6

Figure 3.19 IPv6 header layout



- Address space: 128 bits
 - $2^{128} \approx 3 \times 10^{38} - 7 \times 10^{23}$ IP addresses per square meter across the entire surface of the Earth (actually ca 1000 addresses if to take into account inefficiency in allocation...)

- Routing speed
 - no checksum to packet content (called also payload)
 - no fragmentation
- Real-time and other services
 - flow label – specific packets to be handled more rapidly
- Future evolution
 - next-header field (if non-zero, defines an extension header included in the packet)
- Multicast and anycast
 - *anycast* – packet delivered to at least one host with a relevant address

- Security
 - implemented through authentication and encrypted security payload extension header

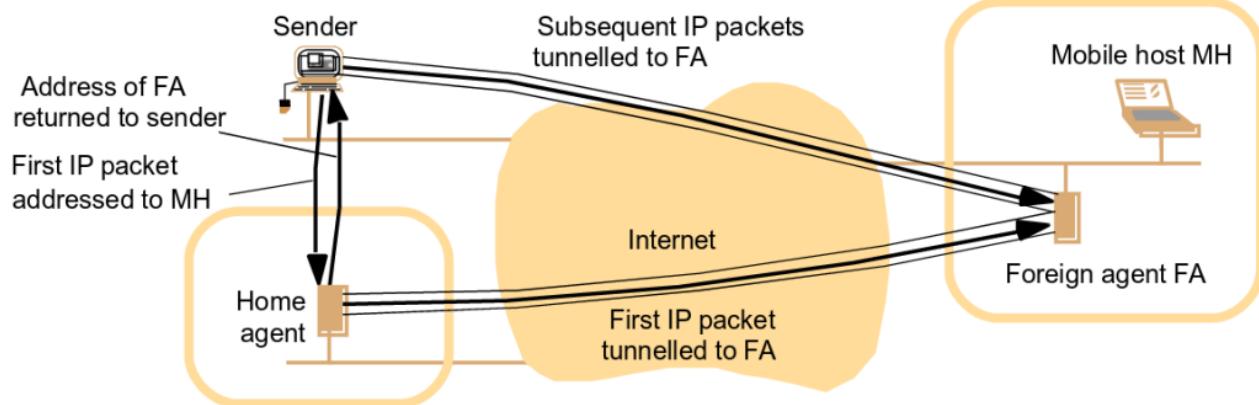
Migration from IPv4

- 'islands' of IPv6 connected via tunnels; gradually merging into larger islands
- IPv4 address space embedded in IPv6 space

3.4.5 MobileIP

home agents and foreign agents etc...

Figure 3.20 The MobileIP routing mechanism



3.4.6 TCP and UDP

Functionality:

Use of ports

IP communication between pairs of computers

Transport protocols – TCP and UDP

Port number – 16-bit integer

UDP features

- almost a transport-level replica of IP
- UDP adds no additional reliability mechanisms except the checksum, which is optional
- restricted to applications and services not requiring reliable delivery of single or multiple messages

TCP features

- reliable delivery of arbitrary long sequences of bytes via stream-based programming abstraction

additional mechanisms to meet reliability guarantees:

- *Sequencing*
 - stream divided into sequence of data segments which are transmitted as IP packets
- *Flow control*
 - reverse flow of data with acknowledgements, which includes window size
 - quantity of data that the sender is permitted to send before the next acknowledgement

- *Retransmission*
 - if no acknowledgement in specified timeout for a certain packet – it is retransmitted
- *Buffering*
 - if receive buffer becomes full, incoming packets start getting dropped and no acknowledgement sent back either – causing retransmission
- *Checksum*
 - Each segment carries a checksum covering the header and the data in the segment. If a received segment does not match its checksum, the segment is dropped

3.4.7 Domain names

- scheme for the use of symbolic names for hosts and networks (for example, binkley.cs.mcgill.ca or essex.ac.uk.)
- organized into a naming hierarchy (which is independent of the physical layout of the networks)
- DNS is implemented as a server process
 - can be run on host computers anywhere in the Internet
 - at least two DNS servers in each domain (often more)
 - servers in each domain hold a partial map of the domain name tree below their domain

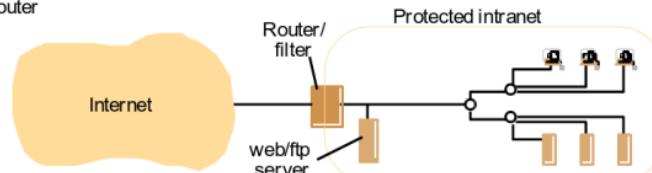
DNS would not be workable without the extensive use of caching, since the ‘root’ name servers would be consulted in almost every case, creating a service access bottleneck.

3.4.8 Firewalls

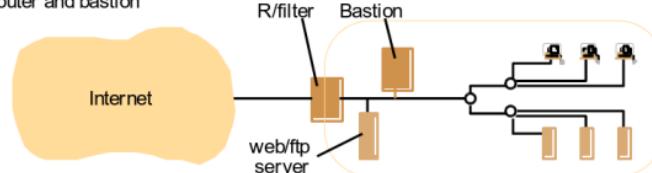
Firewall is implemented by a set of processes that act as a gateway to an intranet

Figure 3.21 Firewall configurations

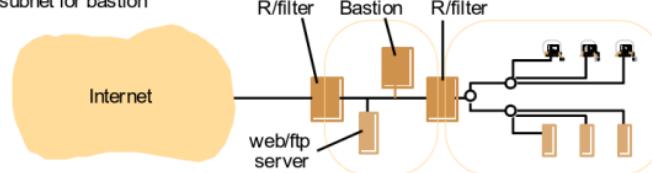
a) Filtering router



b) Filtering router and bastion



c) Screened subnet for bastion



Service control: To determine which services on internal hosts are accessible for external access and to reject all other incoming service requests.

Behaviour control: To prevent behaviour that infringes the organization's policies, is antisocial or has no discernible legitimate purpose

User control: Organizations may wish to give a set of users other privileges than others...

IP packet filtering: for example, based on port numbers – no NFS usage from outside

TCP gateway: TCP segments can be checked for correctness (some denial of service attacks use malformed TCP segments to disrupt client operating systems)

Application-level gateway: An application-level gateway process acts as a proxy for an application process

Virtual private networks (VPNs)

– extend the firewall protection boundary beyond the local intranet by the use of cryptographically protected secure channels at the IP level

3.5 Case studies

Figure 3.22 IEEE 802 network standards

<i>IEEE No.</i>	<i>Name</i>	<i>Title</i>	<i>Reference</i>
802.3	Ethernet	CSMA/CD Networks (Ethernet)	[IEEE 1985a]
802.4		Token Bus Networks	[IEEE 1985b]
802.5		Token Ring Networks	[IEEE 1985c]
802.6		Metropolitan Area Networks	[IEEE 1994]
802.11	WiFi	Wireless Local Area Networks	[IEEE 1999]
802.15.1	Bluetooth	Wireless Personal Area Networks	[IEEE 2002]
802.15.4	ZigBee	Wireless Sensor Networks	[IEEE 2003]
802.16	WiMAX	Wireless Metropolitan Area Networks	[IEEE 2004a]

3.5.1 Ethernet

Home task: read the book (pp 144-154), in particular:

Packet broadcasting

Ethernet paket layout

Packet collisions

Ethernet efficiency

Physical implementations

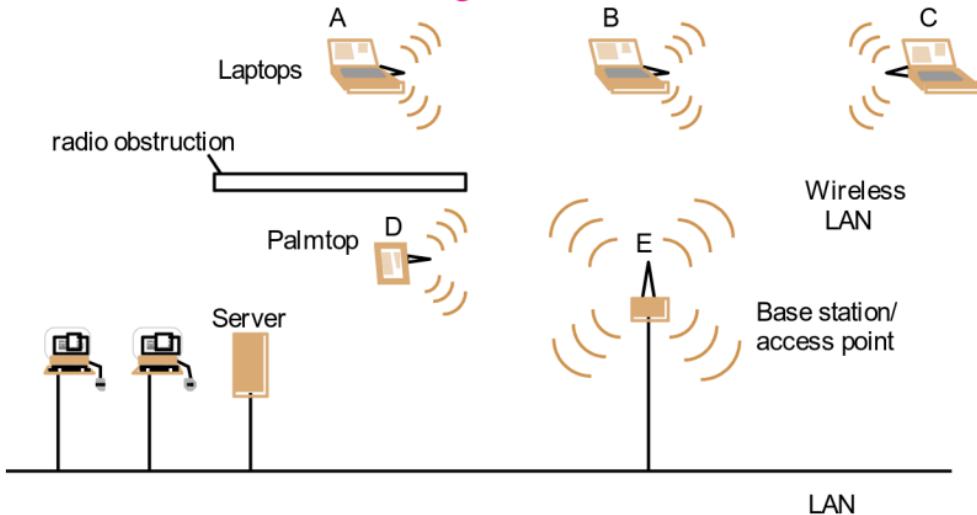
Figure 3.23 Ethernet ranges and speeds

	10Base5	10BaseT	100BaseT	1000BaseT
Data rate	10 Mbps	10 Mbps	100 Mbps	1000 Mbps
<i>Max. segment lengths:</i>				
Twisted wire (UTP)	100 m	100 m	100 m	25 m
Coaxial cable (STP)	500 m	500 m	500 m	25 m
Multi-mode fibre	2000 m	2000 m	500 m	500 m
Mono-mode fibre	25000 m	25000 m	20000 m	2000 m

Ethernet for real-time and quality of service critical applications

3.5.2 IEEE 802.11 (WiFi) wireless LAN

Figure 3.24 Wireless LAN configuration



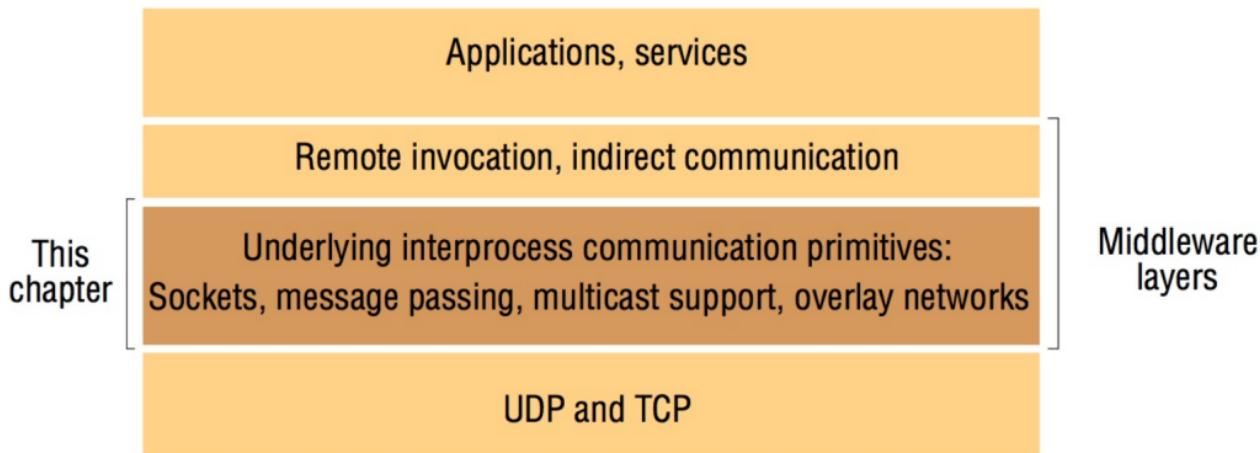
Security

End of week 3

4 Interprocess communication

4.1 Introduction

Figure 4.1 Middleware layers



How middleware and application programs can use UDP and TCP?

What is specific about IP multicast? Why/how could it be made more reliable?

What is an overlay network?

What is MPI?

4.2 The API for the Internet protocols

4.2.1 The characteristics of interprocess communication

Synchronous and asynchronous communication

synchronous – sending and receiving processes synchronize at every message

- both *send* and *receive* – blocking operations
 - whenever *send* is issued – sending process blocked until *receive* is issued
 - whenever *receive* is issued by a process, it is blocked until the message arrives

asynchronous – *send* – nonblocking; *receive* – either blocking or non-blocking

In case threads are supported (Java) blocking receive has no disadvantages – a separate thread is handling the communication while other threads can continue their work

- today's systems do not generally provide the non-blocking *receive*

Message destinations

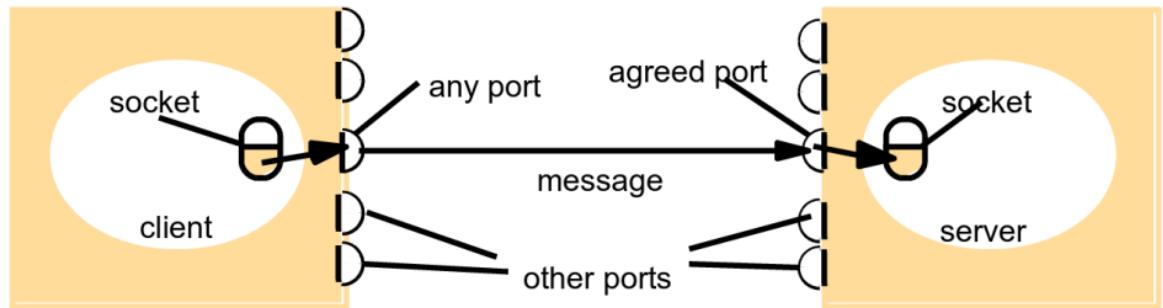
- messages sent to (*Internet address, local port*)

Reliability & ordering – also important factors

4.2.2 Sockets

socket – abstraction providing an endpoint for communication between processes

Figure 4.2 Sockets and ports



Internet address = 138.37.94.248

Internet address = 138.37.88.249

Java API for Internet addresses

- Java class InetAddress referring to Domain Name System (DNS) hostnames

```
InetAddress aComputer = InetAddress .getByName("bruno . dcs . qmul . ac . uk");
```

4.2.3 UDP datagram communication

- datagram transmission without acknowledgement or retries
 - create a socket bound to an Internet address of the local host and a local port
 1. A server will bind its socket to a server port
 2. A client binds its socket to any free local port
 - The receive method returns the Internet address and port of the sender, in addition to the message (allowing the recipient to send a reply)

Issues related to datagram communication:

Message size:

- in IP protocol – $\leq 2^{16}$ (incl. headers), but in most environments ≤ 8 kilobytes

Blocking:

- Sockets normally provide non-blocking *sends* and blocking *receives*

Timeouts:

- if needed, should be fairly large in comparison with the time for message transmission

Receive from any:

- by default every message is placed in a receiving queue
 - but it is possible to connect a datagram socket to a particular remote port and Internet address

Failure model for UDP datagrams

(In Chapter 2: failure model for communication channels – reliable communication in terms of 2 properties – *integrity* and *validity*)

UDP datagrams suffer from

- Omission failures
- Ordering

Applications – provide your own checks!

Use of UDP

- Domain Name System (DNS)
- Voice over IP (VOIP)

No overheads associated with guaranteed message delivery. But overheads on:

- the need to store state information at the source and destination
- transmission of extra messages
- latency for the sender

Java API for UDP datagrams

2 classes: DatagramPacket and DatagramSocket

Class **DatagramPacket** – provides constructor for making an instance out of

- an array of bytes comprising a message
- the length of the message
- and the Internet address and
- local port number of the destination socket

DatagramPacket

array of bytes containing message	length of message	Internet address	port number
-----------------------------------	-------------------	------------------	-------------

On the receiving side: DatagramPacket has another constructor + methods getData, getPort and getAddress

Class **DatagramSocket** – supports sockets for sending and receiving datagrams

- constructor with port number
 - has also no-argument case – system to choose a free port
- Methods:
 - send and receive
 - * argument – DatagramPacket
 - setSoTimeout – block receive for specified time before throwing InterruptedIOException
 - connect – to connect to a particular remote port and internet address for exclusive communication to/from there

Figure 4.3 UDP client sends a message to the server and gets a reply

```
import java.net.*;
import java.io.*;
public class UDPClient{
    public static void main(String args[]){
        // args give message contents and server hostname
        DatagramSocket aSocket = null;
        try {
            aSocket = new DatagramSocket();
            byte [] m = args[0].getBytes();
            InetAddress aHost = InetAddress.getByName(args[1]);
            int serverPort = 6789;
            DatagramPacket request = new DatagramPacket(m,m.length(),aHost,serverPort);
            aSocket.send(request);
            byte[] buffer = new byte[1000];
            DatagramPacket reply = new DatagramPacket(buffer, buffer.length());
            aSocket.receive(reply);
            System.out.println("Reply:" + new String(reply.getData()));
        }catch (SocketException e){System.out.println("Socket:" + e.getMessage());}
        }catch (IOException e){System.out.println("IO:" + e.getMessage());}
        }finally {if(aSocket != null) aSocket.close();}
    }
}
```

Figure 4.4 UDP server repeatedly receives a request and sends it back to the client

```
import java.net.*;
import java.io.*;
public class UDPServer{
    public static void main(String args[]){
        DatagramSocket aSocket = null;
        try{
            aSocket = new DatagramSocket(6789);
            byte[] buffer = new byte[1000];
            while(true){
                DatagramPacket request = new DatagramPacket(buffer,buffer.length);
                aSocket.receive(request);
                DatagramPacket reply = new DatagramPacket(request.getData(),
                    request.getLength(), request.getAddress(), request.getPort());
                aSocket.send(reply);
            }
        } catch (SocketException e){System.out.println("Socket:" + e.getMessage());}
        } catch (IOException e) {System.out.println("IO:" + e.getMessage());}
        } finally {if(aSocket != null) aSocket.close();}
    }
}
```

4.2.4 TCP stream communication

Network characteristics hidden by stream abstraction:

- Message sizes
- Lost messages
- Flow control
- Message duplication and ordering
- Message destinations
 - once connection established – simply read/write to/from stream
 - to establish connection
 - * connect request (from client)
 - * accept request (from server)

Pair of sockets associated with stream – read and write

Issues related to stream communication:

- Matching data items – (e.g. int should be followed by float – matching in both sides)
- Blocking –
 - while trying to read data before it has arrived in queue
 - writing data to the stream, but the TCP flow-control mechanism still waiting for data acknowledgements etc.
- Threads – usually used

Failure model

- integrity

- checksums
- sequence numbers
- validity
 - timeouts
 - retransmission

Use of TCP

HTTP, FTP, Telnet, SMTP

Java API for TCP streams

Classes ServerSocket and Socket

Class **ServerSocket**:

- to listen connect requests from clients

- accept method
 - gets a connect request from the queue or
 - if the queue is empty, blocks until one arrives
 - result of executing accept – an instance of `Socket` – a socket to use for communicating with the client

Class `Socket`:

- for use by pair of processes
- client constructor – to create a socket specifying DNS hostname and port of a server
 - connects to the specified remote computer and port number
- methods:
 - `getInputStream` and `getOutputStream`

Figure 4.5 TCP client makes connection to server, sends request and receives reply

```
import java.net.*;
import java.io.*;
public class TCPClient {
    public static void main (String args[]) {
        // arguments supply message and hostname of destination
        Socket s = null;
        try{
            int serverPort = 7896;
            s = new Socket(args[1], serverPort);
            DataInputStream in = new DataInputStream( s.getInputStream());
            DataOutputStream out = new DataOutputStream( s.getOutputStream());
            out.writeUTF(args[0]); // UTF is a string encoding see Sn 4.3
            String data = in.readUTF();
            System.out.println("Received:"+ data) ;
        }catch (UnknownHostException e){
            System.out.println("Sock:"+e.getMessage());
        }catch (EOFException e){System.out.println("EOF:"+e.getMessage());
        }catch (IOException e){System.out.println("IO:"+e.getMessage());}
        }finally {if(s!=null) try {s.close();} catch (IOException e){System.out.
            println("close:"+e.getMessage());}}
    }
}
```

Figure 4.6 TCP server makes a connection for each client and then echoes the client's request

```
import java.net.*;
import java.io.*;
public class TCPServer {
    public static void main (String args[]) {
        try{
            int serverPort = 7896;
            ServerSocket listenSocket = new ServerSocket(serverPort);
            while(true) {
                Socket clientSocket = listenSocket.accept();
                Connection c = new Connection(clientSocket);
            }
        } catch(IOException e) {System.out.println("Listen "+e.getMessage());}
    }
}
class Connection extends Thread {
    DataInputStream in;
    DataOutputStream out;
    Socket clientSocket;
    public Connection (Socket aClientSocket) {
        try {
            clientSocket = aClientSocket;
```

```
in = new DataInputStream( clientSocket.getInputStream());
out =new DataOutputStream( clientSocket.getOutputStream());
this.start();
} catch(IOException e) {System.out.println("Connection:"+e.getMessage());}
}

public void run() {
    try {                                         // an echo server
        String data = in.readUTF();
        out.writeUTF(data);
    } catch(EOFException e) {System.out.println("EOF:"+e.getMessage());}
    } catch(IOException e) {System.out.println("IO:"+e.getMessage());}
    } finally{ try {clientSocket.close();} catch (IOException e){/*close failed */}}
}
}
```

4.3 External data representation and marshalling

- messages ←
 - data values of many different types
 - different floating-point number representations
 - integers – big-endian, little-endian order
 - ASCII – 1byte; Unicode – 2bytes

⇒ either:

- a) convert data to agreed external format, or
- b) transmit data in sender's format + format used – recipient converts the values if needed

external data representation: agreed standard for the representation of data structures and primitive values

marshalling: process of taking a collection of data items and assembling them into a form suitable for transmission in a message

unmarshalling: process of disassembling a collection data items from a message at the destination

- CORBA's (Common Object Request Broker Architecture) common data representation (bin, just values)
- Java's object serialization (bin, data + type info)
- XML (Extensible Markup Languaga) (txt, may refer to externally defined *namespaces*)
- Google – *protocol buffers* (both stored and transmitted data)
- JSON (JavaScript Object Notation) <http://www.json.org>

4.3.1 CORBA's Common Data Representation (CDR)

primitive types:	4. unsigned	7. char	10. any (which
1. short (16-bit)	long	8. boolean	can represent
2. long (32-bit)	5. float (32-bit)	(TRUE, FALSE)	any basic or constructed
3. unsigned short	6. double (64- bit)	9. octet (8-bit)	type)

Constructed (composite) types: sequence of bytes in a particular order:

Figure 4.7 CORBA CDR for constructed types

Type	Representation
sequence	length (unsigned long) followed by elements in order
string	length (unsigned long) followed by characters in order (can also have wide characters)
array	array elements in order (no length specified because it is fixed)
struct	in the order of declaration of the components
enumerated	unsigned long (the values are specified by the order declared)
union	type tag followed by the selected member

CORBA CDR that contains the three fields of a struct whose respective types are string, string and unsigned long:

- Person struct with value: {‘Smith’, ‘London’, 1984}

Figure 4.8 CORBA CDR message

<i>index in sequence of bytes</i>	<i>notes on representation</i>
0–3	5
4–7	“Smit”
8–11	“h_____”
12–15	6
16–19	“Lond”
20–23	“on_____”
24–27	1984

The flattened form represents a *Person* struct with value: {‘Smith’, ‘London’, 1984}

```
struct Person{  
    string name;  
    string place;  
    unsigned long year;  
};
```

Marshalling through CORBA IDL

Sun XDR standard

- similar to CORBA in many ways
- sending messages between clients and servers in Sun NFS
- <http://www.cdk5.net/ ipc>

4.3.2 Java object serialization

```
public class Person implements Serializable {  
    private String name;  
    private String place;  
    private int year;  
    public Person(String aName, String aPlace, int aYear) {  
        name = aName;  
        place = aPlace;  
        year = aYear;  
    }  
    // followed by methods for accessing the instance variables  
}
```

serialization – flattening an object or a connected set of objects into a serial form suitable for storing on disk or transmitting in a message

deserialization – vica versa, assuming no a priori knowledge about of types of objects
 – self-containness

- serialization of an object + all objects it references as well to ensure that with the object reconstruction, all of its references can be fulfilled at the destination
- recursive procedure

```
Person p = new Person( "Smith" , "London" , 1984) ;
```

Figure 4.9 Indication of Java serialized form

Serialized values				Explanation
Person	8-byte version number		h0	class name, version number
3	int year	java.lang.String name:	java.lang.String place:	number, type and name of instance variables
1984	5 Smith	6 London	h1	values of instance variables

The true serialized form contains additional type markers; h0 and h1 are handles

- serialize:
 - create an instance of the class `ObjectOutputStream` and invoke its `writeObject` method
- deserialize:
 - open an `ObjectInputStream` on the stream and use its `readObject` method to reconstruct the original object

(de)serialization carried out automatically in RMI

Reflection — the ability to enquire about the properties of a class, such as the names and types of its instance variables and methods

- enables classes to be created from their names
- a constructor with given argument types to be created for a given class

- Reflection makes it possible to do serialization and deserialization in a completely generic manner

4.3.3 Extensible Markup Language (XML)

- defined by the World Wide Web Consortium (W3C)
- data items are tagged with ‘markup’ strings
- tags relate to the structure of the text that they enclose
- XML is used to:
 - enable clients to communicate with web services
 - defining the interfaces and other properties of web services
 - many other uses
 - * archiving and retrieval systems

- * specification of user interfaces
- * encoding of configuration files in operating systems
- clients usually use SOAP messages to communicate with web services

SOAP – XML format whose tags are published for use by web services and their clients

XML elements and attributes

Figure 4.10 XML definition of the Person structure

```
<person id="123456789">
    <name>Smith </name>
    <place>London </place>
    <year>1984 </year>
    <!-- a comment -->
</person >
```

Elements: portion of character data surrounded by matching start and end tags

- An empty tag – no content and is terminated with /> instead of >
 - For example, the empty tag <european/> could be included within the <person> ...</person> tag

Attributes: element – generally a container for data, whereas an attribute – used for labelling that data

- Attributes are for simple values
- if data contains substructures or several lines, it must be defined as an element

Names start with letter _ or :

Binary data – expressed in character data in base64

Parsing and well-formed documents

set of rules e.g. XML prolog:

```
<?XML version = "1.0" encoding = "UTF-8" standalone = "yes"?>
```

XML namespaces – URL referring to the file containing the namespace definitions.

- For example:

```
xmlns:pers = "http://www.cdk5.net/person"
```

Figure 4.11 Illustration of the use of a namespace in the Person structure

```
<person pers:id="123456789" xmlns:pers = "http://www.cdk5.net/person">
    <pers:name> Smith </pers:name>
    <pers:place> London </pers:place>
    <pers:year> 1984 </pers:year>
</person>
```

XML schemas [www.w3.org VIII] defines the elements and attributes that can appear in a document, how the elements are nested and the order and number of elements, and whether an element is empty or can include text

- used for encoding and validation

Figure 4.12 An XML schema for the Person structure

```
<xsd:schema xmlns:xsd = URL of XML schema definitions>
  <xsd:element name= "person" type ="personType"/>
    <xsd:complexType name="personType">
      <xsd:sequence>
        <xsd:element name = "name" type="xs:string"/>
        <xsd:element name = "place" type="xs:string"/>
        <xsd:element name = "year" type="xs:positiveInteger"/>
      </xsd:sequence>
      <xsd:attribute name= "id" type = "xs:positiveInteger"/>
    </xsd:complexType>
</xsd:schema>
```

APIs for accessing XML – in Java, Python etc.

4.3.4 Remote object references

Java, CORBA

- *remote object reference* is an identifier for a remote object that is valid throughout a distributed system

Figure 4.13 Representation of a remote object reference

32 bits	32 bits	32 bits	32 bits	interface of remote object
Internet address	port number	time	object number	

4.4 Multicast communication

single message from one process to each of the members of a group of processes, usually in such a way that the membership of the group is transparent to the sender

1. Fault tolerance based on replicated services
2. Discovering services in spontaneous networking
3. Better performance through replicated data
4. Propagation of event notifications

4.4.1 IP multicast – An implementation of multicast communication

Java's API to it via the `MulticastSocket` class

IP multicast

- sender is unaware of the identities of the individual recipients and of the size of the group

- group specified by a Class D Internet address
 - first 4 bits are 1110 in IPv4
- Being a member of a multicast group allows a computer to receive IP packets sent to the group
- membership dynamic
 - computers allowed to join or leave at any time
 - to join an arbitrary number of groups
 - possible to send datagrams to a multicast group without being a member
- At the application programming level, IP multicast available only via UDP
- Multicast routers
- *time to live (TTL)*

Multicast address allocation:

- Local Network Control Block (224.0.0.0 to 224.0.0.225)
- Internet Control Block (224.0.1.0 to 224.0.1.225)
- Ad Hoc Control Block (224.0.2.0 to 224.0.255.0)
- Administratively Scoped Block (239.0.0.0 to 239.255.255.255) – constrained propagation

Failure model for multicast datagrams

- failure characteristics as UDP datagrams
- *unreliable* multicast

Java API to IP multicast

Figure 4.14 Multicast peer joins a group and sends and receives datagrams

```
import java.net.*;
import java.io.*;
public class MulticastPeer{
    public static void main(String args[]){
        // args give message contents & destination multicast group (e.g. "228.5.6.7")
        MulticastSocket s = null;
        try {
            InetAddress group = InetAddress.getByName(args[1]);
            s = new MulticastSocket(6789);
            s.joinGroup(group);
            byte [] m = args[0].getBytes();
            DatagramPacket messageOut =
                new DatagramPacket(m, m.length, group, 6789);
            s.send(messageOut);
            byte [] buffer = new byte[1000];
            for(int i=0; i< 3; i++) { // get messages from others in group
                DatagramPacket messageIn =
                    new DatagramPacket(buffer, buffer.length);
                s.receive(messageIn);
                System.out.println("Received:" + new String(messageIn.getData()));
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

```
        }
        s.leaveGroup(group);
    } catch (SocketException e){System.out.println("Socket:" + e.getMessage());
    } catch (IOException e){System.out.println("IO:" + e.getMessage());
    } finally { if(s != null) s.close();}
}
}
```

End of week 4

4.5 Network virtualization: Overlay networks

Network virtualization – construction of many different virtual networks over an existing network

- each virtual network redefines its own addressing scheme, protocols, routing algorithms – depending on particular application on top

4.5.1 Overlay networks

overlay network – virtual network consisting of nodes and virtual links, which sits on top of an underlying network (such as an IP network) and offers something that is not otherwise provided:

- a service for a class of applications or a particular higher-level service
 - e.g. multimedia content distribution
- more efficient operation in a given networked environment
 - e.g. routing in an ad hoc network
- an additional feature
 - e.g. multicast or secure communication.

This leads to a wide variety of types of overlay as captured by Figure 4.15

Figure 4.15 Types of overlay

Motivation	Type	Description
<i>Tailored for application needs</i>	Distributed hash tables	One of the most prominent classes of overlay network, offering a service that manages a mapping from keys to values across a potentially large number of nodes in a completely decentralized manner (similar to a standard hash table but in a networked environment).
Peer-to-peer file sharing		Overlay structures that focus on constructing tailored addressing and routing mechanisms to support the cooperative discovery and use (for example, download) of files.
Content distribution networks		Overlays that subsume a range of replication, caching and placement strategies to provide improved performance in terms of content delivery to web users; used for web acceleration and to offer the required real-time performance for video streaming [www.kontiki.com].

Figure 4.15 Types of overlay (Continued)

<i>Tailored for network style</i>	Wireless ad hoc networks	Network overlays that provide customized routing protocols for wireless ad hoc networks, including proactive schemes that effectively construct a routing topology on top of the underlying nodes and reactive schemes that establish routes on demand typically supported by flooding.
	Disruption-tolerant networks	Overlays designed to operate in hostile environments that suffer significant node or link failure and potentially high delays.
<i>Offering additional features</i>	Multicast	One of the earliest uses of overlay networks in the Internet, providing access to multicast services where multicast routers are not available; builds on the work by Van Jacobson, Deering and Casner with their implementation of the MBone (or Multicast Backbone) [mbone].
	Resilience	Overlay networks that seek an order of magnitude improvement in robustness and availability of Internet paths [nms.csail.mit.edu].
	Security	Overlay networks that offer enhanced security over the underling IP network, including virtual private networks, for example, as discussed in Section 3.4.8.

- Advantages:
 - new network services changes to the underlying network
 - encourage experimentation with network services and the customization of services to particular classes of application
 - Multiple overlays can coexist
- Disadvantages:
 - extra level of indirection (hence performance penalty)
 - add to the complexity of network services

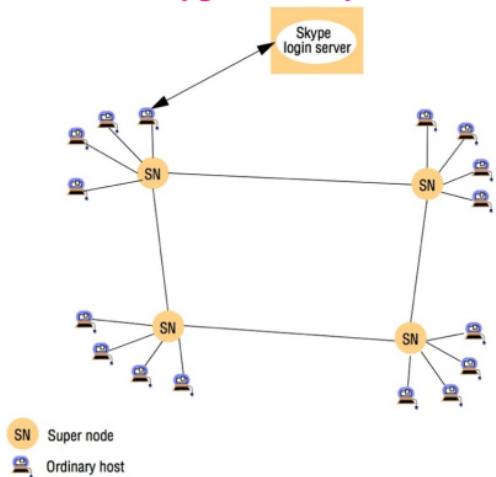
4.5.2 Skype: An example of an overlay network

Peer-to-peer application offering VoIP; 370M users (2009); developed by Kazaa p2p filesharing app

Skype architecture

- hosts and super nodes (which being selected on demand)

Figure 4.16 Skype overlay architecture



User connection

- users authenticated via login server

Search for users

- super nodes – to perform the efficient search of the global index of users distributed across the super nodes
 - On average, eight super nodes are contacted
 - 3-4 seconds to complete for hosts that have a global IP address (5-6 second, if behind a NAT-enabled router)

Voice connection

- TCP for signalling call requests and terminations and either UDP or TCP for the streaming audio

- UDP is preferred
- TCP can be used in certain circumstances to circumvent firewalls

4.6 Case study: MPI

MPI (The Message Passing Interface)

- A message-passing library specification
 - extended message-passing model
 - not a language or compiler specification
 - not a specific implementation or product
- Full featured; for parallel computers, clusters, and heterogeneous networks
- Designed to provide access to advanced parallel hardware for end users, library writers, and tool developers

MPI as STANDARD

Goals of the MPI standard MPI's prime goals are:

- To provide source-code portability
- To allow efficient implementations

MPI also offers:

- A great deal of functionality
- Support for heterogeneous parallel architectures

4 types of MPI calls

1. Calls used to initialize, manage, and terminate communications
2. Calls used to communicate between pairs of processors (Pair communication)
3. Calls used to communicate among groups of processors (Collective communication)
4. Calls to create data types

MPI basic subroutines (functions)

MPI_Init: initialise MPI

MPI_Comm_Size: how many PE?

MPI_Comm_Rank: identify the PE

MPI_Send

MPI_Receive

MPI_Finalise: close MPI

Example (Fortran90) 11.1 Greetings (<http://www.ut.ee/~eero/SC/konspekt/Naited/greetings.f90.html>)

Figure 4.17 An overview of point-to-point communication in MPI

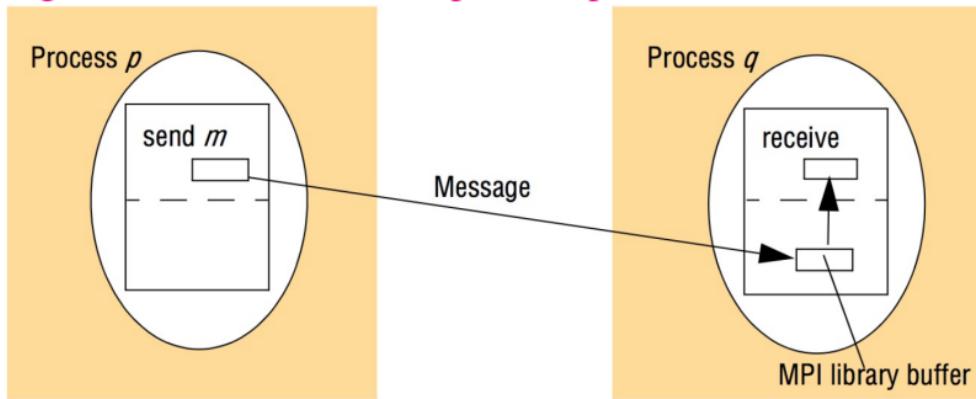


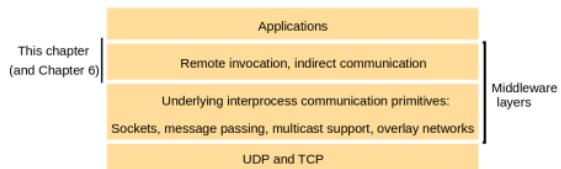
Figure 4.18 Selected send operations in MPI

<i>Send operations</i>	<i>Blocking</i>	<i>Non-blocking</i>
<i>Generic</i>	<i>MPI_Send</i> : the sender blocks until it is safe to return – that is, until the message is in transit or delivered and the sender's application buffer can therefore be reused.	<i>MPI_Isend</i> : the call returns immediately and the programmer is given a communication request handle, which can then be used to check the progress of the call via <i>MPI_Wait</i> or <i>MPI_Test</i> .
<i>Synchronous</i>	<i>MPI_Ssend</i> : the sender and receiver synchronize and the call only returns when the message has been delivered at the receiving end.	<i>MPI_Issend</i> : as with <i>MPI_Isend</i> , but with <i>MPI_Wait</i> and <i>MPI_Test</i> indicating whether the message has been delivered at the receive end.
<i>Buffered</i>	<i>MPI_Bsend</i> : the sender explicitly allocates an MPI buffer library (using a separate <i>MPI_Buffer_attach</i> call) and the call returns when the data is successfully copied into this buffer.	<i>MPI_Ibsend</i> : as with <i>MPI_Isend</i> but with <i>MPI_Wait</i> and <i>MPI_Test</i> indicating whether the message has been copied into the sender's MPI buffer and hence is in transit.
<i>Ready</i>	<i>MPI_Rsend</i> : the call returns when the sender's application buffer can be reused (as with <i>MPI_Send</i>), but the programmer is also indicating to the library that the receiver is ready to receive the message, resulting in potential optimization of the underlying implementation.	<i>MPI_Irsend</i> : the effect is as with <i>MPI_Isend</i> , but as with <i>MPI_Rsend</i> , the programmer is indicating to the underlying implementation that the receiver is guaranteed to be ready to receive (resulting in the same optimizations),

5 Remote invocation

- The remote procedure call (RPC) approach extends the common programming abstraction of the procedure call to distributed environments, allowing a calling process to call a procedure in a remote node as if it is local.
- Remote method invocation (RMI) is similar to RPC but for distributed objects, with added benefits in terms of using object-oriented programming concepts in distributed systems and also extending the concept of an object reference to the global distributed environments, and allowing the use of object references as parameters in remote invocations.

Figure 5.1 Middleware layers



5.1 Introduction

1. Request-reply protocols
2. RPC
3. RMI – in 1990s – RMI extension allowing a local object to invoke methods of remote objects

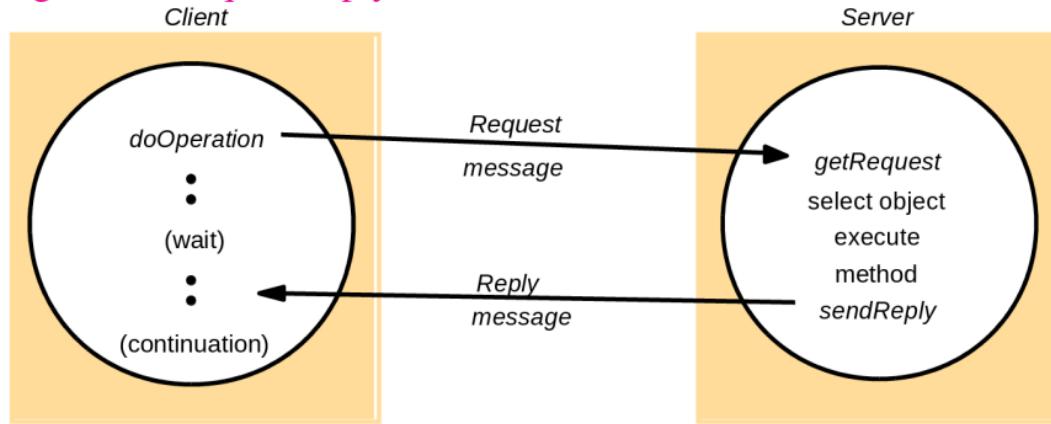
5.2 Request-reply protocols

- typical client-server interactions – request-reply communication is synchronous because the client process blocks until the reply arrives
- Asynchronous request-reply communication – an alternative that may be useful in situations where clients can afford to retrieve replies later

The request-reply protocol

doOperation, *getRequest* and *sendReply*

Figure 5.2 Request-reply communication



doOperation by clients to invoke remote op.; together with additional arguments; return a byte array. Marshaling and unmarshaling!

getRequest by server process to acquire service requests; followed by
sendReply send reply to the client

Figure 5.3 Operations of the request-reply protocol

public byte[] doOperation (RemoteRef s, int operationId, byte[] arguments)

sends a request message to the remote server and returns the reply.

The arguments specify the remote server, the operation to be invoked and the arguments of that operation.

public byte[] getRequest ();

acquires a client request via the server port.

public void sendReply (byte[] reply, InetAddress clientHost, int clientPort);

sends the reply message reply to the client at its Internet address and port.

Figure 5.4 Request-reply message structure

messageType	<i>int (0=Request, 1= Reply)</i>
requestId	<i>int</i>
remoteReference	<i>RemoteRef</i>
operationId	<i>int or Operation</i>
arguments	<i>array of bytes</i>

Message identifiers

1. *requestID* – increasing sequence of integers by the sender
2. server process identifier – e.g. internet address and port

Failure model of the request-reply protocol

A. UDP datagrams

communication failures (omission failures; sender order not guaranteed)
+ possible crash failures
action taken when a timeout occurs depends upon the delivery guarantees being offered

Timeouts – scenarios for a client behaviour

Discarding duplicate request messages – server filtering out duplicates

Lost reply messages

idempotent operation – an operation that can be performed repeatedly with the same effect as if it had been performed exactly once

History

retransmission by server ... problem with memory size ... ← can be cured by the knowledge that the message has arrived, e.g.:

clients can make only one request at a time ⇒ server can interpret each request as an acknowledgement of its previous reply!

Styles of exchange protocols Three different types of protocols (Spector [1982]):

- the request (R) protocol
 - No confirmation needed from server - client can continue right away – UDP-implementation

- the request-reply (RR) protocol
 - most client-server exchanges
- the request-reply-acknowledge reply (RRA) protocol

Figure 5.5 RPC exchange protocols

Name	<i>Messages sent by</i>		
	<i>Client</i>	<i>Server</i>	<i>Client</i>
R	<i>Request</i>		
RR	<i>Request</i>	<i>Reply</i>	
RRA	<i>Request</i>	<i>Reply</i>	<i>Acknowledge reply</i>

B. TCP streams to implement request-reply protocol

- TCP streams
 - transmission of arguments and results of any size
 - * flow-control mechanism
 - ⇒ no need for special measures to avoid overwhelming the recipient
 - request and reply messages are delivered reliably
 - * ⇒ no need for
 - retransmission
 - filtering of duplicates
 - histories

Example: HTTP request-reply protocol

fixed set of methods (GET, PUT, POST, etc)

In addition to invoking methods on web resources:

- *Content negotiation*: information – what data representations client can accept (e.g., language, media type)
- *Authentication*: Credentials and challenges to support password-style authentication
 - When a client receives a challenge, it gets the user to type a name and password and submits the associated credentials with subsequent requests

HTTP – implemented over TCP

Original version of the protocol – client-server interaction steps:

- The client requests and the server accepts a connection at the default server port or at a port specified in the URL

- The client sends a request message to the server
- The server sends a reply message to the client
- The connection is closed

Later version

- *persistent connections* – connections remain open over a series of request-reply exchanges
 - client may receive a message from the server saying that the connection is closed while it is in the middle of sending another request or requests
 - * browser will resend the requests without user involvement, provided that the operations involved are *idempotent* (like GET-method)
 - * otherwise – consult with the user
- Requests and replies are marshalled into messages as ASCII text strings, but

- resources can be represented as byte sequences and may be compressed
- Multipurpose Internet Mail Extensions (MIME) – RFC 2045 – standard for sending multipart data containing, for example, text, images and sound

HTTP methods

- **GET:** Requests the resource whose URL is given as its argument. If the URL refers to data, then the web server replies by returning the data identified
 - Arguments may be added to the URL; for example, GET can be used to send the contents of a form to a program as an argument
- **HEAD:** identical to GET, but does not return any data but instead, all the information about the data
- **POST:** data supplied in the body of the request, action may change data on the server

- **PUT:** Requests that the data supplied in the request is stored with the given URL as its identifier, either as a modification of an existing resource or as a new resource
- **DELETE:** deletes the resource identified by the given URL
- **OPTIONS:** server supplies the client with a list of methods it allows to be applied to the given URL (for example GET, HEAD, PUT) and its special requirements
- **TRACE:** The server sends back the request message. Used for diagnostic purposes

operations PUT and DELETE – idempotent, but POST is not necessarily

Message contents

Figure 5.6 HTTP *Request* message

<i>method</i>	<i>URL or pathname</i>	<i>HTTP version</i>	<i>headers</i>	<i>message body</i>
GET	//www.dcs.qmw.ac.uk/index.html	HTTP/ 1.1		

Figure 5.7 HTTP *Reply* message

<i>HTTP version</i>	<i>status code</i>	<i>reason</i>	<i>headers</i>	<i>message body</i>
HTTP/1.1	200	OK		resource data

5.3 Remote procedure call (RPC)

- Concept by Birrell and Nelson [1984]

5.3.1 Design issues for RPC

Three issues we will look:

- the style of programming promoted by RPC – programming with interfaces
- the call semantics associated with RPC
- the key issue of transparency and how it relates to remote procedure calls

Programming with interfaces

Interfaces in distributed systems: In a distributed program, the modules can run in separate processes

service interface – specification of the procedures offered by a server, defining the types of the arguments of each of the procedures

number of benefits to programming with interfaces in distributed systems (separation between interface and implementation):

- programmers are concerned only with the abstraction offered by the service interface and need not be aware of implementation details
- not need to know the programming language or underlying platform used to implement the service (heterogeneity)
- implementations can change as long as the interface (the external view) remains the same

Distributed nature of the underlying infrastructure:

- not possible for a client module running in one process to access the variables in a module in another process
- parameter-passing mechanisms used in local procedure calls (e.g., call by value; call by reference) – not suitable when the caller and procedure are in different processes

- parameters as input or output
- addresses cannot be passed as arguments or returned as results of calls to remote modules

Interface definition languages (IDLs)

designed to allow procedures implemented in different languages to invoke one another

- IDL provides a notation for defining interfaces in which each of the parameters of an operation may be described as for input or output in addition to having its type specified

Figure 5.8 CORBA IDL example

```
// In file Person.idl
struct Person {
    string name;
    string place;
    long year;
} ;
interface PersonList {
    readonly attribute string listname;
    void addPerson(in Person p) ;
    void getPerson(in string name, out Person p);
    long number();
};
```

RPC call semantics

doOperation implementations with different delivery guarantees:

- Retry request message
- Duplicate filtering
- Retransmission of results

Figure 5.9 Call semantics

Fault tolerance measures			Call semantics
Retransmit request message	Duplicate filtering	Re-execute procedure or retransmit reply	
No	Not applicable	Not applicable	Maybe
Yes	No	Re-execute procedure	At-least-once
Yes	Yes	Retransmit reply	At-most-once

Maybe semantics – remote procedure call may be executed once or not at all

- when no fault-tolerance measures applied, can suffer from
 - omission failures (the request or result message lost)
 - crash failures

At-least-once semantics – can be achieved by retransmission of request messages

- types of failures
 - crash failures when the server containing the remote procedure fails
 - arbitrary failures – in cases when the request message is retransmitted, the remote server may receive it and execute the procedure more than once, possibly causing wrong values stored or returned
 - If the operations in a server can be designed so that all of the procedures in their service interfaces are idempotent operations, then at-least-once call semantics may be acceptable

At-most-once semantics – caller receives either a result or an exception

Transparency

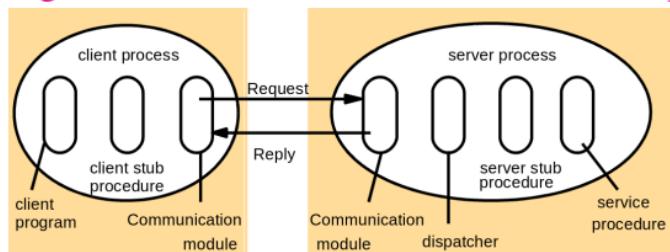
at least location and access transparency

consensus is that remote calls should be made transparent in the sense that the syntax of a remote call is the same as that of a local invocation, but that the difference between local and remote calls should be expressed in their interfaces

End of week 5

5.3.2 Implementation of RPC

Figure 5.10 Role of client and server stub procedures in RPC



stub procedure behaves like a local procedure to the client, but instead of executing the call, it marshals the procedure identifier and the arguments into a request message, which it sends via its communication module to the server

- RPC generally implemented over request-reply protocol
- general choices:
 - *at-least-once* or
 - *at-most-once*

5.3.3 Case study: Sun RPC

- designed for client-server communication in Sun Network File System (NFS)
- interface language called XDR
 - instead of interface names – program number (obtained from central authority) and a version number

- procedure definition specifies a procedure signature and a procedure number
- single input parameter

Figure 5.11 Files interface in Sun XDR

```
const MAX = 1000;
typedef int FileIdentifier;
typedef int FilePointer;
typedef int Length;
struct Data {
    int length;
    char buffer[MAX];
};
struct writeargs {
    FileIdentifier f;
    FilePointer position;
    Data data;
}; //...
```

```
// ... continued:
struct readargs {
    FileIdentifier f;
    FilePointer position;
    Length length;
};

program FILEREADWRITE {
    version VERSION {
        void WRITE(writeargs)=1;      // 1
        Data READ(readargs)=2;        // 2
    }=2; // version number = 2
} = 9999; // program number = 999
```

- interface compiler *rpcgen* can be used to generate the following from an interface definition:
 - client stub procedures
 - server main procedure, dispatcher and server stub procedures
 - XDR marshalling and unmarshalling procedures for use by the dispatcher and client and server stub procedures

Further on Sun RPC: <http://www.cdk5.net/rmi>

5.4 Remote method invocation (RMI)

Remote method invocation (RMI) closely related to RPC but extended into the world of distributed objects

- a calling object can invoke a method in a potentially remote object. As with RPC, the underlying details are generally hidden from the user

Similarities between RMI and RPC, they both:

- support programming with interfaces
- typically constructed on top of request-reply protocols
- can offer a range of call semantics, such as
 - *at-least-once*
 - *at-most-once*
- similar level of transparency –

- local and remote calls employ the same syntax
- remote interfaces
 - * typically expose the distributed nature of the underlying call e.g. supporting remote exceptions

RMI added expressiveness for programming of complex distributed applications and services:

- full expressive power of object-oriented programming
 - use of objects, classes and inheritance
 - object-oriented design methodologies and associated tools
- all objects in an RMI-based system have unique object references (independent of they are local or remote)
 - object references can also be passed as parameters ⇒ offering significantly richer parameter-passing semantics than in RPC

5.4.1 Design issues for RMI

Transition from *objects* to *distributed objects*

The object model

some languages allow accessing object instance variables directly (C++, Java) – in distributed object system, object's data can be accessed only with the help of its methods

Object references: to invoke a method object's reference and method name are given

Interfaces: definition of the signatures of a set of methods without their implementation

Actions: initiated by an object invoking a method in another object

three effects of invocation of a method:

1. The state of the receiver may be changed

2. A new object may be instantiated, for example, by using a constructor in Java or C++
3. Further invocations on methods in other objects may take place

Exceptions: a block of code may be defined to *throw* an exception; another block *catches* the exception

Garbage collection: ...Java vs C++ case...

Distributed objects

Distributed object systems – different possible architectures

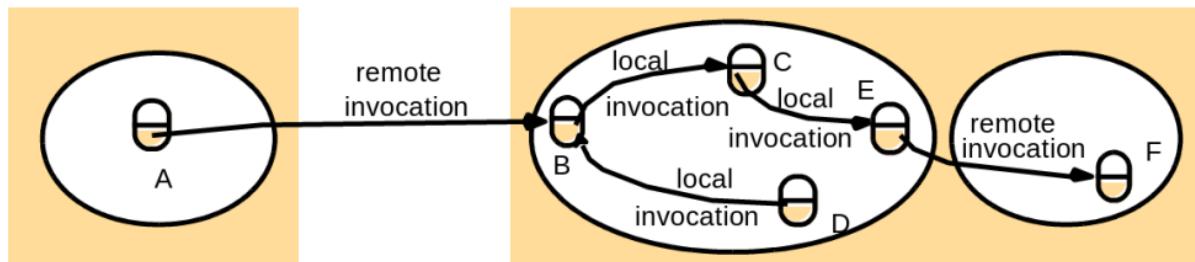
- client-server architecture ... but also possibly:
- replicated objects – for enhanced performance and fault-tolerance
- migrated objects – enhanced availability and performance

The distributed object model

Each process contains a collection of objects

objects that can receive remote invocations – *remote objects*

Figure 5.12 Remote and local method invocations

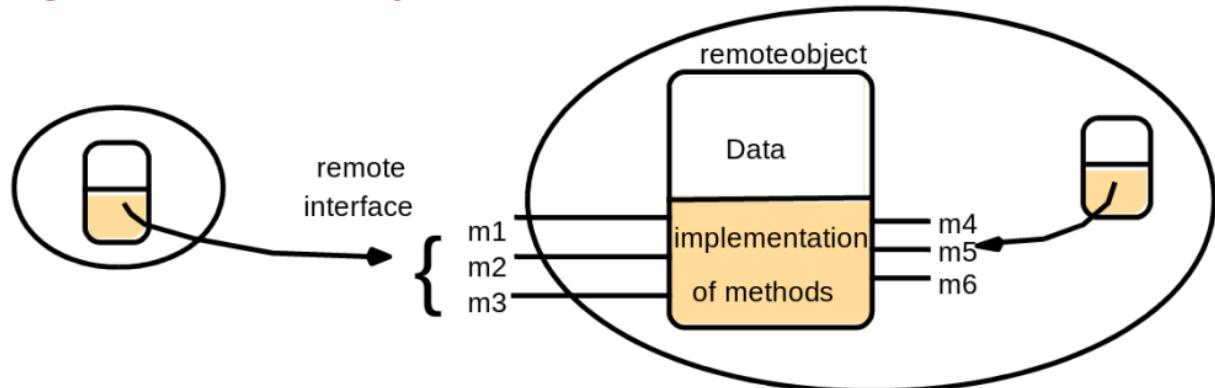


Remote object reference: identifier that can be used throughout a distributed system to refer to a particular unique remote object

- Remote object references may be passed as arguments and results of remote method invocations

Remote interfaces: which of the object methods can be invoked remotely

Figure 5.12 A remote object and its remote interface



- CORBA interface definition language (IDL)
- Java RMI – keyword: *Remote*

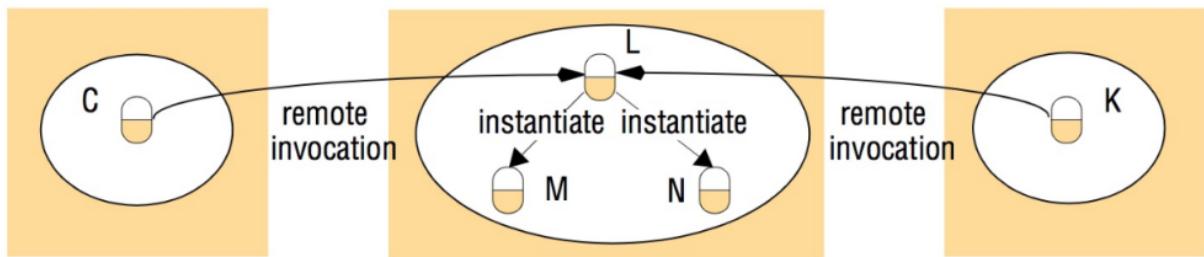
NB! Remote interfaces cannot contain constructors!

Actions in a distributed object system

- remote reference of the object must be available to the invoker

Remote object references may be obtained as the results of remote method invocations

Figure 5.14 Instantiation of remote objects



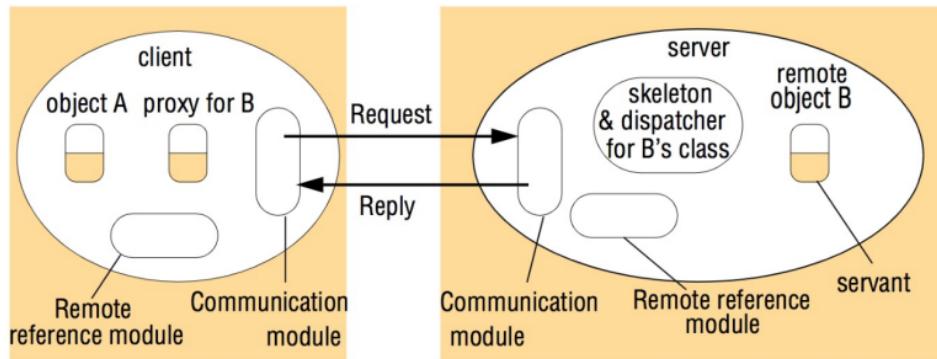
Garbage collection in a distributed-object system:

if garbage collection supported by the language (e.g. Java) – also RMI should allow it + a module for distributed reference counting

Exceptions: usual exceptions + e.g. timeouts

5.4.2 Implementation of RMI

Figure 5.15 The role of proxy and skeleton in remote method invocation



We will discuss:

- What are the roles of each of the components?
- What are communication and remote reference modules?
- What is the role of RMI software that runs over them?
- What is generation of proxies and why is it needed?
- What is binding of names to their remote object references?
- What is the activation and passivation of objects?

Communication module

- responsible for transferring *request* and *reply* messages between the client and server
- uses only 3 fields of the messages: *message type*, *requestId* and *remote reference* (Fig. 5.4)

communication modules are together responsible for providing a specified invocation semantics, for example *at-most-once*

Remote reference module

- responsible for translating between local and remote object references and for creating remote object references

using *remote object table* – correspondence between local object references in that process and remote object references

- An entry for all the remote objects held by the process
- An entry for each local proxy

Actions of the remote reference module:

- When a remote object is to be passed as an argument or a result for the first time, the remote reference module creates a remote object reference, and adds it to its table
- When a remote object reference arrives in a request or reply message, the remote reference module is asked for the corresponding local object reference, which may refer either to a proxy or to a remote object
 - In the case that the remote object reference is not in the table, the RMI software creates a new proxy and asks the remote reference module to add it to the table

Servants

- instance of a class providing the body of a remote object
 - handles the remote requests passed on by the corresponding skeleton

- living within a server process
- created when remote objects instantiated
- remain in use until they are no longer needed (finally being garbage collected or deleted)

The RMI software

Proxy: making remote method invocation transparent to clients – behaving like a local object to the invoker

- forwards invocation in a message to a remote object
- hides the details of:
 - remote object reference
 - marshalling of arguments, unmarshalling of results

- sending and receiving of messages from the client
- just one proxy for each remote object for which a process holds a remote object reference
- implements:
 - the methods in the remote interface of the remote object it represents
 - each method of the proxy marshals:
 - * a reference to the target object
 - * its own *operationId* and its arguments
 - ... into a request message and sends it to the target
- then awaits the reply message
 - unmarshals it and returns the results to the invoker

server has one dispatcher and one skeleton for each class representing a remote object

Dispatcher: receives request messages from the communication module

- uses the *operationId* to select the appropriate method in the skeleton, passing on the request message

Skeleton: implements the methods in the remote interface

- unmarshals the arguments in the request message
- invokes the corresponding method in the servant
- waits for the invocation to complete
- marshals the result (together with any exceptions in a reply message to the sending proxy's method)

Generation of the classes for proxies, dispatchers and skeletons

- generated automatically by an interface compiler

Dynamic invocation: An alternative to proxies

- useful in applications where some of the interfaces of the remote objects cannot be predicted at design time
 - dynamic downloading of classes to clients (available in Java RMI) – an alternative to dynamic invocation
 - Dynamic skeletons
 - Java RMI generic dispatcher and the dynamic downloading of classes to the server
 - (book Chapter 8 on CORBA)

Server and client programs

Server program : classes for

- dispatchers, skeletons, supported servants +

- initialization section
 - creating and initializing at least one of the hosted servants, which can be used to access the rest
 - may also register some of its servants with a binder

Client program: classes for proxies for all of the remote objects that it will invoke

- can use a binder to look up remote object references

Factory methods:

remote object interfaces cannot include constructors ⇒ servants cannot be created this way

- Servants created either in
 - the initialization section or by
 - factory methods – methods that create servants

- factory object – an object with factory methods

Any remote object that needs to be able to create new remote objects on demand for clients must provide methods in its remote interface for this purpose.

→ Such methods are called **factory methods**

The binder in a distributed system

binder – a separate service that maintains a table containing mappings from textual names to remote object references

- binder used by:
 - servers to register their remote objects by name
 - clients to look them up

The Java binder – RMIClient, see case study on Java RMI in Section 5.5

Server threads

- each remote invocation executed on a separate thread – (to avoid blocking)
 - ... programmer has to take it into account...

Activation of remote objects

active-passive modes of service objects – to economise on resources

- *active* object - available for invocation
- *passive* object -
 1. the implementation of its methods
 2. its state in the marshalled form

Activation: creating an active object from the corresponding passive object by

- creating a new instance of its class
- initializing its instance variables from the stored state

An *activator* is responsible for:

- registering passive objects that are available for activation (involves recording the names of servers against the URLs or file names of the corresponding passive objects)
- starting named server processes and activating remote objects in them
- keeping track of the locations of the servers for remote objects that it has already activated
- Java RMI – the ability to make remote objects activatable [java.sun.com IX]
 - uses one activator on each server computer

- CORBA case study in Chapter 8 describes the implementation repository
 - a weak form of activator that starts services containing objects in an initial state

Persistent object stores

An object that is guaranteed to live between activations of processes is called a persistent object

- generally managed by persistent object stores, which store their state in a marshalled form on disk

Object location

remote object reference – Internet address and port number of the process that created the remote object – to guarantee uniqueness

some remote objects exist in series of different processes, possibly on different computers, throughout their lifetime

location service – helping clients to locate remote objects from their remote object references

- using database: remote object reference → probable current location

5.4.3 Distributed garbage collection

Java distributed garbage collection algorithm

- server keeping track, which of its objects are proxied at which clients
 - protocol for creation and removal of proxies with notifications to the server
- based on no client proxies to an object exist and no local references either, garbage collection can remove the object at the server

(more details in TextBook Section 5.4.3)

Leases in Jini

- references to a certain object are *leased* to other (outside) processes
- leases have a certain pre-negotiated time period
- before the lease is about to expire, the client must request a renewal if needed

5.5 Case study: Java RMI

Example: *shared whiteboard*

www.cdk5.net/rmi

Remote interfaces in Java RMI

- extending an interface *Remote* in *java.rmi* package
- must throw *RemoteException*

Figure 5.16 Java Remote interfaces Shape and ShapeList

```
1 import java.rmi.*;
2 import java.util.Vector;
3 public interface Shape extends Remote { // i.e. Shape is a remote interface
4     int getVersion() throws RemoteException;
5     GraphicalObject getAllState() throws RemoteException; // 1
6 }
7 public interface ShapeList extends Remote {
8     Shape newShape(GraphicalObject g) throws RemoteException; // 2
9     Vector allShapes() throws RemoteException;
10    int getVersion() throws RemoteException;
```

11}

Parameter and result passing

In Java RMI:

- parameters of a method – *input* parameters
- result of a method – single *output* parameter

Any object that is serializable – implements the *Serializable* interface – can be passed as an argument or result in Java RMI.

- All primitive types and remote objects are serializable

Passing remote objects: When the type of a parameter or result value is defined as a remote interface, the corresponding argument or result is always passed as a remote object reference

Passing non-remote objects: All serializable non-remote objects are copied and passed by value

The arguments and return values in a remote invocation are serialized to a stream using the method described in Section 4.3.2, with the following modifications:

1. Whenever an object that implements the Remote interface is serialized, it is replaced by its remote object reference, which contains the name of its (the remote object's) class
2. When any object is serialized, its class information is annotated with the location of the class (as a URL), enabling the class to be downloaded by the receiver

Downloading of classes

- If the recipient does not already possess the class of an object passed by value, its code is downloaded automatically
- if the recipient of a remote object reference does not already possess the class for a proxy, its code is downloaded automatically

advantages:

1. There is no need for every user to keep the same set of classes in their working environment
2. Both client and server programs can make transparent use of instances of new classes whenever they are added

RMIregistry

- binder for Java RMI
 - on every server computer that hosts remote objects
 - maintains a table mapping textual, URL-style names to references to remote objects hosted on that computer
 - accessed by methods of the Naming class
 - methods take as an argument a URL-formatted string of the form:

```
// computerName : port / objectName
```

Figure 5.17 The Naming class of Java RMIServer

void rebind (String name, Remote obj)

This method is used by a server to register the identifier of a remote object by name, as shown in Figure 15.18, line 4.

void bind (String name, Remote obj)

This method can alternatively be used by a server to register a remote object by name, but if the name is already bound to a remote object reference an exception is thrown.

void unbind (String name, Remote obj)

This method removes a binding.

Remote lookup(String name)

This method is used by clients to look up a remote object by name, as shown in Figure 5.20 line 1. A remote object reference is returned.

String [] list()

This method returns an array of Strings containing the names bound in the registry.

5.5.1 Building client and server programs

Server program

Figure 5.18 Java class ShapeListServer with main method

```
1 import java.rmi.*;
2 import java.rmi.server.UnicastRemoteObject;
3 public class ShapeListServer{
4     public static void main(String args[]){
5         System.setSecurityManager(new RMISecurityManager());
6         try {
7             ShapeList aShapeList = new ShapeListServant(); // 1
8             ShapeList stub = // 2
9                 (ShapeList) UnicastRemoteObject.exportObject(aShapeList,0); // 3
10            Naming.rebind("//bruno.ShapeList", stub); // 4
11            System.out.println("ShapeList_server_ready");
12        } catch(Exception e) {
13            System.out.println("ShapeList_server_main_" + e.getMessage());
14        }
15    }
}
```

Figure 5.19 Java class ShapeListServant implements interface ShapeList

```
1 import java.util.Vector;
2 public class ShapeListServant implements ShapeList {
3     private Vector theList; // contains the list of Shapes
4     private int version;
5     public ShapeListServant() {...}
6     public Shape newShape(GraphicalObject g) {           // 1
7         version++;
8         Shape s = new ShapeServant( g, version);          // 2
9         theList.addElement(s);
10        return s;
11    }
12    public Vector allShapes() {...}
13    public int getVersion() { ... }
14 }
```

Client program

Figure 5.20 Java client ShapeList

```
1 import java.rmi.*;
2 import java.rmi.server.*;
3 import java.util.Vector;
4 public class ShapeListClient{
5     public static void main(String args[]){
6         System.setSecurityManager(new RMISecurityManager());
7         ShapeList aShapeList = null;
8         try{
9             aShapeList = (ShapeList) Naming.lookup("//bruno.ShapeList"); // 1
10            Vector sList = aShapeList.allShapes(); // 2
11        } catch(RemoteException e) {System.out.println(e.getMessage());}
12        }catch(Exception e) {System.out.println("Client:" + e.getMessage());}
13    }
14 }
```

Callbacks

server should inform its clients whenever certain event occurs

callback – server's action of notifying clients about an event

- client creates a remote object – *callback object* – that implements an interface containing a method for the server to call
- server provides an operation allowing interested clients to inform it of the remote object references of their *callback objects*
- Whenever an event of interest occurs, the server calls the interested clients

Problems with polling solved, but at the same time, attention is needed because:

- server needs to have up-to-date lists of the clients' callback objects, but clients may not always inform the server before they exit, leaving the server with incorrect lists

- leasing technique can be used to overcome this problem
- server needs to make a series of synchronous RMIs to the *callback objects* in the list
- TextBook Chapter 6 gives some ideas on solving this issue

⇒ WhiteboardCallback interface could be defined as:

```
public interface WhiteboardCallback implements Remote {  
    void callback(int version) throws RemoteException;  
};
```

- implemented as a remote object by the client
 - client needs to inform the server about its callback object

ShapeList interface requires additional methods such as register and deregister, defined as follows:

```
int register(WhiteboardCallback callback) throws RemoteException;  
void deregister(int callbackId) throws RemoteException;
```

5.5.2 Design and implementation of Java RMI

Use of reflection

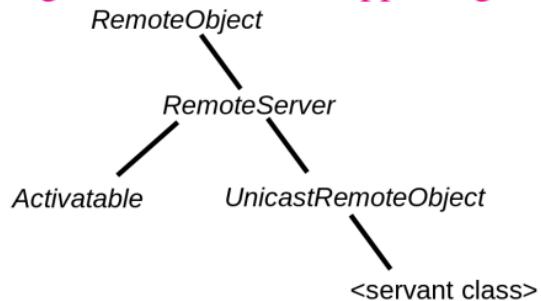
Reflection used to pass information in request messages about the method to be invoked.

- with the help of the Method class in reflection package

Java classes supporting RMI

Inheritance structure of the classes supporting Java RMI servers:

Figure 5.21 Classes supporting Java RMI



End of week 6

6 Indirect communication

6.1 Introduction

Roger Needham, Maurice Wilkes and David Wheeler:

“All problems in computer science can be solved by another level of indirection”

Indirect communication – communication between entities in a distributed system through an intermediary with no direct coupling between the sender and the receiver(s)

2 key properties stemming from the use of an intermediary:

1. *Space uncoupling*

- the sender does not know or need to know the identity of the receiver(s)
- participants (senders or receivers) can be replaced, updated, replicated or migrated

2. *Time uncoupling*

- the sender and receiver(s) can have independent lifetimes
 - ⇒ more volatile environments where senders and receivers may come and go

Figure 6.1 Space and time coupling in distributed systems

	Time-coupled	Time-uncoupled
Space coupling	<p><i>Properties:</i> Communication directed towards a given receiver or receivers; receiver(s) must exist at that moment in time</p> <p><i>Examples:</i> Message passing, remote invocation (see Chapters 4 and 5)</p>	<p><i>Properties:</i> Communication directed towards a given receiver or receivers; sender(s) and receiver(s) can have independent lifetimes</p> <p><i>Examples:</i> See Exercise 6.3</p>
Space uncoupling	<p><i>Properties:</i> Sender does not need to know the identity of the receiver(s); receiver(s) must exist at that moment in time</p> <p><i>Examples:</i> IP multicast (see Chapter 4)</p>	<p><i>Properties:</i> Sender does not need to know the identity of the receiver(s); sender(s) and receiver(s) can have independent lifetimes</p> <p><i>Examples:</i> Most indirect communication paradigms covered in this chapter</p>

The relationship with asynchronous communication

- In asynchronous communication, a sender sends a message and then continues (without blocking) ⇒ no need to meet in time with the receiver to communicate
- Time uncoupling adds the extra dimension that the sender and receiver(s) can have independent existences

6.2 Group communication

Group communication – a message is sent to a group → message is delivered to all members of the group

- the sender is not aware of the identities of the receivers

Key areas of application:

- the reliable dissemination of information to potentially large numbers of clients
- support for collaborative applications

- support for a range of fault-tolerance strategies, including the consistent update of replicated data
- support for system monitoring and management

JGroups toolkit

6.2.1 The programming model

group & group membership ← processes may join or leave the group

`aGroup.send(aMessage))`

process groups

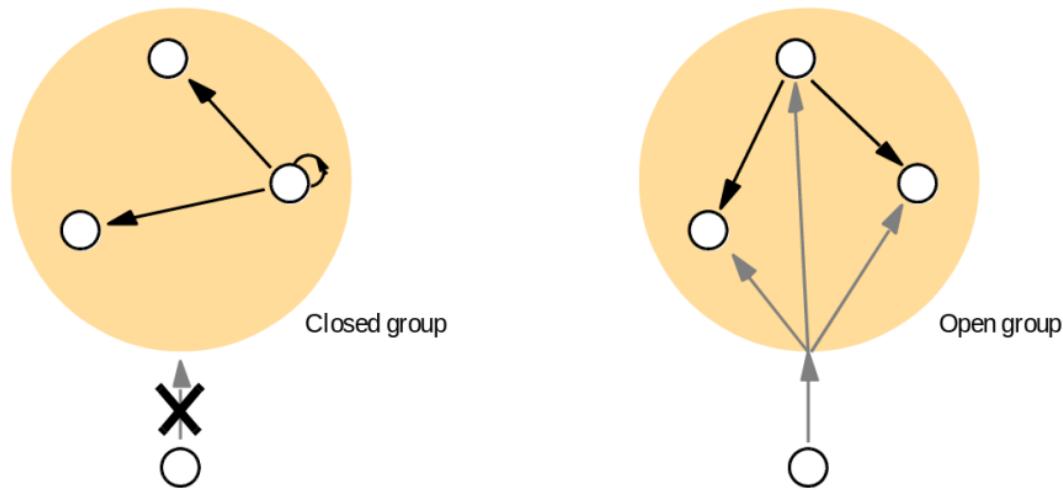
- e.g. RPC

object groups

- marshalling and dispatching as in RMI
- Electra – CORBA-compliant system supporting object groups

closed and open groups

Figure 6.2 Open and closed groups



overlapping and non-overlapping groups

synchronous and asynchronous systems

6.2.2 Implementation issues

Reliability and ordering in multicast

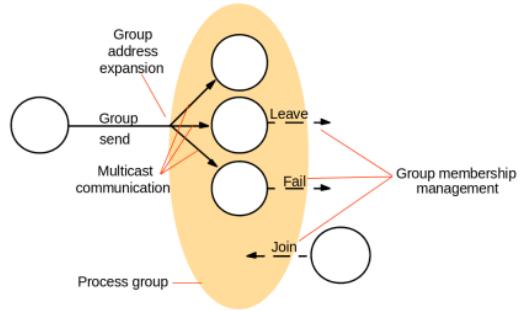
integrity, validity + agreement

ordered multicast possibilities (hybrid solutions also possible) :

- FIFO ordering: First-in-first-out (FIFO) (or source ordering) – if sender sends one before the other, it will be delivered in this order at all group processes
- Casual ordering: – if a message happens before another message in the distributed system, this so-called causal relationship will be preserved in the delivery of the associated messages at all processes
- Total ordering: – if a message is delivered before another message at one process, the same order will be preserved at all processes

Group membership management

Figure 6.3 The role of group membership management



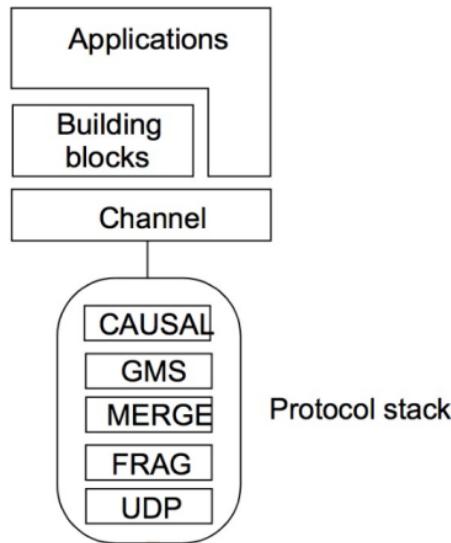
- Providing an interface for group membership changes
- Failure detection
- Notifying members of group membership changes
- Performing group address expansion

IP multicast as a weak case of a group membership service

- IP multicast itself does not provide group members with information about current membership delivery; is not coordinated with membership changes

6.2.3 Case study: the JGroups toolkit

Figure 6.4 The architecture of JGroups



- **Channel** – acts as a handle onto a group

core functions of joining, leaving, sending and receiving

- connect – to a particular named group
if the named group does not exist, it is implicitly created at the time of the first connect
- disconnect – to leave a group
- getView – returns the current member list
- getState – historical application state associated with the group

Figure 6.5 Java class FireAlarmJG

```
1 import org.jgroups.JChannel;
2 public class FireAlarmJG {
3     public void raise() {
4         try {
5             JChannel channel = new JChannel();
6             channel.connect("AlarmChannel");
7             Message msg = new Message(null, null, "Fire!"); // destination, source, payload
8             // destination = null — distribute to whole group; source null — source
9             // added automatically by the system anyway
10            channel.send(msg);
11        }
12        catch(Exception e) {
13        }
14    }
}
```

```
FireAlarmJG alarm = new FireAlarmJG(); // create a new instance of the FireAlarmJG class
alarm.raise(); // raise an alarm
```

Figure 6.6 Java class FireAlarmConsumerJG

```
1 import org.jgroups.JChannel;
2 public class FireAlarmConsumerJG {
3     public String await() {
4         try {
5             JChannel channel = new JChannel();
6             channel.connect("AlarmChannel");
7             Message msg = (Message) channel.receive(0);
8             // parameter: timeout zero — the receive message will block until a message is received
9             // incoming messages are buffered and receive returns the top element in the buffer
10            return (String) msg.GetObject();
11        }
12        catch(Exception e) {
13            return null;
14        }
15    }
16}
```

```
FireAlarmConsumerJG alarmCall = new FireAlarmConsumerJG(); // (...receiver code...)
String msg = alarmCall.await();
System.out.println("Alarm_received:" + msg);
```

- **Building blocks**

- higher-level abstractions, building on the underlying service offered by channels

- *MessageDispatcher*

- * e.g. `castMessage` method that sends a message to a group and blocks until a specified number of replies are received

- *RpcDispatcher* – invokes specified method on all objects associated with a group

- *NotificationBus* – implementation of a distributed event bus, in which an event is any serializable Java object

- **The protocol stack**

- underlying communication protocol, constructed as a stack of composable protocol layers

bidirectional stack of protocol layers

```
public Object up (Event evt);  
public Object down (Event evt);
```

- UDP most common transport layer in JGroups (IP multicast for sending to all members in a group; TCP layer may be preferred; PING for membership discovery etc.)
- FRAG – message packetization to maximum message size (8,192 bytes by default)
- MERGE – unexpected network partitioning and the subsequent merging of subgroups after the partition
- GMS implements a group membership protocol to maintain consistent views of membership across the group

- CAUSAL implements causal ordering (Section 6.2.2 Chapter 15)

6.3 Publish-subscribe systems

also referred to as *distributed event-based* systems

- publishers publish structured events to an event service and subscribers express interest in particular events through subscriptions which can be arbitrary patterns over the structured events
- event notifications
- one-to-many communications paradigm

Applications of publish-subscribe systems

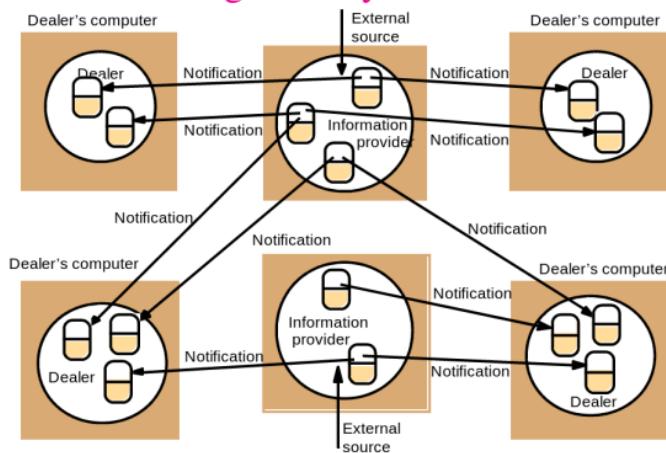
application domains needing large-scale dissemination of events

Examples:

- financial information systems
- other areas with live feeds of real-time data (including RSS feeds)

- support for cooperative working, where a number of participants need to be informed of events of shared interest
- support for ubiquitous computing, including the management of events emanating from the ubiquitous infrastructure (for example, location events)
- a broad set of monitoring applications, including network monitoring in the Internet

Figure 6.7 Dealing room system

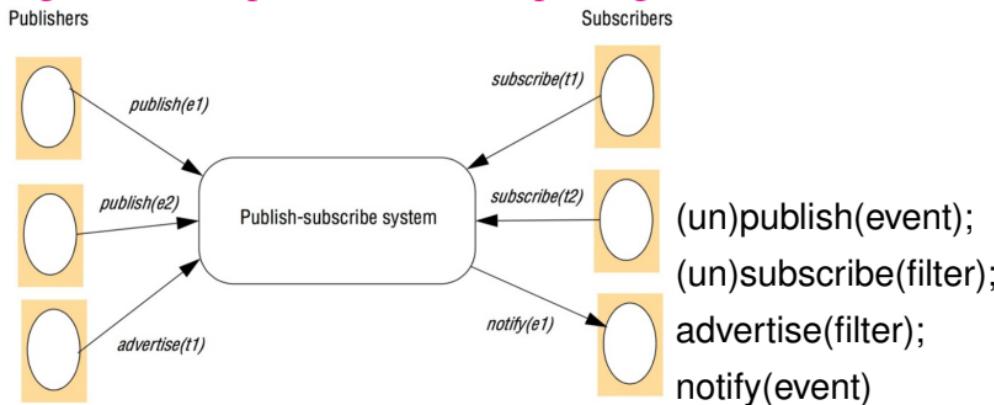


Characteristics of publish-subscribe systems

- *Heterogeneity*
- *Asynchronicity*
- *different delivery guarantees*

6.3.1 The programming model

Figure 6.8 The publish-subscribe paradigm



Expressiveness of publish-subscribe system determined by the **subscription (filter) model**:

- ***Channel-based***

- publishers publish events to named channels
 - subscribers then subscribe to one of these named channels to receive all events sent to that channel
 - * CORBA Event Service (see Chapter 8)

- ***Topic-based*** (also referred to as subject-based):

- each notification is expressed in terms of a number of fields, with one field denoting the topic
 - Subscription defined in terms of topic of interest

- ***Content-based***

- generalization of topic-based approaches allowing the expression of subscriptions over a range of fields in an event notification
- **Type-based**
 - subscriptions defined in terms of types of events
 - matching is defined in terms of types or subtypes of the given filter
- + *concept-based subscription models*
 - filters are expressed in terms of the semantics as well as the syntax of events
- + *complex event processing (or composite event detection)*
 - allows the specification of patterns of events as they occur in the distributed environment

6.3.2 Implementation issues

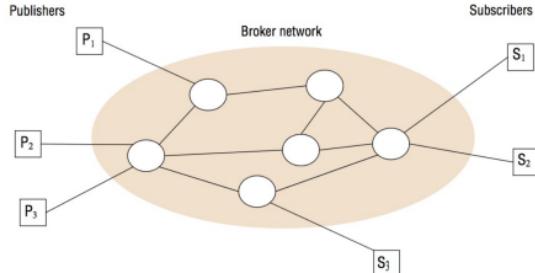
task of a publish-subscribe system: ensure that events are delivered efficiently to all subscribers that have filters defined that match the event

additional requirements in terms of security, scalability, failure handling, concurrency and quality of service

Centralized versus distributed implementations

centralised broker vs. network of brokers

Figure 6.9 A network of brokers

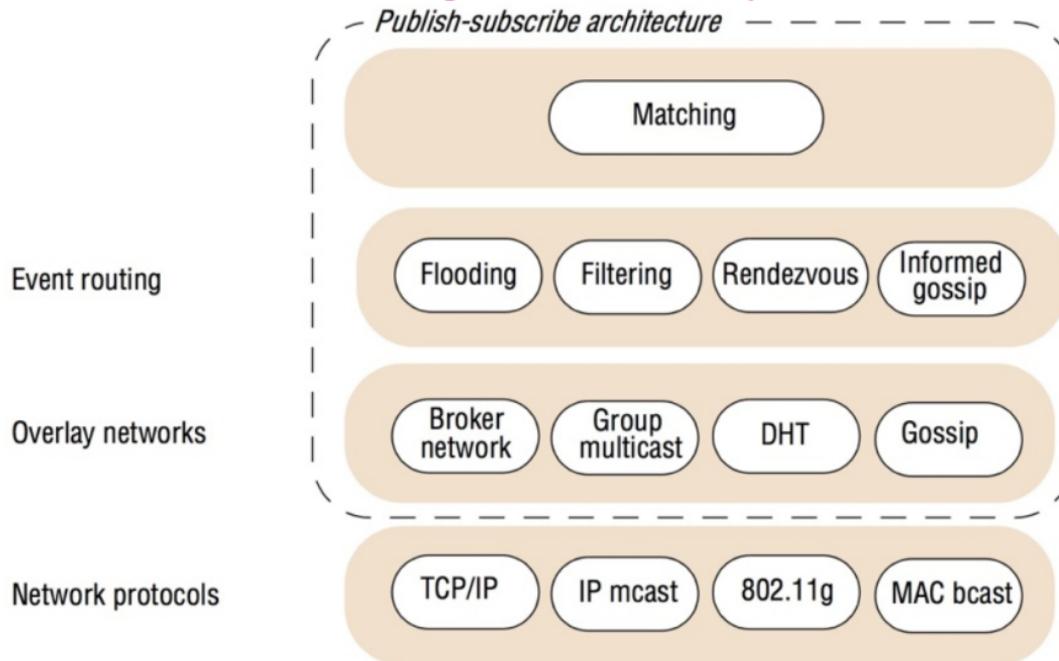


A step further:

fully peer-to-peer implementation of a publish-subscribe system – no distinction between publishers, subscribers and brokers; all nodes act as brokers, cooperatively implementing the required event routing functionality

Overall systems architecture

Figure 6.10 The architecture of publish-subscribe systems



Implementation approaches:

- ***Flooding:***

- sending an event notification to all nodes in the network and then carrying out the appropriate matching at the subscriber end
- alternative – send subscriptions back to all possible publishers, with the matching carried out at the publishing end
- can be implemented
 - * using an underlying broadcast or multicast facility
 - * brokers can be arranged in an acyclic graph in which each forwards incoming event notifications to all its neighbours
- benefit of simplicity but can result in a lot of unnecessary network traffic

- ***Filtering*** (filtering-based routing)

- Brokers forward notifications through the network only where there is a path to a valid subscriber

- each node must maintain
 - * neighbours list containing a list of all connected neighbours in the network of brokers
 - * subscription list containing a list of all directly connected subscribers serviced by this node
 - * routing table

Figure 6.11 Filtering-based routing

```
1 upon receive publish(event e) from node x
2     matchlist := match(e, subscriptions)
3     send notify(e) to matchlist;
4     fwdlist := match(e, routing);
5     send publish(e) to fwdlist - x;
6 upon receive subscribe(subscription s) from node x
7     if x is client then
8         add x to subscriptions;
9     else add(x, s) to routing;
10    send subscribe(s) to neighbours - x;
```

- * subscriptions essentially using a flooding approach back towards all possible publishers
- **Advertisements:** propagating the advertisements towards subscribers in a similar (actually, symmetrical) way to the propagation of subscriptions
- **Rendezvous:** rendezvous nodes, which are broker nodes responsible for a given subset of the event space
 - $SN(s)$ – given subscription, $s \rightarrow$ one or more rendezvous nodes that take responsibility for that subscription
 - $EN(e)$ – given event $e \rightarrow$ one or more rendezvous nodes responsible for matching e against subscriptions in the system

Figure 6.12 Rendezvous-based routing

```
1 upon receive publish(event e) from node x at node i
2     rvlist := EN(e);
3     if i in rvlist then begin
4         matchlist <- match(e, subscriptions);
5         send notify(e) to matchlist;
6     end
7     send publish(e) to rvlist - i;
8 upon receive subscribe(subscription s) from node x at node i
9     rvlist := SN(s);
10    if i in rvlist then
11        add s to subscriptions;
12    else
13        send subscribe(s) to rvlist - i;
```

- distributed hash table (DHT) – can be used
 - hash table distributed over a set of nodes in P2P manner

6.3.3 Examples of publish-subscribe systems

Figure 6.13 Example publish-subscribe systems

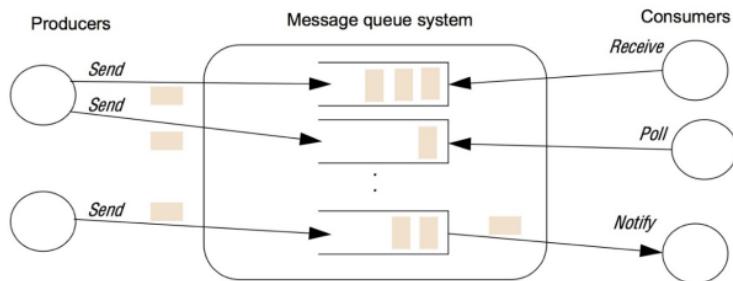
<i>System (and further reading)</i>	<i>Subscription model</i>	<i>Distribution model</i>	<i>Event routing</i>
CORBA Event Service (Chapter 8)	Channel-based	Centralized	-
TIB Rendezvous [Oki <i>et al.</i> 1993]	Topic-based	Distributed	Filtering
Scribe [Castro <i>et al.</i> 2002b]	Topic-based	Peer-to-peer (DHT)	Rendezvous
TERA [Baldoni <i>et al.</i> 2007]	Topic-based	Peer-to-peer	Informed gossip
Siena [Carzaniga <i>et al.</i> 2001]	Content-based	Distributed	Filtering
Gryphon [www.research.ibm.com]	Content-based	Distributed	Filtering
Hermes [Pietzuch and Bacon 2002]	Topic- and content-based	Distributed	Rendezvous and filtering
MEDYM [Cao and Singh 2005]	Content-based	Distributed	Flooding
Meghdoot [Gupta <i>et al.</i> 2004]	Content-based	Peer-to-peer	Rendezvous
Structure-less CBR [Baldoni <i>et al.</i> 2005]	Content-based	Peer-to-peer	Informed gossip

6.4 Message queues

6.4.1 The programming model

- Types of receive operations:
 - blocking receive
 - non-blocking receive
 - notify operation

Figure 6.14 The message queue paradigm



- A number of processes can send messages to the same queue
- a number of receivers can remove messages from a queue
- queuing policy
 - (normally) first-in-first-out (FIFO) but most message queue implementations also support the
 - concept of priority
 - * higher-priority messages delivered first
- Consumer processes can select messages from the queue based on message properties
 - destination (a unique identifier designating the destination queue)
 - metadata associated with the message
 - * priority of the message

- * the delivery mode
 - * body of the message (though body – normally opaque and untouched by the message queue system)
-
- message content serialized
 - length of a message varying (can be 100s megabytes...)

messages are persistent – system preserves messages indefinitely (or until they are consumed)

also system can commit messages to disk – for reliable delivery:

- any message sent is eventually received (validity)
- the message received is identical to the one sent, and no messages are delivered twice (integrity)

can also support additional functionality:

- support for the sending or receiving of a message to be contained within a transaction (all or nothing)
- support for message transformation (e.g. in heterogeneous environments)
- support for security

difference with message-passing systems (MPS):

- MPS have implicit queues associated with senders and receivers (for example, the message buffers in MPI),

message queuing systems have explicit queues that are third-party entities, separate from the sender and the receiver – making it into indirect communication paradigm with the crucial properties of space and time uncoupling

6.4.2 Implementation issues

Case study: WebSphere MQ (textbook pp.272-274)

6.4.3 Case study: The Java Messaging Service (JMS)

JMS – specification of a standardized way for distributed Java programs to communicate indirectly

- unifies the publish-subscribe and message queue paradigms at least superficially by supporting topics and queues as alternative destinations of messages

implementations:

Joram from OW2

Java Messaging from

JBoss

Sun's Open MQ

Apache ActiveMQ

OpenJMS

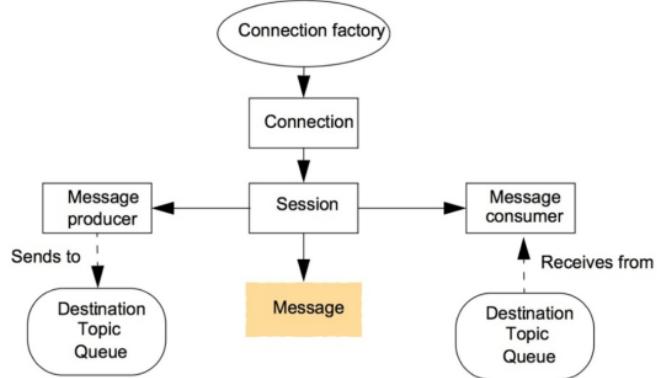
WebSphere MQ provides a JMS interface

Key roles:

- **JMS client** – Java program or component that produces or consumes messages
 - JMS producer – program that creates and produces messages
 - JMS consumer – program that receives and consumes messages
- **JMS provider** – any of the multiple systems that implement the JMS specification
- **JMS message** – object that is used to communicate information between JMS clients (from producers to consumers)
- **JMS destination** – object supporting indirect communication in JMS – either:
 - JMS topic
 - JMS queue

Programming with JMS

Figure 6.16 The programming model offered by JMS



- two types of connection can be established:
 - TopicConnection
 - QueueConnection

Connections can be used to create one or more sessions

- session – series of operations involving the creation, production and consumption of messages related to a logical task
- session object also supports operations to create transactions, supporting all-or-nothing execution of a series of operations
- TopicConnection can support one or more topic sessions
- QueueConnection can support one or more queue sessions (but it is not possible to mix session styles)

session object – central to the operation of JMS – methods for creation of messages, message producers and message consumers:

- ***message*** consists of three parts:

- **header**

- * destination – reference to:

- topic
 - queue
 - * priority
 - * expiration date
 - * message ID
 - * timestamp
- **properties** – user-defined
 - **body** – text message, byte stream, serialized Java object, stream of primitive Java values, structured set of name/value pairs
- **message producer** – object to publish messages under particular topic or to send messages to a queue
 - **message consumer** – object to subscribe to messages with given topic or receive messages from a queue

- filters: *message selector* (over header or properties)
 - * subset of SQL used to specify properties
- can block using a *receive* operation
- can establish *message listener* object
 - * – has to establish method `onMessage`

A simple example

Figure 6.17 Java class FireAlarmJMS

```
1 import javax.jms.*;
2 import javax.naming.*;
3 public class FireAlarmJMS {
4     public void raise() {
5         try {
6             Context ctx = new InitialContext();
7             TopicConnectionFactory topicFactory = // find factory
8                 (TopicConnectionFactory)ctx.lookup("TopicConnectionFactory");
9             Topic topic = (Topic)ctx.lookup("Alarms"); // topic
10            TopicConnection topicConn = // connection
11                topicConnectionFactory.createTopicConnection();
12            TopicSession topicSess = topicConn.createTopicSession(false,
13                            Session.AUTO_ACKNOWLEDGE); // session
14            TopicPublisher topicPub = topicSess.createPublisher(topic);
15            TextMessage msg = topicSess.createTextMessage(); // create message
16            msg.setText("Fire !");
17            topicPub.publish(message); // publish it
18        } catch (Exception e) {
19    }
20}
```

```
//create a new instance of the FireAlarmJMS class and then raise an alarm is:  
alarm = new FireAlarmJMS();  
alarm.raise();
```

Figure 6.18 Java class FireAlarmConsumerJMS

```
1 import javax.jms.*;  
2 import javax.naming.*;  
3 public class FireAlarmConsumerJMS {  
4     public String await() {  
5         try {  
6             Context ctx = new InitialContext();  
7             TopicConnectionFactory topicFactory =  
8                 (TopicConnectionFactory)ctx.lookup("TopicConnectionFactory");  
9             Topic topic = (Topic)ctx.lookup("Alarms");  
10            TopicConnection topicConn =  
11                topicConnectionFactory.createTopicConnection();  
12            TopicSession topicSess = topicConn.createTopicSession(false,  
13                Session.AUTO_ACKNOWLEDGE); // ... identical upto here  
14            TopicSubscriber topicSub = topicSess.createSubscriber(topic);  
15            topicSub.start(); // topic subscriber created and started  
16            TextMessage msg = (TextMessage) topicSub.receive(); // receive  
17            return msg.getText(); // return message as string
```

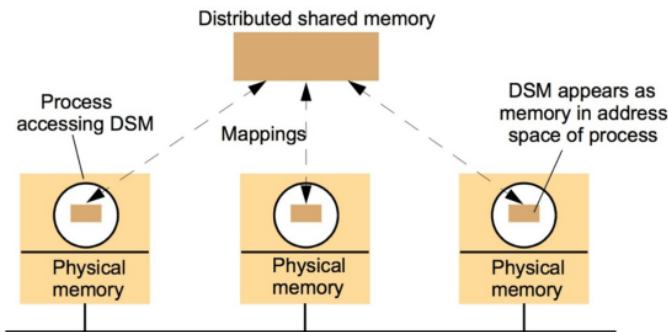
```
18     } catch (Exception e) {
19         return null;
20     }
21 }
```

```
// class usage by a consumer:
FireAlarmConsumerJMS alarmCall = new FireAlarmConsumerJMS();
String msg = alarmCall.await();
System.out.println("Alarm_received: "+msg);
```

6.5 Shared memory approaches

6.5.1 Distributed shared memory (DSM)

Figure 6.19 The distributed shared memory abstraction



- DSM – tool for parallel applications
- shared data items available for access directly
- DSM runtime – sends messages with updates between computers
- managed replicated data for faster access

One of the first examples: Apollo Domain file system [1983] – DSM can be persistent

Non-Uniform Memory Access (NUMA) architecture

- processors see a single address space containing all the memory of all the boards
- access latency for on-board memory less than for a memory module on a different board

Message passing versus DSM

- *service offered*
 - message passing: variable marshalled-unmarshalled into variable on other processor
 - DSM – not possible to run on heterogeneous architectures
- *synchronization*

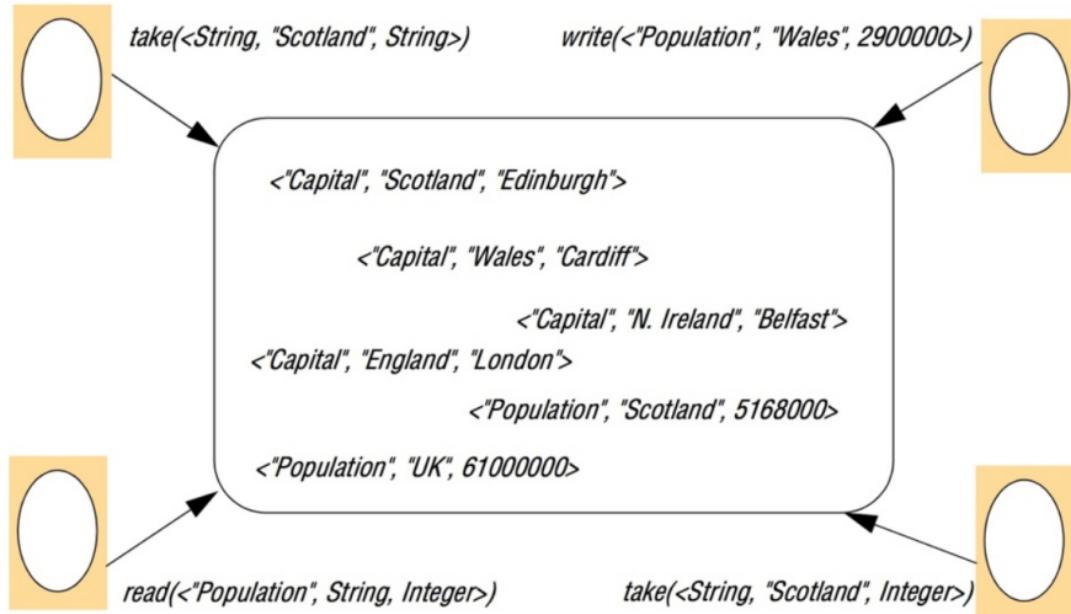
- via message model
 - locks and semaphores in DSM implementations
- DSM can be made persistent
 - message-passing systems: processes have to coexist in time
 - *Efficiency* – very problem-dependent
 - message-passing: suitable for hand-tuning on supercomputer-sized clusters
 - DSM – can be made to perform as well at least for small numbers of processors

6.5.2 Tuple space communication

- David Gelernter [1985], Yale University
- *generative communication*
 - processes communicate indirectly by placing tuples in a tuple space
 - from which other processes can read or remove them
 - Tuples
 - * do not have an address
 - * are accessed by pattern matching on content (*content-addressable memory*)
 - * consist of a sequence of one or more typed data fields such as
 - <"fred", 1958>
 - <"sid", 1964>
 - <4, 9.8, "Yes">

- * tuples are immutable
- Tuple space (TS)
 - * any combination of types of tuples may coexist in the same tuple space
 - * processes share date through it
 - write operation
 - read (or take) operation
 - read – TS not affected
 - take – returns tuple and removes it from TS
 - both blocking operations until there is a matching tuple in TS
- associative addressing – processes provide for read and take operation a specification – any tuple with a matching specification is returned
- Linda programming model – Linda programming language

Figure 6.20 The tuple space abstraction



Properties associated with tuple spaces

- *Space uncoupling:*

- A tuple placed in tuple space may originate from any number of sender processes and may be delivered to any one of a number of potential recipients
 - also referred to as *distributed naming* in Linda
- *Time uncoupling:*
 - A tuple placed in tuple space will remain in that tuple space until removed (potentially indefinitely) ⇒ hence the sender and receiver do not need to overlap in time

a form of *distributed sharing* of shared variables via the tuple space

Variations:

- multiple tuple spaces
- distributed implementation

- Bauhaus Linda:
 - modelling everything as (unordered) sets – that is, tuple spaces are sets of tuples and tuples are sets of values, which may now also include tuples
- turning the tuple space into an *object space*
 - e.g. in JavaSpaces

Implementation issues

centralized vs distributed

- Replication or *state machine* approach (read more in textbook)
- peer-to-peer approaches

Case study: JavaSpaces

tool for tuple space communication developed by Sun

- Sun provides specification, third-party developers offer implementations:
 - [GigaSpaces](#)
 - [Blitz](#)
- strongly dependent on Jini (Sun's discovery service)
 - Jini Technology Starter Kit includes
 - * Outrigger (JavaSpaces implementation)

goals of the JavaSpaces technology are:

- to offer a platform that simplifies the design of distributed applications and services

- to be simple and minimal in terms of the number and size of associated classes
- to have a small footprint
- to allow the code to run on resource-limited devices (such as smart phones)
- to enable replicated implementations of the specification
 - (although in practice most implementations are centralized)

Programming with JavaSpaces

programmer can create any number of instances *space* – shared, persistant repository of objects

an item in JavaSpace – referred to as an *entry*: a group of objects contained in a class that implements `net.jini.core.entry.Entry`

Figure 6.23 The JavaSpaces API

Operation	Effect
<i>Lease</i> <code>write(Entry e, Transaction txn, long lease)</code>	Places an entry into a particular JavaSpace
<code>Entry read(Entry tmpl, Transaction txn, long timeout)</code>	Returns a copy of an entry matching a specified template
<code>Entry readIfExists(Entry tmpl, Transaction txn, long timeout)</code>	As above, but not blocking
<code>Entry take(Entry tmpl, Transaction txn, long timeout)</code>	Retrieves (and removes) an entry matching a specified template
<code>Entry takeIfExists(Entry tmpl, Transaction txn, long timeout)</code>	As above, but not blocking
<code>EventRegistration notify(Entry tmpl, Transaction txn, RemoteEventListener listen, long lease, MarshalledObject handback)</code>	Notifies a process if a tuple matching a specified template is written to a JavaSpace

- placing an entry with `write` operation
 - entry can have an associated lease
 - * numerical value in milliseconds or `Lease.FOREVER`
 - `write` returns granted *Lease* value

- read or take
 - matching specified by a *template*
 - matching entry has the same class or subclass
- notify
 - uses Jini distributed event notification
 - notification via a specified *RemoteEventListener* interface

operations in JavaSpaces can take place in the context of a transaction, ensuring that either all or none of the operations will be executed

Figure 6.24 Java class AlarmTupleJS

```
1 import net.jini.core.entry.*;
2 public class AlarmTupleJS implements Entry {
3     public String alarmType;
4     public AlarmTupleJS() {
5     }
6     public AlarmTupleJS(String alarmType) {
7         this.alarmType = alarmType;
8     }
9 }
```

Figure 6.25 Java class FireAlarmJS

```
1 import net.jini.space.JavaSpace;
2 public class FireAlarmJS {
3     public void raise() {
4         try {
5             JavaSpace space = SpaceAccessor.findSpace("AlarmSpace");
6             AlarmTupleJS tuple = new AlarmTupleJS("Fire !");
7             space.write(tuple, null, 60*60*1000);
8         catch (Exception e) {
9         }
10    }
11 }
```

C

```
// the code can be called using:  
FireAlarmJS alarm = new FireAlarmJS();  
alarm.raise();
```

Figure 6.26 Java class FireAlarmReceiverJS

```
1 import net.jini.space.JavaSpace;  
2 public class FireAlarmConsumerJS {  
3     public String await() {  
4         try {  
5             JavaSpace space = SpaceAccessor.findSpace();  
6             AlarmTupleJS template = new AlarmTupleJS("Fire!");  
7             AlarmTupleJS recv = (AlarmTupleJS) space.read(template, null,  
8                                                 Long.MAX_VALUE);  
9             return recv.alarmType;  
10        }  
11        catch (Exception e) {  
12            return null;  
13        }  
14    }  
15}
```

```
// consumer:  
FireAlarmConsumerJS alarmCall = new FireAlarmConsumerJS();  
String msg = alarmCall.await();  
System.out.println("Alarm received: " + msg);
```

End of week 7

7 Operating systems support

End of week 8

8 Distributed objects and components

8.1 Introduction

Distributed object middleware

- *encapsulation* in object-based solutions – well suited to distributed programming
- *data abstraction* – clean separation between the specification of an object and its implementation ⇒ programmers to deal solely in terms of interfaces and not concern with implementation details
- ⇒ more dynamic and extensible solutions

Examples of distributed objects middleware: Java RMI and CORBA

Component-based middleware

- to overcome a number of limitations with distributed object middleware:

Implicit dependencies: Object interfaces do not describe what the implementation of an object depends on

Programming complexity: need to master many low-level details

Lack of separation of distribution concerns: Application developers need to consider details of security, failure handling and concurrency – largely similar from one application to another

No support for deployment: Object-based middleware provides little or no support for the deployment of (potentially complex) configurations of objects

8.2 Distributed objects

- DS started as client-server architecture
- with emergence of highly popular OO languages (C++, Java) the OO concept spreading to DS
- Unified Modelling Language (UML) in SE has its role too in middleware developments (e.g. CORBA and UML standards developed by the same organisation)

Distributed object (DO) middleware

- Java RMI and CORBA – quite common
- but CORBA – language independent

in DO the term class is avoided – instead factory instantiating new objects from a given template

- in Smalltalk – implementational inheritance
- in DO – interface inheritance:
 - new interface inherits the method signatures of the original interface
 - * + can add extra ones

Figure 8.1 Distributed objects

<i>Objects</i>	<i>Distributed objects</i>	<i>Description of distributed object</i>
Object references	Remote object references	Globally unique reference for a distributed object; may be passed as a parameter.
Interfaces	Remote interfaces	Provides an abstract specification of the methods that can be invoked on the remote object; specified using an interface definition language (IDL).
Actions	Distributed actions	Initiated by a method invocation, potentially resulting in invocation chains; remote invocations use RMI.
Exceptions	Distributed exceptions	Additional exceptions generated from the distributed nature of the system, including message loss or process failure.
Garbage collection	Distributed garbage collection	Extended scheme to ensure that an object will continue to exist if at least one object reference or remote object reference exists for that object, otherwise, it should be removed. Requires a distributed garbage collection algorithm.

OO: objects + class + inheritance \longleftrightarrow DO: encapsulation + data abstraction + design methodologies

The added complexities with DO:

- *Inter-object communication*
 - remote method invocation
 - + often other communications paradigms
 - * (e.g. CORBA's event service + associated notification service)
- *Lifecycle management*
 - creation, migration and deletion of DO

- *Activation and deactivation*
 - # DOs may be very large...
 - node availabilities

- *Persistence*

state of DO need to be preserved across all cycles (like [de]activation, system failures etc.)

- *Additional services*

- e.g. naming, security and transaction services

8.3 Case study: CORBA

Object Management Group (OMG)

- formed in 1989
- designed an interface language
 - independent of any specific implementation language

object request broker (ORB)

- to help a client to invoke a method on an object

Common Object Request Broker Architecture (CORBA)

CORBA 2 specification

CORBA3 – introduction of a component model

8.3.1 CORBA RMI

CORBA's object model

CORBA objects refer to remote objects

wide range of types PL support ⇒ no classes ⇒ instances of classes cannot be passed as arguments

CORBA IDL

Figure 8.2 IDL interfaces *Shape* and *ShapeList*

```
1 struct Rectangle{ // 1
2     long width;
3     long height;
4     long x;
5     long y;
6 };
7 struct GraphicalObject { // 2
8     string type;
9     Rectangle enclosing;
```

```
10    boolean isFilled;  
11};  
12interface Shape { // 3  
13    long getVersion();  
14    GraphicalObject getAllState(); // returns state of the GraphicalObject  
15};  
16typedef sequence <Shape, 100> All; // 4  
17interface ShapeList { // 5  
18    exception FullException{}; // 6  
19    Shape newShape(in GraphicalObject g) raises (FullException); // 7  
20    All allShapes(); // returns sequence of remote object references // 8  
21    long getVersion();  
22};
```

- same lexical rules as C++
 - + distribution keywords
 - * – e.g. interface, any, attribute, in, out, inout, readonly, raises
- grammar of IDL – subset of ANSI C++ + constructs to support method signatures

IDL modules:

module defines a naming scope

Figure 8.3 IDL module *Whiteboard*

```
1 module Whiteboard {
2     struct Rectangle{
3         ...} ;
4     struct GraphicalObject {
5         ...};
6     interface Shape {
7         ...};
8     typedef sequence <Shape, 100> All ;
9     interface ShapeList {
10        ...};
11    };
```

IDL interfaces

- *IDL interface* describes the methods that are available in CORBA objects that implement that interface

- Clients of a CORBA object may be developed just from the knowledge of its IDL interface

IDL methods

The general form of a method signature is:

```
[oneway] <return_type> <method_name> (parameter1, ..., parameterL)
[raises (except1, ..., exceptN)] [context (name1, ..., nameM)];
```

Example:

```
void getPerson(in string name, out Person p);
```

- parameters: in, out, inout
- return value acting as if additional out parameter
 - return type may be void

Passing CORBA objects:

- Any parameter specified by the name of an IDL interface – a reference to a CORBA object
 - the value of a remote object reference is passed

Passing CORBA primitive and constructed types:

- Arguments of primitive and constructed types are copied and passed by value

Invocation semantics

remote invocation call semantics defaults to: *at-most-once*

- to specify method invocation with *maybe semantics*: keyword oneway
 - non-blocking call on the client side
 - ⇒ method should not return a result

Exceptions in CORBA IDL

- optional *raises* expression indicates user-defined exceptions
- exceptions may be defined to contain variables, e.g:
 - exception **FullException {GraphicalObject g};**

IDL types:

- 15 primitive types, *const* keyword
- object - remote object references

Figure 8.4 IDL constructed types

Type	Examples	Use
<i>sequence</i>	<i>typedef sequence <Shape, 100> All;</i> <i>typedef sequence <Shape> All;</i> Bounded and unbounded sequences of Shapes	Defines a type for a variable-length sequence of elements of a specified IDL type. An upper bound on the length may be specified.
<i>string</i>	<i>string name;</i> <i>typedef string<8> SmallString;</i> Unbounded and bounded sequences of characters	Defines a sequence of characters, terminated by the null character. An upper bound on the length may be specified.
<i>array</i>	<i>typedef octet uniqueId[12];</i> <i>typedef GraphicalObject GO[10][8];</i>	Defines a type for a multi-dimensional fixed-length sequence of elements of a specified IDL type.

Type	Examples	Use
<i>record</i>	<i>struct GraphicalObject {</i> <i> string type;</i> <i> Rectangle enclosing;</i> <i> boolean isFilled;</i> <i>};</i>	Defines a type for a record containing a group of related entities.
<i>enumerated</i>	<i>enum Rand</i> <i>(Exp, Number, Name);</i>	The enumerated type in IDL maps a type name onto a small set of integer values.
<i>union</i>	<i>union Exp switch (Rand) {</i> <i> case Exp: string vote;</i> <i> case Number: long n;</i> <i> case Name: string s;</i> <i>};</i>	The IDL discriminated union allows one of a given set of types to be passed as an argument. The header is parameterized by an enum, which specifies which member is in use.

- All arrays or sequences used as arguments must be defined in typedefs

- None of the primitive or constructed data types can contain references
- passing non-CORBA objects (nCO) by value – CORBA's *valuetype*
 - nCO operations cannot be invoked remotely
 - makes it possible to pass a copy of a nCO between client and server
- *valuetype* – struct with additional method signatures (like those of an interface)
- *valuetype* arguments and results – passed by value
 - the state is passed to the remote site and used to produce a new object at the destination
 - if the client and server are both implemented in Java, the code can be downloaded
 - common C++ implementation – the necessary code to be present at both client and server

Attributes

IDL interfaces can have methods and **attributes**

- like public class fields in Java
- may be readonly
- private to CORBA objects
 - pair of attribute value set-retrieve generated by IDL compiler automatically

Inheritance

IDL interfaces may be extended through interface inheritance

Example:

- interface B extends interface A ⇒

- B may add new types, constants, exceptions, methods and attributes to those of A
 - * + can redefine types, constants and exceptions
 - * not allowed to redefine methods

IDL interface may extend more than one interface

```
interface A { };  
interface B: A{ };  
interface C { };  
interface Z : B, C {};
```

(but inheriting common names from two different interfaces not allowed)

IDL type identifiers

- generated by the IDL compiler

- has three parts – the IDL prefix, a type name and a version number
- programmers have to provide a unique mapping to the interfaces – may use *pragma* prefix for this

IDL pragma directives

- for specification of additional non-IDL properties in IDL interface

for example,

- specifying that an interface will be used only locally
- supplying the value of an interface repository ID

Example:

```
#pragma version Whiteboard 2.3
```

CORBA language mappings

primitive types in IDL → corresponding primitive types in that language

structs, enums, unions → Java classes

IDL allows to have multiple return values... can be solved like this:

```
void getPerson(in string name, out Person p); //IDL  
void getPerson(String name, PersonHolder p); //java
```

Asynchronous RMI

CORBA RMI allows clients to make non-blocking invocation requests on CORBA objects

- intended to be implemented in the client - server unaware on invocation synchronous or asynchronous (except e.g. Transaction Service)

Asynchronous RMI invocation semantics:

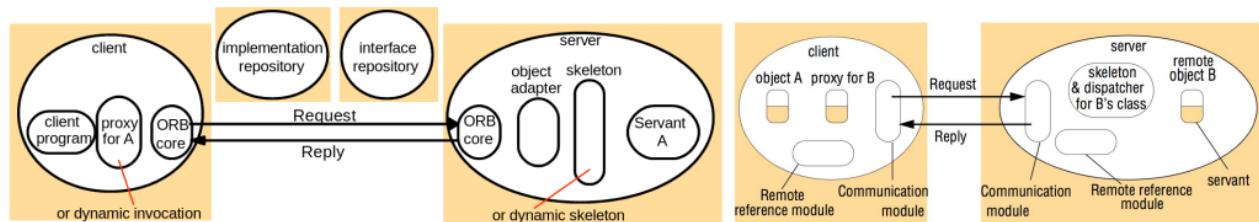
- callback – client passes an extra parameter with a reference to a callback

⇒ server can call back with the results

- polling – server returns a valuetype object that can be used to poll or wait for the reply

8.3.2 The architecture of CORBA

Figure 8.5 The main components of the CORBA architecture



- 3 additional components compared to Figure 5.15 (at right)...
 - object adaptor; implementation repository; interface repository
- a) Static invocation – object interfaces known at compile time – skeleton can be used
- b) Dynamic invocation

ORB core

- role of [Fig. 5.15 communication module] + an interface which includes the following:
 - operations enabling it to be started and stopped
 - operations to convert between remote object references and strings
 - operations to provide argument lists for requests using dynamic invocation

Object adapter (OA)

- role of [Fig. 5.15 reference and dispatcher modules]

CORBA objects with IDL interfaces \longleftrightarrow the programming language interfaces of the corresponding servant classes

OA tasks:

- creates remote object references for CORBA objects (Section 8.3.3)

- dispatches each RMI via a skeleton to the appropriate servant
 - activates and deactivates servants
- gives each CORBA object a unique object name (forms part of its remote object reference)
- keeps a remote object table that maps the names of CORBA objects to their servants
 - also has its own name (forms part of the remote object references of all of the CORBA objects it manages)

Portable Object Adapter (POA)

allows applications and servants to be run on ORBs produced by different developers
supports CORBA objects with two different sorts of lifetimes:

- those whose lifetimes are restricted to that of the process in which their servants are instantiated (transient object references)

- those whose lifetimes can span the instantiations of servants in multiple processes (resistant object references)

...for further details the textbook Section 8.3.2...

Skeletons

Chapter 5.4.2:

Skeleton: implements the methods in the remote interface

- unmarshals the arguments in the request message
- invokes the corresponding method in the servant
- waits for the invocation to complete
- marshals the result (together with any exceptions in a reply message to the sending proxy's method)

Client stubs/proxies

The class of a proxy (for OO languages) or a set of stub procedures (for procedural languages) is generated from an IDL interface by an IDL compiler for the client language

Implementation repository

– responsible for:

- activating registered servers on demand
- locating servers that are currently running
- stores a mapping from the names of object adapters to the pathnames of files containing object implementations

When object implementations are activated in servers, the hostname and port number of the server are added to the mapping

object adapter name	pathname of object implementation	hostname and port number of server
---------------------	-----------------------------------	------------------------------------

Some objects (e.g. callback) created by clients, run once and cease to exist when they are no longer needed – do not use the implementation repository

Interface repository

- information about registered IDL interfaces to clients and servers that require it
 - adds a facility for reflection to CORBA

Every CORBA remote object reference includes a slot that contains the type identifier of its interface, enabling clients that hold it to enquire its type of the interface repository

- applications using static (ordinary) invocation with client proxies and IDL skeletons do not require an interface repository
- Not all ORBs provide an interface repository

Dynamic invocation interface

CORBA does not allow classes for proxies to be downloaded at runtime (as in Java RMI) – The dynamic invocation interface is CORBA's alternative

- used when it is not practical to employ proxies
- The client can obtain from the interface repository the necessary information about the methods available for a given CORBA object
- The client may use this information to construct an invocation with suitable arguments and send it to the server

Dynamic skeletons

- Consider CORBA object whose interface was unknown when the server was compiled

with dynamic skeletons, server can accept invocations on the interface of a CORBA object for which it has no skeleton

- When a dynamic skeleton receives an invocation, it inspects the request contents to discover its
 - target object
 - the method to be invoked
 - the arguments
 - then invokes the target

Legacy code

- The term legacy code refers to existing code that was not designed with distributed objects in mind

A piece of legacy code may be made into a CORBA object by defining an IDL interface for it and providing an implementation of an appropriate object adapter and the necessary skeletons

8.3.3 CORBA remote object reference

called: **interoperable object references (IORs)**

IOR format

1. IDL interface type ID
2. Protocol and address details
3. Object key

interface repository identifier type	IIOP	host domain name	port number	adapter name	object name
---	------	---------------------	-------------	--------------	-------------

1. Note that IDL interface type ID is also identifier for the ORB interface repository (if it is existing)
2. Transport protocol: Internet InterORB protocol (IIOP) – uses TCP
May be repeated to allow possible replications
3. Used by ORB to identify a CORBA object

Transient IOR last only as long as the process that hosts object

Persistent IOR last between activations of the CORBA objects

8.3.4 CORBA services

specification of common services includes in CORBA:

<i>CORBA Service</i>	Role
<i>Naming service</i>	Supports naming in CORBA, in particular mapping names to remote object references within a given naming context (see Chapter 9).
<i>Trading service</i>	Whereas the Naming service allows objects to be located by name, the Trading service allows them to be located by attribute; that is, it is a directory service. The underlying database manages a mapping of service types and associated attributes onto remote object references.
<i>Event service</i>	Allows objects of interest to communicate notifications to subscribers using ordinary CORBA remote method invocations (see Chapter 6 for more on event services generally).

CORBA Service	Role
Notification service	Extends the event service with added capabilities including the ability to define filters expressing events of interest and also to define the reliability and ordering properties of the underlying event channel.
Security service	Supports a range of security mechanisms including authentication, access control, secure communication, auditing and nonrepudiation (see Chapter 11).
Transaction service	Supports the creation of both flat and nested transactions (as defined in Chapters 16 and 17).
Concurrency control service	Uses locks to apply concurrency control to the access of CORBA objects (may be used via the transaction service or as an independent service).

<i>CORBA Service</i>	Role
<i>Persistent state service</i>	Offers a persistent object store for CORBA, used to save and restore the state of CORBA objects (implementations are retrieved from the implementation repository).
<i>Lifecycle service</i>	Defines conventions for creating, deleting, copying and moving CORBA objects; for example, how to use factories to create objects.

8.3.5 CORBA client and server example

compiler *idlj* generates the following items:

- 2 Java interfaces per IDL interface:

Figure 8.7 Java interfaces generated by *idlj* from CORBA interface *ShapeList*

```
1 public interface ShapeListOperations {  
2     Shape newShape(GraphicalObject g) throws ShapeListPackage.FullException;  
3     Shape[] allShapes();  
4     int getVersion();  
5 }  
6  
7 public interface ShapeList extends ShapeListOperations, org.omg.CORBA.Object,  
8     org.omg.CORBA.portable.IDLEntity { }
```

- server skeletons
 - The names of skeleton classes end in POA – for example, *ShapeListPOA*
- The proxy classes or client stubs, one for each IDL interface

- The names of these classes end in Stub – for example, _ShapeListStub
- A Java class to correspond to each of the structs defined with the IDL interfaces
 - In our example, classes Rectangle and GraphicalObject are generated.
 - Each of these classes contains a declaration of one instance variable for each field in the corresponding struct and a pair of constructors, but no other methods.
- Classes called helpers and holders, one for each of the types defined in the IDL interface.
 - A helper class contains the narrow method, which is used to cast down from a given object reference to the class to which it belongs, which is lower down the class hierarchy.
 - * For example, the narrow method in ShapeHelper casts down to class Shape.

- The holder classes deal with out and inout arguments, which cannot be mapped directly onto Java.

Server program

CORBA objects – instances of servant classes.

When a server creates an instance of a servant class, it must register it with the POA (Portable Object Adaptor), which makes the instance into a CORBA object and gives it a remote object reference

Figure 8.8 *ShapeListServant* class of the Java server program for CORBA interface *ShapeList*

```
1 import org.omg.CORBA.*;  
2 import org.omg.PortableServer.POA;  
3 class ShapeListServant extends ShapeListPOA {  
4     private POA theRootpoa;  
5     private Shape theList[];  
6     private int version;  
7     private static int n=0;  
8     public ShapeListServant(POA rootpoa){
```

```
9     theRootpoa = rootpoa;
10    // initialize the other instance variables
11 }
12 public Shape newShape(GraphicalObject g) throws ShapeListPackage.FullException { // 1
13    version++;
14    Shape s = null;
15    ShapeServant shapeRef = new ShapeServant( g, version );
16    try {
17        org.omg.CORBA.Object ref = theRootpoa.servant_to_reference(shapeRef); // 2
18        s = ShapeHelper.narrow(ref);
19    } catch (Exception e) {}
20    if (n >= 100) throw new ShapeListPackage.FullException();
21    theList[n++] = s;
22    return s;
23 }
24 public Shape[] allShapes() { ... }
25 public int getVersion() { ... }
26 }
```

Main method in Server class:

Figure 8.9 Java class *ShapeListServer*

```
1 import org.omg.CosNaming.*;
2 import org.omg.CosNaming.NamingContextPackage.*;
```

```
3 import org.omg.CORBA.*;
4 import org.omg.PortableServer.*;
5 public class ShapeListServer {
6     public static void main(String args[]) {
7         try{
8             ORB orb = ORB.init(args, null); // 1
9             POA rootpoa = POAHelper.narrow(orb.resolve_initial_references("RootPOA")); // 2
10            rootpoa.the_POAManager().activate(); // 3
11            ShapeListServant SLSRef = new ShapeListServant(rootpoa); // 4
12            org.omg.CORBA.Object ref = rootpoa.servant_to_reference(SLSRef); // 5
13            ShapeList SLRef = ShapeListHelper.narrow(ref);
14            org.omg.CORBA.Object objRef = orb.resolve_initial_references("NameService");
15            NamingContext ncRef = NamingContextHelper.narrow(objRef); // 6
16            NameComponent nc = new NameComponent("ShapeList", ""); // 7
17            NameComponent path[] = {nc}; // 8
18            ncRef.rebind(path, SLRef); // 9
19            orb.run(); // 10
20        } catch (Exception e) { ... }
21    }
22}
```

The client program

Figure 8.10 Java client program for CORBA interfaces *Shape* and *ShapeList*

```
1 import org.omg.CosNaming.*;
2 import org.omg.CosNaming.NamingContextPackage.*;
3 import org.omg.CORBA.*;
4 public class ShapeListClient{
5     public static void main(String args[]) {
6         try{
7             ORB orb = ORB.init(args, null); //1
8             org.omg.CORBA.Object objRef =
9                 orb.resolve_initial_references("NameService");
10            NamingContext ncRef = NamingContextHelper.narrow(objRef);
11            NameComponent nc = new NameComponent("ShapeList", "");
12            NameComponent path [] = { nc };
13            ShapeList shapeListRef =
14                ShapeListHelper.narrow(ncRef.resolve(path)); // 2
15            Shape[] sList = shapeListRef.allShapes(); // 3
16            GraphicalObject g = sList[0].getAllState(); // 4
17        } catch(org.omg.CORBA.SystemException e) {...} // 5
18    }
19 }
```

Callbacks

Similar to JavaRMI

```
interface WhiteboardCallback {
    oneway void callback(in int version);
};
```

- implemented by client enabling the server to send version number whenever objects get added
- for this the ShapeList interface requires additional methods:

```
int register(in WhiteboardCallback callback);
void deregister(in int callbackId);
```

8.4 From objects to components

Component-based approaches – a natural evolution from distributed object computing

Issues with object-oriented middleware

Implicit dependencies – internal (encapsulated) behaviour of an object is hidden
– think remote method invocation or other communication paradigms... – not apparent from the interface

- there is a clear requirement to specify not only the interfaces offered by an object but also the dependencies that object has on other objects in the distributed configuration

Interaction with the middleware – too many relatively low-level details associated with the middleware architecture

- clear need to:
 - simplify the programming of distributed applications
 - to present a clean separation of concerns between code related to operation in a middleware framework and code associated with the application
 - to allow the programmer to focus exclusively on the application code

Lack of separation of distribution concerns: Application developers need to deal explicitly with non-functional concerns related to issues such as security, transactions, coordination and replication – largely repeating concerns from one application to another

- the complexities of dealing with such services should be hidden wherever possible from the programmer

No support for deployment: objects must be deployed manually on individual machines – can become a tiresome and error-prone process

- Middleware platforms should provide intrinsic support for deployment so that distributed software can be installed and deployed in the same way as software for a single machine, with the complexities of deployment hidden from the user

→ component based middleware

Essence of components

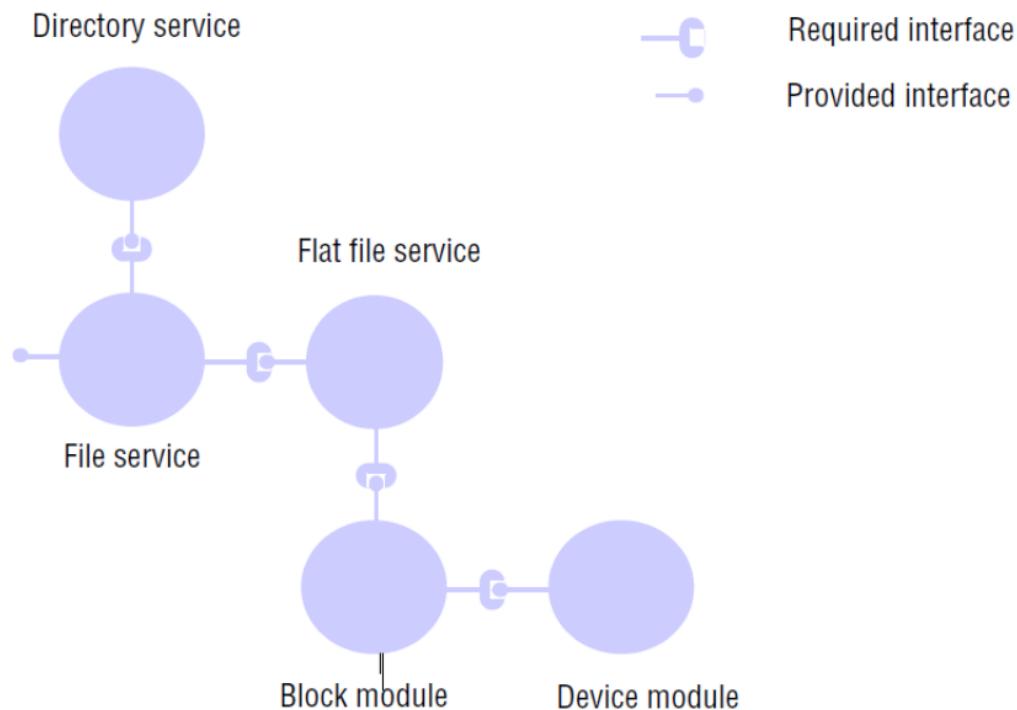
software component – unit of composition with contractually specified interfaces and explicit context dependencies only

- dependencies are also represented as interfaces

- component is specified in terms of a contract, which includes:
 - a set of provided interfaces
 - * – interfaces that the component offers as services to other components
 - a set of required interfaces
 - * – the dependencies that this component has in terms of other components that must be present and connected to this component for it to function correctly
- every required interface must be bound to a provided interface of another component
- → software architecture consisting of components, interfaces and connections between interfaces

Example: Architecture of a simple file system

Figure 8.11 An example software architecture



Many component-based approaches offer two styles of interface:

- interfaces supporting remote method invocation, as in CORBA and Java RMI
- interfaces supporting distributed events (as discussed in Chapter 6)

Component-based system programming concerned with

- development of components
- composition of components

Moving from software development to software assembly

Components and distributed systems

Containers:

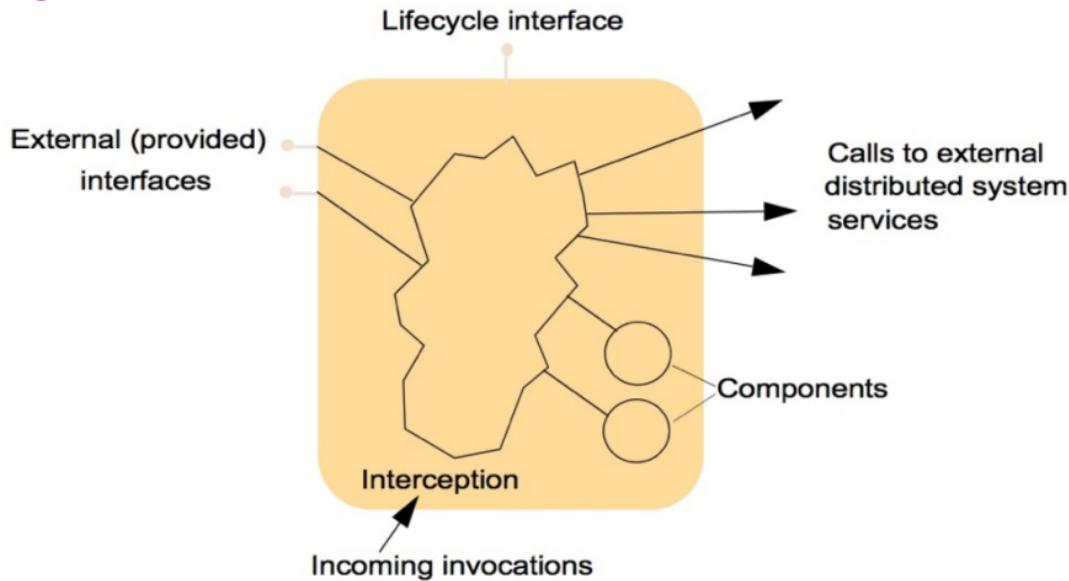
Containers support a common pattern often encountered in distributed applications, which consists of:

- a front-end (perhaps web-based) client
- a container holding one or more components that implement the application or business logic
- system services that manage the associated data in persistent storage

components deal with application concerns

container deals with distributed systems and middleware issues (ensuring that non-functional properties are achieved)

Figure 8.12 The structure of a container



the container does not provide direct access to the components but rather intercepts incoming invocations and then takes appropriate actions to ensure the desired properties of the distributed application are maintained

Middleware supporting the container pattern and the separation of concerns implied by this pattern is known as an *application server*

This style of distributed programming is in widespread use in industry today: – range of application servers:

Technology	Developed by	Further details
<i>WebSphere Application Server</i>	IBM	www.ibm.com
<i>Enterprise JavaBeans</i>	SUN	java.sun.com
<i>Spring Framework</i>	SpringSource (a division of VMware)	www.springsource.org
<i>JBoss</i>	JBoss Community	www.jboss.org
<i>CORBA Component Model</i>	OMG	[Wang <i>et al.</i> 2001] JOnAS]
<i>JOnAS</i>	OW2 Consortium	jonas.ow2.org
<i>GlassFish</i>	SUN	glassfish.dev.java.net

Support for deployment

Component-based middleware provides support for the **deployment of component configuration**

- components are deployed into containers
- deployment descriptors are interpreted by containers to establish the required policies for the underlying middleware and distributed system services

container therefore includes

- a number of components that require the same configuration in terms of distributed system support

Deployment descriptors are typically written in XML with sufficient information to ensure that:

- components are correctly connected using appropriate protocols and associated middleware support

- the underlying middleware and platform are configured to provide the right level of support to the component configuration
- the associated distributed system services are set up to provide the right level of security, transaction support and so on

8.5 Case study: Enterprise JavaBeans

End of week 9

9 Web Services

9.1 Introduction

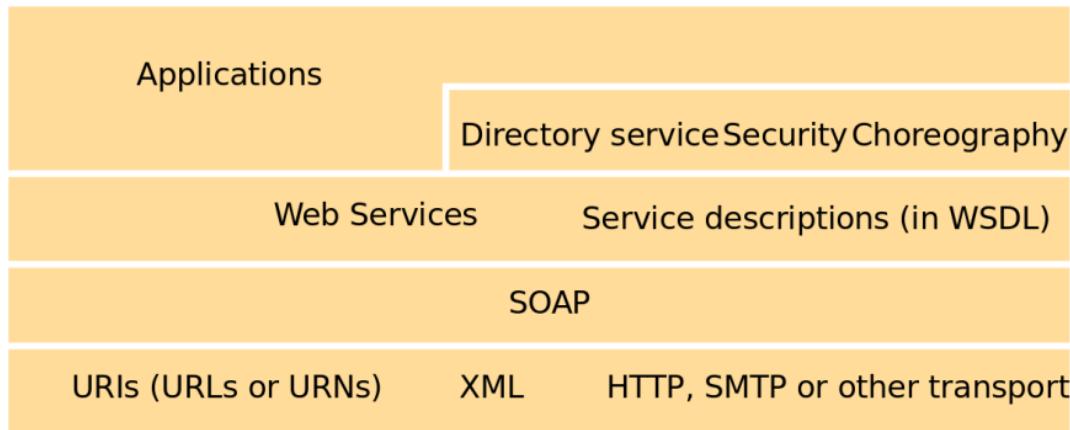
- A web service provides a service interface enabling clients to interact with servers in a more general way than web browsers do
- Clients access the operations in the interface of a web service by means of requests and replies formatted in XML and usually transmitted over HTTP
- Users require a secure means for creating, storing and modifying documents and exchanging them over the Internet
- The secure channels of TLS do not provide all of the necessary requirements
- XML security is intended to breach this gap
- Web services provide an infrastructure for maintaining a richer and more structured form of interoperability between clients and servers

- They provide a basis whereby a client program in one organization may interact with a server in another organization without human supervision
 - External data representation and marshalling of messages exchanged between clients and web services is done in XML.
 - The SOAP (Simple Object Access Protocol) specifies the rules for using XML to package messages, for example to support a request-reply protocol
-
- Web Services
 - One of the dominant paradigms for programming distributed systems
 - Enables business to business integration. (Suppose one organization uses CORBA and another uses .NET) No problem!
 - Enables service oriented architecture (SOA).
 - Adopted by the grid computing community.
 - May exist internally to an organization or externally (in the cloud).

- What are Web Services?
 - Web Services began life when Bill Gates introduced BizTalk in 1999.
 - BizTalk was later renamed .NET.
 - The idea: “to provide the technologies to allow software in different places, written in different languages and resident on different platforms to connect and interoperate.” From “Programming the World Wide” by Sebesta
- Two approaches
 - SOAP Based (WS-*) Web Services
 - REST style web services

9.2 SOAP based Web services

Figure 9.1 Web Services Infrastructure and Components

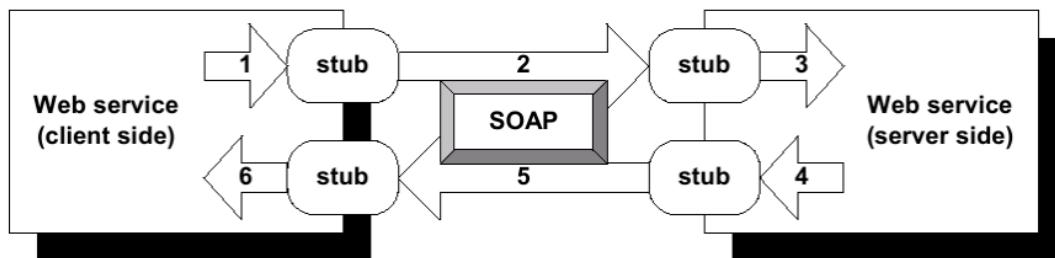
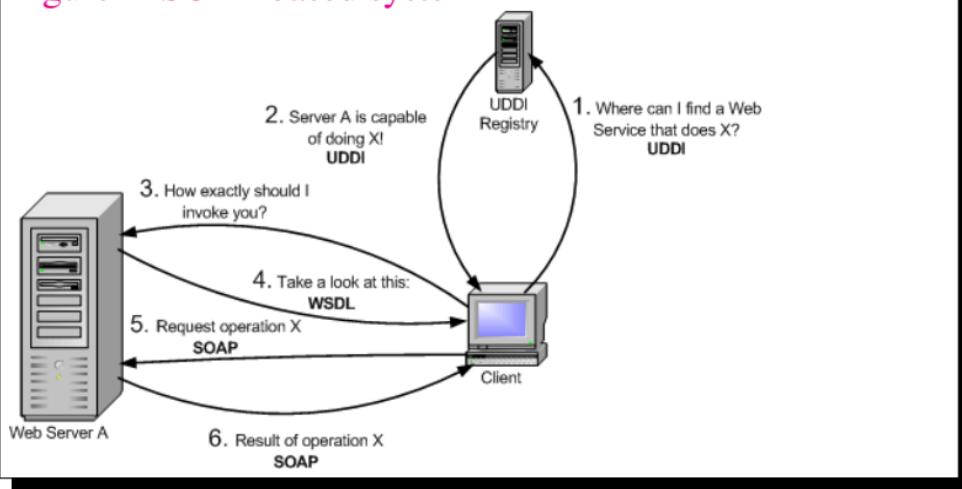


SOAP Style Web Services

- Provide service interfaces
- Communicate using request and reply messages made of SOAP or some other XML document

- Have an Interface Definition Language (IDL) called WSDL (Web Service Definition Language)
- May be looked up in a web service UDDI registry
 - UDDI (Universal Directory and Discovery Service)
- Are language independent
- May be synchronous or asynchronous

Figure A SOAP based system



Communication Patterns

- In general, web services use either a synchronous request-reply pattern of communication with their clients or they communicate by asynchronous messages.
- To allow for a variety of patterns, SOAP is based on the packaging of single one-way messages.
- SOAP is used to hold RPC style parameters or entire documents.
- Originally, SOAP was based on HTTP, but the current version may be used over different transport protocols (SMTP, TCP, UDP, or HTTP)

Service References

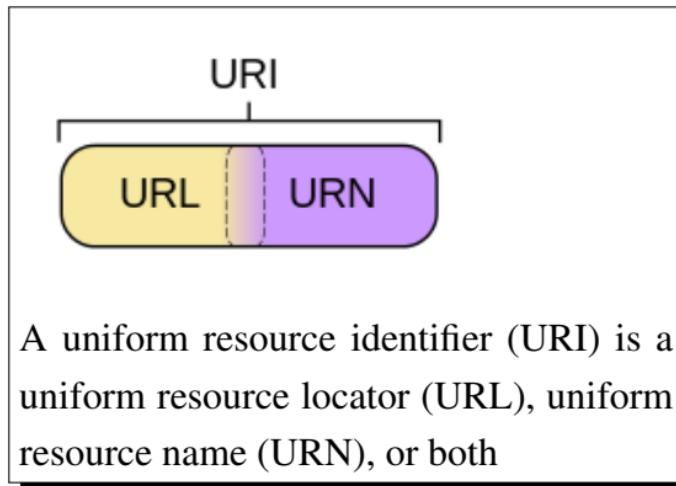
URI's – Uniform Resource Identifiers

URL's – Uniform Resource Locator URI's that include location information

⇒ resources pointed to by URL's are hard to move

URN's – Uniform Resource Name

URI's that include no location information



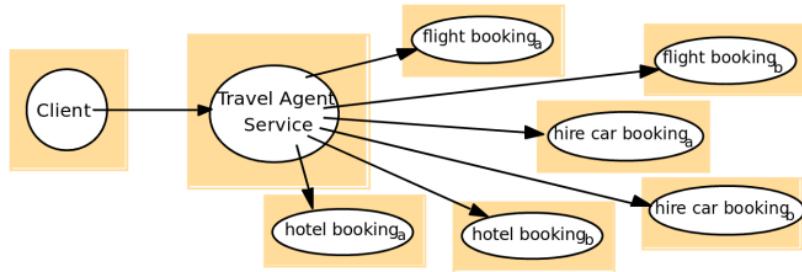
- A URN lookup service may be used to determine a URL from a URN
- URL's are the most frequently used form of URI

Examples:

1. URL: <http://www.cmu.edu/service>
2. URN: urn:ISBN:0-273-76059-9

Web Service Composition (Mashups)

Figure 9.2 The ‘travel agent service’ combines other web services



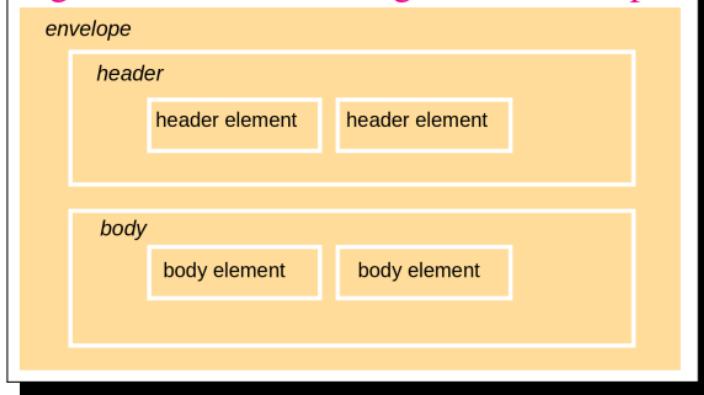
What concerns are not shown? This is an important list: Transactions, Security (privacy, identification, authentication, authorization), Reliability, Orchestration tooling, Interoperability through Standards, RPC or Messaging, Service Level agreements

9.2.1 SOAP

SOAP (Simple Object Access Protocol)

- defines a scheme for using XML to represent the contents of request and reply messages as well as a scheme for the communication of documents
- The SOAP specification states:
 - How XML is to be used to represent the contents of individual messages
 - How a pair of single messages can be combined to produce a request-reply pattern
 - The rules as to how the recipients of messages should process the XML elements that they contain
 - How HTTP and SMTP should be used to communicate SOAP messages
- A SOAP message is carried in an “envelope”
- Inside the envelope there is an optional header and a body

Figure 9.3 SOAP message in an envelope



- Message headers can be used for establishing the necessary context for a service or for keeping a log or audit of operations
- The message body carries an XML document for a particular web service
- The XML elements envelope, header and body, together with other attributes and elements of SOAP messages are defined as a scheme in the SOAP XML namespace

Figure 9.4 Example of a simple request without headers

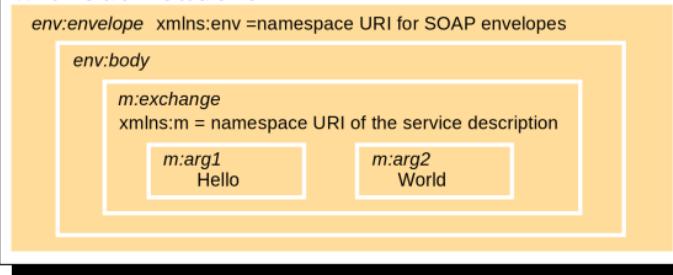


Figure 9.5 Example of a reply corresponding to the request in Figure 9.4

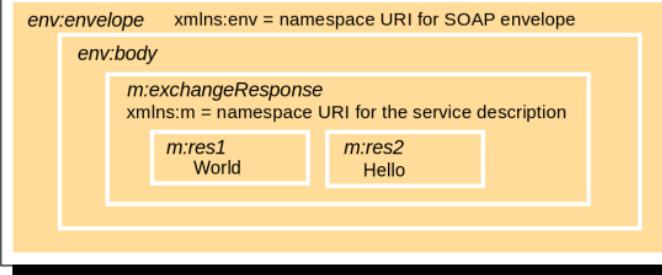
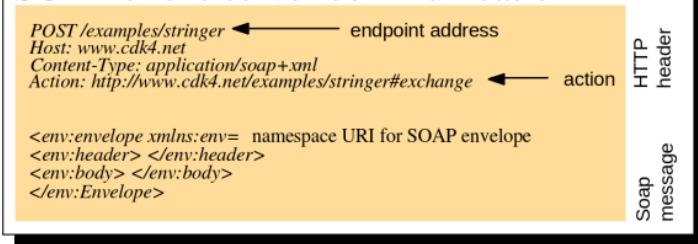


Figure 9.6 Use of HTTP POST Request in
SOAP client-server communication



- A transport protocol is required to send a SOAP document to its destination
- Other transports may be used
 - WS-Addressing may be used to include destination and source
 - Thus, different protocols might be used over different parts of the route of a message

WS-ReliableMessaging: Reliable communication

- SOAP's usual protocol – TCP

- TCP does not guarantee message delivery in certain situations
 - * timeout waiting for acknowledgement – declares the connection is broken – the actual delivery of the information becomes unknown
- Oasis (a global consortium that works on the development, agreement and adoption of e-business and web service standards) has made a recommendation called
 - **WS-ReliableMessaging** [www.oasis.org]
 - allows a SOAP message to be delivered at-least-once, at-most-once or exactly-once (– different from that in RPC number of times server executes a remote procedure)
 - * *At-least-once*: The message is delivered at least once, but an error is reported if it cannot be delivered
 - * *At-most-once*: The message is delivered at most once, but without any error report if it cannot be delivered

- * *Exactly-once*: The message is delivered exactly once, but an error is reported if it cannot be delivered
- Ordering of messages is also provided in combination with any of the above:
 - * In-order: Messages will be delivered to the destination in the order in which they were sent by a particular sender

Traversing firewalls

- transport protocols such as those used by JavaRMI or CORBA normally not able to pass firewalls
- firewalls allow both HTTP and SMTP to pass through

9.2.2 The use of SOAP with Java

Figure 9.7 Java web service interface ShapeList

```
1 import java.rmi.*;  
2 public interface ShapeList extends Remote {  
3     int newShape(GraphicalObject g) throws RemoteException; 1  
4     int numberOfShapes() throws RemoteException;  
5     int getVersion() throws RemoteException;  
6     int getGOVersion(int i) throws RemoteException;  
7     GraphicalObject getAllState(int i) throws RemoteException;  
8 }
```

The Java interface of a web service must conform to the following rules, some of which are illustrated in Figure 9.7:

- It must extend the Remote interface
- It must not have constant declarations, such as *public final static*
- The methods must throw the `java.rmi.RemoteException` or one of its subclasses
- Method parameters and return types must be permitted JAX-RPC types

Figure 9.8 Java implementation of the ShapeList server

```
1 import java.util.Vector;
2 public class ShapeListImpl implements ShapeList {
3     private Vector theList = new Vector();
4     private int version = 0;
5     private Vector theVersions = new Vector();
6     public int newShape(GraphicalObject g) throws RemoteException{
7         version++;
8         theList.addElement(g);
9         theVersions.addElement(new Integer(version));
10        return theList.size();
11    }
12    public int numberOfShapes(){}
13    public int getVersion() {}
14    public int getGOVersion(int i){ }
15    public GraphicalObject getAllState(int i) {}
16 }
```

- There is no main method, and the implementation of the ShapeList interface does not have a constructor
- In effect, a web service is a single object that offers a set of procedures

- *wscompile* and *wsdeploy* can be used to generate the skeleton class and the service description (in WSDL)
- information concerning the URL of the service, its name and description retrieved from a configuration file written in XML

Servlet container

- The service implementation is run as a servlet inside a **Servlet container** whose role is to load, initialize and execute servlets
- The servlet container includes a dispatcher and skeletons (see Section 5.4.2)
- When a request arrives, the dispatcher maps it to a particular skeleton, which translates it into Java and passes on the request to the appropriate method in the servlet

- That method carries out the request and produces a reply, which the skeleton translates back into a SOAP reply
- The URL of a service consists of a concatenation of the URL of the servlet container and the service category and name
 - *e.g.*, `http://localhost:8080/ShapeList-jaxrpc/ShapeList`
- Tomcat [jakarta.apache.org] is a commonly used servlet container

The client program

- may use **static proxies (generated at compile time, example below)**, dynamic proxies or dynamic invocation interface

Figure 9.9 Java implementation of the ShapeList client

```
1 package staticstub;  
2 import javax.xml.rpc.Stub;
```

```
3 public class ShapeListClient {  
4     public static void main(String [] args) { /* pass URL of service */  
5         try {  
6             Stub proxy = createProxy(); // 1 Proxies are sometimes called stubs...  
7             proxy._setProperty( // 2 URL of the service  
8                 (javax.xml.rpc.Stub.ENDPOINT_ADDRESS_PROPERTY, args[0]);  
9             ShapeList aShapeList = (ShapeList)proxy; // 3 The type of the proxy is narrowed to  
10                suit the type of the interface – ShapeList  
11             GraphicalObject g = aShapeList.getAllState(0); // 4 asking the service to return  
12                the object at element 0 in the vector of GraphicalObjects  
13         } catch (Exception ex) { ex.printStackTrace(); }  
14     }  
15     private static Stub createProxy() { // 5  
16         return  
17             (Stub) (new MyShapeListService_Impl().getShapeListPort()); // 6 The class name  
18                for the proxy is formed by adding '_Impl' to the name of the service (–  
19                implementation specific)  
20     }  
21 }
```

Dynamic proxies: dynamic proxy class is created at runtime from the information in the service description and the interface of the service

- avoids the need for involving an implementation-specific name for the proxy class

Dynamic invocation interface: allows a client to call a remote procedure, even if its signature or the name of the service is unknown until runtime

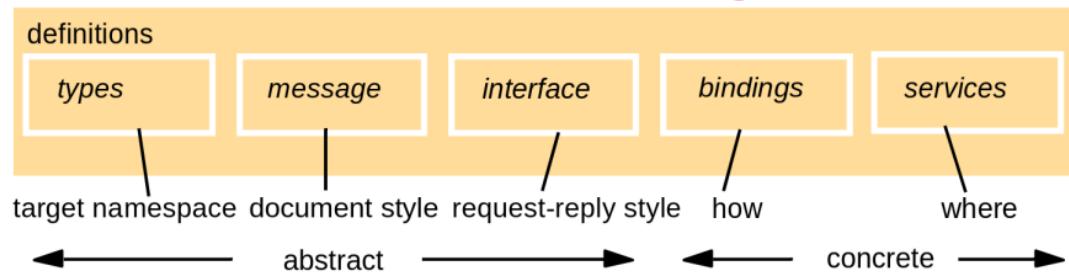
- The client does not require a proxy. Instead, it has to use a series of operations to set the name of the server operation, the return value and each of the parameters before making the procedure call

9.2.3 Service descriptions

- The primary means of describing a web service is by using **WSDL (the Web Services Description Language)**
- XML Schema may be used to describe the input and output parameters
- WSDL describes the operations and makes use of XML Schema to describe an exchange of messages

- A Service Description (WSDL document) is an IDL (interface definition language) + it contains information on how and where the service may be accessed
- It contains an abstract part and a concrete part
 - The abstract part is most like a traditional interface
 - The concrete part tells us how and where to access the service

Figure 9.10 The main elements in a WSDL description



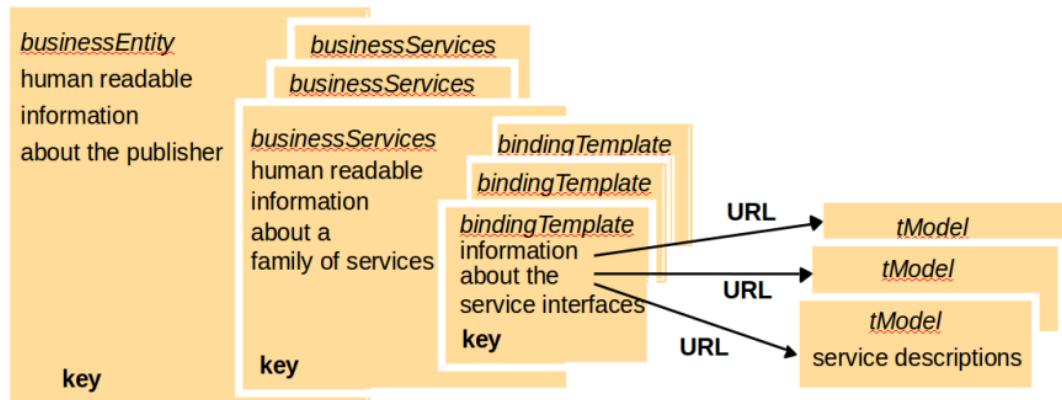
- A binding is a choice of protocols
- A service holds an endpoint address

- Client or server side code may be generated automatically from the WSDL
- A WSDL document may be accessed directly or indirectly through a registry like UDDI (Universal Directory and Discovery Service)
- A message exchange is called an ***operation***
- Related operations are grouped into ***interfaces***
- A ***binding*** specifies concrete details about what goes on the wire
- WSDL is an Interface Definition Language (IDL)
- WSDL
 - Describes the contract between applications
 - Can be automatically generated from a collection of Java or C# classes
 - Can be read by utilities that generate client side proxy code or server side skeletons

9.2.4 WS Directory Service

Universal Description, Discovery and Integration service (UDDI)

Figure 9.15 The main UDDI data structures



9.2.5 XML security

Signing and encrypting

Figure 9.16 Algorithms required for XML signature

Type of algorithm	Name of algorithm	Required	reference
Message digest	SHA-1	Required	Section 7.4.3
Encoding	base64	Required	[Freed and <u>Borenstein</u> 1996]
Signature (asymmetric)	DSA with SHA-1	Required	[NIST 1994]
MAC signature (symmetric)	HMAC-SHA-1	Recommended	Section 7.4.2 and <u>Krawczyk et al.</u> [1997]
Canonicalization	Canonical XML	Required	Page 425

Figure 9.17 Algorithms required for XML encryption

Type of algorithm	Name of algorithm	Required	reference
Block cipher	TRIPLEDES, AES-128 AES-256	required	Section 7.3.1
	AES-192	optional	
Encoding	base64	required	[Freed and Borenstein 1996]
Key transport	RSA-v1.5, RSA-OAEP	required	Section 7.3.2 [Kaliski and Staddon 1998]
Symmetric key wrap (signature by shared key)	TRIPLEDES KeyWrap, AES-128 KeyWrap, AES 256KeyWrap AES-192 KeyWrap	required	[Housley 2002]
Key agreement	Diffie-Hellman	optional	[Rescorla, 1999]

9.2.6 Coordination of web services

Web services choreography

Requirements:

- hierarchical and recursive composition of choreographies
- the ability to add new instances of an existing service and new services
- concurrent paths, alternative paths and the ability to repeat a section of a choreography
- variable timeouts – for example, different periods for holding reservations
- exceptions, for example, to deal with messages arriving out of sequence and user actions such as cancellations
- asynchronous interactions (callbacks)
- reference passing, for example, to allow a car hire company to consult a bank for a credit check on behalf of a user

- marking of the boundaries of the separate transactions that take place, for example, to allow for recovery
- the ability to include human-readable documentation

9.2.7 Applications of web services

- Service-oriented architectures (SOA)
- The Grid
- Cloud computing

Figure 9.19 A selection of Amazon Web Services

<i>Web service</i>	<i>Description</i>
Amazon Elastic Compute Cloud (EC2)	Web-based service offering access to virtual machines of a given performance and storage capacity
Amazon Simple Storage Service (S3)	Web-based storage service for unstructured data
Amazon Simple DB	Web-based storage service for querying structured data
Amazon Simple Queue Service (SQS)	Hosted service supporting message queuing (as discussed in Chapter 6)
Amazon Elastic MapReduce	Web-based service for distributed computation using the MapReduce model (introduced in Chapter 21)
Amazon Flexible Payments Service (FPS)	Web-based service supporting electronic payments

9.3 REST

- ***REST (REpresentational State Transfer)***
 - Notes from [RESTful Java with JAX-RS by Bill Burke](http://it-ebooks.info/book/390/) (<http://it-ebooks.info/book/390/>)
- Roy Fielding's doctoral dissertation (2000)
- The question he tried to answer in his thesis was "Why is the Web so prevalent and ubiquitous"? What makes the Web scale? How can I apply the architecture of the Web to my own applications?

REST Architectural Principles

Addressable resources The key abstraction of information and data in REST is a **resource**, and each resource must be addressable via a **URI** (Uniform Resource Identifier)

A uniform, constrained interface Use a small set of well-defined methods to manipulate your resources

Representation-oriented You interact with services using representations of that service. A resource referenced by one URI can have different formats. Different platforms need different formats. For example, browsers need HTML, JavaScript needs JSON (JavaScript Object Notation), and a Java application may need XML.

Communicate statelessly Stateless applications are easier to scale.

Hypermedia As The Engine Of Application State (HATEOAS) Let your data formats drive state transitions in your applications.

- REST is not protocol specific. It is usually associated with HTTP but its principles are more general.
- SOAP and WS-* use HTTP strictly as a transport protocol

- But HTTP may be used as a rich application protocol
- Browsers usually use only a small part of HTTP
- HTTP is a synchronous request/response network protocol used for distributed, collaborative, document based systems
- Various message formats may be used – XML, JSON,..
- Binary data may be included in the message body

REST – to build distributed services and model service-oriented architectures (SOAs)
– application developers design their systems as a set of reusable, decoupled, distributed services.

9.3.1 Principle: Addressability

Every object and resource in your system is reachable through a unique identifier

- Addressability (not restricted to HTTP)
 - Each HTTP request uses a URI
- The format of a URI is well defined:
 - scheme://host:port/path?queryString#fragment
- The scheme need not be HTTP – may be FTP or HTTPS
- The host field may be a DNS name or a IP address
- The port may be derived from the scheme

- e.g. using HTTP implies port 80
- The path is a set of text segments delimited by the “/”
- The queryString is a list of parameters represented as name=value pairs
 - Each pair is delimited by an “&”
- The fragment is used to point to a particular place in a document
- A space is represented with the ‘+’ characters
- Other characters use % followed by two hex digits

9.3.2 Principle: The Uniform, Constrained interface

Stick to the finite set of operations of the application protocol you're distributing your services upon

- No action parameter in the URI and use only the methods of HTTP for your web services
 - GET - read only operation
 - * *idempotent* (once same as many)
 - * *safe* (no important change to server's state)
 - * may include parameters in the URI
`http://www.example.com/products?pid=123`
 - PUT - store the message body – insert or update
 - * idempotent
 - * not safe

- DELETE - used to delete resources
 - * idempotent
 - * not safe
 - * Each method call may modify the resource in a unique way
 - * The request may or may not contain additional information
 - * The response may or may not contain additional information
- POST
 - * not idempotent
 - * not safe
 - * Each method call may modify the resource in a unique way
 - * The request may or may not contain additional information
 - * The response may or may not contain additional information
 - * The parameters are found within the request body (not within the URI)

- HTTP HEAD, OPTIONS, TRACE and CONNECT are less important when designing RESTful web services
- Does HTTP have too few operations?
 - Note that SQL has only four operations:
 - * SELECT, INSERT, UPDATE and DELETE
 - * JMS and MOM (Message Oriented Middleware) have, essentially, two operations: SEND and RECEIVE

What does a uniform interface buy?

- *Familiarity*
 - We do not need a general IDL that describes a variety of method signatures
 - We do not need stubs
 - We already know the methods

- *Interoperability*
 - WS-* has been a moving target
 - * *application operability*, rather than *vendor operability*
 - HTTP is widely supported
- Scalability
 - Since GET is idempotent and safe, results may be cached by clients or proxy servers
 - Since PUT and DELETE are both idempotent neither the client or the server need worry about handling duplicate message delivery

9.3.3 Principle: Representation-Oriented

Services should be representation-oriented. Each service is addressable through a specific URI and representations are exchanged between the client and service.

- Representations of resources are exchanged
 - GET returns a representation
 - PUT and POST passes representations to the server so that underlying resources may change
 - Representations may be in many formats: XML, JSON, YAML, etc., ...
- HTTP uses the CONTENT-TYPE header to specify the message format the server is sending
- The value of the CONTENT-TYPE is a MIME typed string – capability of the client and server to negotiate the message formats

- Versioning information may be included.
 - Examples:
 - * text/plain
 - text/html
 - application/vnd+xml;version=1.1
 - * “vnd” implies a vendor specific MIME type
- The ACCEPT header in content negotiation
 - An AJAX request might include a request for JSON
 - A Java request might include a request for XML
 - Ruby might ask for YAML

9.3.4 Principle: Communicate Stateless

No client session data stored on the server

- The application may have state but there is no client session data stored on the server
- If there is any session-specific data it should be held and maintained by the client and transferred to the server with each request as needed
- The server is easier to scale. No replication of session data concerns

9.3.5 Principle: HATEOAS

Document centric approach with support for embedding links to other services

- Hypermedia As The Engine Of Application State
- Hypermedia is document centric approach but with the additional support for embedding links to other services
- With each request returned from a server it tells you what interactions you can do next as well as where you can go to transition the state of your application

Example

```
1 <order id="111">
2   <customer>http://customers.myintranet.com/customers/32133</customer>
3   <order-entries>
4     <order-entry>
5       <quantity>5</quantity>
6       <product>http://products.myintranet.com/products/111</product>
7   ...
```

- Client enters a REST application through a simple fixed URL
- All future actions the client may take are discovered within resource representation links returned from the server
- The media types used for these representations, and the link relations they may contain, are standardized
- The client transitions through application states by selecting from the links within a representation or by manipulating the representation in other ways afforded by its media type
- ⇒ RESTful interaction is driven by hypermedia, rather than out-of-band information

Example: Request for a list of products on a web store:

- HTTP GET on <http://example.com/webstore/products> and receive back:

```
<products>
  <product id="123">
    <name>headphones</name>
    <price>$16.99</price>
  </product>
  <product id="124">
    <name>USB Cable</name>
    <price>$5.99</price>
  </product>
  ...
</products>
```

- can become very long list, so better to return only n (say =10) items and add links to next/previous/total...

Contents

0 Introduction	3
0.1 Syllabus	3
0.1.1 Lectures:	3
0.1.2 Discussion seminars	4
0.1.3 Homework	5
0.1.4 Exam	5
0.2 Literature	6
0.2.1 Textbook	6
0.2.2 Additional reading	6
1 Characterization of distributed systems	8
1.1 Introduction	8
1.2 Examples of distributed systems	11
1.2.1 Web search	13

1.2.2	Massively multiplayer online games (MMOGs)	13
1.2.3	Financial trading	14
1.3	Trends in distributed systems	15
1.3.1	Pervasive networking and the modern Internet	15
1.3.2	Mobile and ubiquitous computing	17
1.3.3	Distributed multimedia systems	19
1.3.4	Distributed computing as a utility	20
1.4	Sharing resources	21
1.5	Challenges	23
1.5.1	Heterogeneity	23
1.5.2	Openness	25
1.5.3	Security	27
1.5.4	Scalability	28
1.5.5	Failure handling	29
1.5.6	Concurrency	31
1.5.7	Transparency	31

1.5.8	Quality of service	33
1.6	Case study: The World Wide Web	34
2	System models	38
2.1	Outline	38
2.2	Physical models	39
2.3	Architectural Models	43
2.3.1	Architectural elements	43
2.3.2	Architectural patterns	59
2.3.3	Associated middleware solutions	68
2.4	Fundamental models	71
2.4.1	Interaction model	71
2.4.2	Failure model	76
2.4.3	Security model	81
3	Networking and internetworking	89

3.1	Introduction	89
	3.1.1 Networking issues for distributed systems	91
3.2	Types of network	95
3.3	Network principles	100
	3.3.1 Packet transmission	100
	3.3.2 Data streaming	101
	3.3.3 Switching schemes	102
	3.3.4 Protocols	104
	3.3.5 Routing	112
	3.3.6 Congestion control	116
	3.3.7 Internetworking	117
3.4	Internet protocols	120
	3.4.1 IP addressing	122
	3.4.2 The IP protocol	125
	3.4.3 IP routing	127
	3.4.4 IP version 6	130

3.4.5	MobileIP	133
3.4.6	TCP and UDP	134
3.4.7	Domain names	137
3.4.8	Firewalls	138
3.5	Case studies	140
3.5.1	Ethernet	141
3.5.2	IEEE 802.11 (WiFi) wireless LAN	143
4	Interprocess communication	144
4.1	Introduction	144
4.2	The API for the Internet protocols	145
4.2.1	The characteristics of interprocess communication	145
4.2.2	Sockets	146
4.2.3	UDP datagram communication	147
4.2.4	TCP stream communication	155
4.3	External data representation and marshalling	162

4.3.1	CORBA's Common Data Representation (CDR)	164
4.3.2	Java object serialization	167
4.3.3	Extensible Markup Language (XML)	170
4.3.4	Remote object references	175
4.4	Multicast communication	176
4.4.1	IP multicast – An implementation of multicast communication	176
4.5	Network virtualization: Overlay networks	181
4.5.1	Overlay networks	182
4.5.2	Skype: An example of an overlay network	186
4.6	Case study: MPI	189
5	Remote invocation	194
5.1	Introduction	195
5.2	Request-reply protocols	195
5.3	Remote procedure call (RPC)	207
5.3.1	Design issues for RPC	207

5.3.2	Implementation of RPC	213
5.3.3	Case study: Sun RPC	214
5.4	Remote method invocation (RMI)	217
5.4.1	Design issues for RMI	219
5.4.2	Implementation of RMI	224
5.4.3	Distributed garbage collection	236
5.5	Case study: Java RMI	238
5.5.1	Building client and server programs	245
5.5.2	Design and implementation of Java RMI	250
6	Indirect communication	252
6.1	Introduction	252
6.2	Group communication	255
6.2.1	The programming model	256
6.2.2	Implementation issues	258
6.2.3	Case study: the JGroups toolkit	260

6.3	Publish-subscribe systems	266
6.3.1	The programming model	268
6.3.2	Implementation issues	271
6.3.3	Examples of publish-subscribe systems	277
6.4	Message queues	278
6.4.1	The programming model	278
6.4.2	Implementation issues	282
6.4.3	Case study: The Java Messaging Service (JMS)	282
6.5	Shared memory approaches	291
6.5.1	Distributed shared memory (DSM)	291
6.5.2	Tuple space communication	294
7	Operating systems support	305
8	Distributed objects and components	306
8.1	Introduction	306

8.2	Distributed objects	308
8.3	Case study: CORBA	313
8.3.1	CORBA RMI	314
8.3.2	The architecture of CORBA	327
8.3.3	CORBA remote object reference	335
8.3.4	CORBA services	336
8.3.5	CORBA client and server example	339
8.4	From objects to components	346
8.5	Case study: Enterprise JavaBeans	356
9	Web Services	357
9.1	Introduction	357
9.2	SOAP based Web services	360
9.2.1	SOAP	366
9.2.2	The use of SOAP with Java	372
9.2.3	Service descriptions	377

9.2.4	WS Directory Service	380
9.2.5	XML security	380
9.2.6	Coordination of web services	383
9.2.7	Applications of web services	384
9.3	REST	386
9.3.1	Principle: Addressability	389
9.3.2	Principle: The Uniform, Constrained interface	391
9.3.3	Principle: Representation-Oriented	395
9.3.4	Principle: Communicate Statelessly	397
9.3.5	Principle: HATEOAS	398