Distributed Systems

MTAT.08.009

* * * * *

eero.vainikko@ut.ee

Course homepage (http://courses.cs.ut.ee/2017/ds/fall)

Fall 2017
2 Practical information

Teachers:
- Eero Vainikko
- Amnir Hadachi
- Artjom Lind

Tutors:
- Annika Laumets <annika.laumets@ut.ee>
- Andre Tättar <andre.tattar@ut.ee>
- Andres Namm <andres.namm@ut.ee>
3 Practical information

**Lectures:** WED 14:15, Liivi 2 - 405

**Practical sessions:**
1. group **FRI 10:15 Paabel** (Ülikooli 17) **room 218** *(Amnir Hadachi, tutor: Annika Laumets)*
2. group **WED 8:15 J.Liivi 2 - 403** *(Artjom Lind, tutor: Andre Tättar)*
3. group **FRI 10:15 J.Liivi 2 - 205** *(Eero Vainikko, tutor: Andres Namm)*

Lectures: 32h; Practical sessions: 32h; Independent work: 92h

**Final grade:**

1. **Seminar tasks** (20%)
2. **Homework** (40%=20+20)
3. **Active participation** at lectures (10%)
4. **Final exam** (30%)
4 Practical information

Active participation

Active participation at lectures –

• Create your own theme sheet portfolio for each lecture theme
  – DEADLINE: 1 week after each lecture (or 1/2 points later!)

• Can use whatever way to privately share the link to the portfolio

• Submission of the link through the course page

  – Be inventive! For example – use

  * Google Sites (incl. Google Docs, Slides, videos, quizzes with the help of Google Forms etc...)
  * University of Tartu ownCloud http://owncloud.ut.ee – uploading word/text-file, pictures or scans of your notes taken at the lecture
5 Practical information

Portfolio consist of theme sheets

Each theme sheet is a collection of:

- Your coarse notes
- definitions
- schemes / drawings
- your thoughts / ideas / questions
- external material
  - videos
  - articles
  - web-sites
6 Practical information

– best portfolios awarded with BONUS points!

BONUS points:
Devize and insert questions into the online course study-questionary!
7 Practical information

What we expect from you?

1. Attend regularly the lectures
   
   (a) Full attention needed!
   
   i. Make notes
   
   ii. ask questions
   
   iii. participate in discussions
   
   (b) Review previous material (work with your portfolio!) before coming to the next class

2. Seminar assignments

   (a) Start working on tasks as soon as they are given out

   (b) be a Team player during homework!

   (c) With any problems with the subject – come up early
8 Practical information

EXAM

- Course materials studied at Lectures and Discussion Seminars
- Written exam
- Closed book exam, BUT
- ... you are allowed to take with you a printout of an extract from your Portfolio
  - 0.5 a page per Chapter / theme
  - submission of exam paper together with the printout of the help-materials you prepared for the exam!
- Dates: TBD
  - A. __. January 2017 at 10:15, Liivi 2 - __
  - B. __. January 2017 at 10:15, Liivi 2 - __
Wide background of students

MTAT.08.009 Distributed Systems belongs to following MsC curricula:

- Computer Science (129537) - UT, Institute of Computer Science
- Robotics and Computer Engineering (136637) - UT, Institute of Technology
- Cybersecurity (100946) - TTU & UT, Institute of Computer Science
0.1 Syllabus

0.1.1 Lectures:

0. Introduction to the course

- Characterization of distributed systems
- Networking and internetworking
- Interprocessor communication
- System models
- Remote invocation
- Indirect communication
- Operating systems support
- Distributed objects and components
- Web services
- Peer-to-peer systems
- Security
- Distributed files systems
- Name services
- Time and global state; Coordination and agreement; Distributed transactions
- Designing distributed systems: Google case study
- Big Data paradigm
0.1.2 Practical sessions

**Very important to be present!** (for your own sake... :-)

- 2nd week – Test Evaluation on Python and Networking

- 3-16 weeks **Seminars** every second week followed by **seminar tasks** every following week
  - seminars – step-by-step guides
  - seminar tasks – hands-on exercises to be submitted to collect feedback from tutors

- **Bring your laptop!**

- **Python 2.7**
0.1.3 Homework

- 2 programming tasks with separate deadlines
- Working in groups of up to 4 people
0.2 Literature

0.2.1 Textbook


0.2.2 Additional reading

- POSIX thread programming
- Pthreads API specification
- Introduction to Java threads
- Synchronizing threads in Java
- Java tutorial by SUN
- Fundamentals of multithreading
14 Introduction

- Flick: The Flexible IDL Compiler Kit
- Java IDL Technology
- ONC+ Developer’s Guide
- Microsoft Interface Definition Language (MIDL)
- Introduction to Java RMI
- Java RMI Tutorial
- Annotated WSDL Example
- The NFS Version 4 Protocol
- Microsoft SMB Protocol and CIFS Protocol Overview
- Coda File System
- Remote Filesystems slides
- WebDAV Resources
- Understanding Replication in Databases and Distributed Systems (PDF)
- Linux Virtual Server for Scalable Network Services (PDF)
- NFS Security (PDF)
- Executive Summary: Computer Network Time Synchronization
1 Characterization of distributed systems\textsuperscript{1}

- Introduction
- Examples of Distributed Systems
- Defining Distributed Systems
- Characteristics and Trends of Distributed Systems
- Resource sharing
- Challenges

\textsuperscript{1}Textbook Chapter 1
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!*

Examples of Distributed Systems – **small scale** systems vs **large scale** systems

Distributed Systems – **benefits** vs **challenges**
## 1.2 Examples of distributed systems

### Distributed Systems application domains connected with networking:

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

- heterogeneity of 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
1.2.1 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 event-based systems*

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

```
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
```
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:
Internet scale and complexity
1.3.1 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
1.3.2 Distributed multimedia systems

- live or pre-ordered television broadcasts
- video-on-demand
- music libraries
- audio and video conferencing
1.3.3 Distributed computing as a utility

- Cluster computing
- Grid computing
- Cloud computing
• Cluster: “A type of parallel or distributed processing system, which consists of a collection of interconnected stand-alone computers cooperatively working together as a single, integrated computing resource” [Rajkumar Buyya]

• Cloud: “A type of parallel and distributed system consisting of a collection of interconnected and virtualised computers that are dynamically provisioned and presented as one or more unified computing resources based on service-level agreements established through negotiation between the service provider and consumers” [Rajkumar Buyya]

Prime motivation for Distributed Systems: to share resources

What are the resources?
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 DS Challenge – 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 executable code
1.5.2 **DS Challenge – Openness**

**OPENNESS** of a:

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

- **open 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 DS CHALLENGE – 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*
DS CHALLENGE – 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 **DS CHALLENGE – 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 **DS Challenge** – 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 **DS Challenge** – 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
DS CHALLENGE – 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
Distributed Systems are everywhere

Internet enables users throughout the world to access its (application) services from anywhere

Resource sharing is the main motivating factor for constructing distributed systems

Distributed systems enable globalization:

- Community (Virtual teams, organizations, social networks)
- Science (e-Science)
- Business (e-Banking)
- Entertainment (YouTube, e-Friends)

Construction of DS produces many challenges:
1.5 Challenges

- Heterogeneity, Openness, Security, Scalability, Failure handling, Concurrency, and Transparency

End of Theme 1
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.

### 2.1 Introduction

<table>
<thead>
<tr>
<th>Transmission media</th>
<th>Hardware devices</th>
<th>Software components</th>
</tr>
</thead>
<tbody>
<tr>
<td>wire</td>
<td>routers</td>
<td>protocol stacks</td>
</tr>
<tr>
<td>cable</td>
<td>switches</td>
<td>communication han-</td>
</tr>
<tr>
<td>fibre</td>
<td>bridges</td>
<td>dlers</td>
</tr>
<tr>
<td>wireless channels</td>
<td>hubs</td>
<td>drivers</td>
</tr>
<tr>
<td></td>
<td>repeaters</td>
<td></td>
</tr>
</tbody>
</table>

---

TextoBook Chapter 3

---

\(^2\) Textbook Chapter 3
communication subsystem – hardware and software components that provide the communication facilities for a distributed system

hosts – computers and other devices using the network for communication

node – any computer or switching device attached to a network

Internet constructed from many 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
2.1.1 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 of the Internet 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
- cryptographic 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
2.2 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**

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

- **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
2.2 Types of network

- **Wireless wide area networks (WWANs)**
  - GSM (3G, 4G); UMTS (Universal Mobile Telecommunication System)
  - rates upto 100 Mbps

- **Internetworks**
  - several networks linked together
  - routers, gateways
2.3 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

2.3.1 Packet transmission

- messages – sequences of data items of arbitrary length

  - subdivided into packets

  * packets have a restricted length – to be able to allocate sufficient storage; to avoid delays due to some large messages
2.3.2 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

- **Needed**: channel from source to destination of a multimedia stream
  - predefined route
  - reserved set of resources
  - buffering where appropriate for smoothness

- **ATM** (*Asynchronous Transfer Mode*)

- IPv6 includes features for real-time separate IP stream treatment
2.3.3 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
- ATM networks – prime example
2.3.4 Protocols

A protocol is a well-known set of rules and formats to be used for communication between processes in order to perform a given task. It consists of two important parts:

1. A specification of the sequence of messages that must be exchanged.
2. 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**

A protocol suite (or protocol stack) is a complete set of protocol layers. The diagram illustrates the seven-layer Reference Model for Open Systems Interconnection (OSI). The layers are:

1. **Application**
2. **Presentation**
3. **Session**
4. **Transport**
5. **Network**
6. **Data link**
7. **Physical**

The diagram shows the message flow from the sender to the recipient through the communication medium.
The **OSI model** is a conceptual model, which characterizes and standardizes the communication functions of any computing system or telecommunication without regard to the technology involved or their internal structure.

7. Application layer: Network Process to Application [Serves as the window for users and application processes to access the network services]
6. Presentation layer: Data Representation and Encryption [Formats the data to be presented to the Application layer. It can be viewed as the Translator for the network]
5. Session layer: Inter-host Communication [Allows session establishment between processes running on different stations]
4. Transport layer: End-to-End Connections and Reliability [Ensures that messages are delivered error-free, in-sequence, and with no losses or duplications]
3. Network layer: Path Determination and IP (Logical Addressing) [Controls the operations of the subnet, deciding which physical path the data takes]
2. Data link layer: MAC and LLC (Physical Addressing) [Provides error-free transfer of data frames from one node to another over the Physical layer]
1. Physical layer: Media, Signal, and Binary Transmission [Concerned with the transmission and reception of the unstructured raw bit stream over the physical medium]
## Figure 3.5 OSI protocol summary

<table>
<thead>
<tr>
<th>Layer</th>
<th>Description</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application</td>
<td>Protocols that are designed to meet the communication requirements of specific applications, often defining the interface to a service.</td>
<td>HTTP, FTP, SMTP, CORBA IIOP</td>
</tr>
<tr>
<td>Presentation</td>
<td>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.</td>
<td>Secure Sockets (SSL), CORBA Data Rep.</td>
</tr>
<tr>
<td>Session</td>
<td>At this level reliability and adaptation are performed, such as detection of failures and automatic recovery.</td>
<td>TCP, UDP</td>
</tr>
<tr>
<td>Transport</td>
<td>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.</td>
<td>IP, ATM virtual circuits</td>
</tr>
<tr>
<td>Network</td>
<td>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.</td>
<td>Ethernet MAC, ATM cell transfer, PPP</td>
</tr>
<tr>
<td>Data link</td>
<td>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.</td>
<td>Ethernet base-band signalling, ISDN</td>
</tr>
<tr>
<td>Physical</td>
<td>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).</td>
<td></td>
</tr>
</tbody>
</table>
Figure 3.6 Internetwork layers

Layers

Application

Transport

Internetwork

Network interface

Underlying network

Message

Internetwork protocols

Internetwork packets

Underlying network protocols

Network-specific packets
Packet assembly

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

Note that IP protocol MTU is 64 kbytes (or often 8 kbytes). (While e.g. Ethernet MTU is 1500 bytes.)

Ports

- software-defined destination points at a host computer

Addressing of ports

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

*contact port numbers* – FTP : 21 (initially, then an arbitrary one chosen by server for the rest of the session) ; HTTP :80 (all traffic)
- $\leq 1023$ – well-known ports – restricted to privileged processes
- $1024 \ldots 49151$ – registered ports for which IANA holds service descriptions
- remaining ports upto $65535$ available for private use

**Packet delivery – network layer:**

**A. Datagram packet delivery**

- ‘datagram’ refers to the similarity of this delivery mode to the way in which letters and telegrams are delivered
- packets contain full source and receiver address

**B. Virtual circuit packet delivery:**

- analogous to a telephone network
2.3 Network principles

- A virtual circuit must be set up before packets can pass from a source host A to destination host B.
- **No** sender-receiver addresses attached to packets – only virtual circuit number.
- Virtual circuit currently in use: ATM (Asynchronous Transfer Mode).

### 2.3.5 Routing

In large networks – *adaptive routing* – the best route for communication between two points in the network being 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

<table>
<thead>
<tr>
<th></th>
<th>Routings from A</th>
<th></th>
<th>Routings from B</th>
<th></th>
<th>Routings from C</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>To</strong></td>
<td><strong>Link</strong></td>
<td><strong>Cost</strong></td>
<td><strong>To</strong></td>
<td><strong>Link</strong></td>
<td><strong>Cost</strong></td>
</tr>
<tr>
<td>A</td>
<td>local</td>
<td>0</td>
<td>A</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>B</td>
<td>1</td>
<td>1</td>
<td>B</td>
<td>local</td>
<td>0</td>
</tr>
<tr>
<td>C</td>
<td>1</td>
<td>2</td>
<td>C</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>D</td>
<td>3</td>
<td>1</td>
<td>D</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>E</td>
<td>1</td>
<td>2</td>
<td>E</td>
<td>4</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Routings from D</th>
<th></th>
<th>Routings from E</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>To</strong></td>
<td><strong>Link</strong></td>
<td><strong>Cost</strong></td>
<td><strong>To</strong></td>
</tr>
<tr>
<td>A</td>
<td>3</td>
<td>1</td>
<td>A</td>
</tr>
<tr>
<td>B</td>
<td>3</td>
<td>2</td>
<td>B</td>
</tr>
<tr>
<td>C</td>
<td>6</td>
<td>2</td>
<td>C</td>
</tr>
<tr>
<td>D</td>
<td>local</td>
<td>0</td>
<td>D</td>
</tr>
<tr>
<td>E</td>
<td>6</td>
<td>1</td>
<td>E</td>
</tr>
</tbody>
</table>
A simple routing algorithm

- 'distance vector' algorithm, instance of Bellman’s shortest path algorithm (Bellman 1957) (being basis for *link-state algorithm* (in use since 1979))

**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.

### 2.3.6 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
2.3.7 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 an alien network

**Figure 3.11 Tunnelling for IPv6 migration**

IPv6 encapsulated in IPv4 packets
2.4 Internet protocols

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

Figure 3.12 TCP/IP layers

Layers

- Application
- Transport
- Internet
- Network interface
- Underlying network

Message

- Messages (UDP) or Streams (TCP)
- UDP or TCP packets
- IP datagrams
- Network-specific frames
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
Sockets

socket – abstraction providing an endpoint for communication between processes

Figure 4.2 Sockets and ports

- Messages sent to a particular Internet address and port number can be received only by a process whose socket is associated with that Internet address and port number

- Processes may use the same socket (associated either with UDP or TCP protocol) for sending and receiving messages
Use of ports

IP communication between pairs of computers
Transport protocols – TCP and UDP
Port number – 16-bit integer

UDP (User Datagram Protocol) and TCP (Transmission Control Protocol)

UDP features

- almost a transport-level replica of IP
- UDP adds no additional reliability mechanisms except the checksum (optional in IPv4)
- restricted to applications and services not requiring reliable delivery of single or multiple messages
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 – 64 kB (incl. headers), but in most environments ≤ 8 kB
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
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
# Python: creating a server socket and reply the sender
# with message in UPPERCASE_<message_number>
from socket import *
import random
serverSocket = socket(AF_INET, SOCK_DGRAM)
serverSocket.bind((' ', 12000))
i=0
while True:
    message, address = serverSocket.recvfrom(1024)
    message = message.upper() + '_' + str(i)
    rand = random.randint(0, 10)
    i=i+1
    if rand >= 4: # answer only some of the requests...:
        serverSocket.sendto(message, address)
# Python: client sends a message over UDP and
# gets a reply with the message in UPPER-CASE#:  

```python
import time
from socket import *
for pings in range(10):
    clientSocket = socket(AF_INET, SOCK_DGRAM)
    clientSocket.settimeout(1)
    message = 'This is a Test'
    addr = ('127.0.0.1', 12000)
    start = time.time()
    clientSocket.sendto(message, addr)
    try:
        data, server = clientSocket.recvfrom(1024)
        end = time.time()
        elapsed = end - start
        print '%s%d%.e' % (data, pings, elapsed)
    except timeout:
        print pings, '---REQUEST_TIMED_OUT'
```
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

...more on TCP will follow later...
2.4.1 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

Class A: 0   Network ID            Host ID

Class B: 1 0   Network ID            Host ID

Class C: 1 1 0   Network ID            Host ID

Class D (multicast): 1 1 1 0   Multicast address

Class E (reserved): 1 1 1 1 0   unused
### Figure 3.16 Decimal representation of Internet addresses

<table>
<thead>
<tr>
<th>Class</th>
<th>octet 1</th>
<th>octet 2</th>
<th>octet 3</th>
<th>Range of addresses</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class A:</td>
<td>1 to 127</td>
<td>0 to 255</td>
<td>0 to 255</td>
<td>1.0.0.0 to 127.255.255.255</td>
</tr>
<tr>
<td>Class B:</td>
<td>128 to 191</td>
<td>0 to 255</td>
<td>0 to 255</td>
<td>128.0.0.0 to 191.255.255.255</td>
</tr>
<tr>
<td>Class C:</td>
<td>192 to 223</td>
<td>0 to 255</td>
<td>0 to 255</td>
<td>192.0.0.0 to 223.255.255.255</td>
</tr>
<tr>
<td>Class D (multicast):</td>
<td>224 to 239</td>
<td>0 to 255</td>
<td>0 to 255</td>
<td>224.0.0.0 to 239.255.255.255</td>
</tr>
<tr>
<td>Class E (reserved):</td>
<td>240 to 255</td>
<td>0 to 255</td>
<td>0 to 255</td>
<td>240.0.0.0 to 255.255.255.255</td>
</tr>
</tbody>
</table>

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

Around 1990: – IP addresses likely to run out around 1996
• Classless InterDomain Routing (CIDR) using Variable Length Subnet Masks (VLSM) (See eg video: https://www.youtube.com/watch?v=Q1U9wVXRuHA)

• Specification of IPv6

• Network Address Transition (NAT) scheme to enable unregistered computers to access the Internet

2.4.2 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...!
2.4.3 IP routing

- Routing protocols RIP-1 and RIP-2

- *open shortest path first* (OSPF)

- default routes; key routers; default destination

- 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)

- three blocks of addresses (10.z.y.x, 172.16.y.x or 192.168.y.x) reserved for private internets
Dynamic Host Configuration Protocol (DHCP)
2.4.4 IP version 6

Figure 3.19 IPv6 header layout

<table>
<thead>
<tr>
<th>Version (4 bits)</th>
<th>Traffic class (8 bits)</th>
<th>Flow label (20 bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Payload length (16 bits)</td>
<td>Next header (8 bits)</td>
<td>Hop limit (8 bits)</td>
</tr>
<tr>
<td>Source address (128 bits)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Destination address (128 bits)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Address space: 128 bits

\[ 2^{128} \approx 3 \times 10^{38} \implies 7 \times 10^{23} \] 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
2.4.5 MobileIP

home agents and foreign agents etc...

Figure 3.20 The MobileIP routing mechanism

2.4.6 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.

2.4.7 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 Interprocess communication

3.1 Introduction

OUTLINE:
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?

3Textbook Chapter 4
3.2 The API for the Internet protocols

3.2.1 The characteristics of interprocess communication

Synchronous and asynchronous communication

A queue is associated with each message destination

**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 \((\text{Internet address}, \text{local port})\)
  - port:
    * has exactly one receiver process (except multicast)
    * can have multiple message senders
  - process can have many ports

Reliability & ordering – also important!

point-to-point message service is said to be \textbf{reliable} if messages are guaranteed to be delivered despite a reasonable number of packets being dropped or lost

Some applications demand that messages are delivered in \textit{sending order}
3.2.2 UDP datagram communication

Failure model for UDP datagrams

2 properties of reliable communication: validity and integrity

- **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

UDP datagrams suffer from

- Omission failures

- Ordering

applications need to provide their own checks if needed
3.2.3 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)

- at startup: one process plays client role, other plays server role
- thereafter they could be peers
Pair of sockets associated with a pair of streams – one in each direction

Issues related to stream communication:

- Matching data items – (e.g. int should be followed by float – matching in both side)

- 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

TCP is not completely reliable:

* the severely congested case!
Use of TCP
HTTP, FTP, Telnet, SMTP

Python: TCP example of client-server interaction

```python
# Python: creating a server TCP socket and reply the sender
#       with message in UPPER-CASE and <message_number>
import socket
import sys

# Create a TCP/IP socket
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

# Bind the socket to the port
server_address = ('localhost', 10000)

print >>sys.stderr, 'starting up on %s port %s' % server_address
sock.bind(server_address)

# Listen for incoming connections
sock.listen(1)

count=0

while True:
    count=count+1
```
# Wait for a connection

def wait_for_connection():
    print >>sys.stderr, 'waiting for a connection'
    try:
        connection, client_address = sock.accept()
        print >>sys.stderr, 'connection from', client_address
    except:
        pass

    while True:
        data = connection.recv(128)
        print >>sys.stderr, 'received %s' % data
        if data:
            data = data.upper() + ' ' + str(count)
            print >>sys.stderr, 'sending data back to the client'
            connection.sendall(data)
        else:
            print >>sys.stderr, 'no more data from', client_address
            connection.close()

# Clean up the connection
finally:
    connection.close()
import socket
import sys

# Create a TCP/IP socket
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

# Connect the socket to the port where the server is listening
server_address = ('localhost', 10000)

print('connecting to %s port %s' % server_address)
sock.connect(server_address)

try:
    # Send data
    message = 'Hello message ...
    print('sending %s' % message)
    sock.sendall(message)
    # Look for the response
    amount_received = 0
    amount_expected = len(message)
    while amount_received < amount_expected:
        data = sock.recv(128)
        amount_received += len(data)

    print('received %s' % data)

finally:
    print('closing socket')
    sock.close()
3.3 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 (if needed), 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 (from external data representation)

- CORBA’s (Common Object Request Broker Architecture) common data representation (bin, just values)
- Java’s object serialization (bin, data + type info)
- XML (Extensible Markup Language) (txt, may refer to externally defined namespaces)
- Google – protocol buffers (both stored and transmitted data)
- JSON (JavaScript Object Notation) http://www.json.org
• Python:
  
  – (module marshal – internal python object serialisation – writing pseudo-compiled code into .pyc-files)
  
  – object serialisation module: **pickle**
  
  – + persistent dictionary of objects: **shelve**
3.3.1 CORBA’s Common Data Representation (CDR)

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

<table>
<thead>
<tr>
<th>index in sequence of bytes</th>
<th>4 bytes</th>
<th>notes on representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0–3</td>
<td>5</td>
<td>length of string</td>
</tr>
<tr>
<td>4–7</td>
<td>&quot;Smit&quot;</td>
<td>‘Smith’</td>
</tr>
<tr>
<td>8–11</td>
<td>&quot;h___&quot;</td>
<td></td>
</tr>
<tr>
<td>12–15</td>
<td>6</td>
<td>length of string</td>
</tr>
<tr>
<td>16–19</td>
<td>&quot;Lond&quot;</td>
<td>‘London’</td>
</tr>
<tr>
<td>20–23</td>
<td>&quot;on___&quot;</td>
<td></td>
</tr>
<tr>
<td>24–27</td>
<td>1984</td>
<td>unsigned long</td>
</tr>
</tbody>
</table>

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
Python: omniORBpy module

Sun XDR standard

- similar to CORBA in many ways
- sending messages between clients and servers in Sun NFS
- http://www.cdk5.net/ipc
3.3.2 Java object serialization

```java
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
Interprocess communication 3.3 External data representation and marshalling

**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

<table>
<thead>
<tr>
<th>Serialized values</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Person</td>
<td>8-byte version number</td>
</tr>
<tr>
<td>3</td>
<td>int year</td>
</tr>
<tr>
<td>1984</td>
<td>5 Smith</td>
</tr>
</tbody>
</table>

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

3.3.3 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
121 Interprocess communication 3.3 External data representation and marshalling

* specification of user interfaces
* encoding of configuration files in operating systems

- clients usually use SOAP messages to communicate with web services

**SOAP** (Simple Object Access Protocol) – 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

```xml
<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
123 Interprocess communication 3.3 External data representation and marshalling

set of rules e.g. XML prolog:

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

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

- For example:

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

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

```xml
<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

```xml
<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 (JAXB - Java Architecture for XML Binding)
- Python (PyXB: Python XML Schema Bindings) etc.

3.3.4 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

<table>
<thead>
<tr>
<th>32 bits</th>
<th>32 bits</th>
<th>32 bits</th>
<th>32 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Internet address</td>
<td>port number</td>
<td>time</td>
<td>object number</td>
</tr>
<tr>
<td>interface of remote object</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
3.4 Multicast communication

single message from one process to each of the members of a group of processes
good for constructing distributed systems with following properties:

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

3.4.1 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
  – number of routers it is allowed to pass – *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)
  - to constrain propagation

Failure model for multicast datagrams

- failure characteristics as UDP datagrams
- unreliable multicast
3.5 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
Figure 4.17 An overview of point-to-point communication in MPI

Example (Fortran90) 11.1 Greetings (http://www.ut.ee/~eero/SC/konspekt/Naited/greetings.f90.html)
# Python example on sending an object (pickle under the hood)

```python
from mpi4py import MPI

comm = MPI.COMM_WORLD
rank = comm.Get_rank()

if rank == 0:
    data = {'a': 7, 'b': 3.14}
    comm.send(data, dest=1, tag=11)

elif rank == 1:
    data = comm.recv(source=0, tag=11)
```
### Figure 4.18 Selected send operations in MPI

<table>
<thead>
<tr>
<th>Send operations</th>
<th>Blocking</th>
<th>Non-blocking</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Generic</strong></td>
<td><em>MPI_Send</em>: 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.</td>
<td><em>MPI_Isend</em>: 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 <em>MPI_Wait</em> or <em>MPI_Test</em>.</td>
</tr>
<tr>
<td><strong>Synchronous</strong></td>
<td><em>MPI_Ssend</em>: the sender and receiver synchronize and the call only returns when the message has been delivered at the receiving end.</td>
<td><em>MPI_Isend</em>: as with <em>MPI_Isend</em>, but with <em>MPI_Wait</em> and <em>MPI_Test</em> indicating whether the message has been delivered at the receive end.</td>
</tr>
<tr>
<td><strong>Buffered</strong></td>
<td><em>MPI_Bsend</em>: the sender explicitly allocates an MPI buffer library (using a separate <em>MPI_Buffer_attach</em> call) and the call returns when the data is successfully copied into this buffer.</td>
<td><em>MPI_Ibsend</em>: as with <em>MPI_Isend</em> but with <em>MPI_Wait</em> and <em>MPI_Test</em> indicating whether the message has been copied into the sender's MPI buffer and hence is in transit.</td>
</tr>
<tr>
<td><strong>Ready</strong></td>
<td><em>MPI_Rsend</em>: the call returns when the sender's application buffer can be reused (as with <em>MPI_Send</em>), 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.</td>
<td><em>MPI_Irsend</em>: the effect is as with <em>MPI_Isend</em>, but as with <em>MPI_Rsend</em>, the programmer is indicating to the underlying implementation that the receiver is guaranteed to be ready to receive (resulting in the same optimizations),</td>
</tr>
</tbody>
</table>
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

4 Textbook Chapter 2
Difficulties and threats for distributed systems:

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

4.2 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**

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

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

4.3.1 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

  - What type of transparency do RPC systems offer?
    * 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:
– (publish-subscribe systems – sometimes also called distributed event-based systems)

– publishers distribute information items of interest (events) to possibly a 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

<table>
<thead>
<tr>
<th>Communicating entities (what is communicating)</th>
<th>Communication paradigms (how they communicate)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>System-oriented entities</strong></td>
<td><strong>Interprocess communication</strong></td>
</tr>
<tr>
<td>Nodes</td>
<td>Message passing</td>
</tr>
<tr>
<td>Processes</td>
<td>Request-reply</td>
</tr>
<tr>
<td><strong>Problem-oriented entities</strong></td>
<td><strong>Remote invocation</strong></td>
</tr>
<tr>
<td>Objects</td>
<td>Group communication</td>
</tr>
<tr>
<td>Components</td>
<td>Publish-subscribe</td>
</tr>
<tr>
<td>Web services</td>
<td>Message queues</td>
</tr>
<tr>
<td></td>
<td>Tuple spaces</td>
</tr>
<tr>
<td></td>
<td>DSM</td>
</tr>
</tbody>
</table>
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 and every node in the model
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** is a 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...
4.3.2 Architectural patterns

Architectural Pattern: *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

- **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
Architectural Pattern: 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
• three aspects partitioned into two processes
• (+) low latency
• (-) splitting application logic

• (+) 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**

```javascript
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.)
    ....
}
Architectural Pattern: *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
Architectural Patterns: *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

  * **interoperability** - ability of different information technology systems and software applications to communicate, exchange data, and use the information that has been exchanged

  - 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 and intercession
    * 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

4.3.3 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

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

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

### 4.4.1 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
4.4.2 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**

- cheaf 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**

<table>
<thead>
<tr>
<th>Class of failure</th>
<th>Affects</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fail-stop</td>
<td>Process</td>
<td>Process halts and remains halted. Other processes may detect this state.</td>
</tr>
<tr>
<td>Crash</td>
<td>Process</td>
<td>Process halts and remains halted. Other processes may not be able to detect this state.</td>
</tr>
<tr>
<td>Omission</td>
<td>Channel</td>
<td>A message inserted in an outgoing message buffer never arrives at the other end’s incoming message buffer.</td>
</tr>
<tr>
<td>Send-omission</td>
<td>Process</td>
<td>A process completes a <em>send</em>, but the message is not put in its outgoing message buffer.</td>
</tr>
<tr>
<td>Receive-omission</td>
<td>Process</td>
<td>A message is put in a process’s incoming message buffer, but that process does not receive it.</td>
</tr>
<tr>
<td>Arbitrary (Byzantine)</td>
<td>Process or channel</td>
<td>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.</td>
</tr>
</tbody>
</table>
Timing failures

- applicable in synchronous distributed systems

**Figure 2.16 Timing failures**

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

Failure masking

- 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
4.4.3 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

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

Performing security analysis

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
5 Remote invocation

5.1 Introduction

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.
5.2 Request-reply protocols

1. Request-reply protocols; provide direct support also to:

2. RPC (1984) and

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
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

```java
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

<table>
<thead>
<tr>
<th>messageType</th>
<th>int  (0=Request, 1=Reply)</th>
</tr>
</thead>
<tbody>
<tr>
<td>requestId</td>
<td>int</td>
</tr>
<tr>
<td>remoteReference</td>
<td>RemoteRef</td>
</tr>
<tr>
<td>operationId</td>
<td>int or Operation</td>
</tr>
<tr>
<td>arguments</td>
<td>array of bytes</td>
</tr>
</tbody>
</table>
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

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

Message content histories

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**

<table>
<thead>
<tr>
<th>Name</th>
<th><strong>Messages sent by</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><strong>Client</strong></td>
</tr>
<tr>
<td>R</td>
<td>Request</td>
</tr>
<tr>
<td>RR</td>
<td>Request</td>
</tr>
<tr>
<td>RRA</td>
<td>Request</td>
</tr>
</tbody>
</table>
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
Remote invocation

5.2 Request-reply protocols

- 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 for 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)

    * not idempotent – consult with the user

- Requests and replies are marshalled into messages as ASCII text strings, but
Remote invocation

5.2 Request-reply protocols

- 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

<table>
<thead>
<tr>
<th>method</th>
<th>URL or pathname</th>
<th>HTTP version</th>
<th>headers</th>
<th>message body</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET</td>
<td>//www.dcs.qmw.ac.uk/index.html</td>
<td>HTTP/1.1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Figure 5.7 HTTP Reply message

<table>
<thead>
<tr>
<th>HTTP version</th>
<th>status code</th>
<th>reason</th>
<th>headers</th>
<th>message body</th>
</tr>
</thead>
<tbody>
<tr>
<td>HTTP/1.1</td>
<td>200</td>
<td>OK</td>
<td></td>
<td>resource data</td>
</tr>
</tbody>
</table>
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
Remote invocation

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
Remote invocation

5.3 Remote procedure call (RPC)

- 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();
};
Remote procedure call (RPC)

**RPC call semantics**

*doOperation* implementations with different delivery guarantees:

- Retry request message
- Duplicate filtering
- Retransmission of results

Figure 5.9 Call semantics

<table>
<thead>
<tr>
<th>Fault tolerance measures</th>
<th>Call semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Retransmit request message</td>
<td>Duplicate filtering</td>
</tr>
<tr>
<td>No</td>
<td>Not applicable</td>
</tr>
<tr>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>
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 – if the request message is retransmitted, the remote procedure may be executed more than once => possibly wrong values stored or returned
  - if idempotent operations on server – 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

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
Remote invocation

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

Figure 5.11 Files interface in Sun XDR

```c
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 = 9999
```
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](http://www.cdk5.net/rmi)
5.3.4 Python RPC

- **Native Python-based protocols:**
  - RPyC (Remote Python Call)
  - Circuits

- **RPC frameworks with a lot of underlying protocols:**
  - Spyne (see lightning talk)

- **JSON-RPC based frameworks:**
  - python-symmetric-jsonrpc
  - rpcbd
  - Ladon

- **SOAP:**
  - ZSI
  - SOAPpy
  - Ladon
  - Spyne

- **XML-RPC based frameworks:**
  - XMLRPC, using the xmlrpc-pclib and SimpleXMLRPCServer modules in the standard library.
# Example of SimpleXML–RPC server in Python
# https://docs.python.org/2/library/simplerpcserver.html
from SimpleXMLRPCServer import SimpleXMLRPCServer
from SimpleXMLRPCServer import SimpleXMLRPCRequestHandler
from time import time

# Restrict to a particular path.
class RequestHandler(SimpleXMLRPCRequestHandler):
    rpc_paths = (’/RPC2’,)
if __name__ == '__main__':
    # Create server
    server = SimpleXMLRPCServer((’127.0.0.1’, 8000),
        requestHandler=RequestHandler)
    server.register_introspection_functions()
    # Register a function under a different name
    def adder_function(x, y):
        return x + y
    server.register_function(adder_function, ’add’)
    def get_time_function():
        return time()
    server.register_function(get_time_function, ’get_time’)
    # Run the server ’s main loop
    server.serve_forever()
# Example of SimpleXML–RPC client in Python

# https://docs.python.org/2/library/simplexmlrpcserver.html

```python
import xmlrpclib
from time import localtime, asctime

if __name__ == '__main__':
    # Connect to XML–RPC server
    s = xmlrpclib.ServerProxy('http://127.0.0.1:8000')
    print 'Connected to XML-RPC server'

    # Print list of available methods
    print 'Methods on XML-RPC server:
    print s.system.listMethods()

    print '2+3=', s.add(2,3)  # Returns 5
    print 'Current time is:', asctime(localtime(s.get_time()))
```

210  Remote invocation  5.3 Remote procedure call (RPC)
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 –
Remote invocation

Remote method invocation (RMI)

- local and remote calls employ the same syntax
- remote interfaces
  * typically expose the distributed nature of the underlying call e.g. support 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
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 need to be 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

![Diagram of 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
CORBA (Common Object Request Broker Architecture) interface definition language (IDL)

Java RMI – keyword: *Remote*

- **NB!** Remote interfaces like all interfaces, do not have 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 (for example, Figure 5.12 – remote object F reference could be obtained by A through remote invocation of a method in B)

- if the object L in Figure 5.14 contains a method for creating remote objects, then the remote invocations from C and K could lead to the instantiation of the objects M and N, respectively:

**Figure 5.14 Instantiation of remote objects**

![Diagram of remote object instantiation](image)
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 is communication module and remote reference module?
- 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
Remote invocation

- An entry for all the remote objects held by the process

- An entry for each local proxy

Actions of the remote reference module:

- remote object being passed as an argument or a result for the first time – create a remote object reference, and add it to the table

- remote object reference arriving in a request or reply message – look for the corresponding local object reference (which may refer either to a proxy or to a remote object)

  - if 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
• proxy 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
But what if a client program receives a remote reference to an object whose remote interface was not available at compile time?

**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

  - (book Chapter 8 on CORBA)

  - but Java RMI has generic dispatcher and the dynamic downloading of classes to the server
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
Remote method invocation (RMI)

- 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:
Remote invocation

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

CORBA Naming Service – Chapter 8

The Java binder – RMIRegistry, 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
Remote invocation

5.4 Remote method invocation (RMI)

- 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 $\rightarrow$ 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)
Remote interfaces in Java RMI

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

Example: `shared whiteboard`

Figure 5.16 Java Remote interfaces Shape and ShapeList

```java
import java.rmi.*;
import java.util.Vector;

public interface Shape extends Remote {
    // i.e. Shape is a remote interface
    int getVersion() throws RemoteException;
    GraphicalObject getAllState() throws RemoteException; // 1
}

public interface ShapeList extends Remote {
    // the return value of the method newShape is defined as Shape – a remote interface:
    Shape newShape(GraphicalObject g) throws RemoteException; // 2
    Vector allShapes() throws RemoteException;
    int getVersion() throws RemoteException;
}
```
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. 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; two possibilities:

- 1. 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:
2. Setting up a system-wide binding service

- run an instance of the RMI registry in the networked environment
- use the class LocateRegistry in java.rmi.registry, to discover this registry
  
  * getRegistry method that returns an object of type Registry representing the remote binding service:

```java
public static Registry getRegistry() throws RemoteException
```

- issue a call of rebind on this returned Registry object
Figure 5.17 The Naming class of Java RMIregistry

```java
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

```java
package examples.RMIShape;
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;
public class ShapeListServer {
    public static void main(String args[]){
        System.setSecurityManager(new RMISecurityManager());
        try {
            ShapeList aShapelist = new ShapeListServant();
            ShapeList stub = (ShapeList) UnicastRemoteObject.exportObject(aShapelist, 0);
            Naming.rebind("ShapeList", aShapelist);
            System.out.println("ShapeList server ready");
        } catch(Exception e) {
            System.out.println("ShapeList server main" + e.getMessage());
        }
    }
}
```
Figure 5.19 Java class ShapeListServant implements interface ShapeList

```java
import java.util.Vector;
public class ShapeListServant implements ShapeList{
    private Vector theList;
    private int version;
    public ShapeListServant() throws RemoteException{
        theList = new Vector();
        version = 0;
    }
    public Shape newShape(GraphicalObject g) throws RemoteException{
        // 1
        version++;
        Shape s = new ShapeServant(g, version); //2
        theList.addElement(s);
        return s;
    }
    public Vector allShapes() throws RemoteException{
        return theList;
    }
    public int getVersion() throws RemoteException{
        return version;
    }
}
```
Client program

Figure 5.20 Java client ShapeList

```java
import java.rmi.*;
import java.rmi.server.*;
import java.util.Vector;
import java.awt.Rectangle;
import java.awt.Color;

public class ShapeListClient{
    public static void main(String args[]){
        String option = "Read";
        String shapeType = "Rectangle";
        if(args.length > 0) option = args[0]; // read or write
        if(args.length > 1) shapeType = args[1]; // specify Circle, Line etc
        System.out.println("option= "+option+" shape= "+shapeType);
        if(System.getSecurityManager() == null){
            System.setSecurityManager(new RMISecurityManager());
        } else System.out.println("Already has a security manager, so cant set RMI SM");
        ShapeList aShapeList = null;
        try{
            aShapeList = (ShapeList) Naming.lookup("//Jean.torriano.net/ShapeList"); // 1
            System.out.println("Found server");
            Vector sList = aShapeList.allShapes(); // 2
        }
    }
}
```
Remote invocation

System.out.println("Got vector");
if (option.equals("Read")) {
    for (int i = 0; i < sList.size(); i++) {
        GraphicalObject g = ((Shape)sList.elementAt(i)).getAllState();
        g.print();
    }
} else {
    GraphicalObject g = new GraphicalObject(shapeType, new Rectangle(50, 50, 300, 400), Color.red,
                                             Color.blue, false);
    System.out.println("Created graphical object");
    aShapeList.newShape(g);
    System.out.println("Stored shape");
}
} catch (RemoteException e) {System.out.println("allShapes: "+ e.getMessage());}
} catch (Exception e) {System.out.println("Lookup: "+ e.getMessage());}
Remote invocation

5.5 Case study: Java RMI

code of ShapeServant class

```java
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;

public class ShapeServant extends UnicastRemoteObject implements Shape {
    int myVersion;
    GraphicalObject theG;

    public ShapeServant(GraphicalObject g, int version) throws RemoteException {
        theG = g;
        myVersion = version;
    }

    public int getVersion() throws RemoteException {
        return myVersion;
    }

    public GraphicalObject getAllState() throws RemoteException {
        return theG;
    }
}
```
import java.awt.Rectangle;
import java.awt.Color;
import java.io.Serializable;
public class GraphicalObject implements Serializable{
    public String type;
    public Rectangle enclosing;
    public Color line;
    public Color fill;
    public boolean isFilled;

    // constructors
    public GraphicalObject() { }

    public GraphicalObject(String aType, Rectangle anEnclosing, Color aLine, Color aFill, boolean anIsFilled) {
        type = aType;
        enclosing = anEnclosing;
        line = aLine;
        fill = aFill;
        isFilled = anIsFilled;
    }
}
public void print() {
    System.out.print(type);
    System.out.print(enclosing.x + "\,\,\) + enclosing.y + "\,\,\) + enclosing.
    width + "\,\,\) + enclosing.height);
    if (isFilled) System.out.println("filled"); else System.out.println("not filled");
}

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
Remote invocation

5.5 Case study: Java RMI

- 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:

```java
int register(WhiteboardCallback callback) throws RemoteException;
void deregister(int callbackId) throws RemoteException;
```
# Example of Pyro4 remote-objects server in Python
# http://pythonhosted.org/Pyro4/intro.html#simple-example
# saved as greeting-server.py

import Pyro4

@Pyro4.expose
class GreetingMaker(object):
    def get_fortune(self, name):
        return "Hello, {0}. Here is your fortune message:\n" "Behold the warranty—-the bold print giveth
" "and the fine print taketh away.". format(name)

if __name__ == '__main__':
    # make a Pyro daemon
    daemon = Pyro4.Daemon()

    # register the greeting maker as a Pyro object
    uri = daemon.register(GreetingMaker)

    # print the uri so we can use it in the client later
    print "Ready. Object uri =", uri

    # start the event loop of the server to wait for calls
    daemon.requestLoop()
# Example of Pyro4 remote-objects client in Python
# http://pythonhosted.org/Pyro4/intro.html#simple-example
# saved as greeting-client.py

import Pyro4

if __name__ == '__main__':

    uri = raw_input("Pyro uri of the greeting object? ").strip()
    name = raw_input("What is your name? ").strip()

    # get a Pyro proxy to the greeting object
    greeting_maker = Pyro4.Proxy(uri)

    # call method normally
    print greeting_maker.get_fortune(name)
6 Operating systems support

See please separate slides: Artjom Lind: Operating systems support
7 Distributed objects and components

7.1 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, CORBA, Pyro

---

6Textbook Chapter 8
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
7.2 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, CORBA, Pyro – quite common

- but CORBA – language independent

in DO the term class is avoided – instead factory instantiating new objects from a given template
• in DO – interface inheritance:
  
  – new interface inherits the method signatures of the original interface
    
    + can add extra ones
## Figure 8.1 Distributed objects

<table>
<thead>
<tr>
<th>Objects</th>
<th>Distributed objects</th>
<th>Description of distributed object</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object references</td>
<td>Remote object references</td>
<td>Globally unique reference for a distributed object; may be passed as a parameter.</td>
</tr>
<tr>
<td>Interfaces</td>
<td>Remote interfaces</td>
<td>Provides an abstract specification of the methods that can be invoked on the remote object; specified using an interface definition language (IDL).</td>
</tr>
<tr>
<td>Actions</td>
<td>Distributed actions</td>
<td>Initiated by a method invocation, potentially resulting in invocation chains; remote invocations use RMI.</td>
</tr>
<tr>
<td>Exceptions</td>
<td>Distributed exceptions</td>
<td>Additional exceptions generated from the distributed nature of the system, including message loss or process failure.</td>
</tr>
<tr>
<td>Garbage collection</td>
<td>Distributed garbage collection</td>
<td>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.</td>
</tr>
</tbody>
</table>
Distributed objects and components

7.2 Distributed objects

OO: objects + class + inheritance $\leftrightarrow$ 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*
  
  – number of 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
7.3 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)
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: a range of application servers:

<table>
<thead>
<tr>
<th>Technology</th>
<th>Developed by</th>
<th>Further details</th>
</tr>
</thead>
<tbody>
<tr>
<td>WebSphere Application Server</td>
<td>IBM</td>
<td><a href="http://www.ibm.com">www.ibm.com</a></td>
</tr>
<tr>
<td>Enterprise JavaBeans</td>
<td>SUN</td>
<td>java.sun.com</td>
</tr>
<tr>
<td>Spring Framework</td>
<td>SpringSource</td>
<td><a href="http://www.springsource.org">www.springsource.org</a></td>
</tr>
<tr>
<td></td>
<td>(a division of VMware)</td>
<td></td>
</tr>
<tr>
<td>JBoss</td>
<td>JBoss Community</td>
<td><a href="http://www.jboss.org">www.jboss.org</a></td>
</tr>
<tr>
<td>CORBA Component Model</td>
<td>OMG</td>
<td>[Wang et al. 2001JOnAS]</td>
</tr>
<tr>
<td>JOnAS</td>
<td>OW2 Consortium</td>
<td>jonas.ow2.org</td>
</tr>
<tr>
<td>GlassFish</td>
<td>SUN</td>
<td>glassfish.dev.java.net</td>
</tr>
</tbody>
</table>
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, etc
7.4 Case study: Enterprise JavaBeans

- **Enterprise JavaBeans (EJB)** – server-side component model

- Benefits of Enterprise Beans
  - EJB container provides system-level services to enterprise beans, the bean developer can concentrate on solving business problems
  - Client developer can focus on the presentation of the client
  - Application assembler can build new applications from existing beans

- When should one use enterprise beans?
  - The application must be scalable. To accommodate a growing number of users, you may need to distribute an application’s components across multiple machines. Not only can the enterprise beans of an application run on different machines, but also their location will remain transparent to the clients.
Transactions must ensure data integrity. Enterprise beans support transactions, the mechanisms that manage the concurrent access of shared objects.

The application will have a variety of clients. With only a few lines of code, remote clients can easily locate enterprise beans. These clients can be thin, various, and numerous.

- **components** – known as beans
  - cover the application (or business) logic
  - possible to separate from data logic on persistent storage
    i.e. three-tier architecture

- EJB is managed – the container pattern is used to support key DS services:
  - transactions
  - security
- life-cycle support

- key services – either
  - container-managed – completely hidden from the developer
  - bean-managed – bean developer can take more control over these operations

- EJB specification (>200 pages) – following key roles:
  - bean provider – develops component(s)’ application logic
  - application assembler – makes an application configurations out of beans
  - deployer – deploys a given application assembly in a given operational environment
  - service provider – fundamental DS services
– *persistance provider* – managing persistent data and database connections

– *container provider* – builds upon above two roles (e.g. transactions, security)

– *system administrator* – role of monitoring, controlling at runtime to ensure correct operation

Clients talk to

- **EJB Server**, which hosts
  - **EJB Container**, which host
    - *Enterprise bean 1*
    - *Enterprise bean 2*
    - *
    - *Enterprise bean (n)*
• **EJB Server** – provides a Runtime Environment
  – provides system services and manages resources (through API)
    * Process and thread management
    * System resources management
    * Database connection pooling and caching

• **EJB Container** – provides a Run-time Environment for an Enterprise Bean
  – Hosts the Enterprise JavaBeans
  – Provides services to Enterprise JavaBeans
    * Naming
    * Life cycle management
    * Persistence (state management)
    * Transaction Management
Security

- Likely provided by server vendor

- Enterprise JavaBeans
  - A specialized Java class where the real business logic lives
    - May be developer-written or tool-generated
  - Distributed over a network
  - Transactional
  - Secure
  - Server vendors provide tools that automatically generate distribution, transaction and security behavior

- EJB Clients
– Client access is controlled by the container in which the enterprise Bean is deployed

– Clients locates an Enterprise JavaBean through Java Naming and Directory Interface (JNDI)

– RMI is the standard method for accessing a bean over a network

● 2 main types of beans:

– Session beans
  * stateful
    · persistent state
    · non-persistent state
  * stateless

Enterprise JavaBeans

![Diagram of Enterprise JavaBeans]
• **Message-driven beans**

  – clients interact with sessions beans via remote or local invocation

    ∗ EJB 2.0 supports indirect communication
      
      · message queues
      
      · topics

    ∗ listener-style interface reflecting the event-driven nature of the bean

Some simple examples on EJB 3.0:

https://docs.oracle.com/cd/E13222_01/wls/docs100.ejb30/examples.html
8 Security

See please separate slides: Amnir Hadachi: Security

7Textbook Chapter 11
10 Indirect communication

10.1 Introduction

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

(“... except for the problem of too many layers of indirection”)

Indirection – one of the fundamental concepts in computer science

(For example in computer programming: the ability to reference something using a name, reference, or container instead of the value itself)

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

8Textbook Chapter 6
Indirect communication

2 key properties stemming from the use of an intermediary:

1. *Space uncoupling*
   - the sender does not (need to) know the identity of the receiver(s)
   - receiver does not have to know the sender

   ⇒ 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

<table>
<thead>
<tr>
<th>Space coupling</th>
<th>Time-coupled</th>
<th>Time-uncoupled</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Properties:</strong> Communication directed towards a given receiver or receivers; receiver(s) must exist at that moment in time</td>
<td><strong>Properties:</strong> Communication directed towards a given receiver or receivers; sender(s) and receiver(s) can have independent lifetimes</td>
<td></td>
</tr>
<tr>
<td><strong>Examples:</strong> Message passing, remote invocation</td>
<td><strong>Examples:</strong> systems similar to e-mail exchange...</td>
<td></td>
</tr>
<tr>
<td><strong>Properties:</strong> Sender does not need to know the identity of the receiver(s); receiver(s) must exist at that moment in time</td>
<td><strong>Properties:</strong> Sender does not need to know the identity of the receiver(s); sender(s) and receiver(s) can have independent lifetimes</td>
<td></td>
</tr>
<tr>
<td><strong>Examples:</strong> IP multicast (introduced earlier)</td>
<td><strong>Examples:</strong> Most indirect communication paradigms covered in this chapter</td>
<td></td>
</tr>
</tbody>
</table>
The relationship with asynchronous communication

- In asynchronous communication, a sender sends a message and then continues (without blocking) \( \Rightarrow \) 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

10.2 Group communication

Group communication – a message is sent to a group \( \rightarrow \) 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

10.2.1 The programming model

- `group & group` membership ← processes may join or leave the group
  (In Java: `aGroup.send(aMessage)`)

**process groups**

- achievable using e.g. RPC
- **`JGroups toolkit`**

**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 – can a node belong to multiple groups?

synchronous and asynchronous systems – both cases are of interest
10.2.2 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

- **Causal 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 is 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
- Group communication is most effective in small scale
10.2.3 Case study: the JGroups toolkit

Figure 6.4 The architecture of JGroups

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

<table>
<thead>
<tr>
<th>Protocol stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>CAUSAL</td>
</tr>
<tr>
<td>GMS</td>
</tr>
<tr>
<td>MERGE</td>
</tr>
<tr>
<td>FRAG</td>
</tr>
<tr>
<td>UDP</td>
</tr>
</tbody>
</table>

- **Applications**
- **Building blocks**
- **Channel**

The 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
import org.jgroups.JChannel;

public class FireAlarmJG {
    public void raise() {
        try {
            JChannel channel = new JChannel();
            channel.connect("AlarmChannel");
            Message msg = new Message(null, null, "Fire!"); // destination, source, payload
            // destination = null — distribute to whole group; source null — source added automatically by the system anyway
            channel.send(msg);
        }
        catch (Exception e) {
        }
    }
}

FireAlarmJG alarm = new FireAlarmJG(); // create a new instance of the FireAlarmJG class
alarm.raise(); // raise an alarm
Figure 6.6 Java class FireAlarmConsumerJG

```java
import org.jgroups.JChannel;

public class FireAlarmConsumerJG {
    public String await() {
        try {
            JChannel channel = new JChannel();
            channel.connect("AlarmChannel");
            Message msg = (Message) channel.receive(0);
            // parameter: timeout zero — the receive message will block until a message is received
            // incoming messages are buffered and receive returns the top element in the buffer
            return (String) msg.getObject();
        } catch (Exception e) {
            return null;
        }
    }
}

FireAlarmConsumerJG alarmCall = new FireAlarmConsumerJG(); // (...receiver code...)
String msg = alarmCall.await();
System.out.println("Alarm received:");
10.3 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**

### 10.3.1 The programming model

**Figure 6.8 The publish-subscribe paradigm**

- (un)publish(event);
- (un)subscribe(filter);
- advertise(filter);
- notify(event)
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 (Textbook 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**
Indirect communication

10.3 Publish-subscribe systems

- 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
10.3.2 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 for content-based routing (CBR):
• **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)
Indirect communication

10.3 Publish-subscribe systems

- 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

```plaintext
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
```
* 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

```plaintext
upon receive publish(event e) from node x at node i
    rvlist := EN(e);
    if i in rvlist then begin
        matchlist <- match(e, subscriptions);
        send notify(e) to matchlist;
    end
    send publish(e) to rvlist - i;

upon receive subscribe(subscription s) from node x at node i
    rvlist := SN(s);
    if i in rvlist then
        add s to subscriptions;
    else
        send subscribe(s) to rvlist - i;
```

- distributed hash table (DHT) – can be used
  - hash table distributed over a set of nodes in P2P manner
Gossiping – nodes in the network periodically and probabilistically exchange events (or data) with neighbouring nodes

10.3.3 Examples of publish-subscribe systems

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

10.4.1 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
Indirect communication

10.4 Message queues

- 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:
Indirect communication

10.4 Message queues

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

10.4.3 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, properties and body:

– 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

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

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

```java
import javax.jms.*;
import javax.naming.*;
public class FireAlarmConsumerJMS {
    public String await() {
        try {
            Context ctx = new InitialContext();
            TopicConnectionFactory topicFactory =
                (TopicConnectionFactory) ctx.lookup("TopicConnectionFactory");
            Topic topic = (Topic) ctx.lookup("Alarms");
            TopicConnection topicConn =
                topicConnectionFactory.createTopicConnection();
            TopicSession topicSess = topicConn.createTopicSession(false,
                Session.AUTO_ACKNOWLEDGE); // ... identical unto here
            TopicSubscriber topicSub = topicSess.createSubscriber(topic);
            topicSub.start(); // topic subscriber created and started
            TextMessage msg = (TextMessage) topicSub.receive(); // receive
            return msg.getText(); // return message as string
        }
    }
}
```
```java
} catch (Exception e) {
    return null;
}
```

// class usage by a consumer:
FireAlarmConsumerJMS alarmCall = new FireAlarmConsumerJMS();
String msg = alarmCall.await();
System.out.println("Alarm received: "+msg);
Example: Python snakeMQ example

```
Simple Peer-to-Peer messaging based on SnakeMQ Framework

# Installing SnakeMQ
pip install snakemq

# Running the Peer "Alice"
python snakemq-peer.py -N Alice

# Running the Peer "Bob"
python snakemq-peer.py -P 7778 -H 127.0.0.1 -p 7777 -N Bob

# Make sure there is at least 2 peers interconnected
# Init user input by pressing Enter
# Type in the message to send or type Q to exit, hit Enter

import snakemq.link
import snakemq.packeter
import snakemq.messaging
import snakemq.message
from threading import Thread
from argparse import ArgumentParser
DEFAULT_LISTEN_INET_ADDR = '127.0.0.1'
DEFAULT_LISTEN_PORT = 7777
```
class SnakePeer:
    def __init__(self, myname, mysock_addr):
        self.__myname = myname
        self.__link = snakemq.link.Link()
        self.__packeter = snakemq.packeter.Packeter(self.__link)
        self.__messaging = 
            snakemq.messaging.Messaging(myname, '', self.__packeter)
        self.__mysock_addr = mysock_addr
        self.__mypeers = []

    def listen(self):
        self.__link.add_listener(self.__mysock_addr)
        print 'MyPeer[%s] is listening on %s:%d'

    def connect(self, peersock_addr):
        self.__link.add_connector(peersock_addr)
        print 'MyPeer[%s:%d] is connected to %s:%d'

    def run(self):
        def __on_recv(conn, ident, message):
            print '
Message from [%s]: %s' % (ident, message.data)
        def __on_connect(conn, ident):
            print '
Peer [%s] connected' % ident
            self.__mypeers.append(ident)
        def __on_disconnect(conn, ident):
```python
print ('
Peer_%s disconnected' % ident)
self.__mypeers.remove(ident)
self.__messaging.on_message_recv.add(__on_recv)
selself.__messaging.on_connect.add(__on_connect)
selself.__messaging.on_disconnect.add(__on_disconnect)
selself.__link.loop()

def send(self, to, message):
    # drop after 600 seconds if the message can’t be delivered
    message = snakemq.message.Message(message, ttl=600)
    self.__messaging.send_message(to, message)

def sendtoall(self, message):
    if len(self.__mypeers) > 0:
        for p in self.__mypeers:
            self.send(p, message)

def getonlinepeers(self):
    return [p for p in self.__mypeers]

def stop(self):
    self.__link.stop()
    self.__link.cleanup()

def send_user_input(mypeer):
    print ('Starting input processor'
while 1:
    raw_input('Hit Enter to init user−input... ')
    print ('Currently online peers:'
```
```python
print('[%s]'.join(mypeer.getonlinepeers()))
m = raw_input('Enter message to send to all or Q to exit: ')
if len(m) <= 0:
    continue
elif m == 'Q':
    mypeer.stop()
    return
else:
    mypeer.sendtoall(m)
if __name__ == '__main__':
    parser = ArgumentParser(description='Simple Peer-to-Peer messaging based on SnakeMQ Framework',
        version='0.0.0.1')
    parser.add_argument('−L', '−−listenaddr',
        help='Bind listener socket to INET address',
        'defaults to %s' % DEFAULT_LISTEN_INET_ADDR, 
        default=DEFAULT_LISTEN_INET_ADDR)
    parser.add_argument('−P', '−−listenport', type=int,
        help='Bind listener socket to port',
        'defaults to %d' % DEFAULT_LISTEN_PORT, 
        default=DEFAULT_LISTEN_PORT)
    parser.add_argument('−H', '−−connecthost',
        help='Connect to remote peer\'s IP',

```python
default='')
parser.add_argument(’–p’, ’—connectport’, type=int, \ 
     help=’Connect_to_remote_peer’’s_port’, \
     default=0)
parser.add_argument(’–N’, ’—myname’, \ 
     help=’The_name_to_show_to_the_other_peers’, \
     required=True)
args = parser.parse_args()
listen_addr = (args.listenaddr, int(args.listenport))
mypeer = SnakePeer(args.myname, listen_addr)
mypeer.listen()
if len(args.connecthost) >0:
    remote_peeraddr = (args.connecthost, int(args.connectport))
    mypeer.connect(remote_peeraddr)
t = Thread(name=’InputProcessor’’,\ 
     target=send_user_input , args=(mypeer,))
t.start()
mypeer.run()
t.join()
print ’Terminating...’
```
10.5 Shared memory approaches

10.5.1 Distributed shared memory (DSM)

- 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*
Indirect communication

Shared memory approaches

- 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
10.5.2 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 data 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
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 smartphones)

• 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, persistent 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

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

- 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

```java
import net.jini.core.entry.*;

public class AlarmTupleJS implements Entry {
    public String alarmType;
    public AlarmTupleJS() {
    }
    public AlarmTupleJS(String alarmType) {
        this.alarmType = alarmType;
    }
}
```

Figure 6.25 Java class FireAlarmJS

```java
import net.jini.space.JavaSpace;

public class FireAlarmJS {
    public void raise() {
        try {
            JavaSpace space = SpaceAccessor.findSpace("AlarmSpace");
            AlarmTupleJS tuple = new AlarmTupleJS("Fire!");
            space.write(tuple, null, 60*60*1000);
        } catch (Exception e) {
        }
    }
}
```
// the code can be called using:
FireAlarmJS alarm = new FireAlarmJS();
alarm.raise();

Figure 6.26 Java class FireAlarmReceiverJS

```java
import net.jini.space.JavaSpace;

public class FireAlarmConsumerJS {
    public String await() {
        try {
            JavaSpace space = SpaceAccessor.findSpace();
            AlarmTupleJS template = new AlarmTupleJS("Fire!");
            AlarmTupleJS recvd = (AlarmTupleJS) space.read(template, null,
                                                            Long.MAX_VALUE);

            return recvd.alarmType;
        }
        catch (Exception e) {
            return null;
        }
    }
}
```
// consumer:
FireAlarmConsumerJS alarmCall = new FireAlarmConsumerJS();
String msg = alarmCall.await();
System.out.println("Alarm received: " + msg);
Contents

0 Introduction 9

0.1 Syllabus .................................................................................................................. 10
0.1.1 Lectures: ............................................................................................................. 10
0.1.2 Practical sessions ................................................................................................. 11
0.1.3 Homework ............................................................................................................ 12

0.2 Literature .................................................................................................................. 13
0.2.1 Textbook ............................................................................................................... 13
0.2.2 Additional reading ............................................................................................... 13

1 Characterization of distributed systems9 15

1.1 Introduction ............................................................................................................... 16
1.2 Examples of distributed systems ............................................................................. 18
1.2.1 Web search ......................................................................................................... 20

9Textbook Chapter 1
1.2.2 Massively multiplayer online games (MMOGs) ................. 20
1.2.3 Financial trading ............................................. 21

1.3 Trends in distributed systems ..................................... 22
1.3.1 Pervasive networking and the modern Internet ............... 22
1.3.1 Mobile and ubiquitous computing .............................. 25
1.3.2 Distributed multimedia systems ................................. 27
1.3.3 Distributed computing as a utility .............................. 28

1.4 Sharing resources .................................................. 30
1.5 Challenges .......................................................... 32
1.5.1 DS CHALLENGE – Heterogeneity ............................ 32
1.5.2 DS CHALLENGE – Openness ................................. 34
1.5.3 DS CHALLENGE – Security .................................. 36
1.5.4 DS CHALLENGE – Scalability ............................... 37
1.5.5 DS CHALLENGE – Failure handling ......................... 38
1.5.6 DS CHALLENGE – Concurrency ............................. 40
1.5.7 DS CHALLENGE – Transparency ............................ 40
1.5.8 DS CHALLENGE – Quality of service .......................... 42

2 Networking and internetworking

2.1 Introduction ........................................................................ 45

2.1.1 Networking issues for distributed systems ......................... 47

2.2 Types of network ................................................................. 51

2.3 Network principles ............................................................... 56

2.3.1 Packet transmission ......................................................... 56

2.3.2 Data streaming ................................................................. 57

2.3.3 Switching schemes ......................................................... 58

2.3.4 Protocols ......................................................................... 60

2.3.5 Routing .......................................................................... 69

2.3.6 Congestion control ......................................................... 73

2.3.7 Internetworking ............................................................... 74

2.4 Internet protocols ............................................................... 77

\(^{10}\)Textbook Chapter 3
## Contents

### 2.4.1 IP addressing  

2.4.2 The IP protocol  

2.4.3 IP routing  

2.4.4 IP version 6  

2.4.5 MobileIP  

2.4.6 Domain names  

2.4.7 Firewalls  

### 3 Interprocess communication\(^{11}\)  

3.1 Introduction  

3.2 The API for the Internet protocols  

3.2.1 The characteristics of interprocess communication  

3.2.2 UDP datagram communication  

3.2.3 TCP stream communication  

3.3 External data representation and marshalling  

\(^{11}\)Textbook Chapter 4
3.3.1 CORBA’s Common Data Representation (CDR) .................................. 115
3.3.2 Java object serialization ................................................................. 117
3.3.3 Extensible Markup Language (XML) ............................................. 120
3.3.4 Remote object references ............................................................ 125
3.4 Multicast communication ................................................................. 126
3.4.1 IP multicast – An implementation of multicast communication 126
3.5 Case study: MPI ............................................................................. 129

4 System models\textsuperscript{12} ................................................................ 135
4.1 Outline ......................................................................................... 135
4.2 Physical models ........................................................................... 136
4.3 Architectural Models ..................................................................... 140
4.3.1 Architectural elements ............................................................... 140
4.3.2 Architectural patterns ................................................................. 156
4.3.3 Associated middleware solutions .............................................. 164

\textsuperscript{12}Textbook Chapter 2
4.4 Fundamental models ........................................ 167
  4.4.1 Interaction model ................................... 167
  4.4.2 Failure model ........................................ 172
  4.4.3 Security model ...................................... 177

5 Remote invocation\textsuperscript{13} 185
  5.1 Introduction ............................................. 185
  5.2 Request-reply protocols ................................ 186
  5.3 Remote procedure call (RPC) .......................... 198
    5.3.1 Design issues for RPC ............................ 198
    5.3.2 Implementation of RPC ............................ 204
    5.3.3 Case study: Sun RPC .............................. 205
    5.3.4 Python RPC ........................................ 208
  5.4 Remote method invocation (RMI) .................... 211
    5.4.1 Design issues for RMI ............................ 213

\textsuperscript{13}Textbook Chapter 5
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.4.2</td>
<td>Implementation of RMI</td>
<td>219</td>
</tr>
<tr>
<td>5.4.3</td>
<td>Distributed garbage collection</td>
<td>232</td>
</tr>
<tr>
<td>5.5</td>
<td>Case study: Java RMI</td>
<td>234</td>
</tr>
<tr>
<td>5.5.1</td>
<td>Building client and server programs</td>
<td>240</td>
</tr>
<tr>
<td>5.6</td>
<td>Example: Python Pyro4</td>
<td>249</td>
</tr>
<tr>
<td>6</td>
<td>Operating systems support</td>
<td>251</td>
</tr>
<tr>
<td>7</td>
<td>Distributed objects and components</td>
<td>252</td>
</tr>
<tr>
<td>7.1</td>
<td>Introduction</td>
<td>252</td>
</tr>
<tr>
<td>7.2</td>
<td>Distributed objects</td>
<td>254</td>
</tr>
<tr>
<td>7.3</td>
<td>From objects to components</td>
<td>259</td>
</tr>
<tr>
<td>7.4</td>
<td>Case study: Enterprise JavaBeans</td>
<td>270</td>
</tr>
<tr>
<td>8</td>
<td>Security</td>
<td>278</td>
</tr>
</tbody>
</table>

14 Textbook Chapter 8
15 Textbook Chapter 11
10 Indirect communication

10.1 Introduction

10.2 Group communication

10.2.1 The programming model
10.2.2 Implementation issues
10.2.3 Case study: the JGroups toolkit

10.3 Publish-subscribe systems

10.3.1 The programming model
10.3.2 Implementation issues
10.3.3 Examples of publish-subscribe systems

10.4 Message queues

10.4.1 The programming model
10.4.2 Implementation issues
10.4.3 Case study: The Java Messaging Service (JMS)

10.5 Shared memory approaches

16 Textbook Chapter 6
10.5.1 Distributed shared memory (DSM) ........................................ 320
10.5.2 Tuple space communication ....................................................... 323