MSIT-116D: Distributed Systems
Course Design and Editorial Committee

Prof. M.G. Krishnan
Vice Chancellor
Karnataka State Open University
Mukthagangotri, Mysore – 570 006

Prof. Vikram Raj Urs
Dean (Academic) & Convener
Karnataka State Open University
Mukthagangotri, Mysore – 570 006

Head of the Department and Course Co-Ordinator

Rashmi B.S
Assistant Professor & Chairperson
DoS in Information Technology
Karnataka State Open University
Mukthagangotri, Mysore – 570 006

Course Editor

Ms. Nandini H.M
Assistant professor of Information Technology
DoS in Information Technology
Karnataka State Open University
Mukthagangotri, Mysore – 570 006

Course Writers

Dr. Suresha
Professor
DoS in Computer Science,
University of Mysore, Mysore.

Smt. L. Hamsaveni
Assistant Professor
DoS in Computer Science,
University of Mysore, Mysore.

Publisher

Registrar
Karnataka State Open University
Mukthagangotri, Mysore – 570 006

Developed by Academic Section, KSOU, Mysore
Karnataka State Open University, 2014

All rights reserved. No part of this work may be reproduced in any form, by mimeograph
or any other means, without permission in writing from the Karnataka State Open
University.
Further information on the Karnataka State Open University Programmes may be
obtained from the University’s Office at Mukthagangotri, Mysore – 6.
Printed and Published on behalf of Karnataka State Open University, Mysore-6 by the
Registrar (Administration)
<table>
<thead>
<tr>
<th>Module 1</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Unit-1</td>
<td>Introduction to Distributed Systems</td>
</tr>
<tr>
<td>Unit-2</td>
<td>System Models</td>
</tr>
<tr>
<td>Unit-3</td>
<td>Architecture of Distributed Systems</td>
</tr>
<tr>
<td>Unit-4</td>
<td>Networking and Internetworking</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Module 2</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Unit-5</td>
<td>Inter process Communication</td>
</tr>
<tr>
<td>Unit-6</td>
<td>External Data Representation</td>
</tr>
<tr>
<td>Unit-7</td>
<td>Distributed Objects and Remote Invocation</td>
</tr>
<tr>
<td>Unit-8</td>
<td>Remote Procedure Call, Events and Notification</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Module 3</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Unit-9</td>
<td>Operating Systems</td>
</tr>
<tr>
<td>Unit-10</td>
<td>Communication and Invocation</td>
</tr>
<tr>
<td>Unit-11</td>
<td>Overview of Security Techniques</td>
</tr>
<tr>
<td>Unit-12</td>
<td>Digital Signatures</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Module 4</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Unit-13</td>
<td>File Service Architecture</td>
</tr>
<tr>
<td>Unit-14</td>
<td>Name Services</td>
</tr>
<tr>
<td>Unit-15</td>
<td>Peer to Peer Systems</td>
</tr>
<tr>
<td>Unit-16</td>
<td>Time and Clocks</td>
</tr>
</tbody>
</table>
Preface

The word distributed in terms such as "distributed system", "distributed programming", and "distributed algorithm" originally referred to computer networks where individual computers were physically distributed within some geographical area. The terms are nowadays used in a much wider sense, even referring to autonomous processes that run on the same physical computer and interact with each other by message passing. While there is no single definition of a distributed system, the following defining properties are commonly used:

- There are several autonomous computational entities, each of which has its own local memory.
- The entities communicate with each other by message passing.

In this article, the computational entities are called computers or nodes.

A distributed system may have a common goal, such as solving a large computational problem. Alternatively, each computer may have its own user with individual needs, and the purpose of the distributed system is to coordinate the use of shared resources or provide communication services to the users.

Organization of the material: The book introduces its topics in ascending order of complexity and is divided into four modules, containing four units each.

In the first module, we begin with an introduction to Distributed Systems, System Models and Architecture of distributed systems.

In the second module, we discussed Inter process communication, External Data Representation and Remote Procedure Call concepts.

The third module contains concepts of Operating System and overview of Security techniques with Digital Signatures.

In the fourth module, File Service Architecture, Name Services, Peer to Peer Systems are discussed.

Happy reading to all the students!!!
UNIT 1: Introduction to Distributed Systems

Structure:

1.0 Objectives
1.1 Introduction
1.2 Definition
1.3 Examples of distributed systems
1.4 Resource Sharing
1.5 Challenges of distributed systems
1.6 Summary
1.7 Keywords
1.8 Unit-end exercises and answers
1.9 Suggested readings

1.0 OBJECTIVES

At the end of this unit you will be able to know:

- Definition of distributed systems
- History of distributed systems
- Benefits of resource sharing
- Challenges of distributed systems.

1.1 INTRODUCTION

This unit is about the introduction to distributed systems. We start with definition of distributed systems. It lists a list of examples of distributed systems. It also introduces Resource Sharing and Challenges of distributed systems. The goal is to provide motivational examples of contemporary distributed systems, illustrating both the
pervasive role of distributed systems and the great diversity of the associated applications.

1.2 DEFINITION

A distributed system can be defined as a “collection of independent computers that appear to the users of the system as a single computer”. The computers in a distributed system are essentially independent machines. This means that, architecturally, the machines are capable of operating independently. The software enables this set of connected machines to appear as a single computer to the users of the system.

Another definition: A distributed system can be defined as a system of networked computers that coordinate their activity only by message passing.

In a distributed system, each computer has its own memory, has its own clock, and each computer runs its own operating systems.

Why distributed systems?

It is mainly because of availability of powerful yet cheap microprocessors (PCs, workstations, PDAs, embedded systems, etc.) and continuing advances in communication technology.

The benefits of distributed systems:

Price/performance ratio: You don't get twice the performance for twice the price in buying computers. Processors are only so fast and the price/performance curve becomes nonlinear and steep very quickly. With multiple CPUs, we can get (almost) double the performance for double the money (as long as we can figure out how to keep the processors busy and the overhead negligible).
Distributing machines: It makes sense to put the CPUs for ATM cash machines at the source, each networked with the bank. Each bank can have one or more computers networked with each other and with other banks. For computer graphics, it makes sense to put the graphics processing at the user's terminal to maximize the bandwidth between the device and processor.

Computer supported cooperative networking: Users that are geographically separated can now work and play together. Examples of this are electronic whiteboards, distributed document systems, audio/video teleconferencing, email, file transfer, and games such as Doom, Quake, Age of Empires, and Duke Nuke’em, Starcraft, and scores of others.

Increased reliability: If a small percentage of machines break, the rest of the system remains intact and can do useful work.

Incremental growth: A company may buy a computer. Eventually the workload is too great for the machine. The only option is to replace the computer with a faster one. Networking allows you to add on to an existing infrastructure.

Remote services: Users may need to access information held by others at their systems. Examples of this include web browsing, remote file access, and programs such as Napster and Gnutella to access MP3 music.

Mobility: Users move around with their laptop computers, Palm Pilots, and WAP phones. It is not feasible for them to carry all the information they need with them.

Advantages and disadvantages:

A distributed system has distinct advantages over a set of non-networked smaller computers. Data can be shared dynamically – giving private copies (via floppy disk, for example) does not work if the data is changing. Peripherals can also be shared. Some peripherals are expensive and/or infrequently used so it is not justifiable to give each PC
a peripheral. These peripherals include optical and tape jukeboxes, typesetters, large format color printers and expensive drum scanners. Machines themselves can be shared and workload can be distributed amongst idle machines. The advantages are Economics, Speed, Inherent distribution, Reliability, Incremental growth.

There are many problems with distributed systems: Designing, implementing and using distributed software may be difficult. Issues of creating operating systems and/or languages that support distributed systems arise. The network may lose messages and/or become overloaded. Rewiring the network can be costly and difficult. Security becomes a far greater concern. Easy and convenient data access from anywhere creates security problems. The disadvantages are software, network, more components to fail, security

1.3. EXAMPLES OF DISTRIBUTED SYSTEMS

There many examples of distributed systems. They are: the Internet and the associated World Wide Web, web search, online gaming, email, social networks, eCommerce, etc. Distributed systems encompass many of the most significant technological developments of recent years and hence an understanding of the underlying technology is absolutely central to knowledge of modern computing.

We now look at more specific examples of distributed systems to further illustrate the diversity and indeed complexity of distributed systems provision today.

Web search: Web search has emerged as a major growth industry in the last decade, with recent figures indicating that the global number of searches has risen to over 10 billion per calendar month. The task of a web search engine is to index the entire contents of the World Wide Web, encompassing a wide range of information styles including web pages, multimedia sources and (scanned) books. This is a very complex task, as current estimates state that the Web consists of over 63 billion pages and one trillion unique web addresses. Given that most search engines analyze the entire web content and then carry
out sophisticated processing on this enormous database, this task itself represents a major challenge for distributed systems design.

Mobile and ubiquitous computing: Technological advances in device miniaturization and wireless networking have led increasingly to the integration of small and portable computing devices into distributed systems. These devices include:

- Laptop computers.
- Handheld devices, including mobile phones, smart phones, GPS-enabled devices, pagers, personal digital assistants (PDAs), video cameras and digital cameras.
- Wearable devices, such as smart watches with functionality similar to a PDA.
- Devices embedded in appliances such as washing machines, hi-fi systems, cars and refrigerators.

The portability of many of these devices, together with their ability to connect conveniently to networks in different places, makes mobile computing possible. Mobile computing is the performance of computing tasks while the user is on the move, or visiting places other than their usual environment. In mobile computing, users who are away from their ‘home’ intranet (the intranet at work, or their residence) are still provided with access to resources via the devices they carry with them.

Wide area network applications: There are many wide area network applications, namely email (electronic mail), bbs (bulletin board systems), netnews (group discussions on single subject), gopher (text retrieval service), WWW (world wide web, is the biggest example of distributed system).

The Internet: network of networks, global access to “everybody” (data, service, other actor; open ended), enormous size (open ended), no single authority

Intranet: it is a portion of the internet managed by an organization. A single authority, protected access.
Key characteristics of distributed systems

The following are the key characteristics of distributed systems.

- Resource sharing
- Openness
- Concurrency
- Scalability
- Fault Tolerance
- Transparency
- No global clock
- Independent failures

1.4. RESOURCE SHARING

Note that the users are so accustomed to the benefits of resource sharing that they may easily overlook their significance. We routinely share hardware resources such as printers, data resources such as files, and resources with more specific functionality such as search engines.

When we look at from the point of view of hardware provision, we share equipment such as printers and disks to reduce costs. But of far greater significance to users is the sharing of the higher-level resources that play a part in their applications and in their everyday work and social activities. For example, users are concerned with sharing data in the form of a shared database or a set of web pages. Similarly, users think in terms of shared resources such as a search engine or a currency converter, without regard for the server or servers that provide these.

We use the term *service* for a distinct part of a computer system that manages a collection of related resources and presents their functionality to users and applications. The only access we have to the service is via the set of operations that it exports.
The fact that services restrict resource access to a well-defined set of operations is in part standard software engineering practice. For effective sharing, each resource must be managed by a program that offers a communication interface enabling the resource to be accessed and updated reliably and consistently.

The term server refers to a running program (a process) on a networked computer that accepts requests from programs running on other computers to perform a service and responds appropriately. The requesting processes are referred to as clients, and the overall approach is known as client-server computing. In this approach, requests are sent in messages from clients to a server and replies are sent in messages from the server to the clients. A complete interaction between a client and a server, from the point when the client sends its request to when it receives the server's response, is called a remote invocation.

Many, but certainly not all, distributed systems can be constructed entirely in the form of interacting clients and servers. The World Wide Web, email and networked printers all fit this model.

There are two types of resources: they are hardware resources (e.g., disks and printers) and software resources (e.g., files, windows, and data objects).

Hardware sharing is used for convenience and reduction of cost. Where as Data sharing (shared usage of information) is used for consistency (compilers and libraries), exchange of information (database), and cooperative work (groupware).

Service resources are used as search engines, computer-supported cooperative working and so on.

**Resource Manager:** Software module that manages a set of resources. Each resource requires its own management policies and methods.

Client server model: server processes act as resource managers for a set of resources and
a set of clients.

Object based model: resources are objects that can move. Object manager is movable. Request for a task on an object is sent to the current manager. Manager must be co-located with object.

Examples for distributed systems in use: ARGUS, Amoeba, Mach, Arjuna, Clouds, Emerald

Quality of service
Once users are provided with the functionality that they require of a service, such as the file service in a distributed system, we can go on to ask about the quality of the service provided. The main nonfunctional properties of systems that affect the quality of the service experienced by clients and users are reliability, security and performance. Adaptability to meet changing system configurations and resource availability has been recognized as a further important aspect of service quality.

1.5. CHALLENGES OF DISTRIBUTED SYSTEMS

In this section we describe the main challenges of distributed systems.

Heterogeneity
The Internet enables users to access services and run applications over a heterogeneous collection of computers and networks. Heterogeneity (that is, variety and difference) applies to all of the following:

- Networks;
- Computer hardware;
- Operating systems;
- Programming languages;
- Implementations by different developers.
Although the Internet consists of many different sorts of network, their differences are masked by the fact that all of the computers attached to them use the Internet protocols to communicate with one another.

**Openness**

The openness of a computer system is the characteristic that determines whether the system can be extended and re-implemented in various ways. The openness of distributed systems is determined primarily by the degree to which new resource-sharing services can be added and be made available for use by a variety of client programs.

Openness cannot be achieved unless the specification and documentation of the key software interfaces of the components of a system are made available to software developers.

A further benefit that is often cited for open systems is their independence from individual vendors.

To summarize:

- Open systems are characterized by the fact that their key interfaces are published.
- Open distributed systems are based on the provision of a uniform communication mechanism and published interfaces for access to shared resources.
- Open distributed systems can be constructed from heterogeneous hardware and software, possibly from different vendors.

**Security**

Many of the information resources that are made available and maintained in distributed systems have a high intrinsic value to their users. Their security is therefore of considerable importance. Security for information resources has three components: confidentiality (protection against disclosure to unauthorized individuals), integrity (protection against alteration or corruption), and availability (protection against interference with the means to access the resources).
In a distributed system, clients send requests to access data managed by servers, which involves sending information in messages over a network. For example:
1. A doctor might request access to hospital patient data or send additions to that data.
2. In electronic commerce and banking, users send their credit card numbers across the Internet.

In both examples, the challenge is to send sensitive information in a message over a network in a secure manner.
However, security challenges have not yet been fully met: for ‘Denial of service attacks’ and ‘Security of mobile code’.

**Scalability**
Distributed systems operate effectively and efficiently at many different scales, ranging from a small intranet to the Internet. A system is described as scalable if it will remain effective when there is a significant increase in the number of resources and the number of users. The number of computers and servers in the Internet has increased dramatically.

The design of scalable distributed systems presents the following challenges:

- *Controlling the cost of physical resources*
- *Controlling the performance loss*
- *Preventing software resources running out*
- *Avoiding performance bottlenecks*

Some shared resources are accessed very frequently; for example, many users may access the same web page, causing a decline in performance. Ideally, the system and application software should not need to change when the scale of the system increases, but this is difficult to achieve. The issue of scale is a dominant theme in the development of distributed systems.
**Failure handling**

Computer systems sometimes fail. When faults occur in hardware or software, programs may produce incorrect results or may stop before they have completed the intended computation.

Failures in a distributed system are partial – that is, some components fail while others continue to function. Therefore the handling of failures is particularly difficult. The following techniques for dealing with failures are discussed throughout the book:

*Detecting failures:* Some failures can be detected. For example, checksums can be used to detect corrupted data in a message or a file.

*Masking failures:* Some failures that have been detected can be hidden or made less severe. Two examples of hiding failures:

1. Messages can be retransmitted when they fail to arrive.
2. File data can be written to a pair of disks so that if one is corrupted, the other may still be correct.

*Tolerating failures:* Most of the services in the Internet do exhibit failures – it would not be practical for them to attempt to detect and hide all of the failures that might occur in such a large network with so many components. Their clients can be designed to tolerate failures, which generally involve the users tolerating them as well.

*Recovery from failures:* Recovery involves the design of software so that the state of permanent data can be recovered or ‘rolled back’ after a server has crashed. In general, the computations performed by some programs will be incomplete when a fault occurs, and the permanent data that they update (files and other material stored in permanent storage) may not be in a consistent state.
Redundancy: Services can be made to tolerate failures by the use of redundant components. Consider the following examples:

1. There should always be at least two different routes between any two routers in the Internet.
2. In the Domain Name System, every name table is replicated in at least two different servers.
3. A database may be replicated in several servers to ensure that the data remains accessible after the failure of any single server; the servers can be designed to detect faults in their peers; when a fault is detected in one server, clients are redirected to the remaining servers.

Distributed systems provide a high degree of availability in the face of hardware faults. The availability of a system is a measure of the proportion of time that it is available for use. When one of the components in a distributed system fails, only the work that was using the failed component is affected. A user may move to another computer if the one that they were using fails; a server process can be started on another computer.

Concurrency

Both services and applications provide resources that can be shared by clients in a distributed system. There is therefore a possibility that several clients will attempt to access a shared resource at the same time. For example, a data structure that records bids for an auction may be accessed very frequently when it gets close to the deadline time. In this case, their operations on the shared resource may conflict with one another and produce inconsistent results.

A shared resource in a distributed system must be responsible for ensuring that it operates correctly in a concurrent environment. This applies not only to servers but also to objects in applications. Therefore any programmer who takes an implementation of an object that was not intended for use in a distributed system must do whatever is necessary to make it safe in a concurrent environment. For an object to be safe in a concurrent environment, its operations must be synchronized in such a way that its data remains consistent. This
can be achieved by standard techniques such as semaphores, which are used in most operating systems.

Transparency

Transparency is defined as the concealment from the user and the application programmer of the separation of components in a distributed system, so that the system is perceived as a whole rather than as a collection of independent components.

The various transparencies are:

*Access transparency:* enables local and remote resources to be accessed using identical operations.

*Location transparency:* enables resources to be accessed without knowledge of their physical or network location (for example, which building or IP address).

*Concurrency transparency:* enables several processes to operate concurrently using shared resources without interference between them.

*Replication transparency:* enables multiple instances of resources to be used to increase reliability and performance without knowledge of the replicas by users or application programmers.

*Failure transparency:* enables the concealment of faults, allowing users and application programs to complete their tasks despite the failure of hardware or software components.

*Mobility transparency:* allows the movement of resources and clients within a system without affecting the operation of users or programs.

*Performance transparency:* allows the system to be reconfigured to improve performance as loads vary.

*Scaling transparency:* allows the system and applications to expand in scale without change to the system structure or the application algorithms.

1.6. SUMMARY
In this unit we introduced distributed systems. We started with definition of distributed systems. We presented a list of examples of distributed systems. This unit also introduced Resource Sharing and Challenges of distributed systems. In this unit we also discussed about different transparencies.

1.7. KEYWORDS

A distributed system: can be defined as a “collection of independent computers that appear to the users of the system as a single computer”.

Resource Manager: Software module that manages a set of resources. Each resource requires its own management policies and methods.

Transparency: is defined as the concealment from the user and the application programmer of the separation of components in a distributed system, so that the system is perceived as a whole rather than as a collection of independent components

1.8. UNIT-END EXERCISES AND ANSWERS

1. Give definition of distributed systems.
2. Give some examples of distributed systems.
3. Explain the characteristics of distributed systems.
4. Discuss about resource sharing.
5. What are the challenges of distributed systems? Explain
6. What is transparency? Explain various transparency.

Answers: SEE

1. 1.2
2. 1.3
3. 1.3
1.9  SUGGESTED READINGS

- DISTRIBUTED SYSTEMS Principles and Paradigm By: Andrew S. Tanenbaum, Maarten Van Steen
At the end of this unit you will be able to know about:

- Design of distributed systems
- Various system models

This unit provides an explanation of three important and complementary ways in which the design of distributed systems can usefully be described and discussed. The purpose of the System models is to illustrate/describe common properties and design choices for distributed system in a single descriptive model.

There are three types of models: namely,

1. Physical models consider the types of computers and devices that constitute a system and their interconnectivity, without details of specific technologies.
Physical models: capture the hardware composition of a system in terms of computers and other devices and their interconnecting network.

2. Architectural models describe a system in terms of the computational and communication tasks performed by its computational elements; the computational elements being individual computers or aggregates of them supported by appropriate network interconnections. Client-server and peer-to-peer are two of the most commonly used forms of architectural model for distributed systems.

3. Fundamental models take an abstract perspective in order to describe solutions to individual issues faced by most distributed systems.

There is no global time in a distributed system, so the clocks on different computers do not necessarily give the same time as one another. All communication between processes is achieved by means of messages. Message communication over a computer network can be affected by delays, can suffer from a variety of failures and is vulnerable to security attacks. These issues are addressed by three models: namely, the interaction model, the failure model, and the security model.

2.2 PHYSICAL MODELS

A physical model is a representation of the underlying hardware elements of a distributed system that abstracts away from specific details of the computer and networking technologies employed. The end result is a physical architecture with a significant increase in the level of heterogeneity embracing, for example, the tiniest embedded devices utilized in ubiquitous computing through to complex computational elements found in Grid computing. These systems deploy an increasingly varied set of networking technologies and offer a wide variety of applications and services. Such systems potentially involve up to hundreds of thousands of nodes.
2.3 ARCHITECTURAL MODELS

The architecture of a system is its structure in terms of separately specified components and their interrelationships. The overall goal is to ensure that the structure will meet present and likely future demands on it. Major concerns are to make the system reliable, manageable, adaptable and cost-effective. Architecture models define the main components of the system, what their roles are and how they interact (software architecture), and how they are deployed in an underlying network of computers (system architecture). Architecture model is concerned with the placement of its parts, namely how components are mapped to underlying network and the relationship between them, that is, their functional roles and patterns of communication between them. An architectural model simplifies and abstracts the functions of the individual components of a distributed system and then it considers the placement of the components across a network of an architectural model, the interrelationships between the components.

Architectural elements:

To understand the fundamental building blocks of a distributed system, it is necessary to consider four key questions:
What are the entities that are communicating in the distributed system?
How do they communicate, or, more specifically, what communication paradigms 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)?

Now we examine two architectural styles stemming from the role of individual processes: client-server and peer-to-peer.
Client-server: This is the architecture that is most often cited when distributed systems are discussed. It is historically the most important and remains the most widely employed. The Figure 1.2.1 illustrates the simple structure in which processes take on the roles of being clients or servers. In particular, client processes interact with individual server processes in potentially separate host computers in order to access the shared resources that they manage.

![Client Server Model](image)

**Figure 1.2.1: Client Server Model**

Peer-to-peer: In this architecture all of the processes involved in a task or activity play similar roles, interacting cooperatively as peers without any distinction between client and server processes or the computers on which they run. In practical terms, all participating processes run the same program and offer the same set of interfaces to each other. While the client-server model offers a direct and relatively simple approach to the sharing of data and other resources, it scales poorly. The centralization of service provision and management implied by placing a service at a single address does not scale well beyond the capacity of the computer that hosts the service and the bandwidth of its network connections. The Figure 1.2.2 shows Peer-to-Peer model.
2. 4  FUNDAMENTAL MODELS

Fundamental models deal with formal description of the properties that is common to architecture models. Since no global time in a distributed system, so the clocks on different computers do not necessarily give the same time as one another. Messages communications can be affected by delays and suffer from a variety of failures and vulnerable to security attacks. All system models have some common fundamental properties.

There are three fundamental models: 1) interaction models, 2) failure models and 3) security models.

1  Interaction: Processes communicate with messages and coordinate via synchronization and ordering of activities. The message delays are often of considerable duration, the coordination between processes is limited by lack of global clock. The interaction model deals with performance and with difficulty of setting time limits.
2 Failure: The correct operation is threatened whenever a fault occurs in any of the computers and network. We should define types of faults in order to tolerate them for the system to continue to run correctly. The failure model attempts to give a precise specification of the faults that can be exhibited by processes and communication channels.

3 Security: The modular feature of distributed system and their openness exposes them to attack by both external and internal agents. Security model defines and classifies the forms of attack may take, providing a basis for the analysis of threats to a system and for the design of system that are able to resist them. The security model discusses the possible threats to processes and communication channels. It introduces the concept of a secure channel, which is secure against those threats

2.5 SUMMARY

In this unit, we have discussed about three types of system models namely, Physical models, Architecture models and Fundamental models.

2.6 KEYWORDS

Physical model: considers the types of computers and devices that constitute a system and their interconnectivity, without details of specific technologies

Architectural model: describes a system in terms of the computational and communication tasks performed by its computational elements.

Fundamental model: takes an abstract perspective in order to describe solutions to individual issues faced by most distributed systems

Client server model: The client–server model of computing is a distributed application structure that partitions tasks or workloads between the providers of a resource or service, called servers, and service requesters, called clients.
Peer-to-peer model: In this architecture all of the processes involved in a task or activity play similar roles, interacting cooperatively as peers without any distinction between client and server processes or the computers on which they run.

2.7 UNIT-END EXERCISES AND ANSWERS

1. What are the four key questions to understand the fundamental building blocks of a distributed system?
2. Explain client server model with a neat diagram.
3. Explain peer-to-peer model with a neat diagram.
4. Compare client server model and peer-to-peer model.
5. Explain fundamental models.

Answers: SEE

1. 2.3
2. 2.3
3. 2.3
4. 2.3
5. 2.4

2.8 SUGGESTED READINGS

- DISTRIBUTED SYSTEMS Principles and Paradigm By : Andrew S. Tanenbaum, Maarten Van Steen
UNIT 3 ARCHITECTURES OF DISTRIBUTED SYSTEMS

Structure:

3.0 Objectives
3.1 Introduction
3.2 The Architectures Suitable for Distributed Systems
3.3 Layered (software) Architecture for Client-Server Systems
3.4 System Architecture
3.5 Overlay Networks
3.6 Summary
3.7 Keywords
3.8 Unit-end exercises and answers
3.9 Suggested readings

3.0 OBJECTIVES

At the end of this unit you will be able to know:

- Various architectures of distributed systems.
- Layered (software) Architecture for Client-Server Systems
- Overlay Networks

3.1 INTRODUCTION

This unit introduces the various architectures of distributed systems.

In this unit we discuss the architectures suitable for distributed systems, We also discuss about Layered (software) Architecture for Client-Server Systems, System Architecture and Overlay Networks.

Broadly we have the following architectures.

**Software Architectures**: describe the organization and interaction of software components; focuses on logical organization of software (component interaction, etc.).

**System Architectures**: describe the placement of software components on physical machines.
The realization of an architecture may be centralized (most components located on a single machine), decentralized (most machines have approximately the same functionality), or hybrid (some combination).

**Architectural Styles**: An architectural style describes a particular way to configure a collection of components and connectors. A **Component** is a module with well-defined interfaces; reusable, replaceable. A **Connector** is a communication link between modules.

### 3.2 THE ARCHITECTURES SUITABLE FOR DISTRIBUTED SYSTEMS

The following are the architectures suitable for distributed systems.
- Layered architectures
- Object-based architectures
- Data-centered architectures
- Event-based architectures

![Layered Architecture Diagram](image)

(a)
Figure 1.3.1: The (a) layered architectural style and (b) The object-based architectural style.

**Data-Centered Architectures:** Main purpose is data access and update. Processes interact by reading and modifying data in some shared repository in either active or passive manner.

- Traditional data base (passive): responds to requests
- Blackboard system (active): clients solve problems collaboratively; system updates clients when information changes.

Communication via event propagation, in distributed systems seen often in Publish/Subscribe. e.g., register interest in market information; get email updates. Decouples sender and receiver, use asynchronous communication

**Event-based architecture:** It supports several communication styles:

- Publish-subscribe
- Broadcast
- Point-to-point
Figure 1.3.2: (a) The event-based architectural style (b) The shared data-space architectural style.

Data Centric Architecture; e.g., shared distributed file systems or Web-based distributed systems
Combination of data-centered and event based architectures. Processes communicate asynchronously.

System Architectures for Distributed Systems

The system architecture for distributed systems can be either centralized or decentralized or hybrid.
**Centralized**: characterized by traditional client-server structure.
- Vertical (or hierarchical) organization of communication and control paths (as in layered software architectures)
- Logical separation of functions into client (requesting process) and server (responder)

**Decentralized**: characterized by peer-to-peer.
- Horizontal rather than hierarchical communication and control
- Communication paths are less structured; symmetric functionality.

**Hybrid**: It is combination of elements of Client Server and Peer-to-Peer.
Classification of a system as centralized or decentralized refers to communication and control organization, primarily.

**Traditional Client-Server system**: In traditional Client-Server system, the Processes are divided into two groups (clients and servers). They use Synchronous communication: request-reply protocol. In LANs often implemented with a connectionless protocol (unreliable). In WANs, communication is typically connection-oriented TCP/IP (reliable), with high likelihood of communication failures.

![Diagram](image)

Figure 1.3.3: General interaction between a client and a server.
Transmission Failures: with connectionless transmissions, failure of any sort means no reply.
The possibilities are:

- Request message was lost
- Reply message was lost
- Server failed either before, during or after performing the service

### 3.3 LAYERED (SOFTWARE) ARCHITECTURE FOR CLIENT-SERVER SYSTEMS

The layered architecture consists of three levels:

1) **User-interface level**: It usually uses GUIs for interacting with end users.
2) **Processing level**: It is concerned with data processing applications, which is the core functionality.
3) **Data level**: interacts with data base or file system. Data usually is persistent; exists even if no client is accessing it. It is concerned with File or database system.

The Figure 1.3.4 shows the application layering.
3.4 SYSTEM ARCHITECTURE

The system architecture involves the Mapping the software architecture to system hardware, which has close correspondence between logical software modules and actual computers.

**Multi-tiered architectures**

Multi-tier architecture (often referred to as n-tier architecture) is a client–server architecture in which presentation, application processing, and data management functions are physically separated. The most widespread use of multi-tier architecture is the three-tier architecture. N-tier application architecture provides a model by which developers can create flexible and reusable applications. By segregating an application into tiers, developers acquire the option of modifying or adding a specific layer, instead of reworking the entire application. Layer and tier are roughly equivalent terms, but layer typically implies software and tier is more likely to refer to hardware. Two-tier and three-tier are the most common architectures.
Two-tiered Client-Server Architectures

Two-tiered architecture is a technological hardware and software configuration in which the presentation and application components of the architecture are resident on one system in a two-system configuration and the database component is resident on the other system in the configuration.

Thin Client approach: Server provides processing and data management; client provides simple graphical display. This increases work load at server. It is easier to manage, more reliable, client machines don’t need to be so large and powerful.

Fat-Client approach: All application processing and some data reside at the client. This reduces work load at server and more scalable. But, it is harder to manage by system admin and less secure.

Three-tiered Client-Server Architectures

In some applications servers may also need to be clients, which lead to a three level architecture. Three-tiered architecture is a technological hardware and software configuration in which the presentation, application and database components of the architecture are resident on separate and distinct systems within the configuration. Some examples are Distributed transaction processing and Web servers that interact with database servers. In Three-tiered Architectures, functionality distributed across three levels of machines instead of two.

Figure 1.3.5: An example of a server acting as client.
Centralized versus Decentralized Architectures

**Vertical distribution:** Traditional client-server architectures exhibit **vertical distribution.** Each level serves a different purpose in the system. In this *logically* different components reside on different nodes.

**Horizontal distribution** (P2P): each node has roughly the same processing capabilities and stores/manages part of the total system data. It has better load balancing, more resistant to denial-of-service attacks, harder to manage than client-server. Communication and control is not hierarchical; all about equal.

**Peer-to-Peer**

A **peer-to-peer (P2P) network** is a type of **decentralized** and **distributed network architecture** in which individual nodes in the network (called *"peers"*) act as both suppliers and consumers of resources. It is a network of personal computers, each of which acts as both client and server, so that each can exchange files and folders directly with every other computer on the network.

In Peer-to-peer Nodes act as both client and server and interaction is symmetric. Each node acts as a server for part of the total system data. Overlay networks connect nodes in the P2P system. Nodes in the overlay use their own addressing system for storing and retrieving data in the system. Nodes can route requests to locations that may not be known by the requester.

### 3.5 OVERLAY NETWORKS

Overlay networks are logical or *virtual* networks, built on top of a physical network. A link between two nodes in the overlay may consist of several physical links. Messages in the overlay are sent to logical addresses, not physical (IP) addresses. Various approaches used to resolve logical addresses to physical.

Each node in a P2P system knows how to contact several other nodes. The overlay network may be structured (nodes and content are connected according to some design
that simplifies later lookups) or unstructured (content is assigned to nodes without regard to the network topology).

**Structured P2P Architectures**

In structured P2P architectures, a common approach is to use a distributed hash table (DHT) to organize the nodes. Traditional hash functions convert a key to a hash value, which can be used as an index into a hash table. The keys are unique, each represents an object to store in the table. The hash function value is used to insert an object in the hash table and to retrieve it.

In a DHT, data objects and nodes are each assigned a key which hashes to a random number from a very large identifier space (to ensure uniqueness). A mapping function assigns objects to nodes, based on the hash function value. A lookup, also based on hash function value, returns the network address of the node that stores the requested object.

**Characteristics of DHT**

It is scalable to thousands, even millions of network nodes. Search time increases more slowly than size; usually $O(\log(N))$. It is also fault tolerant, able to re-organize itself when nodes fail. It is decentralized, no central coordinator.

**Unstructured P2P**

Unstructured P2P organizes the overlay network as a random graph. Each node knows about a subset of nodes, its "neighbors". Neighbors are chosen in different ways: physically close nodes, nodes that joined at about the same time, etc. Data items are randomly mapped to some node in the system and lookup is random.

**Locating a Data Object by Flooding**

Send a request to all known neighbors If not found, neighbors forward the request to their neighbors. Works well in small to medium sized networks, doesn’t scale well. “Time-to-live” counter can be used to control number of hops. Example system: Gnutella & Freenet (Freenet uses a caching system to improve performance)

**Structured P2P Architectures**

The hybrid architectures combine client-server and P2P architectures.

For example,
- Edge-server systems; e.g. ISPs, which act as servers to their clients, but cooperate with other edge servers to host shared content
- Collaborative distributed systems; e.g., BitTorrent, which supports parallel downloading and uploading of chunks of a file. First, interact with C/S system, then operates in decentralized manner.

![Diagram of the Internet as consisting of a collection of edge servers.](image)

Figure 1.3.6: Viewing the Internet as consisting of a collection of edge servers.

**P2P verses Client/Server**

- P2P computing allows end users to communicate without a dedicated server.
- Communication is still usually synchronous (blocking)
- There is less likelihood of performance bottlenecks since communication is more distributed. Data distribution leads to workload distribution.
- Resource discovery is more difficult than in centralized client-server computing and look-up/retrieval is slower
- P2P can be more fault tolerant, more resistant to denial of service attacks because network content is distributed. Individual hosts may be unreliable, but overall, the system should maintain a consistent level of service.
3.6 SUMMARY

In this unit we introduced the various architectures of distributed systems. We also discussed about the architectures suitable for distributed systems, Layered (software) Architecture for Client-Server Systems, System Architecture and Overlay Networks.

3.7 KEYWORDS

Hybrid: It is combination of elements of Client Server and Peer-to-Peer.

Data-Centered Architectures: Main purpose is data access and update. Processes interact by reading and modifying data in some shared repository in either active or passive manner.

Multi-tier architecture (often referred to as n-tier architecture) is a client–server architecture in which presentation, application processing, and data management functions are physically separated.

3.8 UNIT-END EXERCISES AND ANSWERS

2. Define Two-tiered C/S Architectures and Three-tiered C/S Architectures
3. Compare Structured and Unstructured P2P Architectures
4. Explain Structured P2P Architectures
5. Compare P2P verses Client/Server

Answers: SEE

1. 3.3
2. 3.4
3. 3.5
4. 3.5
5. 3.4
3.9 SUGGESTED READINGS


- DISTRIBUTED SYSTEMS Principles and Paradigm By: Andrew S. Tanenbaum, Maarten Van Steen
UNIT 4: NETWORKING AND INTERNETWORKING

Structure:

4.0 Objectives
4.1 Introduction
4.2 Networking issues
4.3 Types of Networks
4.4 Internet Protocols
4.5 Summary
4.6 Keywords
4.7 Unit-end exercises and answers
4.8 Suggested readings

4.0 OBJECTIVES

At the end of this unit you will be able to understand:

- Definition of networks
- Networking issues
- Types of Networks
- Internet Protocols

4.1 INTRODUCTION

Distributed systems use local area networks, wide area networks and internetworks for communication. The performance, reliability, scalability, mobility and quality of service characteristics of the underlying networks impact the behaviour of distributed systems and hence affect their design. Changes in user requirements have resulted in the emergence of wireless networks and of high-performance networks with quality of service guarantees.
The principles on which computer networks are based include protocol layering, packet switching, routing and data streaming. Internetworking techniques enable heterogeneous networks to be integrated. The Internet is the major example; its protocols are almost universally used in distributed systems. The addressing and routing schemes used in the Internet have withstood the impact of its enormous growth. They are now undergoing revision to accommodate future growth and to meet new application requirements for mobility, security and quality of service.

The networks used in distributed systems are built from a variety of transmission media, including wire, cable, fiber and wireless channels; hardware devices, including routers, switches, bridges, hubs, repeaters and network interfaces; and software components, including protocol stacks, communication handlers and drivers. The resulting functionality and performance available to distributed system and application programs is affected by all of these. We shall refer to the collection of hardware and software components that provide the communication facilities for a distributed system as a communication subsystem. The computers and other devices that use the network for communication purposes are referred to as hosts. The term node is used to refer to any computer or switching device attached to a network.

The Internet is a single communication subsystem providing communication between all of the hosts that are connected to it. The Internet is constructed from many subnets. A subnet is a unit of routing (delivering data from one part of the Internet to another); it is a collection of nodes that can all be reached on the same physical network. The Internet’s infrastructure includes an architecture and hardware and software components that effectively integrate diverse subnets into a single data communication service.

The design of a communication subsystem is strongly influenced by the characteristics of the operating systems used in the computers of which the distributed system is composed as well as the networks that interconnect them. In this unit, we consider the impact of network technologies on the communication subsystem;
4.2 NETWORKING ISSUES

More recently, following the growth and commercialization of the Internet and the emergence of many new modes of use, more stringent requirements for reliability, scalability, mobility, security and quality of service have emerged, including the performance requirement to meet the needs of interactive applications. In this unit, we define and describe the nature of each of these requirements.

Performance: The network performance parameters that are of primary interest for our purposes are those affecting the speed with which individual messages can be transferred between two interconnected computers. These are the latency and the point-to-point data transfer rate:

*Latency* is the delay that occurs after a send operation is executed and before data starts to arrive at the destination computer. It can be measured as the time required to transfer an empty message.

*Data transfer rate* is the speed at which data can be transferred between two computers in the network once transmission has begun, usually quoted in bits per second.

The performance of networks deteriorates in conditions of overload – when there are too many messages in the network at the same time. The precise effect of overload on the latency, data transfer rate and total system bandwidth of a network depends strongly on the network technology.

Scalability: Computer networks are an indispensable part of the infrastructure of modern societies. The Growth of host computers and web servers connected to the Internet so rapid and diverse that it is difficult to find recent reliable statistics. The potential future size of the Internet is commensurate with the population of the planet. It is realistic to expect it to include several billion nodes and hundreds of millions of active hosts. These numbers indicate the future changes in size and load that the Internet must handle. The network technologies on which it is based were not designed to cope with even the
Internet’s current scale, but they have performed remarkably well. Some substantial changes to the addressing and routing mechanisms are in progress in order to handle the next phase of the Internet’s growth. For simple client-server applications such as the Web, we would expect future traffic to grow at least in proportion to the number of active users. The ability of the Internet’s infrastructure to cope with this growth will depend upon the economics of use, in particular charges to users and the patterns of communication that actually occur.

**Reliability:** Many applications are able to recover from communication failures and hence do not require guaranteed error-free communication. The end-to-end argument further supports the view that the communication subsystem need not provide totally error-free communication; the detection of communication errors and their correction is often best performed by application-level software. The reliability of most physical transmission media is very high. When errors occur they are usually due to failures in the software at the sender or receiver or buffer overflow rather than errors in the network.

**Security:** The first level of defense adopted by most organizations is to protect its networks and the computers attached to them with a *firewall*. A firewall creates a protection boundary between the organization’s intranet and the rest of the Internet. The purpose of the firewall is to protect the resources in all of the computers inside the organization from access by external users or processes and to control the use of resources outside the firewall by users inside the organization.

A firewall runs on a gateway – a computer that stands at the network entry point to an organization’s intranet. The firewall receives and filters all of the messages travelling into and out of an organization. It is configured according to the organization’s security policy to allow certain incoming and outgoing messages to pass through it and to reject all others. To enable distributed applications to move beyond the restrictions imposed by firewalls there is a need to produce a secure network environment in which a wide range of distributed applications can be deployed, with end-to-end authentication, privacy and security. This finer-grained and more flexible form of security can be achieved through
the use of cryptographic techniques. It is usually applied at a level above the communication subsystem and hence is not dealt with here.

**Mobility:** Mobile devices such as laptop computers and Internet-capable mobile phones are moved frequently between locations and reconnected at convenient network connection points or even used while on the move. Wireless networks provide connectivity to such devices, but the addressing and routing schemes of the Internet were developed before the advent of these mobile devices and are not well adapted to their need for intermittent connection to many different subnets. 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:** The quality of service is defined as the ability to meet deadlines when transmitting and processing streams of real-time multimedia data. This imposes major new requirements on computer networks. Applications that transmit multimedia data require guaranteed bandwidth and bounded latencies for the communication channels that they use. Some applications vary their demands dynamically and specify both a minimum acceptable quality of service and a desired optimum.

**Multicasting:** Most communication in distributed systems is between pairs of processes, but there often is also a need for one-to-many communication. While this can be simulated by sends to several destinations, that is more costly than necessary and may not exhibit the fault-tolerance characteristics required by applications. For these reasons, many network technologies support the simultaneous transmission of messages to several recipients.

### 4.3 TYPES OF NETWORKS

The main types of network that are used to support distributed systems are discussed here: *personal area networks, local area networks, wide area networks, metropolitan*
area networks and the wireless variants of them. Internetworks such as the Internet are constructed from networks of all these types.

We refer to networks that are composed of many interconnected networks, integrated to provide a single data communication medium, as internetworks. The Internet is the prototypical internetwork; it is composed of millions of local, metropolitan and wide area networks.

**Personal area networks (PANs):** PANs are a subcategory of local networks in which the various digital devices carried by a user are connected by a low-cost, low-energy network. Wired PANs are not of much significance because few users wish to be encumbered by a network of wires on their person, but wireless personal area networks (WPANs) are of increasing importance due to the number of personal devices such as mobile phones, tablets, digital cameras, music players and so on that are now carried by many people.

**Local area networks (LANs):** LANs carry messages at relatively high speeds between computers connected by a single communication medium, such as twisted copper wire, coaxial cable or optical fiber.

A *segment* is a 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. Larger local networks (such as those that serve a campus or an office building) are composed of many segments interconnected by switches or hubs. In local area networks, the total system bandwidth is high and latency is low, except when message traffic is very high.

**Wide area networks (WANs):** WANs carry messages at lower speeds between nodes that are often in different organizations and may be separated by large distances. They
may be located in different cities, countries or continents. The communication medium is a set of communication circuits linking a set of dedicated computers called routers.

They manage the communication network and route messages or packets to their destinations. In most networks, the routing operations introduce a delay at each point in the route, so the total latency for the transmission of a message depends on the route that it follows and the traffic loads in the various network segments that it traverses. The speed of electronic signals in most media is close to the speed of light, and this sets a lower bound on the transmission latency for long-distance networks.

**Metropolitan area networks (MANs):** This type of network is based on the high-bandwidth copper and fiber optic cabling recently installed in some towns and cities for the transmission of video, voice and other data over distances of up to 50 kilometers. A variety of technologies have been used to implement the routing of data in MANs, ranging from Ethernet to ATM. The DSL (Digital Subscriber Line) and cable modem connections now available in many countries are an example.

**Wireless local area networks (WLANs):** WLANs are designed for use in place of wired LANs to provide connectivity for mobile devices, or simply to remove the need for a wired infrastructure to connect computers within homes and office buildings to each other and the Internet. They are in widespread use in several variants of the IEEE 802.11 standard (WiFi).

**Wireless metropolitan area networks (WMANs):** The IEEE 802.16 WiMAX standard is targeted at this class of network. It aims to provide an alternative to wired connections to home and office buildings and to supersede 802.11 WiFi networks in some applications.

**Wireless wide area networks (WWANs):** Most mobile phone networks are based on digital wireless network technologies such as the GSM (Global System for Mobile communication) standard, which is used in most countries of the world. Mobile phone
networks are designed to operate over wide areas (typically entire countries or continents) through the use of cellular radio connections; their data transmission facilities therefore offer wide area mobile connections to the Internet for portable devices.

**Internetworks:** An internetwork is a communication subsystem in which several networks are linked together to provide common data communication facilities that overlay the technologies and protocols of the individual component networks and the methods used for their interconnection.

Internetworks are needed for the development of extensible, open distributed systems. The openness characteristic of distributed systems implies that the networks used in distributed systems should be extensible to very large numbers of computers, whereas individual networks have restricted address spaces and some have performance limitations that are incompatible with their large-scale use. In internetworks, a variety of local and wide area network technologies can be integrated to provide the networking capacity needed by each group of users. Thus internetworks bring many of the benefits of open systems to the provision of communication in distributed systems.

Internetworks are constructed from a variety of component networks. They are interconnected by dedicated switching computers called *routers* and general-purpose computers called *gateways*, and an integrated communication subsystem is produced by a software layer that supports the addressing and transmission of data to computers throughout the internetwork. The result can be thought of as a ‘virtual network’ constructed by overlaying an internetwork layer on a communication medium that consists of the underlying networks, routers and gateways. The Internet is the major instance of internetworking, and its TCP/IP protocols are an example of this integration layer.
4.4 INTERNET PROTOCOLS

We describe here the main features of the TCP/IP suite of protocols and discuss their advantages and limitations when used in distributed systems.

The Internet emerged from two decades of research and development work on wide area networking in the USA, commencing in the early 1970s with the ARPANET – the first large-scale computer network development. An important part of that research was the development of the TCP/IP protocol suite. TCP stands for Transmission Control Protocol, IP for Internet Protocol. The widespread adoption of the TCP/IP and Internet application protocols in national research networks, and more recently in commercial networks in many countries, has enabled the national networks to be integrated into a single internetwork that has grown extremely rapidly to its present size, with more than 60 million hosts. Many application services and application-level protocols now exist based on TCP/IP, including the Web (HTTP), email (SMTP, POP), netnews (NNTP), file transfer (FTP) and Telnet (telnet). TCP is a transport protocol; it can be used to support applications directly, or additional protocols can be layered on it to provide additional features.

The Internet protocols were originally developed primarily to support simple wide area applications such as file transfer and electronic mail, involving communication with relatively high latencies between geographically dispersed computers, but they turned out to be efficient enough to support the requirements of many distributed applications on both wide area and local networks and they are now almost universally used in distributed systems. The resulting standardization of communication protocols has brought immense benefits.

There are two transport protocols – TCP (Transport Control Protocol) and UDP (User Datagram Protocol). TCP is a reliable connection-oriented protocol, and UDP is a datagram protocol that does not guarantee reliable transmission. The Internet Protocol is
the underlying ‘network’ protocol of the Internet virtual network – that is, IP datagrams provide the basic transmission mechanism for the Internet and other TCP/IP networks.

The TCP/IP specifications do not specify the layers below the Internet datagram layer – IP packets in the Internet layer are transformed into packets for transmission over almost any combination of underlying networks or data links. For example, IP ran initially over the ARPANET, which consisted of hosts and an early version of routers (called PSEs) connected by long-distance data links. Today it is used over virtually every known network technology, including ATM, local area networks such as Ethernets, and token ring networks. IP is implemented over serial lines and telephone circuits via the PPP protocol, enabling it to be used for communication with modem connections and other serial links.

The success of TCP/IP is based on the protocols’ independence from the underlying transmission technology, enabling internetworks to be built up from many heterogeneous networks and data links. Users and application programs perceive a single virtual network supporting TCP and UDP and implementers of TCP and UDP see a single virtual IP network, hiding the diversity of the underlying transmission media. Figure 1.4.1 illustrates this view.

**Figure 1.4.1** The programmer's conceptual view of a TCP/IP Internet

IP addressing: Perhaps the most challenging aspect of the design of the Internet protocols was the construction of schemes for naming and addressing hosts and for routing IP packets to their destinations. The scheme used for assigning host addresses to networks and the computers connected to them had to satisfy the following requirements:
• It must be universal – any host must be able to send packets to any other host in the Internet.

• It must be efficient in its use of the address space – it is impossible to predict the ultimate size of the Internet and the number of network and host addresses likely to be required. The address space must be carefully partitioned to ensure that addresses will not run out. In 1978–82, when the specifications for the TCP/IP protocols were being developed, provision for $2^{32}$ or approximately 4 billion addressable hosts (about the same as the population of the world at that time) was considered adequate. This judgment has proved to be short-sighted, for two reasons:
  – The rate of growth of the Internet has far outstripped all predictions.
  – The address space has been allocated and used much less efficiently than expected.

• The addressing scheme must lend itself to the development of a flexible and efficient routing scheme, but the addresses themselves cannot contain very much of the information needed to route a packet to its destination.

Today the overwhelming majority of Internet traffic continues to use the IP version 4 address and packet format defined three decades ago. The scheme assigns an IP address to each host in the Internet – a 32-bit numeric identifier containing a network identifier, which uniquely identifies one of the subnetworks in the Internet, and a host identifier, which uniquely identifies the host’s connection to that network. It is these addresses that are placed in IP packets and used to route them to their destinations.

The design adopted for the Internet address space is shown in Figure 1.4.2. There are four allocated classes of Internet address – A, B, C and D. Class D is reserved for Internet multicast communication, which is implemented in only some Internet routers. Class E contains a range of unallocated addresses, which are reserved for future requirements.
These 32-bit Internet addresses, containing a network identifier and host identifier, are usually written as a sequence of four decimal numbers separated by dots. Each decimal number represents one of the four bytes, or octets, of the IP address. The permissible values for each class of network address are shown in Figure 1.4.3.

Figure 1.4.2: Internet address structure, showing field sizes in bits

Figure 1.4.3: Decimal representation of Internet addresses
Three classes of address were designed to meet the requirements of different types of organization. The Class A addresses, with a capacity for $2^{24}$ hosts on each subnet, are reserved for very large networks such as the US NSFNet and other national wide area networks. Class B addresses are allocated to organizations that operate networks likely to contain more than 255 computers, and Class C addresses are allocated to all other network operators.

Internet addresses with host identifiers 0 and all 1s (binary) are used for special purposes. Addresses with the host identifier set to 0 are used to refer to ‘this host’, and a host identifier that is all 1s is used to address a broadcast message to all of the hosts connected to the network specified in the network identifier part of the address. Network identifiers are allocated by the Internet Assigned Numbers Authority (IANA) to organizations with networks connected to the Internet. Host identifiers for the computers on each network connected to the Internet are assigned by the managers of the relevant networks.

Since host addresses include a network identifier, any computer that is connected to more than one network must have separate addresses on each, and whenever a computer is moved to a different network, its Internet address must change. These requirements can lead to substantial administrative overheads, for example in the case of portable computers.

In practice, the IP address allocation scheme has not turned out to be very effective. The main difficulty is that network administrators in user organizations cannot easily predict future growth in their need for host addresses, and they tend to overestimate, requesting Class B addresses when in doubt. Around 1990 it became evident that based on the rate of allocation at the time, IP addresses were likely to run out around 1996. Three steps were taken. The first was to initiate the development of a new IP protocol and addressing scheme, the result of which was the specification of IPv6.
The second step was to radically modify the way in which IP addresses were allocated. A new address allocation and routing scheme designed to make more effective use of the IP address space was introduced, called classless interdomain routing (CIDR).

The third step was to enable unregistered computers to access the Internet indirectly through routers that implement a Network Address Translation (NAT) scheme.

The IP protocol

The IP protocol transmits datagrams from one host to another, if necessary via intermediate routers. The full IP packet format is rather complex, but Figure 1.4.4 shows the main components. There are several header fields, not shown in the diagram, that are used by the transmission and routing algorithms. IP provides a delivery service that is described as offering unreliable or best-effort delivery semantics, because there is no guarantee of delivery. Packets can be lost, duplicated, delayed or delivered out of order, but these errors arise only when the underlying networks fail or buffers at the destination are full. The only checksum in IP is a header checksum, which is inexpensive to calculate and ensures that any corruptions in the addressing and packet management data will be detected. There is no data checksum, which avoids overheads when crossing routers, leaving the higher-level protocols (TCP and UDP) to provide their own checksums – a practical instance of the end-to-end argument.

![Figure 1.4.4: IP packet layout](image)

The IP layer puts IP datagrams into network packets suitable for transmission in the underlying network (which might, for example, be an Ethernet). When an IP datagram is longer than the MTU of the underlying network, it is broken into smaller packets at the
source and reassembled at its final destination. Packets can be further broken up to suit
the underlying networks encountered during the journey from source to destination.
(Each packet has a fragment identifier to enable out-of-order fragments to be collected.)

The IP layer must also insert a ‘physical’ network address of the message destination to
the underlying network. It obtains this from the address resolution module in the Internet
network interface layer.

**Address resolution:** The address resolution module is responsible for converting Internet
addresses to network addresses for a specific underlying network (sometimes called
physical addresses). For example, if the underlying network is an Ethernet, the address
resolution module converts 32-bit Internet addresses to 48-bit Ethernet addresses.
This translation is network technology dependent:

- Some hosts are connected directly to Internet packet switches; IP packets can be routed
to them without address translation.
- Some local area networks allow network addresses to be assigned to hosts dynamically,
and the addresses can be conveniently chosen to match the host identifier portion of the
Internet address – translation is simply a matter of extracting the host identifier from the
IP address.
- For Ethernets and some other local networks, the network address of each computer is
hard-wired into its network interface hardware and bears no direct relation to its Internet
address – translation depends upon knowledge of the correspondence between IP
addresses and addresses for the hosts on the local network and is done using an address
resolution protocol (ARP).

**IP spoofing:** We have seen that IP packets include a source address – the IP address of
the sending computer. This, together with a port address encapsulated in the data field
(for UDP and TCP packets), is often used by servers to generate a return address.

Unfortunately, it is not possible to guarantee that the source address given is in fact the
address of the sender. A malicious sender can easily substitute an address that is different
from its own. This loophole has been the source of several well-known attacks, including the distributed denial of service attacks of February 2000. The method used was to issue many ping service requests to a large number of computers at several sites (ping is a simple service designed to check the availability of a host). These malicious ping requests all contained the IP address of a target computer in their sender address field. The ping responses were therefore all directed to the target, whose input buffers were overwhelmed, preventing any legitimate IP packets getting through.

IP routing: The IP layer routes packets from their source to their destination. Each router in the Internet implements IP-layer software to provide a routing algorithm.

**Backbones:** The topological map of the Internet is partitioned conceptually into *autonomous systems* (ASs), which are subdivided into *areas*. The intranets of most large organizations such as universities and large companies are regarded as ASs, and they will usually include several areas. Every AS in the topological map has a *backbone* area. The collection of routers that connect non-backbone areas to the backbone and the links that interconnect those routers are called the backbone of the network. The links in the backbone are usually of high bandwidth and are replicated for reliability. This hierarchic structure is a conceptual one that is exploited primarily for the management of resources and the maintenance of the components. It does not affect the routing of IP packets.

**Routing protocols:** RIP-1, the first routing algorithm used in the Internet, is a version of the distance-vector algorithm. RIP-2 (described in RFC 1388) was developed from it to accommodate several additional requirements, including classless interdomain routing, better multicast routing and the need for authentication of RIP packets to prevent attacks on the routers.

As the scale of the Internet has expanded and the processing capacity of routers has increased, there has been a move towards the adoption of algorithms that do not suffer from the slow convergence and potential instability of distance-vector algorithms.
The direction of the move is towards the link-state class of algorithms and the algorithm called open shortest path first (OSPF). This protocol is based on a path-finding algorithm that is due to Dijkstra and has been shown to converge more rapidly than the RIP algorithm.

**Default routes:** Up to now, our discussion of routing algorithms has suggested that every router maintains a full routing table showing the route to every destination (subnet or directly connected host) in the Internet. At the current scale of the Internet this is clearly infeasible (the number of destinations is probably already in excess of 1 million and still growing very rapidly).

Two possible solutions to this problem come to mind, and both have been adopted in an effort to alleviate the effects of the Internet’s growth. The first solution is to adopt some form of topological grouping of IP addresses. Prior to 1993, nothing could be inferred from an IP address about its location. In 1993, as part of the move to simplify and economize on the allocation of IP addresses that is discussed below under CIDR, the decision was taken that for future allocations, the following regional locations would be applied:

- Addresses 194.0.0.0 to 195.255.255.255 are in Europe
- Addresses 198.0.0.0 to 199.255.255.255 are in North America
- Addresses 200.0.0.0 to 201.255.255.255 are in Central and South America
- Addresses 202.0.0.0 to 203.255.255.255 are in Asia and the Pacific

The default routing scheme is heavily used in Internet routing; no single router holds routes to all destinations in the Internet.

**Routing on a local subnet:** Packets addressed to hosts on the same network as the sender are transmitted to the destination host in a single hop, using the host identifier part of the address to obtain the address of the destination host on the underlying network. The IP layer simply uses ARP to get the network address of the destination and then uses the underlying network to transmit the packets. If the IP layer in the sending computer
discovers that the destination is on a different network, it must send the message to a local router. It uses ARP to get the network address of the gateway or router and then uses the underlying network to transmit the packet to it. Gateways and routers are connected to two or more networks and they have several Internet addresses, one for each network to which they are attached.

**Classless inter-domain routing (CIDR):** The shortage of IP led to the introduction in 1996 of this scheme for allocating addresses and managing the entries in routing tables. The main problem was a scarcity of Class B addresses – those for subnets with more than 255 hosts connected. Plenty of Class C addresses were available. The CIDR solution for this problem is to allocate a batch of contiguous Class C addresses to a subnet requiring more than 255 addresses. The CIDR scheme also makes it possible to subdivide a Class B address space for allocation to multiple subnets.

Batching Class C addresses sounds like a straightforward step, but unless it is accompanied by a change in routing table format, it has a substantial impact on the size of routing tables and hence the efficiency of the algorithms that manage them. The change adopted was to add a *mask* field to the routing tables. The mask is a bit pattern that is used to select the portion of an IP address that is compared with the routing table entry. This effectively enables the host/subnet address to be any portion of the IP address, providing more flexibility than the classes A, B and C – hence the name *classless* inter-domain routing. Once again, these changes to routers are made on an incremental basis, so some routers perform CIDR and others use the old class-based algorithms. This works because the newly allocated ranges of Class C addresses are assigned modulo 256, so each range represents an integral number of Class C sized subnet addresses. On the other hand, some subnets also make use of CIDR to subdivide the range of addresses in a single network, of Class A, B or C. If a collection of subnets is connected to the rest of the world entirely by CIDR routers, then the ranges of IP addresses used within the collection can be allocated to individual subnets in chunks determined by a binary mask of any size.
Unregistered addresses and Network Address Translation (NAT): Not all of the computers and devices that access the Internet need to be assigned globally unique IP addresses. Computers that are attached to a local network and access to the Internet through a NAT-enabled router can rely upon the router to redirect incoming UDP and TCP packets for them.

The network includes Internet-enabled computers that are connected to the router by a wired Ethernet connection as well as others that are connected through a WiFi access point. For completeness some Bluetooth-enabled devices are shown, but these are not connected to the router and hence cannot access the Internet directly. The home network has been allocated a single registered IP address (83.215.152.95) by its Internet service provider. The approach described here is suitable for any organization wishing to connect computers without registered IP addresses to the Internet.

All of the Internet-enabled devices on the home network have been assigned unregistered IP addresses on the 192.168.1.x Class C subnet. Most of the internal computers and devices are allocated individual IP addresses dynamically by a Dynamic Host Configuration Protocol (DHCP) service running on the router. In our illustration the numbers above 192.168.1.100 are used by the DHCP service and the nodes with lower numbers (such as PC 1) have been allocated numbers manually, for a reason explained later in this subsection. Although all of these addresses are completely hidden from the rest of the Internet by the NAT router, it is conventional to use a range of addresses from one of three blocks of addresses (10.z.y.x, 172.16.y.x or 192.168.y.x) that IANA has reserved for private internets.

NAT is described in RFC 1631 and extended in RFC 2663. NAT-enabled routers maintain an address translation table and exploit the source and destination port number fields in the UDP and TCP packets to assign each incoming reply message to the internal computer that sent the corresponding request message. Note that the source port given in a request message is always used as the destination port in the corresponding reply message. The most commonly used variant of NAT addressing works as follows:
When a computer on the internal network sends a UDP or TCP packet to a computer outside it, the router receives the packet and saves the source IP address and port number to an available slot in its address translation table.

The router replaces the source address in the packet with the router’s IP address and the source port with a virtual port number that indexes the table slot containing the sending computer’s address information.

The packet with the modified source address and port number is then forwarded towards its destination by the router. The address translation table now holds a mapping from virtual port numbers to real internal IP addresses and port numbers for all packets sent recently by computers on the internal network.

When the router receives a UDP or TCP packet from an external computer it uses the destination port number in the packet to access a slot in the address translation table. It replaces the destination address and destination port in the received packet with those stored in the slot and forwards the modified packet to the internal computer identified by the destination address. The router will retain a port mapping and reuse it as long as it appears to be in use. A timer is reset each time the router accesses an entry in the table. If the entry is not accessed again before the timer expires, the entry is removed from the table.

The scheme described above deals satisfactorily with the commonest modes of communication for nonregistered computers, in which they act as clients to external services such as web servers. But it does not enable them to act as servers to handle incoming requests. To deal with that case, NAT routers can be configured manually to forward all of the incoming requests on a given port to one particular internal computer.

Computers that act as servers must retain the same internal IP address and this is achieved by allocating their addresses manually (as was done for PC 1). This solution to the problem of providing external access to services is satisfactory as long as there is no requirement for more than one internal computer to offer a service on any given port.
NAT was introduced as a short-term solution to the problem of IP address allocation for personal and home computers. It has enabled the expansion of Internet use to proceed far further than was originally anticipated, but it does impose some limitations, of which the last point is an example. IPv6 must be seen as the next step, enabling full Internet participation for all computers and portable devices.

4.5 SUMMARY

In this unit, we have introduced Networking, networking issues and types of Networks. We also discussed in depth Internet Protocols.

4.6 KEYWORDS

**Hosts:** The computers and other devices that use the network for communication purposes are referred to as *hosts*.

**Node:** The term *node* is used to refer to any computer or switching device attached to a network.

**Internet:** The Internet is a single communication subsystem providing communication between all of the hosts that are connected to it.

**Internetworks:** An internetwork is a communication subsystem in which several networks are linked together to provide common data communication facilities that overlay the technologies and protocols of the individual component networks and the methods used for their interconnection.

4.7 UNIT-END EXERCISES AND ANSWERS

1. Explain various networking issues. 4.2
2. What are the main types of network that are used to support distributed systems? 4.3
3. Explain PANs, LANs, WANs, MANs, WLANs, WMANs, WWANs 4.3
4. Explain the TCP/IP protocol. 4.4
5. Compare TCP and UDP 4.4
6. Explain various classes of Internet address. 4.4
7. Explain Routing protocols. 4.4

Answers: SEE
1. 4.2
2. 4.3
3. 4.3
4. 4.4
5. 4.4
6. 4.4
7. 4.4

4.8 SUGGESTED READINGS

- DISTRIBUTED SYSTEMS Principles and Paradigm By: Andrew S. Tanenbaum, Maarten Van Steen.
Module 2

UNIT 5: INTERPROCESS COMMUNICATION

Structure:

5.0 Objective
5.1 Introduction
5.3 Interprocess communication
5.4 The API for the internet protocols
5.5 Summary
5.6 Keywords
5.7 Unit-end exercises and answers
5.8 Suggested readings

5.0 OBJECTIVES

At the end of this unit you will be able to know:

- The definition of Interprocess communication
- The API for the internet protocols

5.1 INTRODUCTION

This unit is concerned with the characteristics of protocols for communication between processes in a distributed system – that is, interprocess communication.

Interprocess communication in the Internet provides both datagram and stream communication. The Java APIs for these are presented, together with a discussion of their failure models. They provide alternative building blocks for communication protocols. This is complemented by a study of protocols for the representation of collections of data
objects in messages and of references to remote objects. Together, these services offer support for the construction of higher-level communication services.

The interprocess communication primitives support point-to-point communication, yet it is equally useful to be able to send a message from one sender to a group of receivers. The unit also considers multicast communication, including IP multicast and the key concepts of reliability and ordering of messages in multicast communication.

Multicast is an important requirement for distributed applications and must be provided even if underlying support for IP multicast is not available. This is typically provided by an overlay network constructed on top of the underlying TCP/IP network. Overlay networks can also provide support for file sharing, enhanced reliability and content distribution.

The Message Passing Interface (MPI) is a standard developed to provide an API for a set of message-passing operations with synchronous and asynchronous variants

## 5.2 INTERPROCESS COMMUNICATION

This is concerned with the communication aspects of middleware, although the principles discussed are more widely applicable. This unit introduces the characteristics of interprocess communication and then discusses UDP and TCP from a programmer’s point of view, presenting the Java interface to each of these two protocols, together with a discussion of their failure models.

The application program interface to UDP provides a message passing abstraction—the simplest form of interprocess communication. This enables a sending process to transmit a single message to a receiving process. The independent packets containing these messages are called datagrams. In the Java and UNIX APIs, the sender specifies the destination using a socket—an indirect reference to a particular port used by the destination process at a destination computer.
The application program interface to TCP provides the abstraction of a two-way stream between pairs of processes. The information communicated consists of a stream of data items with no message boundaries. Streams provide a building block for producer-consumer communication. A producer and a consumer form a pair of processes in which the role of the first is to produce data items and the role of the second is to consume them. The data items sent by the producer to the consumer are queued on arrival at the receiving host until the consumer is ready to receive them. The consumer must wait when no data items are available. The producer must wait if the storage used to hold the queued data items is exhausted.

5.3 THE API FOR THE INTERNET PROTOCOLS

In this section, we discuss the general characteristics of interprocess communication and then discuss the Internet protocols as an example, explaining how programmers can use them, either by means of UDP messages or through TCP streams.

The characteristics of interprocess communication

Message passing between a pair of processes can be supported by two message communication operations, send and receive, defined in terms of destinations and messages. To communicate, one process sends a message (a sequence of bytes) to a destination and another process at the destination receives the message. This activity involves the communication of data from the sending process to the receiving process and may involve the synchronization of the two processes.

Synchronous and asynchronous communication: A queue is associated with each message destination. Sending processes cause messages to be added to remote queues and receiving processes remove messages from local queues. Communication between the sending and receiving processes may be either synchronous or asynchronous. In the synchronous form of communication, the sending and receiving processes synchronize at
every message. In this case, both send and receive are blocking operations. Whenever a send is issued the sending process (or thread) is blocked until the corresponding receive is issued. Whenever a receive is issued by a process (or thread), it blocks until a message arrives.

In the asynchronous form of communication, the use of the send operation is nonblocking in that the sending process is allowed to proceed as soon as the message has been copied to a local buffer, and the transmission of the message proceeds in parallel with the sending process. The receive operation can have blocking and non-blocking variants. In the non-blocking variant, the receiving process proceeds with its program after issuing a receive operation, which provides a buffer to be filled in the background, but it must separately receive notification that its buffer has been filled, by polling or interrupt.

In a system environment such as Java, which supports multiple threads in a single process, the blocking receive has no disadvantages, for it can be issued by one thread while other threads in the process remain active, and the simplicity of synchronizing the receiving threads with the incoming message is a substantial advantage. Non-blocking communication appears to be more efficient, but it involves extra complexity in the receiving process associated with the need to acquire the incoming message out of its flow of control. For these reasons, today’s systems do not generally provide the nonblocking form of receive.

Message destinations: In the Internet protocols, messages are sent to (Internet address, local port) pairs. A local port is a message destination within a computer, specified as an integer. A port has exactly one receiver (multicast ports are an exception) but can have many senders. Processes may use multiple ports to receive messages. Any process that knows the number of a port can send a message to it. Servers generally publicize their port numbers for use by clients. If the client uses a fixed Internet address to refer to a service, then that service must always run on the same computer for its address to remain valid. This can be avoided by using the following approach to providing location transparency:
• Client programs refer to services by name and use a name server or binder to translate their names into server locations at runtime. This allows services to be relocated but not to migrate – that is, to be moved while the system is running.

**Reliability:** Reliable communication in defined terms of validity and integrity. As far as the validity property is concerned, a point-to-point message service can be described as reliable if messages are guaranteed to be delivered despite a ‘reasonable’ number of packets being dropped or lost. In contrast, a point-to-point message service can be described as unreliable if messages are not guaranteed to be delivered in the face of even a single packet dropped or lost. For integrity, messages must arrive uncorrupted and without duplication.

**Ordering:** Some applications require that messages be delivered in *sender order* – that is, the order in which they were transmitted by the sender. The delivery of messages out of sender order is regarded as a failure by such applications.

**Sockets**

Both forms of communication (UDP and TCP) use the *socket* abstraction, which provides an endpoint for communication between processes. Sockets originate from BSD UNIX but are also present in most other versions of UNIX, including Linux as well as Windows and the Macintosh OS. Interprocess communication consists of transmitting a message between a socket in one process and a socket in another process. For a process to receive messages, its socket must be bound to a local port and one of the Internet addresses of the computer on which it runs. 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 for sending and receiving messages. Each computer has a large number (2^16) of possible port numbers for use by local processes for receiving messages. Any process may make use of multiple ports to receive messages, but a process cannot share ports with other processes on the same
computer. (Processes using IP multicast are an exception in that they do share ports) However, any number of processes may send messages to the same port. Each socket is associated with a particular protocol – either UDP or TCP.

**Java API for Internet addresses:** As the IP packets underlying UDP and TCP are sent to Internet addresses, Java provides a class, `InetAddress`, that represents Internet addresses. Users of this class refer to computers by Domain Name System (DNS) hostnames. For example, instances of `InetAddress` that contain Internet addresses can be created by calling a static method of `InetAddress`, giving a DNS hostname as the argument. The method uses the DNS to get the corresponding Internet address. For example, to get an object representing the Internet address of the host, whose DNS name is `bruno.dcs.qmul.ac.uk`, use:

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

This method can throw an `UnknownHostException`. Note that the user of the class does not need to state the explicit value of an Internet address. In fact, the class encapsulates the details of the representation of Internet addresses. Thus the interface for this class is not dependent on the number of bytes needed to represent Internet addresses – 4 bytes in IPv4 and 16 bytes in IPv6.

**UDP datagram communication**

A datagram sent by UDP is transmitted from a sending process to a receiving process without acknowledgement or retries. If a failure occurs, the message may not arrive. A datagram is transmitted between processes when one process *sends* it and another *receives* it. To send or receive messages a process must first create a socket bound to an Internet address of the local host and a local port. A server will bind its socket to a *server port* – one that it makes known to clients so that they can send messages to it. 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.
The following are some issues relating to datagram communication:

**Message size:** The receiving process needs to specify an array of bytes of a particular size in which to receive a message. If the message is too big for the array, it is truncated on arrival. The underlying IP protocol allows packet lengths of up to 216 bytes, which includes the headers as well as the message. However, most environments impose a size restriction of 8 kilobytes. Any application requiring messages larger than the maximum must fragment them into chunks of that size. Generally, an application, for example DNS, will decide on a size that is not excessively large but is adequate for its intended use.

**Blocking:** Sockets normally provide non-blocking *sends* and blocking *receives* for datagram communication (a non-blocking *receive* is an option in some implementations). The *send* operation returns when it has handed the message to the underlying UDP and IP protocols, which are responsible for transmitting it to its destination. On arrival, the message is placed in a queue for the socket that is bound to the destination port. The message can be collected from the queue by an outstanding or future invocation of *receive* on that socket. Messages are discarded at the destination if no process already has a socket bound to the destination port.

The method *receive* blocks until a datagram is received, unless a timeout has been set on the socket. If the process that invokes the *receive* method has other work to do while waiting for the message, it should arrange to use a separate thread. For example, when a server receives a message from a client, the message may specify work to do, in which case the server will use separate threads to do the work and to wait for messages from other clients.

**Timeouts:** The *receive* that blocks forever is suitable for use by a server that is waiting to receive requests from its clients. But in some programs, it is not appropriate that a process that has invoked a *receive* operation should wait indefinitely in situations where the sending process may have crashed or the expected message may have been lost. To allow for such requirements, timeouts can be set on sockets. Choosing an appropriate
timeout interval is difficult, but it should be fairly large in comparison with the time required to transmit a message.

*Receive from any:* The *receive* method does not specify an origin for messages. Instead, an invocation of *receive* gets a message addressed to its socket from any origin. The *receive* method returns the Internet address and local port of the sender, allowing the recipient to check where the message came from. It is possible to connect a datagram socket to a particular remote port and Internet address, in which case the socket is only able to send messages to and receive messages from that address.

**Failure model for UDP datagrams:** Reliable communication is defined in terms of two properties: integrity and validity. The integrity property requires that messages should not be corrupted or duplicated. The use of a checksum ensures that there is a negligible probability that any message received is corrupted. UDP datagrams suffer from the following failures:

*Omission failures:* Messages may be dropped occasionally, either because of a checksum error or because no buffer space is available at the source or destination. To simplify the discussion, we regard send-omission and receive-omission failures as omission failures in the communication channel.

*Ordering:* Messages can sometimes be delivered out of sender order. Applications using UDP datagrams are left to provide their own checks to achieve the quality of reliable communication they require. A reliable delivery service may be constructed from one that suffers from omission failures by the use of acknowledgements.

**Use of UDP:** For some applications, it is acceptable to use a service that is liable to occasional omission failures. For example, the Domain Name System, which looks up DNS names in the Internet, is implemented over UDP. Voice over IP (VOIP) also runs over UDP. UDP datagrams are sometimes an attractive choice because they do not suffer
from the overheads associated with guaranteed message delivery. There are three main sources of overhead:

- The need to store state information at the source and destination;
- The transmission of extra messages;
- Latency for the sender.

**Java API for UDP datagrams:** The Java API provides datagram communication by means of two classes: `DatagramPacket` and `DatagramSocket`.

`DatagramPacket`: This class provides a constructor that makes an instance out of an array of bytes comprising a message, the length of the message and the Internet address and local port number of the destination socket, as follows:

```
Datagram packet

| array of bytes containing message | length of message | Internet address | port number |
```

An instance of `DatagramPacket` may be transmitted between processes when one process sends it and another receives it.

This class provides another constructor for use when receiving a message. Its arguments specify an array of bytes in which to receive the message and the length of the array. A received message is put in the `DatagramPacket` together with its length and the Internet address and port of the sending socket. The message can be retrieved from the `DatagramPacket` by means of the method `getData`. The methods `getPort` and `getAddress` access the port and Internet address.
import java.net.*;
import java.io.*;

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

Figure 5.1.1 UDP client sends a message to the server and gets a reply

DatagramSocket: This class supports sockets for sending and receiving UDP datagrams. It provides a constructor that takes a port number as its argument, for use by processes that need to use a particular port. It also provides a no-argument constructor that allows the system to choose a free local port. These constructors can throw a SocketException if
the chosen port is already in use or if a reserved port (a number below 1024) is specified when running over UNIX.

The class `DatagramSocket` provides methods that include the following:

`send` and `receive`: These methods are for transmitting datagrams between a pair of sockets. The argument of `send` is an instance of `DatagramPacket` containing a message and its destination. The argument of `receive` is an empty `DatagramPacket` in which to put the message, its length and its origin. The methods `send` and `receive` can throw `IOExceptions`.

`setSoTimeout`: This method allows a timeout to be set. With a timeout set, the `receive` method will block for the time specified and then throw an `InterruptedIOException`.

`connect`: This method is used for connecting to a particular remote port and Internet address, in which case the socket is only able to send messages to and receive messages from that address.

```java
import java.net.*;
import java.io.*;
public class UDPServer{
    public static void main(String args[]){
        DatagramSocket aSocket = null;
        try{
            aSocket = new DatagramSocket(6789);
            byte[] buffer = new byte[1000];
            while(true){
                DatagramPacket request = new DatagramPacket(buffer, buffer.length);
                aSocket.receive(request);
                DatagramPacket reply = new DatagramPacket(request.getData(),
```
```java
request.getLength(), request.getAddress(), request.getPort());

aSocket.send(reply);
}
}

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

Figure 5.1.2 UDP server repeatedly receives a request and sends it back to the client.

Figure 2.1.1 shows the program for a client that creates a socket, sends a message to a server at port 6789 and then waits to receive a reply. The arguments of the main method supply a message and the DNS hostname of the server. The message is converted to an array of bytes, and the DNS hostname is converted to an Internet address. Figure 2.1.2 shows the program for the corresponding server, which creates a socket bound to its server port (6789) and then repeatedly waits to receive a request message from a client, to which it replies by sending back the same message.

**TCP stream communication**

The API to the TCP protocol, which originates from BSD 4.x UNIX, provides the abstraction of a stream of bytes to which data may be written and from which data may be read. The following characteristics of the network are hidden by the stream abstraction:

*Message sizes*: The application can choose how much data it writes to a stream or reads from it. It may deal in very small or very large sets of data. The underlying implementation of a TCP stream decides how much data to collect before transmitting it as one or more IP packets. On arrival, the data is handed to the application as requested. Applications can, if necessary, force data to be sent immediately.
**Lost messages:** The TCP protocol uses an acknowledgement scheme. As an example of a simple scheme (which is not used in TCP), the sending end keeps a record of each IP packet sent and the receiving end acknowledges all the arrivals. If the sender does not receive an acknowledgement within a timeout, it retransmits the message. The more sophisticated sliding window scheme cuts down on the number of acknowledgement messages required.

**Flow control:** The TCP protocol attempts to match the speeds of the processes that read from and write to a stream. If the writer is too fast for the reader, then it is blocked until the reader has consumed sufficient data.

**Message duplication and ordering:** Message identifiers are associated with each IP packet, which enables the recipient to detect and reject duplicates, or to reorder messages that do not arrive in sender order.

**Message destinations:** A pair of communicating processes establishes a connection before they can communicate over a stream. Once a connection is established, the processes simply read from and write to the stream without needing to use Internet addresses and ports. Establishing a connection involves a `connect` request from client to server followed by an `accept` request from server to client before any communication can take place. This could be a considerable overhead for a single client-server request and reply.

The API for stream communication assumes that when a pair of processes are establishing a connection, one of them plays the client role and the other plays the server role, but thereafter they could be peers. The client role involves creating a stream socket bound to any port and then making a `connect` request asking for a connection to a server at its server port. The server role involves creating a listening socket bound to a server port and waiting for clients to request connections. The listening socket maintains a queue of incoming connection requests. In the socket model, when the server `accepts` a
connection, a new stream socket is created for the server to communicate with a client, meanwhile retaining its socket at the server port for listening for connect requests from other clients.

The pair of sockets in the client and server are connected by a pair of streams, one in each direction. Thus each socket has an input stream and an output stream. One of the pair of processes can send information to the other by writing to its output stream, and the other process obtains the information by reading from its input stream. When an application closes a socket, this indicates that it will not write any more data to its output stream. Any data in the output buffer is sent to the other end of the stream and put in the queue at the destination socket, with an indication that the stream is broken. The process at the destination can read the data in the queue, but any further reads after the queue is empty will result in an indication of end of stream. When a process exits or fails, all of its sockets are eventually closed and any process attempting to communicate with it will discover that its connection has been broken.

The following are some outstanding issues related to stream communication:

**Matching of data items**: Two communicating processes need to agree as to the contents of the data transmitted over a stream. For example, if one process writes an `int` followed by a `double` to a stream, then the reader at the other end must read an `int` followed by a `double`. When a pair of processes does not cooperate correctly in their use of a stream, the reading process may experience errors when interpreting the data or may block due to insufficient data in the stream.

**Blocking**: The data written to a stream is kept in a queue at the destination socket. When a process attempts to read data from an input channel, it will get data from the queue or it will block until data becomes available. The process that writes data to a stream may be blocked by the TCP flow-control mechanism if the socket at the other end is queuing as much data as the protocol allows.
Threads: When a server accepts a connection, it generally creates a new thread in which to communicate with the new client. The advantage of using a separate thread for each client is that the server can block when waiting for input without delaying other clients. In an environment in which threads are not provided, an alternative is to test whether input is available from a stream before attempting to read it; for example, in a UNIX environment the `select` system call may be used for this purpose.

Failure model: To satisfy the integrity property of reliable communication, TCP streams use checksums to detect and reject corrupt packets and sequence numbers to detect and reject duplicate packets. For the sake of the validity property, TCP streams use timeouts and retransmissions to deal with lost packets. Therefore, messages are guaranteed to be delivered even when some of the underlying packets are lost. But if the packet loss over a connection passes some limit or the network connecting a pair of communicating processes is severed or becomes severely congested, the TCP software responsible for sending messages will receive no acknowledgements and after a time will declare the connection to be broken. Thus TCP does not provide reliable communication, because it does not guarantee to deliver messages in the face of all possible difficulties.

When a connection is broken, a process using it will be notified if it attempts to read or write. This has the following effects:

- The processes using the connection cannot distinguish between network failure and failure of the process at the other end of the connection.
- The communicating processes cannot tell whether the messages they have sent recently have been received or not.

Use of TCP: Many frequently used services run over TCP connections, with reserved port numbers. These include the following:

HTTP: The Hypertext Transfer Protocol is used for communication between web browsers and web servers.
**FTP**: The File Transfer Protocol allows directories on a remote computer to be browsed and files to be transferred from one computer to another over a connection.

**Telnet**: Telnet provides access by means of a terminal session to a remote computer.

**SMTP**: The Simple Mail Transfer Protocol is used to send mail between computers.

**Java API for TCP streams**: The Java interface to TCP streams is provided in the classes `ServerSocket` and `Socket`:

*ServerSocket*: This class is intended for use by a server to create a socket at a server port for listening for `connect` requests from clients. Its `accept` method gets a `connect` request from the queue or, if the queue is empty, blocks until one arrives. The result of executing `accept` is an instance of `Socket` – a socket to use for communicating with the client.

*Socket*: This class is for use by a pair of processes with a connection. The client uses a constructor to create a socket, specifying the DNS hostname and port of a server. This constructor not only creates a socket associated with a local port but also `connects` it to the specified remote computer and port number. It can throw an `UnknownHostException` if the hostname is wrong or an `IOException` if an IO error occurs.

The *Socket* class provides the methods `getInputStream` and `getOutputStream` for accessing the two streams associated with a socket. The return types of these methods are `InputStream` and `OutputStream`, respectively – abstract classes that define methods for reading and writing bytes. The return values can be used as the arguments of constructors for suitable input and output streams. Our example uses `DataInputStream` and `DataOutputStream`, which allow binary representations of primitive data types to be read and written in a machine-independent manner

```java
import java.net.*;
import java.io.*;
```
public class TCPClient {

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

Figure 5.1.3 TCP client makes connection to server, sends request and receives reply

Figure 2.1.3 shows a client program in which the arguments of the main method supply a message and the DNS hostname of the server. The client creates a socket bound to the hostname and server port 7896. It makes a DataInputStream and a DataOutputStream from the socket’s input and output streams, then writes the message to its output stream and waits to read a reply from its input stream. The server program in Figure 2.1.4 opens a server socket on its server port (7896) and listens for connect requests. When one arrives, it makes a new thread in which to communicate with the client. The new thread
creates a `DataInputStream` and a `DataOutputStream` from its socket’s input and output streams and then waits to read a message and write the same on e back.

```java
import java.net.*;
import java.io.*;

public class TCPServer {
    public static void main (String args[]) {
        try{
            int serverPort = 7896;
            ServerSocket listenSocket = new ServerSocket(serverPort);
            while(true) {
                Socket clientSocket = listenSocket.accept();
                Connection c = new Connection(clientSocket);
            }
        } catch(IOException e) {System.out.println("Listen :"+e.getMessage());}
    }
}

class Connection extends Thread {
    DataInputStream in;
    DataOutputStream out;
    Socket clientSocket;
    public Connection (Socket aClientSocket) {
        try {
            clientSocket = aClientSocket;
            in = new DataInputStream( clientSocket.getInputStream());
            out =new DataOutputStream( clientSocket.getOutputStream());
            this.start();
        } catch(IOException e) {System.out.println("Connection:"+e.getMessage());}
    }
    public void run(){
```
try { // an echo server
    String data = in.readUTF();
    out.writeUTF(data);
} catch(EOFException e) {System.out.println("EOF:"+e.getMessage());
} catch(IOException e) {System.out.println("IO:"+e.getMessage());
} finally { try {clientSocket.close();}catch (IOException e){/*close failed*/}}

Figure 5.1.4 TCP server makes a connection for each client and then echoes the client’s request

As our message consists of a string, the client and server processes use the method writeUTF of DataOutputStream to write it to the output stream and the method readUTF of DataInputStream to read it from the input stream. UTF-8 is an encoding that represents strings in a particular format.

When a process has closed its socket, it will no longer be able to use its input and output streams. The process to which it has sent data can read the data in its queue, but any further reads after the queue is empty will result in an EOFException. Attempts to use a closed socket or to write to a broken stream result in an IOException.

5.6. SUMMARY

In this unit we have discuss about interprocess communication and APIs for the internet protocol.
5.7. KEYWORDS

Interprocess communication: communication between processes in a distributed system.

The Message Passing Interface (MPI): is a standard developed to provide an API for a set of message-passing operations with synchronous and asynchronous variants.

HTTP: The Hypertext Transfer Protocol is used for communication between web browsers and web servers.

5.8. UNIT-END EXERCISES AND ANSWERS

1. Explain interprocess communication. 1.2
2. Explain the characteristics of interprocess communication. 1.3
3. Explain UDP datagram communication. 1.3
4. Explain the use of Use of TCP: 1.3
5. Compare Synchronous and asynchronous communication: 1.3
6. Discuss about Java API for Internet addresses. 1.3

Answers: SEE

1. 5.2
2. 5.3
3. 5.3
4. 5.3
5. 5.3
6. 5.3
5.9 SUGGESTED READINGS

- DISTRIBUTED SYSTEMS Principles and Paradigm By: Andrew S. Tanenbaum, Maarten Van Steen.
### UNIT 6: EXTERNAL DATA REPRESENTATION

<table>
<thead>
<tr>
<th>Structure:</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.0  Objectives</td>
</tr>
<tr>
<td>6.1  Introduction</td>
</tr>
<tr>
<td>6.2  External data representation and Marshalling</td>
</tr>
<tr>
<td>6.3  Client server communication</td>
</tr>
<tr>
<td>6.4  Group Communication</td>
</tr>
<tr>
<td>6.5  Summary</td>
</tr>
<tr>
<td>6.6  Keywords</td>
</tr>
<tr>
<td>6.7  Unit-end exercises and answers</td>
</tr>
<tr>
<td>6.8  Suggested readings</td>
</tr>
</tbody>
</table>

### 6.0 OBJECTIVES
At the end of this unit you will be able to know:
- Data representation
- Marshalling
- Client server communication
- Group communication

### 6.1 INTRODUCTION
In this unit, we discuss about the external data representation, marshalling, Client server communication, and Group communication.

The information stored in running programs is represented as data structures – for example, by sets of interconnected objects – whereas the information in messages consists of sequences of bytes. Irrespective of the form of communication used, the data structures must be flattened (converted to a sequence of bytes) before transmission and rebuilt on arrival. The individual primitive data items transmitted in messages can be data values of many different types, and not all computers store primitive values such as integers in the same order. The representation of floating-point numbers also differs
between architectures. There are two variants for the ordering of integers: the so-called *big-endian* order, in which the most significant byte comes first; and *little-endian* order, in which it comes last. Another issue is the set of codes used to represent characters: for example, the majority of applications on systems such as UNIX use ASCII character coding, taking one byte per character, whereas the Unicode standard allows for the representation of texts in many different languages and takes two bytes per character.

### 6.2 EXTERNAL DATA REPRESENTATION AND MARSHALLING

One of the following methods can be used to enable any two computers to exchange binary data values:

- The values are converted to an agreed external format before transmission and converted to the local form on receipt; if the two computers are known to be the same type, the conversion to external format can be omitted.
- The values are transmitted in the sender’s format, together with an indication of the format used, and the recipient converts the values if necessary.

Note, however, that bytes themselves are never altered during transmission. To support RMI or RPC, any data type that can be passed as an argument or returned as a result must be able to be flattened and the individual primitive data values represented in an agreed format. An agreed standard for the representation of data structures and primitive values is called an *external data representation*.

*Marshalling* is the process of taking a collection of data items and assembling them into a form suitable for transmission in a message. *Unmarshalling* is the process of disassembling them on arrival to produce an equivalent collection of data items at the destination. Thus marshalling consists of the translation of structured data items and primitive values into an external data representation. Similarly, unmarshalling consists of the generation of primitive values from their external data representation and the rebuilding of the data structures.
Three alternative approaches to external data representation and marshalling are discussed:

- **CORBA’s common data representation**, which is concerned with an external representation for the structured and primitive types that can be passed as the arguments and results of remote method invocations in CORBA. It can be used by a variety of programming languages.
- **Java’s object serialization**, which is concerned with the flattening and external data representation of any single object or tree of objects that may need to be transmitted in a message or stored on a disk. It is for use only by Java.
- **XML (Extensible Markup Language)**, which defines a textual format for representing structured data. It was originally intended for documents containing textual self-describing structured data – for example documents accessible on the Web – but it is now also used to represent the data sent in messages exchanged by clients and servers in web services.

In the first two cases, the marshalling and unmarshalling activities are intended to be carried out by a middleware layer without any involvement on the part of the application programmer. Even in the case of XML, which is textual and therefore more accessible to hand-encoding, software for marshalling and unmarshalling is available for all commonly used platforms and programming environments. Because marshalling requires the consideration of all the finest details of the representation of the primitive components of composite objects, the process is likely to be error-prone if carried out by hand. Compactness is another issue that can be addressed in the design of automatically generated marshalling procedures.

In the first two approaches, the primitive data types are marshalled into a binary form. In the third approach (XML), the primitive data types are represented textually. The textual representation of a data value will generally be longer than the equivalent binary representation. The HTTP protocol is another example of the textual approach.
Another issue with regard to the design of marshalling methods is whether the marshalled data should include information concerning the type of its contents. For example, CORBA’s representation includes just the values of the objects transmitted, and nothing about their types. On the other hand, both Java serialization and XML do include type information, but in different ways. Java puts all of the required type information into the serialized form, but XML documents may refer to externally defined sets of names (with types) called *namespaces*.

Although we are interested in the use of an external data representation for the arguments and results of RMIs and RPCs, it does have a more general use for representing data structures, objects or structured documents in a form suitable for transmission in messages or storing in files.

Two other techniques for external data representation are worthy of mention. Google uses an approach called *protocol buffers* to capture representations of both stored and transmitted data. There is also considerable interest in JSON (JavaScript Object Notation) as an approach to external data representation. Protocol buffers and JSON represent a step towards more lightweight approaches to data representation (when compared, for example, to XML).

**CORBA’s Common Data Representation (CDR)**

CORBA CDR is the external data representation defined with CORBA 2.0. CDR can represent all of the data types that can be used as arguments and return values in remote invocations in CORBA. These consist of 15 primitive types, which include *short* (16-bit), *long* (32-bit), *unsigned short*, *unsigned long*, *float* (32-bit), *double* (64-bit), *char*, *boolean* (TRUE, FALSE), *octet* (8-bit), and *any* (which can represent any basic or constructed type); together with a range of composite types, which are described in Figure 2.2.1. Each argument or result in a remote invocation is represented by a sequence of bytes in the invocation or result message.
<table>
<thead>
<tr>
<th>Type</th>
<th>Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequence</td>
<td>length (unsigned long) followed by elements in order</td>
</tr>
<tr>
<td>String</td>
<td>length (unsigned long) followed by characters in order (can also have wide characters)</td>
</tr>
<tr>
<td>Array</td>
<td>array elements in order (no length specified because it is fixed)</td>
</tr>
<tr>
<td>Struct</td>
<td>in the order of declaration of the components</td>
</tr>
<tr>
<td>Enumerated</td>
<td>unsigned long (the values are specified by the order declared)</td>
</tr>
<tr>
<td>Union</td>
<td>type tag followed by the selected member</td>
</tr>
</tbody>
</table>

Figure 6.2.1: CORBA CDR for constructed types

**Primitive types:** CDR defines a representation for both big-endian and little-endian orderings. Values are transmitted in the sender’s ordering, which is specified in each message. The recipient translates if it requires a different ordering. For example, a 16-bit short occupies two bytes in the message, and for big-endian ordering, the most significant bits occupy the first byte and the least significant bits occupy the second byte. Each primitive value is placed at an index in the sequence of bytes according to its size. Suppose that the sequence of bytes is indexed from zero upwards. Then a primitive value of size $n$ bytes (where $n = 1, 2, 4$ or $8$) is appended to the sequence at an index that is a multiple of $n$ in the stream of bytes. Floating-point values follow the IEEE standard, in which the sign, exponent and fractional part are in bytes $0–n$ for big-endian ordering and the other way round for little-endian. Characters are represented by a code set agreed between client and server.

**Constructed types:** The primitive values that comprise each constructed type are added to a sequence of bytes in a particular order, as shown in Figure 2.2.1.
Figure 6.2.2: CORBA CDR message

Figure 6.2.2 shows a message in CORBA CDR that contains the three fields of a `struct` whose respective types are `string`, `string` and `unsigned long`. The figure shows the sequence of bytes with four bytes in each row. The representation of each string consists of an `unsigned long` representing its length followed by the characters in the string. For simplicity, we assume that each character occupies just one byte. Variable-length data is padded with zeros so that it has a standard form, enabling marshalled data or its checksum to be compared. Note that each `unsigned long`, which occupies four bytes, starts at an index that is a multiple of four. The figure does not distinguish between the big- and little-endian orderings. Although the example in Figure 6.2.2 is simple, CORBA CDR can represent any data structure that can be composed from the primitive and constructed types, but without using pointers.

Another example of an external data representation is the Sun XDR standard, which is specified in RFC. It was developed by Sun for use in the messages exchanged between clients and servers in Sun NFS.

The type of a data item is not given with the data representation in the message in either the CORBA CDR or the Sun XDR standard. This is because it is assumed that the sender and recipient have common knowledge of the order and types of the data items in a
message. In particular, for RMI or RPC, each method invocation passes arguments of particular types, and the result is a value of a particular type.

**Marshalling in CORBA** Marshalling operations can be generated automatically from the specification of the types of data items to be transmitted in a message. The types of the data structures and the types of the basic data items are described in CORBA IDL, which provides a notation for describing the types of the arguments and results of RMI methods. For example, we might use CORBA IDL to describe the data structure in the message in Figure 2.2.2 as follows:

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

The CORBA interface compiler generates appropriate marshalling and unmarshalling operations for the arguments and results of remote methods from the definitions of the types of their parameters and results.

**Remote object references**

This section applies only to languages such as Java and CORBA that support the distributed object model. It is not relevant to XML. When a client invokes a method in a remote object, an invocation message is sent to the server process that hosts the remote object. This message needs to specify which particular object is to have its method invoked. A *remote object reference* is an identifier for a remote object that is valid throughout a distributed system. A remote object reference is passed in the invocation message to specify which object is to be invoked.

Each remote object has a single remote object reference and that remote object references can be compared to see whether they refer to the same remote object. Here, we discuss
the external representation of remote object references. Remote object references must be generated in a manner that ensures uniqueness over space and time. In general, there may be many processes hosting remote objects, so remote object references must be unique among all of the processes in the various computers in a distributed system. Even after the remote object associated with a given remote object reference is deleted, it is important that the remote object reference is not reused, because its potential invokers may retain obsolete remote object references. Any attempt to invoke a deleted object should produce an error rather than allow access to a different object.

There are several ways to ensure that a remote object reference is unique. One way is to construct a remote object reference by concatenating the Internet address of its host computer and the port number of the process that created it with the time of its creation and a local object number. The local object number is incremented each time an object is created in that process.

The port number and time together produce a unique process identifier on that computer. With this approach, remote object references might be represented with a format such as that shown in Figure 2.2.3. In the simplest implementations of RMI, remote objects live only in the process that created them and survive only as long as that process continues to run. In such cases, the remote object reference can be used as the address of the remote object. In other words, invocation messages are sent to the Internet address in the remote reference and to the process on that computer using the given port number.

<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>
</tbody>
</table>

Figure 6.2.3: Representation of a remote object reference

To allow remote objects to be relocated into a different process on a different computer, the remote object reference should not be used as the address of the remote object.
The peer-to-peer overlay systems use a form of remote object reference that is completely independent of location. Messages are routed to resources by means of a distributed routing algorithm.

The last field of the remote object reference shown in Figure 2.2.3 contains some information about the interface of the remote object, for example, the interface name. This information is relevant to any process that receives a remote object reference as an argument or as the result of a remote invocation, because it needs to know about the methods offered by the remote object.

### 6.3 CLIENT-SERVER COMMUNICATION

The client-server communication is designed to support the roles and message exchanges in typical client-server interactions. In the normal case, request-reply communication is synchronous because the client process blocks until the reply arrives from the server. Asynchronous request-reply communication is an alternative that is useful where clients can afford to retrieve replies later. It is often built over UDP datagrams. Client-server protocol consists of request/response pairs, hence no acknowledgements at transport layer are necessary. So, it has no connection establishment overhead. 

There is no need for flow control due to small amounts of data being transferred. The request-reply protocol was based on a trio of communication primitives: doOperation, getRequest, and sendReply shown in Figure 2.2.4.
The designed request-reply protocol matches requests to replies. If UDP datagrams are used, the delivery guarantees must be provided by the request-reply protocol, which may use the server reply message as an acknowledgement of the client request message. Figure 2.2.5 outlines the three communication primitives.

\begin{verbatim}
public byte[] doOperation (RemoteObjectRef o, int methodId, byte[] arguments)
    sends a request message to the remote object and returns the reply.
    The arguments specify the remote object, the method to be invoked and the arguments of that method.

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.
\end{verbatim}

The information to be transmitted in a request message or a reply message is shown in Figure 6.2.6.
In a protocol message:

- The first field indicates whether the message is a request or a reply message.
- The second field request id contains a message identifier.
- The third field is a remote object reference.
- The forth field is an identifier for the method to be invoked.

Message identifier: A message identifier consists of two parts:

- A requestId, which is taken from an increasing sequence of integers by the sending process
- An identifier for the sender process, for example its port and Internet address.

Failure model of the request-reply protocol: If the three primitive doOperation, getRequest, and sendReply are implemented over UDP datagram, they have the same communication failures.

- Omission failure
- Messages are not guaranteed to be delivered in sender order.

RPC exchange protocols: Three protocols are used for implementing various types of RPC.

- The request (R) protocol.
- The request-reply (RR) protocol.
- The request-reply-acknowledge (RRA) protocol.
In the R protocol, a single request message is sent by the client to the server. The R protocol may be used when there is no value to be returned from the remote method. The RR protocol is useful for most client-server exchanges because it is based on request-reply protocol. RRA protocol is based on the exchange of three messages: request-reply-acknowledge reply. HTTP is an example of a request-reply protocol. HTTP is a request-reply protocol for the exchange of network resources between web clients and web servers.

HTTP protocol has the following steps:

- Connection establishment between client and server at the default server port or at a port specified in the URL
- client sends a request
- server sends a reply
- connection closure

HTTP 1.1 uses persistent connections. Persistent connections are connections that remain open over a series of request-reply exchanges between client and server. Resources can have MIME-like structures in arguments and results. A Mime type specifies a type and a subtype, for example text/plain, text/html, image/gif, image/jpeg.

The following are the HTTP methods:
GET: It requests the resource, identified by URL as argument. If the URL refers to data, then the web server replies by returning the data. If the URL refers to a program, then the web server runs the program and returns the output to the client.

<table>
<thead>
<tr>
<th>method</th>
<th>URL</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 2.2.8: HTTP request message

HEAD: This method is similar to GET, but only meta-data on resource is returned (like date of last modification, type, and size)

POST: It specifies the URL of a resource (for instance, a server program) that can deal with the data supplied with the request. This method is designed to deal with:

- Providing a block of data to a data-handling process
- Posting a message to a bulletin board, mailing list or news group.
- Extending a dataset with an append operation

PUT: It stores the supplied data in the given URL as its identifier.

DELETE: The server deletes an identified resource by the given URL on the server.

OPTIONS: A server supplies the client with a list of methods. It allows to be applied to the given URL.

TRACE: The server sends back the request message. A reply message specifies. The protocol version, A status code, Reason, Some headers, An optional message body

<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>

Figure 2.2.9 HTTP reply message
6.4 GROUP COMMUNICATION

Group communication provides an example of an indirect communication paradigm. *Group communication* offers a service whereby a message is sent to a group and then this message is delivered to all members of the group. In this action, the sender is not aware of the identities of the receivers. Group communication represents an abstraction over multicast communication and may be implemented over IP multicast or an equivalent overlay network, adding significant extra value in terms of managing group membership, detecting failures and providing reliability and ordering guarantees. With the added guarantees, group communication is to IP multicast what TCP is to the point-to-point service in IP.

Group communication is an important building block for distributed systems, and particularly reliable distributed systems, with key areas of application including:

- The reliable dissemination of information to potentially large numbers of clients, including in the financial industry, where institutions require accurate and up-to-date access to a wide variety of information sources;

- Support for collaborative applications, where again events must be disseminated to multiple users to preserve a common user view – for example, in multiuser games.

- Support for a range of fault-tolerance strategies, including the consistent update of replicated data or the implementation of highly available (replicated) servers;

- Support for system monitoring and management, including for example load balancing strategies.
We look at group communication in more detail below, examining the programming model offered and the associated implementation issues. We examine the JGroups toolkit as a case study of a group communication service.

**The programming model**

In group communication, the central concept is that of a *group* with associated *group-membership*, whereby processes may *join* or *leave* the group. Processes can then send a message to this group and have it propagated to all members of the group with certain guarantees in terms of reliability and ordering. Thus, group communication implements *multicast* communication, in which a message is sent to all the members of the group by a single operation. Communication to *all* processes in the system, as opposed to a subgroup of them, is known as *broadcast*, whereas communication to a single process is known as *unicast*.

The essential feature of group communication is that a process issues only one multicast operation to send a message to each of a group of processes (in Java this operation is `aGroup.send(aMessage)`) instead of issuing multiple send operations to individual processes.

The use of a single multicast operation instead of multiple send operations amounts to much more than a convenience for the programmer: it enables the implementation to be efficient in its utilization of bandwidth. It can take steps to send the message no more than once over any communication link, by sending it over a distribution tree; and it can use network hardware support for multicast where this is available. The implementation can also minimize the total time taken to deliver the message to all destinations, as compared with transmitting it separately and serially.

To see these advantages, compare the bandwidth utilization and the total transmission time taken when sending the same message from a computer in London to two computers on the same Ethernet in Palo Alto, (a) by two separate UDP sends, and (b) by a single IP
multicast operation. In the former case, two copies of the message are sent independently, and the second is delayed by the first. In the latter case, a set of multicast aware routers forward a single copy of the message from London to a router on the destination LAN in California. That router then uses hardware multicast (provided by the Ethernet) to deliver the message to both destinations at once, instead of sending it twice.

The use of a single multicast operation is also important in terms of delivery guarantees. If a process issues multiple independent send operations to individual processes, then there is no way for the implementation to provide guarantees that affect the group of processes as a whole. If the sender fails halfway through sending, then some members of the group may receive the message while others do not. In addition, the relative ordering of two messages delivered to any two group members is undefined.

Group communication, however, has the potential to offer a range of guarantees in terms of reliability and ordering.

Group communication has been the subject of many research projects, including the V-system, Chorus, Amoeba, Trans/Total, Delta-4, Isis, Horus, Totem and Transis.

**Process groups and object groups:** Most work on group services focuses on the concept of *process groups*, that is, groups where the communicating entities are processes. Such services are relatively low-level in that:

- Messages are delivered to processes and no further support for dispatching is provided.
- Messages are typically unstructured byte arrays with no support for marshalling of complex data types.

The level of service provided by process groups is therefore similar to that of sockets. In contrast, *object groups* provide a higher-level approach to group computing. An object group is a collection of objects (normally instances of the same class) that process the same set of invocations concurrently, with each returning responses. Client objects need
not be aware of the replication. They invoke operations on a single, local object, which acts as a proxy for the group. The proxy uses a group communication system to send the invocations to the members of the object group. Object parameters and results are marshalled as in RMI and the associated calls are dispatched automatically to the right destination objects/methods.

Electra is a CORBA-compliant system that supports object groups. An Electra group can be interfaced to any CORBA-compliant application. Electra was originally built on top of the Horus group communication system, which it uses to manage the membership of the group and to multicast invocations. In ‘transparent mode’, the local proxy returns the first available response to a client object. In ‘non-transparent mode’, the client object can access all the responses returned by the group members. Electra uses an extension of the standard CORBA Object Request Broker interface, with functions for creating and destroying object groups and managing their membership. Eternal and the Object Group Service also provide CORBA-compliant support for object groups.

Despite the promise of object groups, however, process groups still dominate in terms of usage. For example, the popular JGroups toolkit, is a classical process group approach.

Other key distinctions: A wide range of group communication services has been developed, and they vary in the assumptions they make:

*Closed and open groups*: A group is said to be closed if only members of the group may multicast to it. A process in a closed group delivers to itself any message that it multicasts to the group. A group is open if processes outside the group may send to it. (The categories ‘open’ and ‘closed’ also apply with analogous meanings to mailing lists.) Closed groups of processes are useful, for example, for cooperating servers to send messages to one another that only they should receive. Open groups are useful, for example, for delivering events to groups of interested processes.
Overlapping and non-overlapping groups: In overlapping groups, entities (processes or objects) may be members of multiple groups, and non-overlapping groups imply that membership does not overlap (that is, any process belongs to at most one group). Note that in real-life systems, it is realistic to expect that group membership will overlap.

Synchronous and asynchronous systems: There is a requirement to consider group communication in both environments. Such distinctions can have a significant impact on the underlying multicast algorithms. For example, some algorithms assume that groups are closed. The same effect as openness can be achieved with a closed group by picking a member of the group and sending it a message (one-to-one) for it to multicast to its group. Rodrigues et al. discuss multicast to open groups. Issues related to open and closed groups arise, when algorithms for reliability and ordering are considered. We must also consider the impact of overlapping groups and whether the system is synchronous or asynchronous on such protocols.

6.5 SUMMARY

In this unit we introduced external data representation and marshalling. We also discussed about client server communication and group communication.

6.6 KEYWORDS

Marshalling: It is the process of taking a collection of data items and assembling them into a form suitable for transmission in a message.

Unmarshalling: It is the process of disassembling them on arrival to produce an equivalent collection of data items at the destination.

There are two variants for the ordering of integers: the so-called big-endian order, in which the most significant byte comes first; and little-endian order, in which it comes last.
Closed and open groups: A group is said to be *closed* if only members of the group may multicast to it. A process in a closed group delivers to itself any message that it multicasts to the group. A group is *open* if processes outside the group may send to it.

### 6.7 UNIT-END EXERCISES AND ANSWERS

1. What are the methods that can be used to enable any two computers to exchange binary data values? List them.
2. Explain marshalling and unmarshalling.
3. List three alternative approaches to external data representation and marshalling.
4. Explain CORBA’s Common Data Representation (CDR).
5. With a neat diagram, explain about client server communication.
6. Discuss about group communication in distributed systems.

**Answers:** SEE

1. 6.2
2. 6.2
3. 6.2
4. 6.2
5. 6.3
6. 6.4

### 6.8 SUGGESTED READINGS

- DISTRIBUTED SYSTEMS Principles and Paradigm By: Andrew S. Tanenbaum, Maarten Van Steen.
UNIT 7: DISTRIBUTED OBJECTS AND REMOTE INVOCATION

Structure:
7.0 Objectives
7.1 Introduction
7.2 Distributed Objects
7.3 Remote invocation
7.4 Communication between distributed Objects
7.5 Summary
7.6 Keywords
7.7 Unit-end exercises and answers
7.8 Suggested readings

7.0 OBJECTIVES

At the end of this unit you will be able to know:

- Definition of distributed objects
- Remote invocation
- Communication between distributed objects

7.1 INTRODUCTION

This unit is concerned with programming models for distributed applications. This unit introduces communication between distributed objects by means of remote method Invocation (RMI). Objects that can receive remote method invocations are called remote objects and they implement a remote interface. Due to the possibility of independent failure of invoker and Invoked objects, AMPS have different semantics from local calls. They can be made to look very similar to local invocations, but total transparency is not necessarily desirable.
7.2 DISTRIBUTED OBJECTS

The code for marshalling and unmarshalling arguments and sending request and reply messages can be generated automatically by an interface compiler from the definition of the remote interface. Remote procedure call is to RMI as procedure call is to object invocation.

Distributed event-based systems allow objects to subscribe to events occurring at remote objects of interest and in turn to receive notifications when such events occur. Events and notifications provide a way for heterogeneous Objects to communicate with one another asynchronously.

Here we discuss the programming models for distributed applications — that is, those applications that are composed of cooperating programs running in several different processes. Such programs need to be able to invoke operations in other processes, often running in different computers. To achieve this, some familiar programming models that have been extended to apply to distributed programs:

- The earliest and perhaps the best-known of these was the extension of the conventional procedure call model to the remote procedure call model, which allows client programs to call procedures in server programs running in separate processes and generally in different computers from the client.

- More recently, the object-based programming model has been extended to allow objects in different processes to communicate with one another by means of remote method invocation (RMI). RMI is an extension of local method invocation that allows an object living in one process to invoke the methods of an object living in another process.

- The event-based programming model allows objects to receive notification of the events at other objects in which they have registered interest. This model has been extended to allow distributed event-based programs to be written.

Note that we use the term `RMI' to refer to remote method invocation in a generic way — this should not be confused with particular examples of remote method invocation such as Java RMI.
7.3 REMOTE INVOCATION

Most current distributed systems software is written in object-oriented languages and RPC can be understood in relation to RMI.

Therefore this unit concentrates on the RMI and event paradigms, each of which applies to distributed objects and communication between distributed objects.

**Middleware:** It is a software that provides a programming model about the basic building blocks of processes and messages passing. The middleware layer uses protocols based on messages between processes to provide its higher-level abstractions such as remote invocations and events as shown in Figure 2.3.1.

An important aspect of middleware is the provision of location transparency and independence from the details of communication protocols, operating systems and computer hardware. Some forms of middleware allow the separate components to be written in different programming languages.

**Location transparency:** In RPC, the client that calls a procedure cannot tell whether the procedure runs in the same process or in a different process, possibly on a different computer, nor does the client need to know the location of the server. Similarly, in RMI the objects making the invocation cannot tell whether the object it invokes is local or not and does not need to know its location. Also in distributed event-based programs, the objects generating events and the objects that receive notifications of those events need not be aware of one another’s' locations.

![Figure 7.3.1 Middleware layer](image-url)
Communication protocols: The protocols that support the middleware abstractions are independent of the underlying transport protocols. For example, the request-reply protocol can be implemented over either LTDP or TCP.

Computer hardware: Two agreed standards for external data representation are described in Unit 2 of Module 2. These are used when marshalling and unmarshalling messages. They hide the differences due to hardware architectures, such as byte ordering.

Operating systems: The higher-level abstractions provided by the middleware layer are independent of the underlying operating systems.

Use of several programming languages: Some middleware is designed to allow distributed applications to use more than one programming language. In particular, CORBA allows clients written in one language to invoke methods in objects that live in server programs written in another language. This is achieved by using an interface definition language or IDL to define interfaces.

Interfaces
Most modern programming languages provide a means of organizing a program as a set of modules that can communicate with one another. Communication between modules can be by means of procedure calls between modules or by direct access to the variables in another module. In order to control the possible interactions between modules, an explicit interface is defined for each module. The interface of a module specifies the procedures and the variables that can be accessed from other modules. Modules are implemented so as to hide all the information about them except that which is available through its interface. So long as its interface remains the same, the implementation may be changed without affecting the users of the module.

Interfaces in distributed systems: In a distributed program, the modules can run in separate processes. It is not possible for a module running in one process to access the variables in a module in another process. Therefore, the interface of a module that is intended for RPC or RMI cannot specify direct access to variables. Note that CORBA IDL interfaces can specify attributes, which seems to break this rule. However, the attributes are not accessed directly but by means of some getter and setter procedures.
added automatically to the interface. The parameter-passing mechanisms (for example call by value and call by reference) used in local procedure call are not suitable when the caller and procedure are in different processes. The specification of a procedure or method in the interface of a module in a distributed program describes the parameters as input or output or sometimes both. Input parameters are passed to the remote module by sending the values of the arguments in the request message and then supplying them as arguments to the operation to be executed in the server. Output parameters are returned in the reply message and are used as the result of the call or to replace the values of the corresponding variables in the calling environment. When a parameter is used for both input and output, the value must be transmitted in both the request and reply messages. Another difference between local and remote modules is that pointers in one process are not valid in another remote one. Therefore, pointers cannot be passed as arguments or returned as results of calls to remote modules.

Now we discuss the interfaces used in the original client-server model for RPC and in the distributed object model for RMI.

Service interfaces: In the client-server model, each server provides a set of procedures that are available for use by clients. For example, a file server would provide procedures for reading and writing files. The term service interface is used to refer to the specification of the procedures offered by a server, defining the types of the input and output arguments of each of the procedures.

Remote interfaces: In the distributed object model, a remote interface specifies the methods of an object that are available for invocation by objects in other processes, defining the types of the input and output arguments of each of them. However, the big difference is that the methods in remote interfaces can pass objects as arguments and results of methods. In addition, references to remote objects may also be passed — these should not be confused with pointers, which refer to specific memory locations.

Neither service interfaces nor remote interfaces may specify direct access to variables. In the latter case, this prohibits direct access to the instance variables of an object.

**Interface definition languages:** An RMI mechanism can be integrated with a particular programming language if it includes an adequate notation for defining interfaces, allowing input and output parameters to be mapped onto the language's normal use of
parameters. Java RMI is an example in which an RMI mechanism has been added to an object-oriented programming language. This approach is useful when all the parts of a distributed application can be written in the same language. It is also convenient because it allows the programmer to use a single language for local and remote invocation. However, many existing useful services are written in C++ and other languages. It would be beneficial to allow programs written in a variety of languages, including Java, to access them remotely. Interface definition languages (or IDLs) are designed to allow objects implemented in different languages to invoke one another. An IDL provides a notation for defining interfaces in which each of the parameters of a method may be described as for input or output in addition to having its type specified.

```
// 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();
};
```

Figure 7.3.2 CORBA IDL example

Figure 2.3.2 shows a simple example of CORBA IDL. The interface named PersonList specifies the methods available for RMI in a remote object that implements that interface. For example, the method addPerson specifies its argument as in, meaning that it is an input argument; and the method getPerson that retrieves an instance of Person by name specifies its second argument as out, meaning that it is an output argument. Some examples include CORBA IDL as an example of an IDL for RMI; Sun XDR as an IDL.
for RPC; the interface definition language for the RPC system in the OSF's Distributed Computing Environment (DCE): which uses C language syntax and is called IDL; and DCOM IDL which is based on DCE IDL and is used in Microsoft's Distributed Common Object Model.

7.4 COMMUNICATION BETWEEN DISTRIBUTED OBJECTS

The object-based model for a distributed system extends the model supported by object-oriented programming languages to make it apply to distributed objects. Here we address communication between distributed objects by means of RMI. The material is presented under the following headings:

**The object model:** A brief review of the relevant aspects of the object model, suitable for the reader with a basic knowledge of an object-oriented programming language, for example Java or C++.

**Distributed objects:** A presentation of object-based distributed systems, which argues that the object model is very appropriate for distributed systems.

**The distributed object model:** A discussion of the extensions to the object model necessary for it to support distributed objects.

**Design issues:** A set of arguments about the design alternatives:

1. Local invocations are executed exactly once, but what suitable semantics is possible for remote invocations?
2. How can RMI semantics be made similar to those of local method invocation and what differences cannot be eliminated?

**Implementation:** An explanation as to how a layer of middleware above the request-reply protocol may be designed to support RMI between application-level distributed objects.

**Distributed garbage collection:** A presentation of an algorithm for distributed garbage collection, that is suitable for use with the RMI implementation.

**The object model**

An object-oriented program, for example in Java or C++, consists of a collection of interacting objects, each of which consists of a set of data and a set of methods. An object
communicates with other objects by invoking their methods, generally passing arguments and receiving results. Objects can encapsulate their data and the code of their methods. Some languages, for example Java and C++, allow programmers to define objects whose instance variables can be accessed directly. But for use in a distributed object system, an object's data should be accessible only via its methods.

**Object references:** Objects can be accessed via object references. For example, in Java, a variable that appears to hold an object actually holds a reference to that object. To invoke a method in an object, the object reference and method name are given, together with any necessary arguments. The object whose method is invoked is sometimes called the target and sometimes the receiver. Object references are first-class values, meaning that they may, for example, be assigned to variables, passed as arguments and returned as results of methods.

**Interfaces:** An interface provides a definition of the signatures of a set of methods (that is, the types of their arguments, return values and exceptions) without specifying their implementation. An object will provide a particular interface if its class contains code that implements the methods of that interface. In Java, a class may implement several interfaces, and the methods of an interface may be implemented by any class. An interface also defines a type that can be used to declare the type of variables or of the parameters and return values of methods. Note that interfaces do not have constructors.

**Actions:** Action in an object-oriented program is initiated by an object invoking a method in another object. An invocation can include additional information (arguments) needed to carry out the method. The receiver executes the appropriate method and then returns control to the invoking object, sometimes supplying a result. An invocation of a method can have two effects:

1. The state of the receiver may be changed, and
2. Further invocations on methods in other objects may take place.

As an invocation can lead to further invocations of methods in other objects, an action is a chain of related method invocations, each of which eventually returns. This explanation does not take account of exceptions.

**Exceptions:** Programs can encounter many sorts of errors and unexpected conditions of varying seriousness. During the execution of a method, many different problems may be
discovered: for example, inconsistent values in the object's variables, or failure in attempts to read or write to files or network sockets. When programmers need to insert tests in their code to deal with all possible unusual or erroneous cases, this detracts from the clarity of the normal case. Exceptions provide a clean way to deal with error conditions without complicating the code. In addition, each method heading explicitly lists as exceptions the error conditions it might encounter, allowing users of the method to deal with them. A block of code may be defined to throw an exception whenever particular unexpected conditions or errors arise. This means that control passes to another block of code that catches the exception. Control does not return to the place where the exception was thrown.

**Garbage collection:** It is necessary to provide a means of freeing the space occupied by objects when they are no longer needed. A language, for example Java, that can detect automatically when an object is no longer accessible recovers the space and makes it available for allocation to other objects. This process is called garbage collection. When a language (for example C++) does not support garbage collection, the programmer has to cope with the freeing of space allocated to objects. This can be a major source of errors.

**Distributed objects**

The state of an object consists of the values of its instance variables. In the object-based paradigm the state of a program is partitioned into separate parts, each of which is associated with an object. Since object-based programs are logically partitioned, the physical distribution of objects into different processes or computers in a distributed system is a natural extension. Distributed object systems may adopt the client-server architecture. In this case, objects are managed by servers and their clients invoke their methods using remote method invocation. In RMI, the client's request to invoke a method of an object is sent in a message to the server managing the object. The invocation is carried out by executing a method of the object at the server and the result is returned to the client in another message. To allow for chains of related invocations, objects in servers are allowed to become clients of objects in other servers. Distributed objects can assume the other architectural models. For example, objects can be replicated in order to obtain the usual benefits of fault tolerance and enhanced performance, and objects can be migrated with a view to enhancing their performance and availability. Having client and
server objects in different processes enforces encapsulation. That is, the state of an object can be accessed only by the methods of the object, which means that it is not possible for unauthorized methods to act on the state. For example, the possibility of concurrent RMIs from objects in different computers implies that an object may be accessed concurrently. Therefore, the possibility of conflicting accesses arises. However, the fact that the data of an object is accessed only by its own methods allows objects to provide methods for protecting themselves against incorrect accesses. For example, they may use synchronization primitives such as condition variables to protect access to their instance variables.

Another advantage of treating the shared state of a distributed program as a collection of objects is that an object may be accessed via RMI or it may be copied into a local cache and accessed directly provided that the class implementation is available locally. The fact that objects are accessed only via their methods gives another advantage for heterogeneous systems in that different data formats may be used at different sites — these formats will be unnoticed by clients that use RMI to access the methods of the objects.

**The distributed object model**

Here we discuss extensions to the object model to make it applicable to distributed objects. Each process contains a collection of objects, some of which can receive both local and remote invocations, whereas the other objects can receive only local invocations, as shown in Figure 2.3.3. Method invocations between objects in different processes, whether in the same computer or not, are known as remote method invocation. Method invocations between objects in the same process are called local method invocations.

![Figure 2.3.3 Remote and local method invocation](image)

Figure 2.3.3 Remote and local method invocation
We refer to objects that can receive remote invocations as remote objects. All objects can receive local invocations, although they can receive them only from other objects that hold references to them. The following paragraphs discuss remote object references, remote interfaces and other aspects of the distributed object model.

Remote object references: The notion of object reference is extended to allow any object that can receive an RM1 to have a remote object reference. A remote object reference is an identifier that can be used throughout a distributed system to refer to a particular unique remote object. It’s representation, which is generally different from that of local object references. Remote object references are analogous to local ones in that:

1. The remote object to receive a remote method invocation is specified as a remote object reference; and
2. Remote object references may be passed as arguments and results of remote method invocations.

Remote interfaces: The class of a remote object implements the methods of its remote interface, for example as public instance methods in Java. Objects in other processes can invoke only the methods that belong to its remote interface. Local objects can invoke the methods in the remote interface as well as other methods implemented by a remote object. Note that remote interfaces, like all interfaces, do not have constructors.

7.5 SUMMARY

In this unit you have understood definition of distributed objects and remote invocation. We also discussed about communication between distributed objects.

7.6 KEYWORDS

RMI: refer to remote method invocation, in a generic way, communication between distributed.

Middleware: It is a software that provides a programming model about the basic building blocks of processes and messages passing.

Interface: The interface of a module specifies the procedures and the variables that can be accessed from other modules.
7.7 UNIT-END EXERCISES AND ANSWERS

1. What are the familiar programming models have that been extended to apply to distributed programs?
2. Explain middleware in the context of remote invocation.
3. Write a short note on Interfaces.
4. Explain communication between distributed objects.
5. With a neat diagram explain remote and local method invocation.

Answers: SEE

1. 7.2
2. 7.3
3. 7.3
4. 7.4
5. 7.4

7.8 SUGGESTED READINGS

- DISTRIBUTED SYSTEMS Principles and Paradigm By: Andrew S. Tanenbaum, Maarten Van Steen.
## UNIT 8: REMOTE PROCEDURE CALL, EVENTS AND NOTIFICATIONS

<table>
<thead>
<tr>
<th>Structure:</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.0 Objectives</td>
</tr>
<tr>
<td>8.1 Introduction</td>
</tr>
<tr>
<td>8.2 Remote Procedure Call</td>
</tr>
<tr>
<td>8.3 Events and Notifications</td>
</tr>
<tr>
<td>8.4 Summary</td>
</tr>
<tr>
<td>8.5 Keywords</td>
</tr>
<tr>
<td>8.6 Unit-end exercises and answers</td>
</tr>
<tr>
<td>8.7 Suggested readings</td>
</tr>
</tbody>
</table>

### 8.0 OBJECTIVES

At the end of this unit you will be able to know:

- RPC mechanism
- Role of client and server stub procedures
- Binding and Authentication
- Events and their types
- The participants in distributed event notification

### 8.1 INTRODUCTION

This unit explains about the remote procedure call (RPC), events and notifications. The concept of a remote procedure call (RPC) represents a major intellectual breakthrough in distributed computing, with the goal of making the programming of distributed systems look similar, if not identical, to conventional programming – that is, achieving a high level of distribution transparency. This unification is achieved in a very simple manner, by extending the abstraction of a procedure call to distributed environments. In particular, in RPC, procedures on remote machines can be called as if they are procedures in the local address space.
8.2 REMOTE PROCEDURE CALL

The underlying RPC system hides important aspects of distribution, including the encoding and decoding of parameters and results, the passing of messages and the preserving of the required semantics for the procedure call. This concept was first introduced by Birrell and Nelson in the year 1984 and paved the way for many of the developments in distributed systems programming used today.

Design issues for RPC

Before looking at the implementation of RPC systems, we look at three issues that are important in understanding this concept:

- 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: Most modern programming languages provide a means of organizing a program as a set of modules that can communicate with one another. Communication between modules can be by means of procedure calls between modules or by direct access to the variables in another module. In order to control the possible interactions between modules, an explicit interface is defined for each module. The interface of a module specifies the procedures and the variables that can be accessed from other modules. Modules are implemented so as to hide all the information about them except that which is available through its interface. So long as its interface remains the same, the implementation may be changed without affecting the users of the module.

Interfaces in distributed systems: In a distributed program, the modules can run in separate processes. In the client-server model, in particular, each server provides a set of procedures that are available for use by clients. For example, a file server would provide procedures for reading and writing files. The term service interface is used to refer to the
specification of the procedures offered by a server, defining the types of the arguments of each of the procedures.

There are a number of benefits to programming with interfaces in distributed systems, stemming from the important separation between interface and implementation:

- As with any form of modular programming, programmers are concerned only with the abstraction offered by the service interface and need not be aware of implementation details.
- Extrapolating to (potentially heterogeneous) distributed systems, programmers also do not need to know the programming language or underlying platform used to implement the service (an important step towards managing heterogeneity in distributed systems).
- This approach provides natural support for software evolution in that implementations can change as long as long as the interface (the external view) remains the same. More correctly, the interface can also change as long as it remains compatible with the original.

The definition of service interfaces is influenced by the distributed nature of the underlying infrastructure:

- It is not possible for a client module running in one process to access the variables in a module in another process. Therefore the service interface cannot specify direct access to variables. Note that CORBA IDL interfaces can specify attributes, which seems to break this rule. However, the attributes are not accessed directly but by means of some getter and setter procedures added automatically to the interface.
- The parameter-passing mechanisms used in local procedure calls – for example, call by value and call by reference, are not suitable when the caller and procedure are in different processes. In particular, call by reference is not supported. Rather, the specification of a procedure in the interface of a module in a distributed program describes the parameters as input or output, or sometimes both. Input parameters are passed to the remote server by sending the values of the arguments
in the request message and then supplying them as arguments to the operation to be executed in the server. *Output* parameters are returned in the reply message and are used as the result of the call or to replace the values of the corresponding variables in the calling environment. When a parameter is used for both input and output, the value must be transmitted in both the request and reply messages.

- Another difference between local and remote modules is that addresses in one process are not valid in another remote one. Therefore, addresses cannot be passed as arguments or returned as results of calls to remote modules.

These constraints have a significant impact on the specification of interface definition languages, as discussed below.

**Interface definition languages:** An RPC mechanism can be integrated with a particular programming language if it includes an adequate notation for defining interfaces, allowing input and output parameters to be mapped onto the language’s normal use of parameters. This approach is useful when all the parts of a distributed application can be written in the same language. It is also convenient because it allows the programmer to use a single language, for example, Java, for local and remote invocation. However, many existing useful services are written in C++ and other languages. It would be beneficial to allow programs written in a variety of languages, including Java, to access them remotely. *Interface definition languages* (IDLs) are designed to allow procedures implemented in different languages to invoke one another. An 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.
// 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();
};

Figure 8.4.1: CORBA IDL example

Figure 2.4.1 shows a simple example of CORBA IDL. The Person structure is the same as the one used to illustrate marshalling. The interface named PersonList specifies the methods available for RMI in a remote object that implements that interface. For example, the method addPerson specifies its argument as in, meaning that it is an input argument, and the method getPerson that retrieves an instance of Person by name specifies its second argument as out, meaning that it is an output argument.

The concept of an IDL was initially developed for RPC systems but applies equally to RMI and also web services. Our case studies include:

- Sun XDR as an example of an IDL for RPC;
- CORBA IDL as an example of an IDL for RMI;
- The Web Services Description Language (WSDL), which is designed for an Internet-wide RPC supporting web services;
- And protocol buffers used at Google for storing and interchanging many kinds of structured information.
**RPC call semantics:** In Request-reply protocols, *doOperation* can be implemented in different ways to provide different delivery guarantees. The main choices are:

*Retry request message:* Controls whether to retransmit the request message until either a reply is received or the server is assumed to have failed.

*Duplicate filtering:* Controls when retransmissions are used and whether to filter out duplicate requests at the server.

*Retransmission of results:* Controls whether to keep a history of result messages to enable lost results to be retransmitted without re-executing the operations at the server.

![Figure 8.4.2: Role of client and server stub procedures in RPC](image)

Combinations of these choices lead to a variety of possible semantics for the reliability of remote invocations as seen by the invoker. Figure 2.4.2 shows the choices of interest, with corresponding names for the semantics that they produce. Note that for local procedure calls, the semantics are *exactly once*, meaning that every procedure is executed exactly once (except in the case of process failure). The choices of RPC invocation semantics are defined as follows.

**Maybe semantics:** With *maybe* semantics, the remote procedure call may be executed once or not at all. Maybe semantics arises when no fault-tolerance measures are applied and can suffer from the following types of failure:
• Omission failures if the request or result message is lost.
• Crash failures when the server containing the remote operation fails.

If the result message has not been received after a timeout and there are no retries, it is uncertain whether the procedure has been executed. If the request message was lost, then the procedure will not have been executed. On the other hand, the procedure may have been executed and the result message lost. A crash failure may occur either before or after the procedure is executed. Moreover, in an asynchronous system, the result of executing the procedure may arrive after the timeout. *Maybe* semantics is useful only for applications in which occasional failed calls are acceptable.

**At-least-once semantics:** With *at-least-once* semantics, the invoker receives either a result, in which case the invoker knows that the procedure was executed at least once, or an exception informing it that no result was received. *At-least-once* semantics can be achieved by the retransmission of request messages, which masks the omission failures of the request or result message. *At-least-once* semantics can suffer from the following types of failure:

• Crash failures when the server containing the remote procedure fails.
• Arbitrary failures – in cases when the request message is retransmitted, the remote server may receive it and execute the procedure more than once, possibly causing wrong values to be stored or returned.

An *idempotent operation is defined* as one that can be performed repeatedly with the same effect as if it had been performed exactly once. Non-idempotent operations can have the wrong effect if they are performed more than once. For example, an operation to increase a bank balance by $10 should be performed only once; if it were to be repeated, the balance would grow and grow! If the operations in a server can be designed so that all of the procedures in their service interfaces are idempotent operations, then *at-least-once* call semantics may be acceptable.
**At-most-once semantics:** With *at-most-once* semantics, the caller receives either a result, in which case the caller knows that the procedure was executed exactly once, or an exception informing it that no result was received, in which case the procedure will have been executed either once or not at all. *At-most-once* semantics can be achieved by using all of the fault-tolerance measures outlined in Figure 2.4.3. As in the previous case, the use of retries masks any omission failures of the request or result messages. This set of fault tolerance measures prevents arbitrary failures by ensuring that for each RPC a procedure is never executed more than once. Sun RPC provides at-least-once call semantics.

---

<table>
<thead>
<tr>
<th>Fault tolerance measures</th>
<th>Call semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Retransmit request message</td>
<td></td>
</tr>
<tr>
<td>Duplicate filtering</td>
<td></td>
</tr>
<tr>
<td>Re-execute procedure or retransmit reply</td>
<td></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>

**Figure 8.4.3: Call Semantics**

**Transparency** • The originators of RPC, Birrell and Nelson, aimed to make remote procedure calls as much like local procedure calls as possible, with no distinction in syntax between a local and a remote procedure call. All the necessary calls to marshalling and message-passing procedures were hidden from the programmer making the call. Although request messages are retransmitted after a timeout, this is transparent to the caller to make the semantics of remote procedure calls like that of local procedure calls.
More precisely, RPC strives to offer at least location and access transparency, hiding the physical location of the (potentially remote) procedure and also accessing local and remote procedures in the same way. Middleware can also offer additional levels of transparency to RPC.

However, remote procedure calls are more vulnerable to failure than local ones, since they involve a network, another computer and another process. Whichever of the above semantics is chosen, there is always the chance that no result will be received, and in the case of failure, it is impossible to distinguish between failure of the network and of the remote server process. This requires that clients making remote calls are able to recover from such situations.

The latency of a remote procedure call is several orders of magnitude greater than that of a local one. This suggests that programs that make use of remote calls need to be able to take this factor into account, perhaps by minimizing remote interactions. The designers of Argus suggested that a caller should be able to abort a remote procedure call that is taking too long in such a way that it has no effect on the server. To allow this, the server would need to be able to restore things to how they were before the procedure was called.

Remote procedure calls also require a different style of parameter passing, as discussed above. In particular, RPC does not offer call by reference.

Waldo et al say that the difference between local and remote operations should be expressed at the service interface, to allow participants to react in a consistent way to possible partial failures. Other systems went further than this by arguing that the syntax of a remote call should be different from that of a local call: in the case of Argus, the language was extended to make remote operations explicit to the programmer.

The choice as to whether RPC should be transparent is also available to the designers of IDLs. For example, in some IDLs, a remote invocation may throw an exception when the client is unable to communicate with a remote procedure. This requires that the client
program handle such exceptions, allowing it to deal with such failures. An IDL can also provide a facility for specifying the call semantics of a procedure. This can help the designer of the service – for example, if at-least-once call semantics is chosen to avoid the overheads of at-most-once, the operations must be designed to be idempotent.

The current 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.

**Implementation of RPC**

![RPC implementation diagram](image)

**Figure 8.4.4: RPC implementation**

The software components required to implement RPC are shown in Figure 2.4.4. The client that accesses a service includes one *stub procedure* for each procedure in the service interface. The 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. When the reply message arrives, it unmarshals the results. The server process contains a dispatcher together with one server stub procedure and one service procedure for each procedure in
the service interface. The dispatcher selects one of the server stub procedures according to the procedure identifier in the request message. The server stub procedure then unmarshals the arguments in the request message, calls the corresponding service procedure and marshals the return values for the reply message. The service procedures implement the procedures in the service interface. The client and server stub procedures and the dispatcher can be generated automatically by an interface compiler from the interface definition of the service.

RPC is generally implemented over a request-reply protocol. The contents of request and reply messages are the same as those illustrated for request-reply protocols. RPC may be implemented to have one of the choices of invocation semantics – at-least-once or at-most-once is generally chosen. To achieve this, the communication module will implement the desired design choices in terms of retransmission of requests, dealing with duplicates and retransmission of results.

8.3 EVENTS AND NOTIFICATIONS

The idea behind the use of events is that one object can react to a change occurring in another object. The actions done by the user are seen as events that cause state changes in objects. The objects are notified whenever the state changes. Local event model can be extended to distributed event-based systems by using the publish-subscribe paradigm. Distributed event-based systems extend the local event model by allowing multiple objects at different locations to be notified of events taking place at an object. They use the publish-subscribe paradigm, in which an object that generates events publishes the type of events that it will make available for observation by other objects. Objects that want to receive notifications from an object that has published its events subscribe to the types of events that are of interest to them. Different event types may, for example, refer to the different methods executed by the object of interest. Objects that represent events are called notifications. Notifications may be stored, sent in messages, queried and applied in a variety of orders to different things. When a publisher experiences an event, subscribers that expressed an interest in that type of event will receive notifications.
Subscribing to a particular type of event is also called registering interest in that type of event.

Events and notifications can be used in a wide variety of different applications, for example to communicate a shape added to a drawing, a modification to a document, the fact that a person has entered or left a room, or that a piece of equipment or an electronically tagged book is at a new location. The latter two examples are made possible with the use of active badges or embedded devices.

Distributed event-based systems have two main characteristics:

**Heterogeneous:** When event notifications are used as a means of communication between distributed objects, components in a distributed system that were not designed to interoperate can be made to work together. All that is required is that event-generating objects publish the types of events they offer, and that other objects subscribe to events and provide an interface for receiving notifications. For example, Bates et al. describe how event-based systems can be used to connect heterogeneous components in the Internet. They describe a system in which applications can be made aware of users' locations and activities, such as using computers, printers or electronically tagged books. They envisage its future use in the context of a home network with commands such as: 'if the children come home, turn on the central heating'.

**Asynchronous:** Notifications are sent asynchronously by event-generating objects to all the objects that have subscribed to them to prevent publishers needing to synchronize with subscribers — publishers and subscribers need to be decoupled. Mushroom is a distributed event-based system designed to support collaborative work, in which the user interface displays objects representing users and information objects such as documents and notepads within shared workspaces called network places. The state of each place is replicated at the computers of users currently in that place. Events are used to describe changes to objects and to a user's focus of interest. For example, an event could specify that a particular user has entered or left a place or has performed a particular action on an object. Each replica of any object to which particular types of events are relevant subscribes to them and receives notifications when they occur. But subscribers are decoupled from objects experiencing events, because different users are active at different times.
A situation in which events can be useful is illustrated in the following dealing room example:

**Simple dealing room system:** Consider a simple dealing room system whose task is to allow dealers using computers to see the latest information about the market prices of the stocks they deal in. The market price for a single named stock is represented by an object with several instance variables. The information arrives in the dealing room from several different external sources in the form of updates to some or all of the instance variables of the objects representing the stocks and is collected by processes we call information providers. Dealers are typically interested only in their specialist stocks. A dealing room system could be modeled by processes with two different tasks:

- An information provider process continuously receives new trading information from a single external source and applies it to the appropriate stock objects. Each of the updates to a stock object is regarded as an event. The stock object experiencing such events notifies all of the dealers who have subscribed to the corresponding stock. There will be a separate information provider process for each external source.

- A dealer process creates an object to represent each named stock that the user asks to have displayed. This local object subscribes to the object representing that stock at the relevant information provider. It then receives all the information sent to it in notifications and displays it to the user.
The communication of notifications is shown in Figure 2.4.5.

**Event types**: An event source can generate events of one or more different types. Each event has attributes that specify information about that event, such as the name or identifier of the object that generated it, the operation, its parameters and the time (or a sequence number). Types and attributes are used both in subscribing to events and in notifications. When subscribing to an event, the type of event is specified, sometimes modified with a criterion as to the values of the attributes. Whenever an event of that type occurs that matches the attributes, the interested parties will be notified. In the dealing room example, there is one type of event (the arrival of an update to a stock), and the attributes might specify the name of a stock, its current price, and latest rise or fall. Dealers may for example specify that they are interested in all the events relating to a stock with a particular name.

### 8.4 SUMMARY

In this unit we studied RPC mechanism, role of client and server stub procedures, binding and authentication, and events and their types.
8.7. **KEYWORDS**

RPC: remote procedure call

8.8. **UNIT-END EXERCISES AND ANSWERS**

1. What is RPC?
2. What are the design issues for RPC? Explain
3. Explain about Interface definition languages.
4. What are the two main characteristics of distributed event-based systems? Explain.
5. In the context of events, explain dealing room system.
6. Explain the choices of RPC invocation semantics.

**Answers: SEE**

1. 8.1
2. 8.2
3. 8.2
4. 8.3
5. 8.3
6. 8.3

8.9 **SUGGESTED READINGS**

- DISTRIBUTED SYSTEMS Principles and Paradigm By: Andrew S. Tanenbaum, Maarten Van Steen.
UNIT 9: OPERATING SYSTEM

Structure:
9.0 Objectives
9.1 Introduction
9.2 Operating system layer
9.3 Protection
9.4 Processes threads
9.5 Summary
9.6 Keywords
9.7 Unit-end exercises and answers
9.8 Suggested readings

9.0 OBJECTIVES

At the end of this unit you will be able to know:

- Operating system in the context of distributed systems.
- Protection
- Process
- Threads

9.1 INTRODUCTION

This unit describes how middleware is supported by the operating system facilities at the nodes of a distributed system. The operating system facilitates the encapsulation and protection of resources inside servers and it supports the mechanisms required to access these resources, including communication and scheduling.
We have learned that an important aspect of distributed systems is resource sharing. Client applications invoke operations on resources that are often on another node or at least in another process. Applications (in the form of clients) and services (in the form of resource managers) use the middleware layer for their interactions. Middleware enables remote communication between objects or processes at the nodes of a distributed system.

Below the middleware layer is the operating system (OS) layer. Here we examine the relationship between the two, and in particular how well the requirements of middleware can be met by the operating system. Those requirements include efficient and robust access to physical resources, and the flexibility to implement a variety of resource-management policies.

The task of any operating system is to provide problem-oriented abstractions of the underlying physical resources – the processors, memory, networks, and storage media. An operating system such as UNIX (and its variants, such as Linux and Mac OS X) or Windows (and its variants, such as XP, Vista and Windows 7) provides the programmer with, for example, files rather than disk blocks, and with sockets rather than raw network access. It takes over the physical resources on a single node and manages them to present these resource abstractions through the system-call interface.

The two operating system concepts that have come about during the development of distributed systems: network operating systems and distributed operating systems. Definitions vary, but the concepts behind them are something like the following.

Both UNIX and Windows are examples of network operating systems. They have a networking capability built into them and so can be used to access remote resources. Access is network-transparent for some – not all – types of resource. For example, through a distributed file system such as NFS, users have network-transparent access to files. That is, many of the files that users access are stored remotely, on a server, and this is largely transparent to their applications.
But the defining characteristic is that the nodes running a network operating system retain autonomy in managing their own processing resources. In other words, there are multiple system images, one per node. With a network operating system, a user can remotely log into another computer, using `ssh`, for example, and run processes there. However, while the operating system manages the processes running at its own node, it does not manage processes across the nodes.

By contrast, one could envisage an operating system in which users are never concerned with where their programs run, or the location of any resources. There is a *single system image*. The operating system has control over all the nodes in the system, and it transparently locates new processes at whatever node suits its scheduling policies. For example, it could create a new process at the least-loaded node in the system, to prevent individual nodes becoming unfairly overloaded.

An operating system that produces a single system image like this for all the resources in a distributed system is called a *distributed operating system*.

**Middleware and network operating systems:** In fact, there are no distributed operating systems in general use, only network operating systems such as UNIX, Mac OS and Windows. This is likely to remain the case, for two main reasons. The first is that users have much invested in their application software, which often meets their current problem-solving needs; they will not adopt a new operating system that will not run their applications, whatever efficiency advantages it offers. Attempts have been made to emulate UNIX and other operating system kernels on top of new kernels, but the emulations’ performance has not been satisfactory. Anyway, keeping emulations of all the major operating systems up-to-date as they evolve would be a huge undertaking.

The second reason against the adoption of distributed operating systems is that users tend to prefer to have a degree of autonomy for their machines, even in a closely knit organization. This is particularly so because of performance. For example, Jones needs
good interactive responsiveness while she writes her documents and would resent it if Smith’s programs were slowing her down.

The combination of middleware and network operating systems provides an acceptable balance between the requirement for autonomy on the one hand and network transparent resource access on the other. The network operating system enables users to run their favourite word processors and other standalone applications. Middleware enables them to take advantage of services that become available in their distributed system.

### 9.2 OPERATING SYSTEM LAYER

Users will only be satisfied if their middleware–OS combination has good performance. Middleware runs on a variety of OS–hardware combinations (platforms) at the nodes of a distributed system. The OS running at a node – a kernel and associated user-level services such as communication libraries – provides its own flavour of abstractions of local hardware resources for processing, storage and communication. Middleware utilizes a combination of these local resources to implement its mechanisms for remote invocations between objects or processes at the nodes.

Figure 3.1.1 shows how the operating system layer at each of two nodes supports a common middleware layer in providing a distributed infrastructure for applications and services.

![Figure 9.1.1: System layers](image-url)

133
Our goal in this unit is to examine the impact of particular OS mechanisms on middleware’s ability to deliver distributed resource sharing to users. Kernels and the client and server processes that execute upon them are the chief architectural components that concern us. Kernels and server processes are the components that manage resources and present clients with an interface to the resources. As such, we require at least the following of them:

**Encapsulation:** They should provide a useful service interface to their resources – that is, a set of operations that meet their clients’ needs. Details such as management of memory and devices used to implement resources should be hidden from clients.

**Protection:** Resources require protection from illegitimate accesses – for example, files are protected from being read by users without read permissions, and device registers are protected from application processes.

**Concurrent processing:** Clients may share resources and access them concurrently. Resource managers are responsible for achieving concurrency transparency.

Clients access resources by making, for example, remote method invocations to a server object, or system calls to a kernel. We call a means of accessing an encapsulated resource an *invocation mechanism*, however it is implemented. A combination of libraries, kernels and servers may be called upon to perform the following invocation related tasks:

**Communication:** Operation parameters and results have to be passed to and from resource managers, over a network or within a computer. **Scheduling:** When an operation is invoked, its processing must be scheduled within the kernel or server.

Figure 9.1.1 shows the core OS functionality that we shall be concerned with: process and thread management, memory management and communication between processes on the same computer (horizontal divisions in the figure denote dependencies). The kernel supplies much of this functionality – all of it in the case of some operating systems.
OS software is designed to be portable between computer architectures where possible. This means that the majority of it is coded in a high-level language such as C, C++ or Modula-3, and that its facilities are layered so that machine-dependent components are reduced to a minimal bottom layer. Some kernels can execute on shared-memory multiprocessors, which are described below.

**Shared-memory multiprocessors:** Shared-memory multiprocessor computers are equipped with several processors that share one or more modules of memory (RAM). The processors may also have their own private memory. Multiprocessor computers can be constructed in a variety of forms. The simplest and least expensive multiprocessors are constructed by incorporating a circuit board holding a few (2–8) processors in a personal computer.

In the common *symmetric processing architecture*, each processor executes the same kernel and the kernels play largely equivalent roles in managing the hardware resources. The kernels share key data structures, such as the queue of runnable threads, but some of their working data is private. Each processor can execute a thread simultaneously, accessing data in the shared memory, which may be private (hardware-protected) or shared with other threads.

Multiprocessors can be used for many high-performance computing tasks. In distributed systems, they are particularly useful for the implementation of high performance servers, because the server can run a single program with several threads that handle several requests from clients simultaneously – for example, providing access to a shared database.

**The core OS components and their responsibilities are:**

**Process manager:** Creation of and operations upon processes. A process is a unit of resource management, including an address space and one or more threads.
Thread manager: Thread creation, synchronization and scheduling. Threads are schedulable activities attached to processes.

Communication manager: Communication between threads attached to different processes on the same computer. Some kernels also support communication between threads in remote processes. Other kernels have no notion of other computers built into them, and an additional service is required for external communication.

Memory manager: Management of physical and virtual memory.

Supervisor: Dispatching of interrupts, system call traps and other exceptions; control of memory management unit and hardware caches; processor and floating-point unit register manipulations. This is known as the Hardware Abstraction Layer in Windows.

9.3 PROTECTION

Note that the resources require protection from illegitimate accesses. However, threats to a system’s integrity do not come only from maliciously contrived code. Benign code that contains a bug or that has unanticipated behaviour may cause part of the rest of the system to behave incorrectly.

To understand what we mean by an ‘illegitimate access’ to a resource, consider a file. Let us suppose, for the sake of explanation, that open files have only two operations, read and write. Protecting the file consists of two sub-problems. The first is to ensure that each of the file’s two operations can be performed only by clients with the right to perform it. For example, Smith, who owns the file, has read and write rights to it. Jones may only perform the read operation. An illegitimate access here would be if Jones somehow managed to perform a write operation on the file. A complete solution to this resource-protection sub-problem in a distributed system requires cryptographic techniques.
The other type of illegitimate access, which we address here, is where a misbehaving client sidesteps the operations that a resource exports. In our example, this would be if Smith or Jones somehow managed to execute an operation that was neither read nor write. Suppose, for example, that Smith managed to access the file pointer variable directly. She could then construct a `setFilePointerRandomly` operation, that sets the file pointer to a random number. Of course, this is a meaningless operation that would upset normal use of the file.

We can protect resources from illegitimate invocations such as `setFilePointerRandomly`. One way is to use a type-safe programming language, such as Sing#, an extension of C# used in the Singularity project or Modula-3. In type-safe languages, no module may access a target module unless it has a reference to it – it cannot make up a pointer to it, as would be possible in C or C++ and it may only use its reference to the target module to perform the invocations (method calls or procedure calls) that the programmer of the target made available to it. It may not, in other words, arbitrarily change the target’s variables. By contrast, in C++ the programmer may cast a pointer however she likes, and thus perform non-type-safe invocations.

We can also employ hardware support to protect modules from one another at the level of individual invocations, regardless of the language in which they are written. To operate this scheme on a general-purpose computer, we require a kernel.

**Kernels and protection:** The kernel is a program that is distinguished by the facts that it remains loaded from system initialization and its code is executed with complete access privileges for the physical resources on its host computer. In particular, it can control the memory management unit and set the processor registers so that no other code may access the machine’s physical resources except in acceptable ways.

Most processors have a hardware mode register whose setting determines whether privileged instructions can be executed, such as those used to determine which protection tables are currently employed by the memory management unit. A kernel process
executes with the processor in *supervisor* (privileged) mode; the kernel arranges that other processes execute in *user* (unprivileged) mode.

The kernel also sets up *address spaces* to protect itself and other processes from the accesses of an aberrant process, and to provide processes with their required virtual memory layout. An address space is a collection of ranges of virtual memory locations, in each of which a specified combination of memory access rights applies, such as readonly or read-write. A process cannot access memory outside its address space. The terms *user process* or *user-level process* are normally used to describe one that executes in user mode and has a user-level address space (that is, one with restricted memory access rights compared with the kernel’s address space).

When a process executes application code, it executes in a distinct user-level address space for that application; when the same process executes kernel code, it executes in the kernel’s address space. The process can safely transfer from a user-level address space to the kernel’s address space via an exception such as an interrupt or a *system call trap* – the invocation mechanism for resources managed by the kernel. A system call trap is implemented by a machine-level *TRAP* instruction, which puts the processor into supervisor mode and switches to the kernel address space. When the *TRAP* instruction is executed (as with any type of exception) the hardware forces the processor to execute a kernel-supplied handler function, in order that no process may gain illicit control of the hardware.

Programs pay a price for protection. Switching between address spaces may take many processor cycles, and a system call trap is a more expensive operation than a simple procedure or method call.

### 9.4 PROCESSES AND THREADS

The traditional operating system notion of a process that executes a single activity was found in the 1980s to be unequal to the requirements of distributed systems and also to those of more sophisticated single-computer applications that require internal
concurrency. The problem is that the traditional process makes sharing between related activities awkward and expensive.

The solution reached was to enhance the notion of a process so that it could be associated with multiple activities. Nowadays, a process consists of an execution environment together with one or more threads. A thread is the operating system abstraction of an activity (the term derives from the phrase ‘thread of execution’). An execution environment is the unit of resource management: a collection of local kernel managed resources to which its threads have access. An execution environment primarily consists of:

- An address space;
- Thread synchronization and communication resources such as semaphores and communication interfaces (for example, sockets);
- Higher-level resources such as open files and windows.

Execution environments are normally expensive to create and manage, but several threads can share them – that is, they can share all resources accessible within them. In other words, an execution environment represents the protection domain in which its threads execute.

Threads can be created and destroyed dynamically, as needed. The central aim of having multiple threads of execution is to maximize the degree of concurrent execution between operations, thus enabling the overlap of computation with input and output, and enabling concurrent processing on multiprocessors. This can be particularly helpful within servers, where concurrent processing of clients’ requests can reduce the tendency for servers to become bottlenecks. For example, one thread can process a client’s request while a second thread servicing another request waits for a disk access to complete.

An execution environment provides protection from threads outside it, so that the data and other resources contained in it are by default inaccessible to threads residing in other
execution environments. But certain kernels allow the controlled sharing of resources such as physical memory between execution environments residing at the same computer.

**An analogy for threads and processes:** The following memorable, if slightly unsavoury, way to think of the concepts of threads and execution environments was published on the *comp.os.mach* USENET group and is by Chris Lloyd. An execution environment consists of a stoppered jar and the air and food within it. Initially, there is one fly – a thread – in the jar. This fly can produce other flies and kill them, as can its progeny. Any fly can consume any resource (air or food) in the jar. Flies can be programmed to queue up in an orderly manner to consume resources. If they lack this discipline, they might bump into one another within the jar – that is, collide and produce unpredictable results when attempting to consume the same resources in an unconstrained manner. Flies can communicate with (send messages to) flies in other jars, but none may escape from the jar, and no fly from outside may enter it. In this view, originally a UNIX process was a single jar with a single sterile fly within it.

![Figure 3.1.2: Address space](image)

As many older operating systems allow only one thread per process, we shall sometimes use the term *multi-threaded process* for emphasis. Confusingly, in some programming
models and operating system designs the term ‘process’ means what we have called a thread. The reader may encounter in the literature the terms heavy weight process, where an execution environment is taken to be included and light weight process, where it is not.

**Address spaces**

An address space is a unit of management of a process’s virtual memory. It is large (typically up to $2^{32}$ bytes, and sometimes up to $2^{64}$ bytes) and consists of one or more regions, separated by inaccessible areas of virtual memory. A region is an area of contiguous virtual memory that is accessible by the threads of the owning process. Regions do not overlap. Note that we distinguish between the regions and their contents. Each region is specified by the following properties:

- Its extent (lowest virtual address and size);
- Read/write/execute permissions for the process’s threads;
- Whether it can be grown upwards or downwards.

Note that this model is page-oriented rather than segment-oriented. Regions, unlike segments, would eventually overlap if they were extended in size. Gaps are left between regions to allow for growth. This representation of an address space as a sparse set of disjoint regions is a generalization of the UNIX address space, which has three regions: a fixed, unmodifiable text region containing program code; a heap, part of which is initialized by values stored in the program’s binary file, and which is extensible towards higher virtual addresses; and a stack, which is extensible towards lower virtual addresses.

The provision of an indefinite number of regions is motivated by several factors. One of these is the need to support a separate stack for each thread. Allocating a separate stack region to each thread makes it possible to detect attempts to exceed the stack limits and to control each stack’s growth. Unallocated virtual memory lies beyond each stack region, and attempts to access this will cause an exception (a page fault). The alternative is to allocate stacks for threads on the heap, but then it is difficult to detect when a thread has exceeded its stack limit.
Another motivation is to enable files in general – not just the text and data sections of binary files – to be mapped into the address space. A *mapped file* is one that is accessed as an array of bytes in memory. The virtual memory system ensures that accesses made in memory are reflected in the underlying file storage. Section CDK3-18.6 (in www.cdk5.net/oss/mach) describes how the Mach kernel extends the abstraction of virtual memory so that regions can correspond to arbitrary ‘memory objects’ and not just to files.

The need to share memory between processes, or between processes and the kernel, is another factor leading to extra regions in the address space. A *shared memory region* (or *shared region* for short) is one that is backed by the same physical memory as one or more regions belonging to other address spaces. Processes therefore access identical memory contents in the regions that are shared, while their non-shared regions remain protected. The uses of shared regions include the following:

**Libraries:** Library code can be very large and would waste considerable memory if it was loaded separately into every process that used it. Instead, a single copy of the library code can be shared by being mapped as a region in the address spaces of processes that require it.

**Kernel:** Often the kernel code and data are mapped into every address space at the same location. When a process makes a system call or an exception occurs, there is no need to switch to a new set of address mappings.

**Data sharing and communication:** Two processes, or a process and the kernel, might need to share data in order to cooperate on some task. It can be considerably more efficient for the data to be shared by being mapped as regions in both address spaces than by being passed in messages between them.
**Creation of a new process**

The creation of a new process has traditionally been an indivisible operation provided by the operating system. For example, the UNIX `fork` system call creates a process with an execution environment copied from the caller (except for the return value from `fork`). The UNIX `exec` system call transforms the calling process into one executing the code of a named program.

For a distributed system, the design of the process-creation mechanism has to take into account the utilization of multiple computers; consequently, the process-support infrastructure is divided into separate system services.

The creation of a new process can be separated into two independent aspects:

- The choice of a target host, for example, the host may be chosen from among the nodes in a cluster of computers acting as a compute server;
- The creation of an execution environment (and an initial thread within it).

**Choice of process host:** The choice of the node at which the new process will reside – the process allocation decision – is a matter of policy. In general, process allocation policies range from always running new processes at their originator’s workstation to sharing the processing load between a set of computers. Eager *et al.* [1986] distinguish two policy categories for load sharing.

The *transfer policy* determines whether to situate a new process locally or remotely. This may depend, for example, on whether the local node is lightly or heavily loaded.

The *location policy* determines which node should host a new process selected for transfer. This decision may depend on the relative loads of nodes, on their machine architectures or on any specialized resources they may possess. The V system and Sprite both provide commands for users to execute a program at a currently idle workstation (there are often many of these at any given time) chosen by the operating system. In the Amoeba system, the *run server* chooses a host for each process from a shared pool of
processors. In all cases, the choice of target host is transparent to the programmer and the user. Those programming for explicit parallelism or fault tolerance, however, may require a means of specifying process location.

Process location policies may be static or adaptive. The former operate without regard to the current state of the system, although they are designed according to the system’s expected long-term characteristics. They are based on a mathematical analysis aimed at optimizing a parameter such as the overall process throughput. They may be deterministic (‘node A should always transfer processes to node B’) or probabilistic (‘node A should transfer processes to any of nodes B–E at random’). Adaptive policies, on the other hand, apply heuristics to make their allocation decisions, based on unpredictable runtime factors such as a measure of the load on each node.

Load-sharing systems may be centralized, hierarchical or decentralized. In the first case there is one load manager component, and in the second there are several, organized in a tree structure. Load managers collect information about the nodes and use it to allocate new processes to nodes. In hierarchical systems, managers make process allocation decisions as far down the tree as possible, but managers may transfer processes to one another, via a common ancestor, under certain load conditions. In a decentralized load-sharing system, nodes exchange information with one another directly to make allocation decisions. The Spawn system, for example, considers nodes to be ‘buyers’ and ‘sellers’ of computational resources and arranges them in a (decentralized) ‘market economy’.

In sender-initiated load-sharing algorithms, the node that requires a new process to be created is responsible for initiating the transfer decision. It typically initiates a transfer when its own load crosses a threshold. By contrast, in receiver-initiated algorithms, a node whose load is below a given threshold advertises its existence to other nodes so that relatively loaded nodes can transfer work to it.

Migratory load-sharing systems can shift load at any time, not just when a new process is created. They use a mechanism called process migration: the transfer of an executing
process from one node to another. Milojicic *et al.* provide a collection of papers on process migration and other types of mobility. While several process migration mechanisms have been constructed, they have not been widely deployed. This is largely because of their expense and the tremendous difficulty of extracting the state of a process that lies within the kernel, in order to move it to another node.

**Creation of a new execution environment:** Once the host computer has been selected, a new process requires an execution environment consisting of an address space with initialized contents (and perhaps other resources, such as default open files).

There are two approaches to defining and initializing the address space of a newly created process. The first approach is used where the address space is of a statically defined format. For example, it could contain just a program text region, heap region and stack region. In this case, the address space regions are created from a list specifying their extent. Address space regions are initialized from an executable file or filled with zeros as appropriate.

Alternatively, the address space can be defined with respect to an existing execution environment. In the case of UNIX *fork* semantics, for example, the newly created child process physically shares the parent’s text region and has heap and stack regions that are copies of the parent’s in extent (as well as in initial contents). This scheme has been generalized so that each region of the parent process may be inherited by (or omitted from) the child process. An inherited region may either be shared with or logically copied from the parent’s region. When parent and child share a region, the page frames (units of physical memory corresponding to virtual memory pages) belonging to the parent’s region are mapped simultaneously into the corresponding child region.
Copy-on-write is a general technique – for example, it is also used in copying large messages – so we take some time to explain its operation here. Let us follow through an example of regions $RA$ and $RB$, whose memory is shared copy-on-write between two processes, $A$ and $B$ (Figure 3.1.3). For the sake of definiteness, let us assume that process $A$ set region $RA$ to be copy-inherited by its child, process $B$, and that the region $RB$ was thus created in process $B$.

We assume, for the sake of simplicity, that the pages belonging to region $A$ are resident in memory. Initially, all page frames associated with the regions are shared between the two processes’ page tables. The pages are initially write-protected at the hardware level, even though they may belong to regions that are logically writable. If a thread in either process attempts to modify the data, a hardware exception called a page fault is taken. Let us say that process $B$ attempted the write. The page fault handler allocates a new frame for process $B$ and copies the original frame’s data into it byte for byte. The old frame number is replaced by the new frame number in one process’s page table – it does not matter which – and the old frame number is left in the other page table. The two corresponding
pages in processes $A$ and $B$ are then each made writable once more at the hardware level. After all of this has taken place, process $B$’s modifying instruction is allowed to proceed.

![Diagram of a client and server with threads]

**Figure 9.1.4: Client and server with threads**

**Threads**

The next key aspect of a process to consider in more detail is its threads. This section examines the advantages of enabling client and server processes to possess more than one thread. It then discusses programming with threads, using Java threads as a case study, and ends with alternative designs for implementing threads.

Consider the server shown in Figure 3.1.4. The server has a pool of one or more threads, each of which repeatedly removes a request from a queue of received requests and processes it. We shall not concern ourselves for the moment with how the requests are received and queued up for the threads. Also, for the sake of simplicity, we assume that each thread applies the same procedure to process the requests. Let us assume that each request takes, on average, 2 milliseconds of processing plus 8 milliseconds of I/O (input/output) delay when the server reads from a disk (there is no caching). Let us further assume for the moment that the server executes at a single-processor computer.
Consider the maximum server throughput, measured in client requests handled per second, for different numbers of threads. If a single thread has to perform all processing, then the turnaround time for handling any request is on average $2 + 8 = 10$ milliseconds, so this server can handle 100 client requests per second. Any new request messages that arrive while the server is handling a request are queued at the server port.

Now consider what happens if the server pool contains two threads. We assume that threads are independently schedulable – that is, one thread can be scheduled when another becomes blocked for I/O. Then thread number two can process a second request while thread number one is blocked, and vice versa. This increases the server throughput. Unfortunately, in our example, the threads may become blocked behind the single disk drive. If all disk requests are serialized and take 8 milliseconds each, then the maximum throughput is $1000/8 = 125$ requests per second.

The throughput can be increased by using a shared-memory multiprocessor to ease the processor bottleneck. A multi-threaded process maps naturally onto a shared memory multiprocessor. The shared execution environment can be implemented in shared memory, and the multiple threads can be scheduled to run on the multiple processors. Consider now the case in which our example server executes at a multiprocessor with two processors. Given that threads can be independently scheduled to the different processors, then up to two threads can process requests in parallel. The reader should check that two threads can process 444 requests per second and three or more threads, bounded by the I/O time, can process 500 requests per second.

**Architectures for multi-threaded servers:** We have described how multi-threading enables servers to maximize their throughput, measured as the number of requests processed per second. To describe the various ways of mapping requests to threads within a server we summarize the account by Schmidt, who describes the threading architectures of various implementations of the CORBA Object Request Broker (ORB). ORB’s process requests that arrive over a set of connected sockets. Their threading architectures are relevant to many types of server, regardless of whether CORBA is used.
Figure 9.1.4 shows one of the possible threading architectures, the *worker pool architecture*. In its simplest form, the server creates a fixed pool of ‘worker’ threads to process the requests when it starts up. The module marked ‘receipt and queuing’ in Figure 3.1.4 is typically implemented by an ‘I/O’ thread, which receives requests from a collection of sockets or ports and places them on a shared request queue for retrieval by the workers.

There is sometimes a requirement to treat the requests with varying priorities. For example, a corporate web server could prioritize request processing according to the class of customer from which the request derives. We may handle varying request priorities by introducing multiple queues into the worker pool architecture, so that the worker threads scan the queues in the order of decreasing priority. A disadvantage of this architecture is its inflexibility: as we saw with our worked-out example, the number of worker threads in the pool may be too few to deal adequately with the current rate of request arrival. Another disadvantage is the high level of switching between the I/O and worker threads as they manipulate the shared queue.

![Diagram of alternative server threading architectures](image)

Figure 9.1.5: Alternative server threading architectures

In the *thread-per-request architecture* (Figure 3.1.5a) the I/O thread spawns a new worker thread for each request, and that worker destroys itself when it has processed the
request against its designated remote object. This architecture has the advantage that the threads do not contend for a shared queue, and throughput is potentially maximized because the I/O thread can create as many workers as there are outstanding requests. Its disadvantage is the overhead of the thread creation and destruction operations.

The thread-per-connection architecture (Figure 9.1.5b) associates a thread with each connection. The server creates a new worker thread when a client makes a connection and destroys the thread when the client closes the connection. In between, the client may make many requests over the connection, targeted at one or more remote objects. The thread-per-object architecture (Figure 9.1.5c) associates a thread with each remote object. An I/O thread receives requests and queues them for the workers, but this time there is a per-object queue.

In each of these last two architectures the server benefits from lower thread management overheads compared with the thread-per-request architecture. Their disadvantage is that clients may be delayed while a worker thread has several outstanding requests but another thread has no work to perform.

Schmidt describes variations on these architectures as well as hybrids of them, and discusses their advantages and disadvantages in more detail.

Threads within clients: Threads can be useful for clients as well as servers. Figure 9.1.4 also shows a client process with two threads. The first thread generates results to be passed to a server by remote method invocation, but does not require a reply. Remote method invocations typically block the caller, even when there is strictly no need to wait. This client process can incorporate a second thread, which performs the remote method invocations and blocks while the first thread is able to continue computing further results. The first thread places its results in buffers, which are emptied by the second thread. It is only blocked when all the buffers are full.
The case for multi-threaded clients is also evident in the example of web browsers. Users experience substantial delays while pages are fetched; it is essential, therefore, for browsers to handle multiple concurrent requests for web pages.

**Threads versus multiple processes:** We can see from the above examples the utility of threads, which allow computation to be overlapped with I/O and, in the case of a multiprocessor, with other computation. The reader may have noted, however, that the same overlap could be achieved through the use of multiple single-threaded processes.

Why, then, should the multi-threaded process model be preferred? The answer is twofold: threads are cheaper to create and manage than processes, and resource sharing can be achieved more efficiently between threads than between processes because threads share an execution environment.

Figure 9.1.6 shows some of the main state components that must be maintained for execution environments and threads, respectively. An execution environment has an address space, communication interfaces such as sockets, higher-level resources such as open files and thread synchronization objects such as semaphores; it also lists the threads associated with it. A thread has a scheduling priority, an execution state (such as `BLOCKED` or `RUNNABLE`), saved processor register values when the thread is `BLOCKED`, and state concerning the thread’s software interrupt handling. A *software interrupt* is an event that causes a thread to be interrupted (similar to the case of a hardware interrupt). If the thread has assigned a handler procedure, control is transferred to it. UNIX signals are examples of software interrupts.

<table>
<thead>
<tr>
<th>Execution environment</th>
<th>Thread</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address space tables</td>
<td>Saved processor registers</td>
</tr>
<tr>
<td>Communication interfaces, open files</td>
<td>Priority and execution state (such as <code>BLOCKED</code>)</td>
</tr>
<tr>
<td>Semaphores, other synchronization objects</td>
<td>Software interrupt handling information objects</td>
</tr>
<tr>
<td>List of thread identifiers</td>
<td>Pages of address space resident in memory; hardware cache entries</td>
</tr>
</tbody>
</table>
Figure 9.1.6: State associated with execution environments and threads

The figure shows that an execution environment and the threads belonging to it are both associated with pages belonging to the address space held in main memory, and data and instructions held in hardware caches.

We can summarize a comparison of processes and threads as follows:

- Creating a new thread within an existing process is cheaper than creating a process.
- More importantly, switching to a different thread within the same process is cheaper than switching between threads belonging to different processes.
- Threads within a process may share data and other resources conveniently and efficiently compared with separate processes.
- But, by the same token, threads within a process are not protected from one another.

Consider the cost of creating a new thread in an existing execution environment. The main tasks are to allocate a region for its stack and to provide initial values for the processor registers and the thread’s execution state (it may initially be SUSPENDED or RUNNABLE) and priority. Since the execution environment exists, only an identifier for this has to be placed in the thread’s descriptor record (which contains data necessary to manage the thread’s execution).

When the new entity performs some useful work rather than calling a null procedure, there are also long-term costs, which are liable to be greater for a new process than for a new thread within an existing process. In a kernel supporting virtual memory, the new process will incur page faults as data and instructions are referenced for the first time; hardware caches will initially contain no data values for the new process, and it must acquire cache entries as it executes. In the case of thread creation, these long-term overheads may also occur, but they are liable to be smaller. When the thread accesses
code and data that have recently been accessed by other threads within the process, it automatically takes advantage of any hardware or main memory caching that has taken place.

The second performance advantage of threads concerns switching between threads – that is, running one thread instead of another at a given processor. This cost is the most important, because it may be incurred many times in the lifetime of a thread. Switching between threads sharing the same execution environment is considerably cheaper than switching between threads belonging to different processes. The overheads associated with thread switching are related to scheduling (choosing the next thread to run) and context switching.

A processor context comprises the values of the processor registers such as the program counter, and the current hardware protection domain: the address space and the processor protection mode (supervisor or user). A context switch is the transition between contexts that takes place when switching between threads, or when a single thread makes a system call or takes another type of exception. It involves the following:

- The saving of the processor’s original register state, and the loading of the new state;
- In some cases, a transfer to a new protection domain – this is known as a domain transition.

Switching between threads sharing the same execution environment entirely at user level involves no domain transition and is relatively cheap. Switching to the kernel, or to another thread belonging to the same execution environment via the kernel, involves a domain transition. The cost is therefore greater but it is still relatively low if the kernel is mapped into the process’s address space. When switching between threads belonging to different execution environments, however, there are greater overheads. The next paragraph below explains the expensive implications of hardware caching for these domain transitions. Longer-term costs of having to acquire hardware cache entries and
main memory pages are more liable to apply when such a domain transition occurs. Figures quoted by Anderson et al. are 1.8 milliseconds for the Topaz kernel to switch between UNIX processes and 0.4 milliseconds to switch between threads belonging to the same execution environment. Even lower costs (0.04 milliseconds) are achieved if threads are switched at user level. These figures are given as a rough guide only; they do not measure the longer-term caching costs.

**The aliasing problem:** Memory management units usually include a hardware cache to speed up the translation between virtual and physical addresses, called a *translation lookaside buffer* (TLB). TLBs, and also virtually addressed data and instruction caches, suffer in general from the so-called *aliasing problem*. The same virtual address can be valid in two different address spaces, but in general it is supposed to refer to different physical data in the two spaces. Unless their entries are tagged with a context identifier, TLBs and virtually addressed caches are unaware of this and so might contain incorrect data. Therefore the TLB and cache contents have to be flushed on a switch to a different address space. Physically addressed caches do not suffer from the aliasing problem but using virtual addresses for cache lookups is a common practice, largely because it allows the lookups to be overlapped with address translation.

In the example above of the client process with two threads, the first thread generates data and passes it to the second thread, which makes a remote method invocation or remote procedure call. Since the threads share an address space, there is no need to use message passing to pass the data. Both threads may access the data via a common variable. Herein lies both the advantage and the danger of using multi-threaded processes. The convenience and efficiency of access to shared data is an advantage. This is particularly so for servers, as the example of caching file data given above showed. However, threads that share an address space and that are not written in a type-safe language are not protected from one another. An errant thread can arbitrarily alter data used by another thread, causing a fault. If protection is required, then either a type-safe language should be used or it may be preferable to use multiple processes instead of multiple threads.
**Threads programming:** Threads programming is concurrent programming, as traditionally studied in, for example, the field of operating systems. This section refers to the following concurrent programming concepts, which are explained fully by Bacon: *race conditions, critical sections* (Bacon calls these *critical regions*), *monitors, condition variables* and *semaphores*.

Much threads programming is done in a conventional language, such as C, that has been augmented with a threads library. The C Threads package developed for the Mach operating system is an example of this. More recently, the POSIX Threads standard IEEE 1003.1c-1995, known as *pthreads*, has been widely adopted. Boykin *et al.* describe both C Threads and pthreads in the context of Mach.

Some languages provide direct support for threads, including Ada95, Modula-3 and Java. We give an overview of Java threads here.

Like any threads implementation, Java provides methods for creating threads, destroying them and synchronizing them. The Java *Thread* class includes the constructor and management methods listed in Figure 3.1.7. The *Thread* and *Object* synchronization methods are in Figure 3.1.8.

**Thread lifetimes** • A new thread is created on the same Java virtual machine (JVM) as its creator, in the *SUSPENDED* state. After it is made *RUNNABLE* with the `start()` method, it executes the `run()` method of an object designated in its constructor. The JVM and the threads on top of it all execute in a process on top of the underlying operating system. Threads can be assigned a priority, so that a Java implementation that supports priorities will run a particular thread in preference to any thread with lower priority. A thread ends its life when it returns from the `run()` method or when its `destroy()` method is called.
Programs can manage threads in groups. Every thread belongs to one group, which it is assigned at the time of its creation. Thread groups are useful when several applications coexist on the same JVM. One example of their use is security: by default, a thread in one group cannot perform management operations on a thread in another group. For example, an application thread cannot mischievously interrupt a system windowing (AWT) thread.

Thread(ThreadGroup group, Runnable target, String name)
   Creates a new thread in the SUSPENDED state, which will belong to group and be identified as name; the thread will execute the run() method of target.

setPriority(int newPriority), getPriority()
   Set and return the thread’s priority.

run()
   A thread executes the run() method of its target object, if it has one, and otherwise its own run() method (Thread implements Runnable).

start()
   Change the state of the thread from SUSPENDED to RUNNABLE.

sleep(int millisecs)
   Cause the thread to enter the SUSPENDED state for the specified time.

yield()
   Enter the READY state and invoke the scheduler.

destroy()
   Destroy the thread.

Figure 9.1.7: Java thread constructor and management methods

Thread groups also facilitate control of the relative priorities of threads (on Java implementations that support priorities). This is useful for browsers running applets and for web servers running programs called servlets, which create dynamic web pages. An unprivileged thread within an applet or servlet can only create a new thread that belongs to its own group, or to a descendant group created within it (the exact restrictions depend upon the Security Manager in place). Browsers and servers can assign threads belonging to different applets or servlets to different groups and set the maximum priority of each group as a whole (including descendant groups). There is no way for an applet or servlet
thread to override the group priorities set by the manager threads, since they cannot be overridden by calls to `setPriority()`.

**Thread synchronization:** Programming a multi-threaded process requires great care. The main difficult issues are the sharing of objects and the techniques used for thread coordination and cooperation. Each thread’s local variables in methods are private to it – threads have private stacks. However, threads are not given private copies of static (class) variables or object instance variables.

Consider, for example, the shared queues that we described earlier in this section, which I/O threads and worker threads use to transfer requests in some server threading architectures. Race conditions can in principle arise when threads manipulate data structures such as queues concurrently. The queued requests can be lost or duplicated unless the threads’ pointer manipulations are carefully coordinated.

```java
thread.join(int millisecs)
   Blocks the calling thread for up to the specified time until thread has terminated.
thread.interrupt()
   Interrupts thread; causes it to return from a blocking method call such as sleep().
object.wait(long millisecs, int nanosecs)
   Blocks the calling thread until a call made to notify() or notifyAll()
on object wakes the thread, or the thread is interrupted, or the specified time has elapsed.
object.notify(), object.notifyAll()
   Wakes, respectively, one or all of any threads that have called wait() on object.
```

**Figure 9.1.8: Java thread synchronization calls**

Java provides the `synchronized` keyword for programmers to designate the well known monitor construct for thread coordination. Programmers designate either entire methods or arbitrary blocks of code as belonging to a monitor associated with an individual object. The monitor’s guarantee is that at most one thread can execute within it at any time. We could serialize the actions of the I/O and worker threads in our example by designating `addTo()` and `removeFrom()` methods in the `Queue` class as `synchronized` methods. All
accesses to variables within those methods would then be carried out in mutual exclusion with respect to invocations of these methods.

Java allows threads to be blocked and woken up via arbitrary objects that act as condition variables. A thread that needs to block awaiting a certain condition calls an object’s \textit{wait()} method. All objects implement this method, since it belongs to Java’s root \textit{Object} class. Another thread calls \textit{notify()} to unblock at most one thread or \textit{notifyAll()} to unblock all threads waiting on that object. Both notification methods also belong to the \textit{Object} class.

As an example, when a worker thread discovers that there are no requests to process, it calls \textit{wait()} on the instance of \textit{Queue}. When the I/O thread subsequently adds a request to the queue, it calls the queue’s \textit{notify()} method to wake up a worker.

The Java synchronization methods are given in Figure 3.1.8. In addition to the synchronization primitives that we have mentioned, the \textit{join()} method blocks the caller until the target thread’s termination. The \textit{interrupt()} method is useful for prematurely waking a waiting thread. All the standard synchronization primitives, such as semaphores, can be implemented in Java. But care is required, since Java’s monitor guarantees apply only to an object’s \textit{synchronized} code; a class may have a mixture of \textit{synchronized} and non-\textit{synchronized} methods. Note also that the monitor implemented by a Java object has only one implicit condition variable, whereas in general a monitor may have several condition variables.

\textbf{Thread scheduling:} An important distinction is between preemptive and non-preemptive scheduling of threads. In \textit{preemptive scheduling}, a thread may be suspended at any point to make way for another thread, even when the preempted thread would otherwise continue running. In \textit{non-preemptive scheduling} (sometimes called \textit{coroutine scheduling}), a thread runs until it makes a call to the threading system (for example, a system call), when the system may deschedule it and schedule another thread to run.
The advantage of non-preemptive scheduling is that any section of code that does not contain a call to the threading system is automatically a critical section. Race conditions are thus conveniently avoided. On the other hand, non-preemptively scheduled threads cannot take advantage of a multiprocessor, since they run exclusively. Care must be taken over long-running sections of code that do not contain calls to the threading system. The programmer may need to insert special `yield()` calls, whose sole function is to enable other threads to be scheduled and make progress. Nonpreemptively scheduled threads are also unsuited to real-time applications, in which events are associated with absolute times by which they must be processed.

Java does not, by default, support real-time processing, although real-time implementations exist. For example, multimedia applications that process data such as voice and video have real-time requirements for both communication and processing (e.g., filtering and compression). Process control is another example of a real-time domain. In general, each real-time domain has its own thread scheduling requirements. It is therefore sometimes desirable for applications to implement their own scheduling policies. To consider this, we turn now to the implementation of threads.

**Threads implementation:** Many kernels provide native support for multi-threaded processes, including Windows, Linux, Solaris, Mach and Mac OS X. These kernels provide thread-creation and -management system calls, and they schedule individual threads. Some other kernels have only a single-threaded process abstraction. Multithreaded processes must then be implemented in a library of procedures linked to application programs. In such cases, the kernel has no knowledge of these user-level threads and therefore cannot schedule them independently. A threads runtime library organizes the scheduling of threads. A thread would block the process, and therefore all threads within it, if it made a blocking system call, so the asynchronous (non-blocking) I/O facilities of the underlying kernel are exploited. Similarly, the implementation can utilize the kernel-provided timers and software interrupt facilities to time slice between threads.
When no kernel support for multi-threaded processes is provided, a user-level threads implementation suffers from the following problems:

- The threads within a process cannot take advantage of a multiprocessor.
- A thread that takes a page fault blocks the entire process and all threads within it.
- Threads within different processes cannot be scheduled according to a single scheme of relative prioritization.

User-level threads implementations, on the other hand, have significant advantages over kernel-level implementations:

- Certain thread operations are significantly less costly. For example, switching between threads belonging to the same process does not necessarily involve a system call, which entails a relatively expensive trap to the kernel.
- Given that the thread-scheduling module is implemented outside the kernel, it can be customized or changed to suit particular application requirements. Variations in scheduling requirements occur largely because of application-specific considerations such as the real-time nature of multimedia processing.
- Many more user-level threads can be supported than could reasonably be provided by default by a kernel.

It is possible to combine the advantages of user-level and kernel-level threads implementations. One approach, applied, for example, to the Mach kernel, is to enable user-level code to provide scheduling hints to the kernel’s thread scheduler. Another, adopted in the Solaris 2 operating system, is a form of hierarchical scheduling. Each process creates one or more kernel-level threads, known in Solaris as ‘lightweight processes’. User-level threads are also supported. A user-level scheduler assigns each user-level thread to a kernel-level thread. This scheme can take advantage of multiprocessors, and also benefits because some thread-creation and thread-switching operations take place at user level. The scheme’s disadvantage is that it still lacks
flexibility: if a thread blocks in the kernel, then all user-level threads assigned to it are also prevented from running, regardless of whether they are eligible to run.

Several research projects have developed hierarchical scheduling further in order to provide greater efficiency and flexibility. These include work on so-called scheduler activations, the multimedia work of Govindan and Anderson, the Psyche multiprocessor operating system, the Nemesis kernel and the SPIN kernel. The insight driving these designs is that what a user-level scheduler requires from the kernel is not just a set of kernel-supported threads onto which it can map user-level threads. The user-level scheduler also requires the kernel to notify it of the events that are relevant to its scheduling decisions. We describe the scheduler activations design in order to make this clear.

The FastThreads package of Anderson et al. is an implementation of a hierarchic, event-based scheduling system. They consider the main system components to be a kernel running on a computer with one or more processors, and a set of application programs running on it. Each application process contains a user-level scheduler, which manages the threads inside the process. The kernel is responsible for allocating virtual processors to processes. The number of virtual processors assigned to a process depends on such factors as the applications’ requirements, their relative priorities and the total demand on the processors. Figure 7.10(a) shows an example of a three-processor machine, on which the kernel allocates one virtual processor to process $A$, running a relatively low-priority job, and two virtual processors to process $B$. They are virtual processors because the kernel can allocate different physical processors to each process as time goes by, while keeping its guarantee of how many processors it has allocated.

The number of virtual processors assigned to a process can also vary. Processes can give back a virtual processor that they no longer need; they can also request extra virtual processors. For example, if process $A$ has requested an extra virtual processor and $B$ terminates, then the kernel can assign one to $A$. 
Figure 9.1.9: Scheduler activations

Figure 9.1.9(b) shows that a process notifies the kernel when either of two types of event occurs: when a virtual processor is ‘idle’ and no longer needed, or when an extra virtual processor is required.

Figure 9.1.9(b) also shows that the kernel notifies the process when any of four types of event occurs. A scheduler activation (SA) is a call from the kernel to a process, which notifies the process’s scheduler of an event. Entering a body of code from a lower layer (the kernel) in this way is sometimes called an upcall. The kernel creates an SA by loading a physical processor’s registers with a context that causes it to commence execution of code in the process, at a procedure address designated by the user-level scheduler. An SA is thus also a unit of allocation of a time slice on a virtual processor. The user-level scheduler has the task of assigning its READY threads to the set of SAs currently executing within it. The number of those SAs is at most the number of virtual processors that the kernel has assigned to the process.

The four types of event that the kernel notifies the user-level scheduler (which we shall refer to simply as ‘the scheduler’) of are as follows:
**Virtual processor allocated:** The kernel has assigned a new virtual processor to the process, and this is the first time slice upon it; the scheduler can load the SA with the context of a READY thread, which can thus recommence execution.

**SA blocked:** An SA has blocked in the kernel, and the kernel is using a fresh SA to notify the scheduler; the scheduler sets the state of the corresponding thread to BLOCKED and can allocate a READY thread to the notifying SA.

**SA unblocked:** An SA that was blocked in the kernel has become unblocked and is ready to execute at user level again; the scheduler can now return the corresponding thread to the READY list. In order to create the notifying SA, the kernel either allocates a new virtual processor to the process or preempts another SA in the same process. In the latter case, it also communicates the preemption event to the scheduler, which can reevaluate its allocation of threads to SAs.

**SA preempted:** The kernel has taken away the specified SA from the process (although it may do this to allocate a processor to a fresh SA in the same process); the scheduler places the preempted thread in the READY list and reevaluates the thread allocation.

This hierarchical scheduling scheme is flexible because the process’s user-level scheduler can allocate threads to SAs in accordance with whatever policies can be built on top of the low-level events. The kernel always behaves the same way. It has no influence on the user-level scheduler’s behaviour, but it assists the scheduler through its event notifications and by providing the register state of blocked and preempted threads. The scheme is potentially efficient because no user-level thread need stay in the READY state if there is a virtual processor on which to run it.

**9.5 SUMMARY**

In this unit we discussed about operating system in the context of distributed systems, protection, process and threads.
9.7.  KEYWORDS

A distributed operating system: An operating system that produces a single system image like this for all the resources in a distributed system.

Encapsulation: Details such as management of memory and devices used to implement resources are hidden from clients.

Kernels The kernel is a program that is distinguished by the facts that it remains loaded from system initialization and its code is executed with complete access privileges for the physical resources on its host computer. In particular, it can control the memory management unit and set the processor registers so that no other code may access the machine’s physical resources except in acceptable ways.

Thread: A thread is the operating system abstraction of an activity (the term derives from the phrase ‘thread of execution’).

Address spaces: An address space is a unit of management of a process’s virtual memory.

9.8.  UNIT-END EXERCISES AND ANSWERS

1. Define distributed operating system.
2. What are the core OS components and their responsibilities?
3. What is an execution environment? An execution environment primarily consists of what?
4. Explain the process of Creation of a new process.

Answers: SEE

1. 9.1
2. 9.2
3. 9.3
4. 9.4
9.9 SUGGESTED READINGS


- DISTRIBUTED SYSTEMS Principles and Paradigm By: Andrew S. Tanenbaum, Maarten Van Steen.
UNIT 10: COMMUNICATION AND INVOCATION

Structure:

10.0 Objectives remote invocation delay
10.1 Introduction
10.2 Communication and Invocation
10.3 Operating system architecture
10.4 Summary
10.5 Keywords
10.6 Unit-end exercises and answers
10.7 Suggested readings

10.0 OBJECTIVES
At the end of this unit you will be able to know:

- Communication,
- Communication primitives,
- Protocols and openness
- Invocation costs
- Asynchronous operation
- Operating system architecture

10.1 INTRODUCTION
Here we concentrate on communication as part of the implementation of what we have called an invocation – a construct, such as a remote method invocation, remote procedure call or event notification, whose purpose is to bring about an operation on a resource in a different address space.
We cover operating system design issues and concepts by asking the following questions about the OS:

- What communication primitives does it supply?
- Which protocols does it support and how open is the communication implementation?
- What steps are taken to make communication as efficient as possible?
- What support is provided for high-latency and disconnected operation?

10.2 COMMUNICATION AND INVOCATION

Communication primitives: Some kernels designed for distributed systems have provided communication primitives tailored to some types of invocation. Amoeba, for example, provides \textit{doOperation}, \textit{getRequest} and \textit{sendReply} as primitives. Amoeba, the V system and Chorus provide group communication primitives. Placing relatively high-level communication functionality in the kernel has the advantage of efficiency. If, for example, middleware provides RMI over UNIX’s connected (TCP) sockets, then a client must make two communication system calls (socket \texttt{write} and \texttt{read}) for each remote invocation. Over Amoeba, it would require only a single call to \textit{doOperation}. The savings in system call overhead are liable to be even greater with group communication.

Protocols and openness: One of the main requirements of the operating system is to provide standard protocols that enable interworking between middleware implementations on different platforms. Several research kernels developed in the 1980s incorporated their own network protocols tuned to RPC interactions – notably Amoeba RPC, VMTP and Sprite RPC. However, these protocols were not widely used beyond their native research environments. By contrast, the designers of the Mach 3.0 and Chorus kernels decided to leave the choice of networking protocols entirely open. These kernels provide message passing between local processes only, and leave network protocol processing to a server that runs on top of the kernel.
Given the everyday requirement for access to the Internet, compatibility at the level of TCP and UDP is required of operating systems for all but the smallest of networked devices. And the operating system is still required to enable middleware to take advantage of novel low-level protocols. For example, users want to benefit from wireless technologies such as infrared and radio frequency (RF) transmission, preferably without having to upgrade their applications. This requires that corresponding protocols, such as IrDA for infrared networking and Bluetooth or IEEE 802.11 for RF networking, can be integrated.

Protocols are normally arranged in a stack of layers. Many operating systems allow new layers to be integrated statically, by including a layer such as IrDA as a permanently installed protocol ‘driver’. By contrast, dynamic protocol composition is a technique whereby a protocol stack can be composed on the fly to meet the requirements of a particular application, and to utilize whichever physical layers are available given the platform’s current connectivity. For example, a web browser running on a notebook computer should be able to take advantage of a wide area wireless link while the user is on the road, and then a faster Ethernet or IEEE 802.11 connection when the user is back in the office.

**Invocation performance**

Invocation performance is a critical factor in distributed system design. The more designers separate functionality between address spaces, the more remote invocations are required. Clients and servers may make many millions of invocation-related operations in their lifetimes, so small fractions of milliseconds count in invocation costs. Network technologies continue to improve, but invocation times have not decreased in proportion with increases in network bandwidth. This section will explain how software overheads often predominate over network overheads in invocation times – at least, for the case of a LAN or intranet. This is in contrast to a remote invocation over the Internet – for example, fetching a web resource. On the Internet, network latencies are highly variable and relatively high on average; throughput may be relatively low, and server load often predominates over per-request processing costs.
RPC and RMI implementations have been the subject of study because of the widespread acceptance of these mechanisms for general-purpose client-server processing. Much of the research has been carried out into invocations over the network, and particularly into how invocation mechanisms can take advantage of high performance networks. There is also an important special case of RPCs between processes hosted at the same computer.

**Invocation costs:** Calling a conventional procedure or invoking a conventional method, making a system call, sending a message, remote procedure calling and remote method invocation are all examples of invocation mechanisms. Each mechanism causes code to be executed outside the scope of the calling procedure or object. Each involves, in general, the communication of arguments to this code and the return of data values to the caller. Invocation mechanisms can be either synchronous, as for example in the case of conventional and remote procedure calls, or asynchronous.

The important performance-related distinctions between invocation mechanisms, apart from whether or not they are synchronous, are whether they involve a domain transition (that is, whether they cross an address space), whether they involve communication across a network and whether they involve thread scheduling and switching. Figure 3.2.1 shows the particular cases of a system call, a remote invocation between processes hosted at the same computer, and a remote invocation between processes at different nodes in the distributed system.
Invocation over the network: A null RPC (and similarly, a null RMI) is defined as an RPC without parameters that executes a null procedure and returns no values. Its execution involves an exchange of messages carrying some system data but no user data. The time taken by a null RPC between user processes connected by a LAN is on the order of a tenth of a millisecond (see, for example, measurements by Bridges et al. of round-trip UDP times using two 2.2GHz Pentium 3 Xeon PCs across a 100 megabits/second Ethernet). By comparison, a null conventional procedure call takes a small fraction of a microsecond. Approximately 100 bytes in total are passed across the network for a null RPC. With a raw bandwidth of 100 megabits/second, the total network
transfer time for this amount of data is about 0.01 milliseconds. Clearly, much of the observed delay – the total RPC call time experienced by a client – has to be accounted for by the actions of the operating system kernel and user-level RPC runtime code.

Null invocation (RPC, RMI) costs are important because they measure a fixed overhead, the latency. Invocation costs increase with the sizes of arguments and results, but in many cases the latency is significant compared with the remainder of the delay.

Consider an RPC that fetches a specified amount of data from a server. It has one integer request argument, specifying how much data to return. It has two reply arguments, an integer specifying success or failure (the client might have given an invalid size) and, when the call is successful, a array of bytes from the server.

![Figure 10.2.2: RPC delay against parameter size](image)

Figure 10.2.2 shows, schematically, client delay against requested data size. The delay is roughly proportional to the size until the size reaches a threshold at about network packet size. Beyond that threshold, at least one extra packet has to be sent, to carry the extra data. Depending on the protocol, a further packet might be used to acknowledge this extra packet. Jumps in the graph occur each time the number of packets increases.

Delay is not the only figure of interest for an RPC implementation: RPC throughput (or bandwidth) is also of concern when data has to be transferred in bulk. This is the rate of data transfer between computers in a single RPC. If we examine Figure 3.2.2, we can see
that the throughput is relatively low for small amounts of data, when the fixed processing overheads predominate. As the amount of data is increased, the throughput rises as those overheads become less significant.

Recall that the steps in an RPC are as follows (RMI involves similar steps):

- A client stub marshals the call arguments into a message, sends the request message and receives and unmarshals the reply.
- At the server, a worker thread receives the incoming request, or an I/O thread receives the request and passes it to a worker thread; in either case, the worker calls the appropriate server stub.
- The server stub unmarshals the request message, calls the designated procedure, and marshals and sends the reply.

The following are the main components accounting for remote invocation delay, besides network transmission times:

**Marshalling:** Marshalling and unmarshalling, which involve copying and converting data, create a significant overhead as the amount of data grows.

**Data copying:** Potentially, even after marshalling, message data is copied several times in the course of an RPC:

1. Across the user–kernel boundary, between the client or server address space and kernel buffers;
2. Across each protocol layer (for example, RPC/UDP/IP/Ethernet);
3. Between the network interface and kernel buffers.

Transfers between the network interface and main memory are usually handled by direct memory access (DMA). The processor handles the other copies.
Packet initialization: This involves initializing protocol headers and trailers, including checksums. The cost is therefore proportional, in part, to the amount of data sent.

Thread scheduling and context switching: These may occur as follows:

1. Several system calls (that is, context switches) are made during an RPC, as stubs invoke the kernel’s communication operations.
2. One or more server threads is scheduled.
3. If the operating system employs a separate network manager process, then each Send involves a context switch to one of its threads.

Waiting for acknowledgements: The choice of RPC protocol may influence delay, particularly when large amounts of data are sent.

Memory sharing: Shared regions may be used for rapid communication between a user process and the kernel, or between user processes. Data is communicated by writing to and reading from the shared region. Data is thus passed efficiently, without being copied to and from the kernel’s address space. But system calls and software interrupts may be required for synchronization, such as when the user process has written data that should be transmitted, or when the kernel has written data for the user process to consume. Of course, a shared region is only justified if it is used sufficiently to offset the initial cost of setting it up.

Even with shared regions, the kernel still has to copy data from the buffers to the network interface. The U-Net architecture even allows user-level code to have direct access to the network interface itself, so that user-level code can transfer the data to the network without any copying.

Choice of protocol: The delay that a client experiences during request-reply interactions over TCP is not necessarily worse than for UDP and in fact is sometimes better, particularly for large messages. However, care is required when implementing request-
reply interactions on top of a protocol such as TCP, which was not specifically designed for this purpose. In particular, TCP’s buffering behaviour can hinder good performance, and its connection overheads put it at a disadvantage compared with UDP, unless enough requests are made over a single connection to render the overhead per request negligible.

The connection overheads of TCP are particularly evident in web invocations. HTTP 1.0, now relatively little-used, makes a separate TCP connection for every invocation. Client browsers are delayed while the connection is made. Furthermore, TCP’s slow-start algorithm has the effect of delaying the transfer of HTTP data unnecessarily in many cases. The slow-start algorithm operates pessimistically in the face of possible network congestion by allowing only a small window of data to be sent at first, before an acknowledgement is received. Nielsen et al discuss how HTTP 1.1, now widely used instead of HTTP 1.0, makes use of so-called persistent connections, which last over the course of several invocations. The initial connection costs are thus amortized, as long as several invocations are made to the same web server. This is likely, as users often fetch several pages from the same site, each containing several images.

**Invocation within a computer:** Bershad et al. report a study that showed that, in the installation examined, most cross-address-space invocation took place within a computer and not, as might be expected in a client-server installation, between computers. The trend towards placing service functionality inside user-level servers means that more and more invocations will be to a local process. This is especially so as caching is pursued aggressively if the data needed by a client is liable to be held in a local server. The cost of an RPC within a computer is growing in importance as a system performance parameter. These considerations suggest that this local case should be optimized.

Figure 10.2.1 suggests that a cross-address-space invocation is implemented within a computer exactly as it is between computers, except that the underlying message passing happens to be local. Indeed, this has often been the model implemented. Bershad et al. developed a more efficient invocation mechanism for the case of two processes on the
same machine called \textit{lightweight RPC} (LRPC). The LRPC design is based on optimizations concerning data copying and thread scheduling.

First, they noted that it would be more efficient to use shared memory regions for client-server communication, with a different (private) region between the server and each of its local clients. Such a region contains one or more \textit{A} (for argument) \textit{stacks}. Instead of RPC parameters being copied between the kernel and user address spaces involved, the client and server are able to pass arguments and return values directly via an A stack. The same stack is used by the client and server stubs. In LRPC, arguments are copied once: when they are marshalled onto the A stack. In an equivalent RPC, they are copied four times: from the client stub’s stack onto a message; from the message to a kernel buffer, from the kernel buffer to a server message, and from the message to the server stub’s stack. There may be several A stacks in a shared region, because several threads in the same client may call the server at the same time.

![Figure 10.2.3: A lightweight remote procedure call](image)

Bershad \textit{et al.} also considered the cost of thread scheduling. Compare the model of system call and remote procedure calls in Figure 3.2.1. When a system call occurs, most kernels do not schedule a new thread to handle the call but instead perform a context switch on the calling thread so that it handles the system call. In an RPC, a remote
procedure may exist in a different computer from the client thread, so a different thread must be scheduled to execute it. In the local case, however, it may be more efficient for the client thread – which would otherwise be *BLOCKED* – to call the invoked procedure in the server’s address space.

A server must be programmed differently in this case to the way we have described servers before. Instead of setting up one or more threads, which then listen on ports for invocation requests, the server exports a set of procedures that it is prepared to have called. Threads in local processes may enter the server’s execution environment as long as they start by calling one of the server’s exported procedures. A client needing to invoke a server’s operations must first bind to the server interface (not shown in the figure). It does this via the kernel, which notifies the server; when the server has responded to the kernel with a list of allowed procedure addresses, the kernel replies to the client with a capability for invoking the server’s operations.

An invocation is shown in Figure 3.2.3. A client thread enters the server’s execution environment by first trapping to the kernel and presenting it with a capability. The kernel checks this and only allows a context switch to a valid server procedure; if it is valid, the kernel switches the thread’s context to call the procedure in the server’s execution environment. When the procedure in the server returns, the thread returns to the kernel, which switches the thread back to the client execution environment. Note that clients and servers employ stub procedures to hide the details just described from application writers.

**Discussion of LRPC** • There is little doubt that LRPC is more efficient than RPC for the local case, as long as enough invocations take place to offset the memory management costs. Bershad *et al.* record LRPC delays a factor of three smaller than those of RPCs executed locally.

Location transparency is not sacrificed in Bershad’s implementation. A client stub examines a bit set at bind time that records whether the server is local or remote, and proceeds to use LRPC or RPC, respectively. The application is unaware of which is used.
However, migration transparency might be hard to achieve when a resource is transferred from a local server to a remote server or vice versa, because of the need to change invocation mechanisms.

**Asynchronous operation**

We have discussed how the operating system can help the middleware layer to provide efficient remote invocation mechanisms. But in the Internet environment the effects of relatively high latencies, low throughput and high server loads may outweigh any benefits that the OS can provide. We can add to this the phenomena of network disconnection and reconnection, which can be regarded as causing extremely high-latency communication. Users’ mobile computers are not connected to the network all the time. Even if they have wide area wireless access (for example, using cellular communication), they may be peremptorily disconnected when, for example, their train enters a tunnel.

A common technique to defeat high latencies is asynchronous operation, which arises in two programming models: concurrent invocations and asynchronous invocations. These models are largely in the domain of middleware rather than operating system kernel design, but it is useful to consider them here, while we are examining the topic of invocation performance.

**Making invocations concurrently** • In the first model, the middleware provides only blocking invocations, but the application spawns multiple threads to perform blocking invocations concurrently.
A good example of such an application is a web browser. A web page typically contains several images and may contain many. The browser does not need to obtain the images in a particular sequence, so it makes several concurrent requests at a time. That way, the time taken to complete all the image requests is typically lower than the delay that would result from making the requests serially. Not only is the total communication delay less, in general, but the browser can overlap computation such as image rendering with communication.

Figure 10.2.4 shows the potential benefits of interleaving invocations (such as HTTP requests) between a client and a single server on a single-processor machine. In the serialized case, the client marshals the arguments, calls the Send operation and then waits until the reply from the server arrives – whereupon it Receives, unmarshals and then processes the results. After this it can make the second invocation.
In the concurrent case, the first client thread marshals the arguments and calls the Send operation. The second thread then immediately makes the second invocation. Each thread waits to receive its results. The total time taken is liable to be lower than in the serialized case, as the figure shows. Similar benefits apply if the client threads make concurrent requests to several servers, and if the client executes on a multiprocessor even greater throughput is potentially possible, since the two threads’ processing can also be overlapped.

**Asynchronous invocations:** An asynchronous invocation is one that is performed asynchronously with respect to the caller. That is, it is made with a non-blocking call, which returns as soon as the invocation request message has been created and is ready for dispatch.

Sometimes the client does not require any response (except perhaps an indication of failure if the target host could not be reached). For example, CORBA one way invocations have maybe semantics. Otherwise, the client uses a separate call to collect the results of the invocation. For example, the Mercury communication system supports asynchronous invocations. An asynchronous operation returns an object called a promise. Eventually, when the invocation succeeds or is deemed to have failed, the Mercury system places the status and any return values in the promise. The caller uses the claim operation to obtain the results from the promise. The claim operation blocks until the promise is ready, whereupon it returns the results or exceptions from the call. The ready operation is available for testing a promise without blocking – it returns true or false according to whether the promise is ready or blocked.

**Persistent asynchronous invocations:** Traditional asynchronous invocation mechanisms such as Mercury invocations and CORBA oneway invocations are implemented upon TCP streams and fail if a stream breaks – that is, if the network link is down or the target host crashes.
But a more developed form of the asynchronous invocation model, which we shall call *persistent asynchronous invocation*, is becoming increasingly relevant because of disconnected operation. This model is similar to Mercury in terms of the programming operations it provides, but the difference is in its failure semantics. A conventional invocation mechanism (synchronous or asynchronous) is designed to fail after a given number of timeouts have occurred, but these short-term timeouts are often not appropriate where disconnections or very high latencies occur.

A system for persistent asynchronous invocation tries indefinitely to perform the invocation, until it is known to have succeeded or failed, or until the application cancels the invocation. An example is Queued RPC (QRPC) in the Rover toolkit for mobile information access.

As its name suggests, QRPC queues outgoing invocation requests in a stable log while there is no network connection and schedules their dispatch over the network to servers when there is a connection. Similarly, it queues invocation results from servers in what we can consider to be the client’s invocation ‘mailbox’ until the client reconnects and collects them. Requests and results may be compressed when they are queued, before their transmission over a low-bandwidth network.

QRPC can take advantage of different communication links for sending an invocation request and receiving the reply. For example, a request could be dispatched over a cellular data link while the user is on the road, and then the response delivered over an Ethernet link when the user connects her device to the corporate intranet. In principle, the invocation system can even store the invocation results near to the user’s next expected point of connection.

The client’s network scheduler operates according to various criteria and does not necessarily dispatch invocations in FIFO order. Applications can assign priorities to individual invocations. When a connection becomes available, QRPC evaluates its
bandwidth and the expense of using it. It dispatches high-priority invocation requests first, and may not dispatch all of them if the link is slow and expensive (such as a wide area wireless connection), assuming that a faster, cheaper link such as an Ethernet will become available eventually. Similarly, QRPC takes priority into account when fetching invocation results from the mailbox over a low-bandwidth link.

Programming with an asynchronous invocation system (persistent or otherwise) raises the issue of how users can continue using the applications on their client device while the results of invocations are still not known. For example, the user may wonder whether they have succeeded in updating a paragraph in a shared document, or if someone else has made a conflicting update, such as deleting the paragraph.

### 10.3 OPERATING SYSTEM ARCHITECTURE

In this section, we examine the architecture of a kernel suitable for a distributed system. We adopt a first-principles approach of starting with the requirement of openness and examining the major kernel architectures that have been proposed, with this in mind.

An open distributed system should make it possible to:

- Run only that system software at each computer that is necessary for it to carry out its particular role in the system architecture – system software requirements can vary between, for example, mobile phones and server computers, and loading redundant modules wastes memory resources;
- Allow the software (and the computer) implementing any particular service to be changed independently of other facilities;
- Allow for alternatives of the same service to be provided, when this is required to suit different users or applications;
- Introduce new services without harming the integrity of existing ones.
The separation of fixed resource management *mechanisms* from resource management *policies*, which vary from application to application and service to service, has been a guiding principle in operating system design for a long time. For example, we said that an ideal scheduling system would provide mechanisms that enable a multimedia application such as video conferencing to meet its real-time demands while coexisting with a non-real-time application such as web browsing.

![Figure 10.2.5 Monolithic kernel and microkernel](image)

Ideally, the kernel would provide only the most basic mechanisms upon which the general resource management tasks at a node are carried out. Server modules would be dynamically loaded as required, to implement the required resource management policies for the currently running applications.

**Monolithic kernels and microkernels** • There are two key examples of kernel design: the so-called *monolithic* and *microkernel* approaches. These designs differ primarily in the decision as to what functionality belongs in the kernel and what is to be left to server processes that can be dynamically loaded to run on top of it. Although microkernels have not been deployed widely, it is instructive to understand their advantages and disadvantages compared with the typical kernels found today.

The UNIX operating system kernel has been called *monolithic*. This term is meant to suggest that it is *massive* – it performs all basic operating system functions and takes up in the order of megabytes of code and data – and that it is *undifferentiated*, i.e. it is coded
in a non-modular way. The result is that to a large extent it is *intractable*: altering any individual software component to adapt it to changing requirements is difficult. Another example of a monolithic kernel is that of the Sprite network operating system. A monolithic kernel can contain some server processes that execute within its address space, including file servers and some networking. The code that these processes execute is part of the standard kernel configuration.

By contrast, in the case of a microkernel design the kernel provides only the most basic abstractions, principally address spaces, threads and *local* interprocess communication; *all* other system services are provided by servers that are dynamically loaded at precisely those computers in the distributed system that require them (Figure 3.2.5). Clients access these system services using the kernel’s message-based invocation mechanisms.

<table>
<thead>
<tr>
<th>Middleware</th>
<th>Language support subsystem</th>
<th>Language support subsystem</th>
<th>OS emulation subsystem</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Microkernel</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hardware</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The microkernel supports middleware via subsystems

Figure 10.2.6: The role of the microkernel

We said above that users are liable to reject operating systems that do not run their applications. But in addition to extensibility, microkernel designers have another goal: the binary emulation of standard operating systems such as UNIX.

The place of the microkernel – in its most general form – in the overall distributed system design is shown in Figure 10.2.6. The microkernel appears as a layer between the hardware layer and a layer consisting of major system components called *subsystems*. If performance is the main goal, rather than portability, then middleware may use the facilities of the microkernel directly. Otherwise, it uses a language runtime support subsystem, or a higher-level operating system interface provided by an operating system.
emulation subsystem. Each of these, in turn, is implemented by a combination of library procedures linked into applications and a set of servers running on top of the microkernel.

There can be more than one system call interface – more than one ‘operating system’ – presented to the programmer on the same underlying platform. An example is the implementation of UNIX and OS/2 on top of the Mach distributed operating system kernel. Note that operating system emulation is different from machine virtualization.

**Comparison:** The chief advantages of a microkernel-based operating system are its extensibility and its ability to enforce modularity behind memory protection boundaries. In addition, a relatively small kernel is more likely to be free of bugs than one that is larger and more complex.

The advantage of a monolithic design is the relative efficiency with which operations can be invoked. System calls may be more expensive than conventional procedures, but even using the techniques we examined in the previous section, an invocation to a separate user-level address space on the same node is more costly still.

The lack of structure in monolithic designs can be avoided by the use of software engineering techniques such as layering, used in MULTICS, or object oriented design, used for example in Choices. Windows employs a combination of both. But Windows remains ‘massive’, and the majority of its functionality is not designed to be routinely replaceable. Even a modularized large kernel can be hard to maintain, and it provides limited support for an open distributed system. As long as modules are executed within the same address space, using a language such as C or C++ that compiles to efficient code but permits arbitrary data accesses, it is possible for strict modularity to be broken by programmers seeking efficient implementations, and for a bug in one module to corrupt the data in another.

**Some hybrid approaches:** Two of the original microkernels, Mach and Chorus, began their developmental life running servers only as user processes. In this configuration,
modularity is hardware-enforced through address spaces. Where servers require direct access to hardware, special system calls can be provided for these privileged processes, which map device registers and buffers into their address spaces. The kernel turns interrupts into messages, which enables user-level servers to handle interrupts.

Because of performance problems, the Chorus and Mach microkernel designs eventually changed to allow servers to be loaded dynamically either into the kernel address space or into a user-level address space. In each case, clients interact with servers using the same interprocess communication calls. A developer can thus debug a server at user level and then, when the development is deemed complete, allow the server to run inside the kernel’s address space in order to optimize system performance. But such a server then threatens the integrity of the system, should it turn out still to contain bugs.

In an attempt to minimize the dependencies between system modules, the SPIN designers chose an event-based model as a mechanism for interaction between modules grafted into the kernel’s address. The system defines a set of core events, such as network packet arrival, timer interrupts, page fault occurrences and thread state changes. System components operate by registering themselves as handlers for the events that affect them. For example, a scheduler would register itself to handle events.

Operating systems such as Nemesis exploit the fact that, even at the hardware level, an address space is not necessarily also a single protection domain. The kernel coexists in a single address space with all dynamically loaded system modules and all applications. When it loads an application, the kernel places the application’s code and data in regions chosen from those that are available at runtime. The advent of processors with 64-bit addressing has made single-address-space operating systems particularly attractive, since they support very large address spaces that can accommodate many applications.

The kernel of a single-address-space operating system sets the protection attributes on individual regions within the address space to restrict access by user-level code. User-level code still runs with the processor in a particular protection context (determined by
settings in the processor and memory management unit), which gives it full access to its
own regions and only selectively shared access to others. The saving of a single address
space, compared with using multiple address spaces, is that the kernel need never flush
any caches when it implements a domain transition.

Some later kernel designs, such as L4 and the Exokernel, take the approach that what we
have described as ‘microkernels’ still contain too much policy as opposed to mechanism.
L4 is a ‘second-generation’ microkernel design that forces dynamically loaded system
modules to execute in user-level address spaces, but optimizes interprocess
communication to offset the costs of doing so. It offloads much of the kernel’s
complexity by delegating the management of address spaces to user-level servers. The
Exokernel takes a quite different approach, employing user-level libraries instead of user-
level servers to supply functional extensions. It provides protected allocation of
extremely low-level resources such as disk blocks, and it expects all other resource
management functionality – even a file system – to be linked into applications as
libraries.

In the words of one microkernel designer, ‘the microkernel story is full of good ideas and
blind alleys’. As we shall see in the next section, the need to support multiple subsystems
and also enforce protection between these subsystems is now met by the concept of
virtualization which has replaced microkernel approaches as the key innovation in
operating system design.

10.4 SUMMARY

In this unit we introduced Communication, Communication primitives, Protocols and
openness, Invocation costs, Asynchronous operation and Operating system architecture.

10.5 KEYWORDS

DMA: direct memory access.
Asynchronous invocations: An *asynchronous invocation* is one that is performed asynchronously with respect to the caller. That is, it is made with a non-blocking call, which returns as soon as the invocation request message has been created and is ready for dispatch.

### 10.6 UNIT-END EXERCISES AND ANSWERS

1. What steps are taken to make communication as efficient as possible?
2. What support is provided for high-latency and disconnected operation?
3. What are the main components accounting for remote invocation delay, besides network transmission times?
4. Explain Operating system architecture.

**Answers: SEE**

1. 10.2
2. 10.2
3. 10.2
4. 10.3

### 10.6 SUGGESTED READINGS

- DISTRIBUTED SYSTEMS Principles and Paradigm By: Andrew S. Tanenbaum, Maarten Van Steen.
UNIT 11: OVERVIEW OF SECURITY TECHNIQUES

Structure:
11.0 Objectives
11.1 Introduction
11.2 Security
11.3 Cryptographic algorithms
11.4 Summary
11.5 Keywords
11.6 Unit-end exercises and answers
11.7 Suggested readings

11.0 OBJECTIVES

At the end of this unit you will be able to know:

- Security techniques
- Uses of cryptography
- Access control
- Firewalls
- Symmetric and asymmetric cryptographic algorithms
- Block ciphers and stream ciphers
- Cryptographic algorithms

11.1 INTRODUCTION

This unit is about the security techniques and cryptographic algorithms.

There is a pervasive need for measures to guarantee the privacy, integrity and availability of resources in distributed systems. Security attacks take the forms of eavesdropping, masquerading, tampering and denial of service. Designers of secure distributed systems must cope with exposed service interfaces and insecure networks in an environment
where attackers are likely to have knowledge of the algorithms used and to deploy computing resources.

Cryptography provides the basis for the authentication of messages as well as their secrecy and integrity; carefully designed security protocols are required to exploit it. The selection of cryptographic algorithms and the management of keys are critical to the effectiveness, performance and usability of security mechanisms. Public-key cryptography makes it easy to distribute cryptographic keys but its performance is inadequate for the encryption of bulk data. Secret-key cryptography is more suitable for bulk encryption tasks. Hybrid protocols such as Transport Layer Security (TLS) establish a secure channel using public-key cryptography and then use it to exchange secret keys for use in subsequent data exchanges.

### 11.1 OVERVIEW OF SECURITY TECHNIQUES

The aim of this unit is to introduce the reader to some of the more important techniques and mechanisms for securing distributed systems and applications.

**Worst-case assumptions and design guidelines**

**Interfaces are exposed:** Distributed systems are composed of processes that offer services or share information. Their communication interfaces are necessarily open (to allow new clients to access them) – an attacker can send a message to any interface.

**Networks are insecure:** For example, message sources can be falsified – messages can be made to look as though they came from Alice when they were actually sent by Mallory. Host addresses can be ‘spoofed’ – Mallory can connect to the network with the same address as Alice and receive copies of messages intended for her.

**Limit the lifetime and scope of each secret:** When a secret key is first generated we can be confident that it has not been compromised. The longer we use it and the more widely it is known, the greater the risk. The use of secrets such as passwords and shared secret keys should be time-limited, and sharing should be restricted.
**Algorithms and program code are available to attackers:** The bigger and the more widely distributed a secret is, the greater the risk of its disclosure. Secret encryption algorithms are totally inadequate for today’s large-scale network environments. Best practice is to publish the algorithms used for encryption and authentication, relying only on the secrecy of cryptographic keys. This helps to ensure that the algorithms are strong by throwing them open to scrutiny by third parties.

**Attackers may have access to large resources:** The cost of computing power is rapidly decreasing. We should assume that attackers will have access to the largest and most powerful computers projected in the lifetime of a system, then add a few orders of magnitude to allow for unexpected developments.

**Minimize the trusted base:** The portions of a system that are responsible for the implementation of its security, and all the hardware and software components upon which they rely, have to be trusted – this is often referred to as the trusted computing base. Any defect or programming error in this trusted base can produce security weaknesses, so we should aim to minimize its size. For example, application programs should not be trusted to protect data from their users.

<table>
<thead>
<tr>
<th>Notation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$K_A$</td>
<td>Alice’s secret key</td>
</tr>
<tr>
<td>$K_B$</td>
<td>Bob’s secret key</td>
</tr>
<tr>
<td>$K_{AB}$</td>
<td>Secret key shared between Alice and Bob</td>
</tr>
<tr>
<td>$K_{Apriv}$</td>
<td>Alice’s private key (known only to Alice)</td>
</tr>
<tr>
<td>$K_{Apub}$</td>
<td>Alice’s public key (published by Alice for all to read)</td>
</tr>
<tr>
<td>${M}_K$</td>
<td>Message $M$ encrypted with key $K$</td>
</tr>
<tr>
<td>$[M]_K$</td>
<td>Message $M$ signed with key $K$</td>
</tr>
</tbody>
</table>

**Figure 11.3.1 Cryptography notations**
11.1 CRYPTOGRAPHY

Encryption is the process of encoding a message in such a way as to hide its contents. Modern cryptography includes several secure algorithms for encrypting and decrypting messages. They are all based on the use of secrets called *keys*. A cryptographic key is a parameter used in an encryption algorithm in such a way that the encryption cannot be reversed without knowledge of the key.

There are two main classes of encryption algorithm in general use. The first uses *shared secret keys* – the sender and the recipient must share a knowledge of the key and it must not be revealed to anyone else. The second class of encryption algorithms uses *public/private key pairs*. Here the sender of a message uses a *public key* – one that has already been published by the recipient – to encrypt the message. The recipient uses a corresponding *private key* to decrypt the message. Although many principals may examine the public key, only the recipient can decrypt the message, because they have the private key.

Both classes of encryption algorithm are extremely useful and are used widely in the construction of secure distributed systems. Public-key encryption algorithms typically require 100 to 1000 times as much processing power as secret-key algorithms, but there are situations where their convenience outweighs this disadvantage.

**Uses of cryptography**

Cryptography plays three major roles in the implementation of secure systems. In all of our scenarios below, we can assume that Alice, Bob and any other participants have already agreed about the encryption algorithms that they wish to use and have implementations of them. We also assume that any secret keys or private keys that they hold can be stored securely to prevent attackers obtaining them.
Secrecy and integrity: Cryptography is used to maintain the secrecy and integrity of information whenever it is exposed to potential attacks – for example, during transmission across networks that are vulnerable to eavesdropping and message tampering. This use of cryptography corresponds to its traditional role in military and intelligence activities. It exploits the fact that a message that is encrypted with a particular encryption key can only be decrypted by a recipient who knows the corresponding decryption key. Thus it maintains the secrecy of the encrypted message as long as the decryption key is not compromised (disclosed to non-participants in the communication) and provided that the encryption algorithm is strong enough to defeat any possible attempts to crack it. Encryption also maintains the integrity of the encrypted information, provided that some redundant information such as a checksum is included and checked.

Scenario 1: Secret communication with a shared secret key: Alice wishes to send some information secretly to Bob. Alice and Bob share a secret key $K_{AB}$.

1. Alice uses $K_{AB}$ and an agreed encryption function $E(K_{AB}, M)$ to encrypt and send any number of messages $\{M_i\}_{K_{AB}}$ to Bob. (Alice can go on using $K_{AB}$ as long as it is safe to assume that $K_{AB}$ has not been compromised.)

2. Bob decrypts the encrypted messages using the corresponding decryption function $D(K_{AB}, M)$.

Bob can now read the original message $M$. If the decrypted message makes sense, or better, if it includes some value agreed between Alice and Bob (such as a checksum of the message) then Bob knows that the message is from Alice and that it hasn’t been tampered with. But there are still some problems:

Problem 1: How can Alice send a shared key $K_{AB}$ to Bob securely?
**Problem 2:** How does Bob know that any \( M_i \) isn’t a copy of an earlier encrypted message from Alice that was captured by Mallory and replayed later? Mallory needn’t have the key \( K_{AB} \) to carry out this attack – he can simply copy the bit pattern that represents the message and send it to Bob later. For example, if the message is a request to pay some money to someone, Mallory might trick Bob into paying twice.

**Authentication:** Cryptography is used in support of mechanisms for authenticating communication between pairs of principals. A principal who decrypts a message successfully using a particular key can assume that the message is authentic if it contains a correct checksum or some other expected value. They can infer that the sender of the message possessed the corresponding encryption key and hence deduce the identity of the sender if the key is known only to two parties. Thus if keys are held in private, a successful decryption authenticates the decrypted message as coming from a particular sender.

**Scenario 2:** Authenticated communication with a server: Alice wishes to access files held by Bob, a file server on the local network of the organization where she works. Sara is an authentication server that is securely managed. Sara issues users with passwords and holds current secret keys for all of the principals in the system it serves (generated by applying some transformation to the user’s password). For example, it knows Alice’s key \( K_A \) and Bob’s \( K_B \). In our scenario we refer to a ticket. A ticket is an encrypted item issued by an authentication server, containing the identity of the principal to whom it is issued and a shared key that has been generated for the current communication session.

1. Alice sends an (unencrypted) message to Sara stating her identity and requesting a ticket for access to Bob.

2. Sara sends a response to Alice encrypted in \( K_A \) consisting of a ticket (to be sent to Bob with each request for file access) encrypted in \( K_B \) and a new secret key \( K_{AB} \) for use when communicating with Bob. So the response that Alice receives looks like this: \( \{ \{ Ticket \} K_B, K_{AB} \} K_A \).
3. Alice decrypts the response using $K_A$ (which she generates from her password using the same transformation; the password is not transmitted over the network, and once it has been used it is deleted from local storage to avoid compromising it). If Alice has the correct password-derived key $K_A$, she obtains a valid ticket for using Bob’s service and a new encryption key for use in communicating with Bob. Alice can’t decrypt or tamper with the ticket, because it is encrypted in $K_B$. If the recipient isn’t Alice then they won’t know Alice’s password, so they won’t be able to decrypt the message.

4. Alice sends the ticket to Bob together with her identity and a request $R$ to access a file: \{Ticket\}$K_B$, Alice, $R$.

5. The ticket, originally created by Sara, is actually: \{$K_{AB}$, Alice\}$K_B$. Bob decrypts the ticket using his key $K_B$. So Bob gets the authentic identity of Alice (based on the knowledge shared between Alice and Sara of Alice’s password) and a new shared secret key $K_{AB}$ for use when interacting with Alice. (This is called a session key because it can safely be used by Alice and Bob for a sequence of interactions.)

This scenario is a simplified version of the authentication protocol originally developed by Roger Needham and Michael Schroeder and subsequently used in the Kerberos system developed and used at MIT. In our simplified description of their protocol above there is no protection against the replay of old authentication messages.

The authentication protocol we have described depends upon prior knowledge by the authentication server Sara of Alice’s and Bob’s keys, $K_A$ and $K_B$. This is feasible in a single organization where Sara runs on a physically secure computer and is managed by a trusted principal who generates initial values of the keys and transmits them to users by a separate secure channel. But it isn’t appropriate for electronic commerce or other wide area applications, where the use of a separate channel is extremely inconvenient and the requirement for a trusted third party is unrealistic. Public-key cryptography rescues us from this dilemma.
The usefulness of challenges: An important aspect of Needham and Schroeder’s 1978 breakthrough was the realization that a user’s password does not have to be submitted to an authentication service (and hence exposed in the network) each time it is authenticated. Instead, they introduced the concept of a cryptographic challenge. This can be seen in step 2 of our scenario above, where the server, Sara, issues a ticket to Alice encrypted in Alice’s secret key, $K_A$. This constitutes a challenge because Alice cannot make use of the ticket unless she can decrypt it, and she can only decrypt it if she can determine $K_A$, which is derived from Alice’s password. An imposter claiming to be Alice would be defeated at this point.

**Scenario 3:** Authenticated communication with public keys: Assuming that Bob has generated a public/private key pair, the following dialogue enables Bob and Alice to establish a shared secret key, $K_{AB}$:

1. Alice accesses a key distribution service to obtain a public-key certificate giving Bob’s public key. It’s called a certificate because it is signed by a trusted authority – a person or organization that is widely known to be reliable. After checking the signature, she reads Bob’s public key, $K_{Bpub}$, from the certificate.

2. Alice creates a new shared key, $K_{AB}$, and encrypts it using $K_{Bpub}$ with a public key algorithm. She sends the result to Bob, along with with a name that uniquely identifies a public/private key pair (since Bob may have several of them) – that is, Alice sends keyname, $\{K_{AB}\}K_{Bpub}$.

3. Bob selects the corresponding private key, $K_{Bpriv}$, from his private key store and uses it to decrypt $K_{AB}$. Note that Alice’s message to Bob might have been corrupted or tampered with in transit. The consequence would simply be that Bob and Alice don’t share the same key $K_{AB}$. If this is a problem, it can be circumvented by adding an agreed value or string to the message, such as Bob’s and Alice’s names or email addresses, which Bob can check after decrypting.
The above scenario illustrates the use of public-key cryptography to distribute a shared secret key. This technique is known as a hybrid cryptographic protocol and is very widely used, since it exploits useful features of both public-key and secret-key encryption algorithms.

**Problem:** This key exchange is vulnerable to man-in-the-middle attacks. Mallory may intercept Alice’s initial request to the key distribution service for Bob’s public key certificate and send a response containing his own public key. He can then intercept all the subsequent messages. In our description above, we guard against this attack by requiring Bob’s certificate to be signed by a well-known authority. To protect against this attack, Alice must ensure that Bob’s public-key certificate is signed with a public key (as described below) that she has received in a totally secure manner.

**Digital signatures:** Cryptography is used to implement a mechanism known as a digital signature. This emulates the role of a conventional signature, verifying to a third party that a message or a document is an unaltered copy of one produced by the signer.

Digital signature techniques are based upon an irreversible binding to the message or document of a secret known only to the signer. This can be achieved by encrypting the message – or better, a compressed form of the message called a digest – using a key that is known only to the signer. A digest is a fixed-length value computed by applying a secure digest function. A secure digest function is similar to a checksum function, but it is very unlikely to produce a similar digest value for two different messages. The resulting encrypted digest acts as a signature that accompanies the message. Public-key cryptography is generally used for this: the originator generates a signature with their private key, and the signature can be decrypted by any recipient using the corresponding public key. There is an additional requirement: the verifier should be sure that the public key really is that of the principal claiming to be the signer.
1. **Certificate type:** Account number  
2. **Name:** Alice  
3. **Account:** 6262626  
4. **Certifying authority:** Bob’s Bank  
5. **Signature:** \{Digest(field 2 + field 3)\}KBpriv

**Figure 11.3.3:** Alice’s bank account certificate

**Scenario 4:** Digital signatures with a secure digest function: Alice wants to sign a document \( M \) so that any subsequent recipient can verify that she is the originator of it. Thus when Bob later accesses the signed document after receiving it by any route and from any source (for example, it could be sent in a message or it could be retrieved from a database), he can verify that Alice is the originator.

1. Alice computes a fixed-length digest of the document, \( \text{Digest}(M) \).
2. Alice encrypts the digest in her private key, appends it to \( M \) and makes the result, \( M, \{\text{Digest}(M)\}K_{A priv} \), available to the intended users.
3. Bob obtains the signed document, extracts \( M \) and computes \( \text{Digest}(M) \).
4. Bob decrypts \( \{\text{Digest}(M)\}K_{A priv} \) using Alice’s public key, \( K_{A pub} \), and compares the result with his calculated \( \text{Digest}(M) \). If they match, the signature is valid.

**Certificates**

A digital certificate is a document containing a statement (usually short) signed by a principal. We illustrate the concept with a scenario.

**Scenario 5:** The use of certificates: Bob is a bank. When his customers establish contact with him they need to be sure that they are talking to Bob the bank, even if they have never contacted him before. Bob needs to authenticate his customers before he gives them access to their accounts.
For example, Alice might find it useful to obtain a certificate from her bank stating her bank account number (Figure 3.3.2). Alice could use this certificate when shopping to certify that she has an account with Bob’s Bank. The certificate is signed using Bob’s private key, $K_{B_{priv}}$. A vendor, Carol, can accept such a certificate for charging items to Alice’s account provided that she can validate the signature in field 5. To do so, Carol needs to have Bob’s public key and she needs to be sure that it is authentic to guard against the possibility that Alice might sign a false certificate associating her name with someone else’s account. To carry out this attack, Alice would simply generate a new key pair, $K_{B_{pub}'}$, $K_{B_{priv}'}$, and use them to generate a forged certificate purporting to come from Bob’s Bank.

1. **Certificate type:** Public key
2. **Name:** Bob’s Bank
3. **Public key:** $K_{B_{pub}}$
4. **Certifying authority:** Fred – The Bankers Federation
5. **Signature:** $\{\text{Digest(field 2 + field 3)}\} K_{F_{priv}}$

**Figure 11.3.4 Public-key certificate for Bob’s Bank**

What Carol needs is a certificate stating Bob’s public key, signed by a well-known and trusted authority. Let us assume that Fred represents the Bankers Federation, one of whose roles is to certify the public keys of banks. Fred could issue a *public-key certificate* for Bob (Figure 3.3.4).

Of course, this certificate depends upon the authenticity of Fred’s public key, $K_{F_{pub}}$, so we have a recursive problem of authenticity – Carol can only rely on this certificate if she can be sure she knows Fred’s authentic public key, $K_{F_{pub}}$. We can break this recursion by ensuring that Carol obtains $K_{F_{pub}}$ by some means in which she can have confidence – she might be handed it by a representative of Fred or she might receive a signed copy of it from someone she knows and trusts who says they got it directly from Fred. Our example illustrates a certification chain – one with two links, in this case.
We have already alluded to one of the problems arising with certificates – the difficulty of choosing a trusted authority from which a chain of authentications can start. Trust is seldom absolute, so the choice of an authority must depend upon the purpose to which the certificate is to be put. Other problems arise over the risk of private keys being compromised (disclosed) and the permissible length of a certification chain – the longer the chain, the greater the risk of a weak link.

Provided that care is taken to address these issues, chains of certificates are an important cornerstone for electronic commerce and other kinds of real-world transaction. They help to address the problem of scale: there are six billion people in the world, so how can we construct an electronic environment in which we can establish the credentials of any of them?

Certificates can be used to establish the authenticity of many types of statement. For example, the members of a group or association might wish to maintain an email list that is open only to members of the group. A good way to do this would be for the membership manager (Bob) to issue a membership certificate \((S,Bob,\{\text{Digest}(S)\}K_{Bpriv})\) to each member, where \(S\) is a statement of the form *Alice is a member of the Friendly Society* and \(K_{Bpriv}\) is Bob’s private key. A member applying to join the Friendly Society email list would have to supply a copy of this certificate to the list management system, which checks the certificate before allowing the member to join the list.

To make certificates useful, two things are needed:

- A standard format and representation for them so that certificate issuers and certificate users can successfully construct and interpret them;

- Agreement on the manner in which chains of certificates are constructed, and in particular the notion of a trusted authority.
There is sometimes a need to revoke a certificate – for example, Alice might discontinue her membership of the Friendly Society, but she and others would probably continue to hold stored copies of her membership certificate. It would be expensive, if not impossible, to track down and delete all such certificates, and it is not easy to invalidate a certificate – it would be necessary to notify all possible recipients of the revocation. The usual solution to this problem is to include an expiry date in the certificate. Anyone receiving an expired certificate should reject it, and the subject of the certificate must request its renewal. If a more rapid revocation is required, then one of the more cumbersome mechanisms mentioned above must be resorted to.

**Access control**

Here we outline the concepts on which the control of access to resources is based in distributed systems and the techniques by which it is implemented.

Historically, the protection of resources in distributed systems has been largely service-specific. Servers receive request messages of the form \(<op, principal, resource>\), where \(op\) is the requested operation, \(principal\) is an identity or a set of credentials for the principal making the request and \(resource\) identifies the resource to which the operation is to be applied. The server must first authenticate the request message and the principal’s credentials and then apply access control, refusing any request for which the requesting principal does not have the necessary access rights to perform the requested operation on the specified resource.

In object-oriented distributed systems there may be many types of object to which access control must be applied, and the decisions are often application-specific. For example, Alice may be allowed only one cash withdrawal from her bank account per day, while Bob is allowed three. Access control decisions are usually left to the application-level code, but generic support is provided for much of the machinery that supports the decisions. This includes the authentication of principals, the signing and authentication of requests, and the management of credentials and access rights data.
Protection domains: A protection domain is an execution environment shared by a collection of processes: it contains a set of <resource, rights> pairs, listing the resources that can be accessed by all processes executing within the domain and specifying the operations permitted on each resource. A protection domain is usually associated with a given principal – when a user logs in, their identity is authenticated and a protection domain is created for the processes that they will run. Conceptually, the domain includes all of the access rights that the principal possesses, including any rights that they acquire through membership of various groups. For example, in UNIX, the protection domain of a process is determined by the user and group identifiers attached to the process at login time. Rights are specified in terms of allowed operations. For example, a file might be readable and writable by one process and only readable by another.

A protection domain is only an abstraction. Two alternative implementations are commonly used in distributed systems: capabilities and access control lists.

Capabilities: A set of capabilities is held by each process according to the domain in which it is located. A capability is a binary value that acts as an access key, allowing the holder access to certain operations on a specified resource. For use in distributed systems, where capabilities must be unforgeable, they take a form such as:

<table>
<thead>
<tr>
<th>Resource identifier</th>
<th>A unique identifier for the target resource</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operations</td>
<td>A list of the operations permitted on the resource</td>
</tr>
<tr>
<td>Authentication code</td>
<td>A digital signature making the capability unforgeable</td>
</tr>
</tbody>
</table>

Services only supply capabilities to clients when they have authenticated them as belonging to the claimed protection domain. The list of operations in the capability is a subset of the operations defined for the target resource and is often encoded as a bit map.
Different capabilities are used for different combinations of access rights to the same resource.

When capabilities are used, client requests are of the form \(<op, userid, capability>\). That is, they include a capability for the resource to be accessed instead of a simple identifier, giving the server immediate proof that the client is authorized to access the resource identified by the capability with the operations specified by the capability. An access-control check on a request that is accompanied by a capability involves only the validation of the capability and a check that the requested operation is in the set permitted by the capability. This feature is the major advantage of capabilities – they constitute a self-contained access key, just as a physical key to a door lock is an access key to the building that the lock protects.

Capabilities share two drawbacks of keys to a physical lock:

**Key theft:** Anyone who holds the key to a building can use it to gain access, whether or not they are an authorized holder of the key – they may have stolen the key or obtained it in some fraudulent manner.

**The revocation problem:** The entitlement to hold a key changes with time. For example, the holder may cease to be an employee of the owner of the building, but they might retain the key, or a copy of it, and use it in an unauthorized manner.

The only available solutions to these problems for physical keys are (a) to put the illicit key holder in jail – not always feasible on a timescale that will prevent them doing damage – or (b) to change the lock and reissue keys to all key holders – a clumsy and expensive operation.

The analogous problems for capabilities are clear:
• Capabilities may, through carelessness or as a result of an eavesdropping attack, fall into the hands of principals other than those to whom they were issued. If this happens, servers are powerless to prevent them being used illicitly.

• It is difficult to cancel capabilities. The status of the holder may change and their access rights should change accordingly, but they can still use their capabilities.

Solutions to both of these problems, based on the inclusion of information identifying the holder and on timeouts plus lists of revoked capabilities, have been proposed and developed. Although they add complexity to an otherwise simple concept, capabilities remain an important technique – for example, they can be used in conjunction with access control lists to optimize access control on repeated access to the same resource, and they provide the neatest mechanism for the implementation of delegation.

It is interesting to note the similarity between capabilities and certificates. Consider Alice’s certificate of ownership of her bank account introduced. It differs from capabilities as described here only in that there is no list of permitted operations and that the issuer is identified. Certificates and capabilities may be interchangeable concepts in some circumstances. Alice’s certificate might be regarded as an access key allowing her to perform all the operations permitted to account holders on her bank account, provided her identity can be proven.

**Access control lists:** A list is stored with each resource, containing an entry of the form `<domain, operations>` for each domain that has access to the resource and giving the operations permitted to the domain. A domain may be specified by an identifier for a principal or it may be an expression that can be used to determine a principal’s membership of the domain. For example, *the owner of this file* is an expression that can be evaluated by comparing the requesting principal’s identity with the owner’s identity stored with a file.
This is the scheme adopted in most file systems, including UNIX and Windows NT, where a set of access permission bits is associated with each file, and the domains to which the permissions are granted are defined by reference to the ownership information stored with each file.

Requests to servers are of the form \(<op, principal, resource>\). For each request, the server authenticates the principal and checks to see that the requested operation is included in the principal’s entry in the access control list of the relevant resource.

**Implementation:** Digital signatures, credentials and public-key certificates provide the cryptographic basis for secure access control. Secure channels offer performance benefits, enabling multiple requests to be handled without a need for repeated checking of principals and credentials.

Both CORBA and Java offer Security APIs. Support for access control is one of their major purposes. Java provides support for distributed objects to manage their own access control with *Principal*, *Signer* and *ACL* classes and default methods for authentication and support for certificates, signature validation and access-control checks. Secret-key and public-key cryptography are also supported. Farley provides a good introduction to these features of Java. The protection of Java programs that include mobile code is based upon the protection domain concept – local code and downloaded code are provided with different protection domains in which to execute. There can be a protection domain for each download source, with access rights for different sets of local resources depending upon the level of trust that is placed in the downloaded code.

**Credentials**

Credentials are a set of evidence provided by a principal when requesting access to a resource. In the simplest case, a certificate from a relevant authority stating the principal’s identity is sufficient, and this would be used to check the principal’s
permissions in an access control list. This is often all that is required or provided, but the concept can be generalized to deal with many more subtle requirements.

It is not convenient to require users to interact with the system and authenticate themselves each time their authority is required to perform an operation on a protected resource. Instead, the notion that a credential *speaks for* a principal is introduced. Thus a user’s public-key certificate speaks for that user – any process receiving a request authenticated with the user’s private key can assume that the request was issued by that user.

The *speaks for* idea can be carried much further. For example, in a cooperative task, it might be required that certain sensitive actions should only be performed with the authority of two members of the team; in that case, the principal requesting the action would submit their own identifying credential and a backing credential from another member of the team, together with an indication that they are to be taken together when checking the credentials.

Similarly, to vote in an election, a vote request would be accompanied by an elector certificate as well as an identifying certificate. A delegation certificate allows a principal to act on behalf of another, and so on. In general, an access-control check involves the evaluation of a logical formula combining the certificates supplied. Lampson *et al.* have developed a comprehensive logic of authentication for use in evaluating the *speaks for* authority carried by a set of credentials. Wobber *et al.* describe a system that supports this very general approach. Further work on useful forms of credential for use in real-world cooperative tasks can be found in Rowley.

Role-based credentials seem particularly useful in the design of practical access control schemes. Sets of role-based credentials are defined for organizations or for cooperative tasks, and application-level access rights are constructed with reference to them. Roles can then be assigned to specific principals by the generation of role certificates associating principals with named roles in specific tasks or organizations.
**Delegation:** A particularly useful form of credential is one that entitles a principal, or a process acting for a principal, to perform an action with the authority of another principal. A need for delegation can arise in any situation where a service needs to access a protected resource in order to complete an action on behalf of its client. Consider the example of a print server that accepts requests to print files. It would be wasteful of resources to copy the file, so the name of the file is passed to the print server and it is accessed by the print server on behalf of the user making the request. If the file is read-protected, this does not work unless the print server can acquire temporary rights to read the file. Delegation is a mechanism designed to solve problems such as this.

Delegation can be achieved using a delegation certificate or a capability. The certificate is signed by the requesting principal and it authorizes another principal (the print server in our example) to access a named resource (the file to be printed). In systems that support them, capabilities can achieve the same result without the need to identify the principals – a capability to access a resource can be passed in a request to a server. The capability is an unforgeable, encoded set of rights to access the resource.

When rights are delegated, it is common to restrict them to a subset of the rights held by the issuing principal, so that the delegated principal cannot misuse them. In our example, the certificate could be time-limited to reduce the risk of the print server’s code subsequently being compromised and the file disclosed to third parties. The CORBA Security Service includes a mechanism for the delegation of rights based on certificates, with support for the restriction of the rights carried.

**Firewalls**

Firewalls protect intranets, performing filtering actions on incoming and outgoing communications. Here we discuss their advantages and drawbacks as security mechanisms.
In an ideal world, communication would always be between mutually trusting processes and secure channels would always be used. There are many reasons why this ideal is not attainable, some fixable, but others inherent in the open nature of distributed systems or resulting from the errors that are present in most software. The ease with which request messages can be sent to any server, anywhere, and the fact that many servers are not designed to withstand malicious attacks from hackers or accidental errors, makes it easy for information that is intended to be confidential to leak out of the owning organization’s servers. Undesirable items can also penetrate an organization’s network, allowing worm programs and viruses to enter its computers.

Firewalls produce a local communication environment in which all external communication is intercepted. Messages are forwarded to the intended local recipient only for communications that are explicitly authorized.

Access to internal networks may be controlled by firewalls, but access to public services on the Internet is unrestricted because their purpose is to offer services to a wide range of users. The use of firewalls offers no protection against attacks from inside an organization, and it is crude in its control of external access. There is a need for finer-grained security mechanisms, enabling individual users to share information with selected others without compromising privacy and integrity. Abadi et al. describe an approach to the provision of access to private web data for external users based on a web tunnel mechanism that can be integrated with a firewall. It offers access for trusted and authenticated users to internal web servers via a secure proxy based on the HTTPS (HTTP over TLS) protocol.

Firewalls are not particularly effective against denial-of-service attacks such as the one based on IP spoofing that was outlined. The problem is that the flood of messages generated by such attacks overwhelms any single point of defence such as a firewall. Any remedy for incoming floods of messages must be applied well upstream of the target. Remedies based on the use of quality of service mechanisms to restrict the flow of messages from the network to a level that the target can handle seem the most promising.
Cryptographic algorithms

A message is encrypted by the sender applying some rule to transform the plaintext message (any sequence of bits) to a ciphertext (a different sequence of bits). The recipient must know the inverse rule in order to transform the ciphertext back into the original plaintext. Other principals are unable to decipher the message unless they also know the inverse rule. The encryption transformation is defined with two parts, a function \( E \) and a key \( K \). The resulting encrypted message is written \( \{ M \}_K \).

\[
E(K, M) = \{ M \}_K
\]

The encryption function \( E \) defines an algorithm that transforms data items in plaintext into encrypted data items by combining them with the key and transposing them in a manner that is heavily dependent on the value of the key. We can think of an encryption algorithm as the specification of a large family of functions from which a particular member is selected by any given key. Decryption is carried out using an inverse function \( D \), which also takes a key as a parameter. For secret-key encryption, the key used for decryption is the same as that used for encryption:

\[
D(K, E(K, M)) = M
\]

Because of its symmetrical use of keys, secret-key cryptography is often referred to as symmetric cryptography, whereas public-key cryptography is referred to as asymmetric because the keys used for encryption and decryption are different, as we shall see below. In the next section, we describe several widely used encryption functions of both types.

**Symmetric algorithms:** If we remove the key parameter from consideration by defining \( F_K ([M]) = E(K, M) \), then it is a property of strong encryption functions that \( F_K ([M]) \) is relatively easy to compute, whereas the inverse, \( F_K^{-1} ([M]) \), is so hard to compute that
it is not feasible. Such functions are known as one-way functions. The effectiveness of any method for encrypting information depends upon the use of an encryption function $FK$ that has this one-way property. It is this that protects against attacks designed to discover $M$ given $\{M\}_K$.

For well-designed symmetric algorithms such as those described in the next section, their strength against attempts to discover $K$ given a plaintext $M$ and the corresponding ciphertext $\{M\}_K$ depends on the size of $K$. This is because the most effective general form of attack is the crudest, known as a brute-force attack. The brute-force approach is to run through all possible values of $K$, computing $E(K, M)$ until the result matches the value of $\{M\}_K$ that is already known. If $K$ has $N$ bits then such an attack requires $2^{N-1}$ iterations on average, and a maximum of $2^N$ iterations, to find $K$. Hence the time to crack $K$ is exponential in the number of bits in $K$.

**Asymmetric algorithms:** When a public/private key pair is used, one-way functions are exploited in another way. The feasibility of a public-key scheme was first proposed by Diffie and Hellman as a cryptographic method that eliminates the need for trust between the communicating parties. The basis for all public-key schemes is the existence of trapdoor functions. A trap-door function is a one-way function with a secret exit – it is easy to compute in one direction but infeasible to compute the inverse unless a secret is known. It was the possibility of finding such functions and using them in practical cryptography that Diffie and Hellman first suggested. Since then, several practical public-key schemes have been proposed and developed. They all depend upon the use of trapdoor functions involving large numbers.

![Cipher block chaining](Figure 11.3.4: Cipher block chaining)
The pair of keys needed for asymmetric algorithms is derived from a common root. For the RSA algorithm, the root is an arbitrarily chosen pair of very large prime numbers. The derivation of the pair of keys from the root is a one-way function. In the case of the RSA algorithm, the large primes are multiplied together – a computation that takes only a few seconds, even for the very large primes used. The resulting product, $N$, is of course much larger than the multiplicands. This use of multiplication is a one-way function in the sense that it is computationally infeasible to derive the original multiplicands from the product – that is, to factorize the product.

One of the pair of keys is used for encryption. For RSA, the encryption function obscures the plaintext by treating each block of bits as a binary number and raising it to the power of the key, modulo $N$. The resulting number is the corresponding ciphertext block.

The size of $N$ and at least one of the pair of keys is much larger than the safe key size for symmetric keys to ensure that $N$ is not factorizable. For this reason, the potential for brute-force attacks on RSA is small; its resistance to attacks depends on the infeasibility of factorizing $N$.

**Block ciphers:** Most encryption algorithms operate on fixed-size blocks of data; 64 bits is a popular size for the blocks. A message is subdivided into blocks, the last block is padded to the standard length if necessary and each block is encrypted independently. The first block is available for transmission as soon as it has been encrypted.

For a simple block cipher, the value of each block of ciphertext does not depend upon the preceding blocks. This constitutes a weakness, since an attacker can recognize repeated patterns and infer their relationship to the plaintext. Nor is the integrity of messages guaranteed unless a checksum or secure digest mechanism is used. Most block cipher algorithms employ cipher block chaining (CBC) to overcome these weaknesses.
Cipher block chaining: In cipher block chaining mode, each plaintext block is combined with the preceding ciphertext block using the exclusive-or operation (XOR) before it is encrypted. On decryption, the block is decrypted and then the preceding encrypted block (which should have been stored for this purpose) is XOR-ed with it to obtain the new plaintext block. This works because the XOR operation is its own inverse – two applications of it produce the original value.

CBC is intended to prevent identical portions of plaintext encrypting to identical pieces of ciphertext. But there is a weakness at the start of each sequence of blocks – if we open encrypted connections to two destinations and send the same message, the encrypted sequences of blocks will be the same, and an eavesdropper might gain some useful information from this. To prevent this, we need to insert a different piece of plaintext in front of each message. Such text is called an *initialization vector*. A timestamp makes a good initialization vector, forcing each message to start with a different plaintext block. This, combined with CBC operation, will result in different ciphertexts even for two identical plaintexts.

![Figure 11.3.5: Stream cipher](image)

The use of CBC mode is restricted to the encryption of data that is transferred across a reliable connection. Decryption will fail if any blocks of ciphertext are lost, since the decryption process will be unable to decrypt any further blocks. It is therefore unsuitable for use in applications, in which some data loss can be tolerated. A stream cipher should be used in such circumstances.
**Stream ciphers:** For some applications, such as the encryption of telephone conversations, encryption in blocks is inappropriate because the data streams are produced in real time in small chunks. Data samples can be as small as 8 bits or even a single bit, and it would be wasteful to pad each of these to 64 bits before encrypting and transmitting them. Stream ciphers are encryption algorithms that can perform encryption incrementally, converting plaintext to ciphertext one bit at a time.

This sounds difficult to achieve, but in fact it is very simple to convert a block cipher algorithm for use as a stream cipher. The trick is to construct a *keystream generator*. A keystream is an arbitrary-length sequence of bits that can be used to obscure the contents of a data stream by XOR-ing the keystream with the data stream (Figure 3.3.5). If the keystream is secure, then so is the resulting encrypted data stream.

The idea is analogous to a technique used in the intelligence community to foil eavesdroppers, where ‘white noise’ is played to hide the conversation in a room while still recording the conversation. If the noisy room sound and the white noise are recorded separately, the conversation can be played back without noise by subtracting the white noise recording from the noisy room recording.

A keystream generator can be constructed by iterating a mathematical function over a range of input values to produce a continuous stream of output values. The output values are then concatenated to make plaintext blocks, and the blocks are encrypted using a key shared by the sender and the receiver. The keystream can be further disguised by applying CBC. The resulting encrypted blocks are used as the keystream. An iteration of almost any function that delivers a range of different non-integer values will do for the source material, but a random number generator is generally used with a starting value for the iteration agreed between the sender and receiver. To maintain quality of service for the data stream, the keystream blocks should be produced a little ahead of the time at which they will be used, and the process that produces them should not demand so much processing effort that the data stream is delayed.
Thus in principle, real-time data streams can be encrypted just as securely as batched data, provided that sufficient processing power is available to encrypt the keystream in real time. Of course, some devices that could benefit from real-time encryption, such as mobile phones, are not equipped with very powerful processors, and in that case it may be necessary to reduce the security of the keystream algorithm.

**Design of cryptographic algorithms:** There are many well-designed cryptographic algorithms such that $E(K, M) = \{M\}^K$ conceals the value of $M$ and makes it practically impossible to retrieve $K$ more quickly than by brute force. All encryption algorithms rely on information-preserving manipulations of $M$ using principles based on information theory. Schneier describes Shannon’s principles of *confusion* and *diffusion* to conceal the content of a ciphertext block $M$, combining it with a key $K$ of sufficient size to render it proof against brute-force attacks.

**Confusion:** Non-destructive operations such as XOR and circular shifting are used to combine each block of plaintext with the key, producing a new bit pattern that obscures the relationship between the blocks in $M$ and $\{M\}^K$. If the blocks are larger than a few characters this will defeat analysis based on a knowledge of character frequencies.

**Diffusion:** There is usually repetition and redundancy in the plaintext. Diffusion dissipates the regular patterns that result by transposing portions of each plaintext block. If CBC is used, the redundancy is also distributed throughout a longer text. Stream ciphers cannot use diffusion since there are no blocks.

In the next two sections, we describe the design of several important practical algorithms. All of them have been designed in the light of the above principles have been subject to rigorous analysis and are considered to be secure against all known attacks with a considerable margin of safety. With the exception of the TEA algorithm, which is described for illustrative purposes, the algorithms described here are among those most
widely used in applications where strong security is required. In some of them there remain some minor weaknesses or areas of concern.

**Secret-key (symmetric) algorithms**

Many cryptographic algorithms have been developed and published in recent years. Schneier describes more than 25 symmetric algorithms, many of which he identifies as secure against known attacks. Here we have room to describe only three of them. We have chosen the first, TEA, for the simplicity of its design and implementation. We go on to discuss briefly about the DES and IDEA algorithms. DES was a US national standard for many years, but it is now largely of historical interest because its 56-bit keys are too small to resist brute-force attack with modern hardware. IDEA uses a 128-bit key. It is one of the most effective symmetric block encryption algorithms and a good all-round choice for bulk encryption.

```c
void encrypt(unsigned long k[], unsigned long text[]) {
    unsigned long y = text[0], z = text[1];
    unsigned long delta = 0x9e3779b9, sum = 0; int n;
    for (n= 0; n < 32; n++) {
        sum += delta;
        y += ((z << 4) + k[0]) ^ (z+sum) ^ ((z >> 5) + k[1]);
        z += ((y << 4) + k[2]) ^ (y+sum) ^ ((y >> 5) + k[3]);
    }
    text[0] = y; text[1] = z;
}
```

Figure 11.3.6: TEA encryption function
**TEA:** The design principles for symmetric algorithms outlined above are illustrated well in the Tiny Encryption Algorithm (TEA) developed at Cambridge University. The encryption function, programmed in C, is given in its entirety in Figure 3.3.6.

The TEA algorithm uses rounds of integer addition, XOR (the ^ operator) and bitwise logical shifts (<< and >>) to achieve diffusion and confusion of the bit patterns in the plaintext. The plaintext is a 64-bit block represented as two 32-bit integers in the vector `text[]`. The key is 128 bits long, represented as four 32-bit integers.

On each of the 32 rounds, the two halves of the text are repeatedly combined with shifted portions of the key and each other in lines 5 and 6. The use of XOR and shifted portions of the text provides confusion, and the shifting and swapping of the two portions of the text provides diffusion. The non-repeating constant `delta` is combined with each portion of the text on each cycle to obscure the key in case it might be revealed by a section of text that does not vary. The decryption function is the inverse of that for encryption and is given in Figure 11.8.

This short program provides secure and reasonably fast secret-key encryption. It is somewhat faster than the DES algorithm, and the conciseness of the program lends itself to optimization and hardware implementation. The 128-bit key is secure against brute-force attacks. Studies by its authors and others have revealed only two very minor weaknesses, which the authors addressed in a subsequent note.

```c
void decrypt(unsigned long k[], unsigned long text[]) {
    unsigned long y = text[0], z = text[1];
    unsigned long delta = 0x9e3779b9, sum = delta << 5; int n;
    for (n= 0; n < 32; n++) {
        z -= ((y << 4) + k[2]) ^ (y + sum) ^ ((y >> 5) + k[3]);
        y -= ((z << 4) + k[0]) ^ (z + sum) ^ ((z >> 5) + k[1]);
        sum -= delta;
    }
}
```
void tea(char mode, FILE *infile, FILE *outfile, unsigned long k[]) {
    /* mode is 'e' for encrypt, 'd' for decrypt, k[] is the key. */
    char ch, Text[8]; int i;
    while(!feof(infile)) {
        i = fread(Text, 1, 8, infile); /* read 8 bytes from infile into Text */
        if (i <= 0) break;
        while (i < 8) { Text[i++] = ' ';} /* pad last block with spaces */
        switch (mode) {
            case 'e':
                text[0] = y; text[1] = z;
        }
    }

Figure 11.3.7: TEA decryption function

To illustrate its use Figure 3.3.8 shows a simple procedure that uses TEA to encrypt and decrypt a pair of previously opened files (using the C stdio library).

DES: The Data Encryption Standard (DES) was developed by IBM and subsequently adopted as a US national standard for government and business applications. In this standard, the encryption function maps a 64-bit plaintext input into a 64-bit encrypted output using a 56-bit key. The algorithm has 16 key-dependent stages known as rounds, in which the data to be encrypted is bit-rotated by a number of bits determined by the key and three key-independent transpositions.

The algorithm was time-consuming to perform in software on the computers of the 1970s and 1980s, but it was implemented in fast VLSI hardware and can easily be incorporated into network interface and other communication chips.
encrypt(k, (unsigned long*) Text); break;

    case 'd':
        decrypt(k, (unsigned long*) Text); break;
    }

fwrite(Text, 1, 8, outfile); /* write 8 bytes from Text to
outfile */

Figure 11.3.8: TEA in use

The client program was aimed at cracking the particular key used in a known plaintext/ciphertext sample and then using it to decrypt a secret challenge message. The clients interacted with a single server that coordinated their work, issuing each client with ranges of key values to check and receiving progress reports from them. The typical client computer ran the client program only as a background activity and had a performance approximately equal to a 200 MHz Pentium processor. The key was cracked in about 12 weeks, after approximately 25% of the possible $2^{56}$ or $6 \times 10^{16}$ values had been checked. In 1998 a machine was developed by the Electronic Frontier Foundation [EFF 1998] that can successfully crack DES keys in around three days.

Although it is still used in many commercial and other applications, DES in its basic form should be considered obsolete for the protection of all but low-value information. A solution that is frequently used is known as triple-DES (or 3DES). This involves applying DES three times with two keys, $K_1$ and $K_2$:

$$E_{3DES}(K_1, K_2, M) = E_{DES}(K_1, D_{DES}(K_2, E_{DES}(K_1, M)))$$
This gives a strength against brute-force attacks equivalent to a key length of 112 bits – adequate for the foreseeable future – but it has the drawback of poor performance resulting from the triple application of an algorithm that is already slow by modern standards.

**IDEA:** The International Data Encryption Algorithm (IDEA) was developed in the early 1990s as a successor to DES. Like TEA, it uses a 128-bit key to encrypt 64-bit blocks. Its algorithm is based on the algebra of groups and has eight rounds of XOR, addition modulo 216 and multiplication. For both DES and IDEA, the same function is used for encryption and decryption: a useful property for algorithms that are to be implemented in hardware. The strength of IDEA has been extensively analyzed, and no significant weaknesses have been found. It performs encryption and decryption at approximately three times the speed of DES.

**RC4:** RC4 is a stream cipher developed by Ronald Rivest. Keys can be of any length up to 256 bytes. RC4 is easy to implement and performs encryption and decryption about 10 times as fast as DES. It was therefore widely adopted in applications including IEEE 802.11 WiFi networks, but a weakness was subsequently discovered by Fluhrer *et al.* that enabled attackers to crack some keys. This led to a redesign of 802.11 security.

**AES:** The Rijndael algorithm selected to become the Advanced Encryption Standard algorithm by NIST was developed by Joan Daemen and Vincent Rijmen. The cipher has a variable block length and key length, with specifications for keys with a length of 128, 192 or 256 bits to encrypt blocks with a length of 128, 192 or 256 bits. Both block length and key length can be extended by multiples of 32 bits. The number of rounds in the algorithm varies from 9 to 13 depending on the key and block sizes. Rijndael can be implemented efficiently on a wide range of processors and in hardware.

**Public-key (asymmetric) algorithms**
Only a few practical public-key schemes have been developed to date. They depend upon the use of trap-door functions of large numbers to produce the keys. The keys \( K_e \) and \( K_d \) are a pair of very large numbers, and the encryption function performs an operation, such as exponentiation on \( M \), using one of them. Decryption is a similar function using the other key. If the exponentiation uses modular arithmetic, it can be shown that the result is the same as the original value of \( M \); that is:

\[
D(K_d, E(K_e, M)) = M
\]

A principal wishing to participate in secure communication with others makes a pair of keys, \( K K_e \) and \( K_d \), and keeps the decryption key \( K_d \) a secret. The encryption key \( K_e \) can be made known publicly for use by anyone who wants to communicate. The encryption key \( K_e \) can be seen as a part of the one-way encryption function \( E \), and the decryption key \( K_d \) is the piece of secret knowledge that enables principal \( p \) to reverse the encryption. Any holder of \( K_e \) (which is widely available) can encrypt messages \( \{M\} K_e \), but only the principal who has the secret \( K_d \) can operate the trapdoor.

The use of functions of large numbers leads to large processing costs in computing the functions \( E \) and \( D \). We shall see later that this is a problem that has to be addressed by the use of public keys only in the initial stages of secure communication sessions. The RSA algorithm is certainly the most widely known public-key algorithm and we describe it in some detail here. Another class of algorithms is based on functions derived from the behaviour of elliptic curves in a plane. These algorithms offer the possibility of less costly encryption and decryption functions with the same level of security, but their practical application is less advanced and we deal with them only briefly.

**RSA:** The Rivest, Shamir and Adelman (RSA) design for a public-key cipher is based on the use of the product of two very large prime numbers (greater than 10100), relying on the fact that the determination of the prime factors of such large numbers is so computationally difficult as to be effectively impossible.
Despite extensive investigations no flaws have been found in it, and it is now very widely used. An outline of the method follows. To find a key pair \( <e,d> \):

1. Choose two large prime numbers, \( P \) and \( Q \) (each greater than 10100), and form
   \[
   N = P \times Q \\
   Z = (P-1) \times (Q-1)
   \]
2. For \( d \), choose any number that is relatively prime with \( Z \) (that is, such that \( d \) has no common factors with \( Z \)).

We illustrate the computations involved using small integer values for \( P \) and \( Q \):

\[
P = 13, \quad Q = 17 \rightarrow N = 221, \quad Z = 192
\]

\( d = 5 \)

3. To find \( e \), solve the equation:

\[
e \times d = 1 \mod Z
\]

That is, \( e \times d \) is the smallest element divisible by \( d \) in the series \( Z+1, 2Z+1, 3Z+1, \ldots \).

\[
e \times d = 1 \mod 192 = 1, 193, 385, \ldots
\]

385 is divisible by \( d \)

\[
e = 385/5 = 77
\]

To encrypt text using the RSA method, the plaintext is divided into equal blocks of length \( k \) bits, where \( 2^k < N \) (that is, such that the numerical value of a block is always less than \( N \); in practical applications, \( k \) is usually in the range 512 to 1024).

\[
k = 7, \quad \text{since } 2^7 = 128
\]

The function for encrypting a single block of plaintext \( M \) is:

\[
E'(e,N,M) = M^e \mod N
\]
for a message $M$, the cipher text is $M^{77} \mod 221$

The function for decrypting a block of encrypted text $c$ to produce the original plaintext block is:

$$D'(d,N,c) = c^d \mod N$$

Rivest, Shamir and Adelman proved that $E'$ and $D'$ are mutual inverses (that is, $E'(D'(x)) = D'(E'(x)) = x$) for all values of $P$ in the range $0 \leq P \leq N$.

The two parameters $e,N$ can be regarded as a key for the encryption function, and similarly the parameters $d,N$ represent a key for the decryption function. So we can write $K_e = <e,N>$ and $K_d = <d,N>$, and we get the encryption functions $E(K_e, M) = \{M\}_K$ (the notation here indicating that the encrypted message can be decrypted only by the holder of the private key $K_d$) and $D(K_d, \{M\}_K) = M$.

It is worth noting one potential weakness of all public-key algorithms – because the public key is available to attackers, they can easily generate encrypted messages. Thus they can attempt to decrypt an unknown message by exhaustively encrypting arbitrary bit sequences until a match with the target message is achieved. This attack, which is known as a chosen plaintext attack, is defeated by ensuring that all messages are longer than the key length, so that this form of brute-force attack is less feasible than a direct attack on the key.

An intending recipient of secret information must publish or otherwise distribute the pair $<e,N>$ while keeping $d$ secret. The publication of $<e,N>$ does not compromise the secrecy of $d$, because any attempt to determine $d$ requires knowledge of the original prime numbers $P$ and $Q$, and these can only be obtained by the factorization of $N$.

Factoring of large numbers (we recall that $P$ and $Q$ were chosen to be $> 10^{100}$, so $N > 10^{200}$) is extremely time-consuming, even on very high-performance computers. In 1978,
Rivest et al. concluded that factoring a number as large as 10200 would take more than four billion years with the best known algorithm on a computer that performs one million instructions per second. A similar calculation for today’s computers would reduce this time to around a million years.

The above strength calculations assume that the currently known factoring algorithms are the best available. RSA and other forms of asymmetric cryptography that use prime number multiplication as their one-way function will be vulnerable if a faster factorization algorithm is discovered.

**Elliptic curve algorithms:** A method for generating public/private key pairs based on the properties of elliptic curves has been developed and tested. Full details can be found in the book by Menezes devoted to the subject. The keys are derived from a different branch of mathematics, and unlike RSA their security does not depend upon the difficulty of factoring large numbers. Shorter keys are secure, and the processing requirements for encryption and decryption are lower than those for RSA. Elliptic curve encryption algorithms are likely to be adopted more widely in the future, especially in systems such as those incorporating mobile devices, which have limited processing resources. The relevant mathematics involves some quite complex properties of elliptic curves and is beyond the scope of this book.

**Hybrid cryptographic protocols**

Public-key cryptography is convenient for electronic commerce because there is no need for a secure key-distribution mechanism. (There is a need to authenticate public keys, but this is much less onerous, requiring only a public-key certificate to be sent with the key.) But the processing costs of public-key cryptography are too high for the encryption of even the medium-sized messages normally encountered in electronic commerce. The solution adopted in most large-scale distributed systems is to use a hybrid encryption scheme in which public-key cryptography is used to authenticate the parties and to encrypt an exchange of secret keys, which are used for all subsequent communication.
11.6. SUMMARY

In this unit we introduced security techniques, uses of cryptography, access control, firewalls, symmetric and asymmetric cryptographic algorithms, block ciphers and stream ciphers, cryptographic algorithms.

11.7. KEYWORDS

Symmetric-key algorithms: are a class of algorithms for cryptography that use the same cryptographic keys for both encryption of plaintext and decryption of ciphertext.

Asymmetric cryptography (also known as Public-key cryptography): refers to a cryptographic algorithm which requires two separate keys, one of which is secret (or private) and one of which is public.

11.8. UNIT-END EXERCISES AND ANSWERS

5. List and explain worst-case assumptions and design guidelines of security techniques.
6. Explain the uses of cryptography.
7. Write a short note Firewalls.
8. Differentiate between Block ciphers and Stream ciphers.
9. Write TEA encryption function and decryption function.
10. Give an outline of the RSA public-key algorithm.

Answers: SEE

5. 11.2
6. 11.2
7. 11.2
8. 11.3
9. 11.3
10. 11.3
11.9 SUGGESTED READINGS

- DISTRIBUTED SYSTEMS Principles and Paradigm By: Andrew S. Tanenbaum, Maarten Van Steen.
UNIT 12: DIGITAL SIGNATURES

Structure:
12.0 Objectives
12.1 Introduction
12.2 Digital signatures
12.3 Cryptographic pragmatics
12.4 Summary
12.5 Keywords
12.6 Unit-end exercises and answers
12.7 Suggested readings

12.0 OBJECTIVES
At the end of this unit you will be able to know:
- Concept of digital signature
- Digital signing
- Secure digest functions
- Certificate standards authorities
- Performance of cryptographic algorithms

12.1 INTRODUCTION
In this unit we discuss about digital signatures. Strong digital signatures are an essential requirement for secure systems. They are needed in order to certify certain pieces of information – for example, to provide trustworthy statements binding users’ identities to their public keys or binding some access rights or roles to users’ identities.

The need for signatures in many kinds of business and personal transaction is beyond dispute. Handwritten signatures have been used as a means of verifying documents for as long as documents have existed. Handwritten signatures are used to meet the needs of document recipients to verify that the document is:
**Authentic:** It convinces the recipient that the signer deliberately signed the document and it has not been altered by anyone else.

**Unforgeable:** It provides proof that the signer, and no one else, deliberately signed the document. The signature cannot be copied and placed on another document.

**Non-repudiable:** The signer cannot credibly deny that the document was signed by them.

In reality, none of these desirable properties of signing is entirely achieved by conventional signatures – forgeries and copies are hard to detect, documents can be altered after signing and signers are sometimes deceived into signing a document involuntarily or unwittingly – but we are willing to live with their imperfection because of the difficulty of cheating and the risk of detection. Like handwritten signatures, digital signatures depend upon the binding of a unique and secret attribute of the signer to a document. In the case of handwritten signatures, the secret is the handwriting pattern of the signer.

### 12.2 DIGITAL SIGNATURES

The properties of digital documents held in stored files or messages are completely different from those of paper documents. Digital documents are trivially easy to generate copy and alter. Simply appending the identity of the originator, whether as a text string, a photograph or a handwritten image, has no value for verification purposes.

What is needed is a means to irrevocably bind a signer’s identity to the entire sequence of bits representing a document. This should meet the first requirement above, for authenticity. As with handwritten signatures, though, the date of a document cannot be guaranteed by a signature. The recipient of a signed document knows only that the document was signed before they received it.
Regarding non-repudiation, there is a problem that does not arise with handwritten signatures. What if the signer deliberately reveals their private key and subsequently denies having signed, saying that there are others who could have done so, since the key was not private? Some protocols have been developed to address this problem under the heading of *undeniable digital signatures*, but they add considerably to the complexity.

A document with a digital signature can be considerably more resistant to forgery than a handwritten one. But the word ‘original’ has little meaning with reference to digital documents. As we shall see in our discussion of the needs of electronic commerce, digital signatures alone cannot, for example, prevent double-spending of electronic cash – other measures are needed to prevent that. We now describe two techniques for signing documents digitally, binding a principal’s identity to the document. Both depend upon the use of cryptography.

**Digital signing:** An electronic document or message $M$ can be signed by a principal $A$ by encrypting a copy of $M$ with a key $K_A$ and attaching it to a plaintext copy of $M$ and $A$’s identifier. The signed document then consists of: $M$, $A$, $[M]K_A$. The signature can be verified by a principal that subsequently receives the document to check that it was originated by $A$ and that its contents, $M$, have not subsequently been altered.

![Figure 12.4.1: Digital signatures with public keys](image-url)
If a secret key is used to encrypt the document, only principals that share the secret can verify the signature. But if public-key cryptography is used, then the signer uses their private key and anyone who has the corresponding public key can verify the signature. This is a better analogue for conventional signatures and meets a wider range of user needs. The verification of signatures proceeds differently depending on whether secret-key or public-key cryptography is used to produce the signature.

**Digest functions:** Digest functions are also called *secure hash functions* and denoted \( H(M) \). They must be carefully designed to ensure that \( H(M) \) is different from \( H(M') \) for all likely pairs of messages \( M \) and \( M' \). If there are any pairs of different messages \( M \) and \( M' \) such that \( H(M) = H(M') \), then a duplicitous principal could send a signed copy of \( M \), but when confronted with it claim that \( M' \) was originally sent and that it must have been altered in transit.

**Digital signatures with public keys**

Public-key cryptography is particularly well adapted for the generation of digital signatures because it is relatively simple and does not require any communication between the recipient of a signed document and the signer or any third party.

The method for \( A \) to sign a message \( M \) and \( B \) to verify it is as follows (and is illustrated graphically in Figure 3.4.1):

1. \( A \) generates a key pair \( K_{pub} \) and \( K_{priv} \) and publishes the public key \( K_{pub} \) by placing it in a well-known location.
2. \( A \) computes the digest of \( M \), \( H(M) \) using an agreed secure hash function \( H \) and encrypts it using the private key \( K_{priv} \) to produce the signature \( S = \{H(M)\} K_{priv} \).
3. \( A \) sends the signed message \([M]K = M,S\) to \( B \).
4. \( B \) decrypts \( S \) using \( K_{pub} \) and computes the digest of \( M, H(M) \). If they match, the signature is valid.
The RSA algorithm is quite suitable for use in constructing digital signatures. Note that the *private* key of the signer is used to encrypt the signature, in contrast to the use of the recipient’s *public* key for encryption when the aim is to transmit information in secrecy. The explanation for this difference is straightforward – a signature must be created using a secret known only to the signer and it should be accessible to all for verification.

**Digital signatures with secret keys – MACs**

There is no technical reason why a secret-key encryption algorithm should not be used to encrypt a digital signature, but in order to verify such signatures the key must be disclosed, and this causes some problems:

- The signer must arrange for the verifier to receive the secret key used for signing securely.
- It may be necessary to verify a signature in several contexts and at different times – at the time of signing, the signer may not know the identities of the verifiers. To resolve this, verification could be delegated to a trusted third party who holds secret keys for all signers, but this adds complexity to the security model and requires secure communication with the trusted third party.
- The disclosure of a secret key used for signing is undesirable because it weakens the security of signatures made with that key – a signature could be forged by a holder of the key who is not the owner of it.

For all these reasons, the public-key method for generating and verifying signatures offers the most convenient solution in most situations.

An exception arises when a secure channel is used to transmit unencrypted messages but there is a need to verify the authenticity of the messages. Since a secure channel provides secure communication between a pair of processes, a shared secret key can be established using the hybrid method and used to produce low-cost signatures. These signatures are
called *message authentication codes* (MACs) to reflect their more limited purpose – they authenticate communication between pairs of principals based on a shared secret.

A low-cost signing technique based on shared secret keys that has adequate security for many purposes is illustrated in Figure 3.4.2 and outlined below. The method depends upon the existence of a secure channel through which the shared key can be distributed:

1. A generates a random key $K$ for signing and distributes it using secure channels to one or more principals who will need to authenticate messages received from A. The principals are *trusted* not to disclose the shared key.

2. For any document $M$ that A wishes to sign, A concatenates $M$ with $K$, computes the digest of the result, $h = H(M + K)$, and sends the signed document $[M]_K = M, h$ to anyone wishing to verify the signature. (The digest $h$ is a MAC.) $K$ will not be compromised by the disclosure of $h$, since the hash function has totally obscured its value.

3. The receiver, B, concatenates the secret key $K$ with the received document $M$ and computes the digest $h' = H(M + K)$. The signature is verified if $h = h'$.

![Figure 12.4.2: Low-cost signatures with a shared secret key](image_url)
Although this method suffers from the disadvantages listed above, it has a performance advantage because it involves no encryption.

**Secure digest functions**

There are many ways to produce a fixed-length bit pattern that characterizes an arbitrary-length message or document. Perhaps the simplest is to use the XOR operation iteratively to combine fixed-length pieces of the source document. Such a function is often used in communication protocols to produce a short fixed-length hash to characterize a message for error-detection purposes, but it is inadequate as the basis for a digital signature scheme. A secure digest function $h = H(M)$ should have the following properties:

1. Given $M$, it is easy to compute $h$.
2. Given $h$, it is hard to compute $M$.
3. Given $M$, it is hard to find another message $M'$, such that $H(M) = H(M')$.

Such functions are also called *one-way hash functions*. The reason for this name is self-evident based on the first two properties. Property 3 demands an additional feature: even though we know that the result of a hash function cannot be unique (because the digest is an information-reducing transformation), we need to be sure that an attacker, given a message $M$ that produces a hash $h$, cannot discover another message $M'$ that also produces $h$. If an attacker *could* do this, then they could forge a signed document $M'$ without knowledge of the signing key by copying the signature from the signed document $M$ and appending it to $M'$.

Admittedly, the set of messages that hash to the same value is restricted and the attacker would have difficulty in producing a meaningful forgery, but with patience it could be done, so it must be guarded against. The feasibility of doing so is considerably enhanced in the case of a so-called *birthday attack*:
1. Alice prepares two versions, $M$ and $M'$, of a contract for Bob. $M$ is favourable to Bob and $M'$ is not.

2. Alice makes several subtly different versions of both $M$ and $M'$ that are visually indistinguishable from each other by methods such as adding spaces at the ends of lines. She compares the hashes of all the $M$s with all the $M'$s. If she finds two that are the same, she can proceed to the next step; if not, she goes on producing visually indistinguishable versions of the two documents until she gets a match.

3. When she has a pair of documents $M$ and $M'$ that hash to the same value, she gives the favourable document $M$ to Bob for him to sign with a digital signature using his private key. When he returns it, she substitutes the matching unfavourable version $M'$, retaining the signature from $M$.

If our hash values are 64 bits long, we require only 232 versions of $M$ and $M'$ on average. This is too small for comfort. We need to make our hash values at least 128 bits long to guard against this type of attack.

The attack relies on a statistical paradox known as the birthday paradox – the probability of finding a matching pair in a given set is far greater than that for finding a match for a given individual. Stallings gives the statistical derivation for the probability that there will be two people with the same birthday in a set of $n$ people. The result is that for a set of only 23 people the chances are even, whereas we require a set of 253 people for an even chance that there will be one with a birthday on a given day.

To satisfy the properties listed above, a secure digest function needs to be carefully designed. The bit-level operations used and their sequencing are similar to those found in symmetric cryptography, but in this case the operations need not be information-preserving, since the function is definitely not intended to be reversible. So a secure digest function can make use of the full range of arithmetic and bit-wise logical operations. The length of the source text is usually included in the digested data.
Two widely used digest functions for practical applications are the MD5 algorithm (so called because it is the fifth in a sequence of message digest algorithms developed by Ron Rivest) and SHA-1 (the Secure Hash Algorithm), which has been adopted for standardization by the US National Institute for Standards and Technology (NIST). Both have been carefully tested and analyzed and can be considered adequately secure for the foreseeable future, while their implementations are reasonably efficient. We describe them briefly here.

**MD5:** The MD5 algorithm uses four rounds, each applying one of four nonlinear functions to each of 16 32-bit segments of a 512-bit block of source text. The result is a 128-bit digest. MD5 is one of the most efficient algorithms currently available.

**SHA-1:** SHA-1 is an algorithm that produces a 160-bit digest. It is based on Rivest’s MD4 algorithm (which is similar to MD5), with some additional operations. It is substantially slower than MD5, but the 160-bit digest does offer greater security against brute-force and birthday-style attacks. SHA algorithms that deliver longer digests (224, 256 and 512 bits) are also included in the standard. Of course, their additional length implies additional costs for the generation, storage and communication of digital signatures and MACs, but following the publication of attacks on SHA-1’s predecessors, which suggest that SHA-1 is vulnerable, NIST announced that is to be superseded by the longer SHA digest versions in US government software by 2010.

**Using an encryption algorithm to make a digest:** It is possible to use a symmetric encryption algorithm to produce a secure digest. In this case, the key should be published so that the digest algorithm can be applied by anyone wishing to verify a digital
signature. The encryption algorithm is used in CBC mode, and the digest is the result of combining the penultimate CBC value with the final encrypted block.

**Certificate standards and certificate authorities**

X.509 is the most widely used standard format for certificates. Although the X.509 certificate format is a part of the X.500 standard for the construction of global directories of names and attributes, it is commonly used in cryptographic work as a format definition for freestanding certificates.

The structure and content of an X.509 certificate are illustrated in Figure 3.4.3. As can be seen, it binds a public key to a named entity called a *subject*. The binding is in the signature, which is issued by another named entity called the *issuer*. The certificate has a *period of validity*, which is defined by a pair of dates. The *<Distinguished Name>* entries are intended to be the name of a person, organization or other entity together with sufficient contextual information to render it unique. In a full X.500 implementation this contextual information would be drawn from a directory hierarchy in which the named entity appears, but in the absence of global X.500 implementations it can only be a descriptive string.

This format is included in the TLS protocol for electronic commerce and is widely used in practice to authenticate the public keys of services and their clients. Certain well known companies and organizations have established themselves to act as *certificate authorities* (for example, Verisign and CREN), and other companies and individuals can obtain X.509 public-key certificates from them by submitting satisfactory evidence of their identity. This leads to a two-step verification procedure for any X.509 certificate:

1. Obtain the public-key certificate of the issuer (a certification authority) from a reliable source.
2. Validate the signature.
The SPKI approach: The X.509 approach is based on the global uniqueness of distinguished names. It has been pointed out that this is an impractical goal that does not reflect the reality of current legal and commercial practice, in which the identities of individuals are not assumed to be unique but are made unique by reference to other people and organizations. This can be seen in the use of a driving licence or a letter from a bank to authenticate an individual’s name and address (a name alone is unlikely to be unique among the world’s population). This leads to longer verification chains, because there are many possible issuers of public-key certificates, and their signatures must be validated through a chain of verification that leads back to someone known and trusted by the principal performing the verification. But the resulting verification is likely to be more convincing, and many of the steps in such a chain can be cached to shorten the process on future occasions.

The arguments above are the basis for the recently developed Simple Public-Key Infrastructure (SPKI) proposals. This is a scheme for the creation and management of sets of public certificates. It enables chains of certificates to be processed using logical inference to produce derived certificates. For example, ‘Bob believes that Alice’s public key is $K_{A_{pub}}$’ and ‘Carol trusts Bob on Alice’s keys’ implies ‘Carol believes that Alice’s public key is $K_{A_{pub}}$’.

12.3 CRYPTOGRAPHY PRAGMATICS

We compare the performance of the encryption and secure hash algorithms described or mentioned above. We consider encryption algorithms alongside secure hash functions because encryption can also be used as a method for digital signing.
Performance of cryptographic algorithms

<table>
<thead>
<tr>
<th>Key size/hash size (bits)</th>
<th>PRB optimized 90 MHz Pentium 1 (Mbytes/s)</th>
<th>Crypto++ 2.1 GHz Pentium 4 (Mbytes/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>TEA</td>
<td>–</td>
<td>23.801</td>
</tr>
<tr>
<td>DES</td>
<td>2.113</td>
<td>21.340</td>
</tr>
<tr>
<td>Triple-DES</td>
<td>0.775</td>
<td>9.848</td>
</tr>
<tr>
<td>IDEA</td>
<td>1.219</td>
<td>18.963</td>
</tr>
<tr>
<td>AES</td>
<td>–</td>
<td>61.010</td>
</tr>
<tr>
<td>AES</td>
<td>–</td>
<td>53.145</td>
</tr>
<tr>
<td>AES</td>
<td>–</td>
<td>48.229</td>
</tr>
<tr>
<td>MD5</td>
<td>17.025</td>
<td>216.674</td>
</tr>
<tr>
<td>SHA-1</td>
<td>–</td>
<td>67.977</td>
</tr>
</tbody>
</table>

Figure 12.4.4: Performance of symmetric encryption and secure digest algorithms

Figure 12.4.4 compares the speeds of the symmetric encryption algorithms and secure digest functions. Where available, we give two speed measurements. In the column labelled PRB optimized we give figures based on those published by Preneel et al. The figures in the column labelled Crypto++ were obtained much more recently by the authors of the Crypto++ open source library of cryptographic schemes. The column headings indicate the speed of the hardware used for these benchmarks. The PRB implementations were handoptimized assembler programs whereas the Crypto++ ones were C++ programs generated with an optimizing compiler.

The key lengths give an indication of computational cost of a brute-force attack on the key; the true strength of cryptographic algorithms is much more difficult to evaluate and rests on reasoning about the success of the algorithm in obscuring the plain text. Preneel et al. provide a useful discussion on the strength and performance of the main symmetric algorithms.
What do these performance figures signify for real applications of cryptography, such as their use in the TLS scheme for secure web interactions? Web pages are seldom larger than 100 kilobytes, so the contents of a page can be encrypted using any of the symmetric algorithms in a few milliseconds, even with a processor that is quite slow by today’s standards. RSA is used primarily for digital signatures, and that step can also be performed in a few milliseconds. Thus the impact of algorithm performance on the perceived speed of the https application is minimal.

Asymmetric algorithms such as RSA are seldom used for data encryption, but their performance for signing is of interest. The Crypto++ library pages indicate that with the hardware mentioned in the last column of Figure 12.4.4 it takes about 4.75 ms using RSA with a 1024-bit key to sign a secure hash (presumably using 160-bit SHA-1) and about 0.18 ms to verify the signature.

Applications of cryptography and political obstacles

The algorithms described above all emerged during the 1980s and 1990s, when computer networks were beginning to be used for commercial purposes and it was becoming evident that their lack of security was a major problem. As we mentioned in the introduction, the emergence of cryptographic software was strongly resisted by the US government. The resistance had two sources: the US National Security Agency (NSA), which was thought to have a policy to restrict the strength of cryptography available to other nations to a level at which the NSA could decrypt any secret communication for military intelligence purposes; and the US Federal Bureau of Investigation (FBI) which aimed to ensure that its agents could have privileged access to the cryptographic keys used by all private organizations and individuals in the US for law-enforcement purposes.

Cryptographic software was classified as a munition in the United States and was subject to stringent export restrictions. Other countries, especially allies of the US, applied similar (or in some cases even more stringent) restrictions. The problem was
compounded by the general ignorance among politicians and the general public as to what cryptographic software was and its potential non-military applications. US software companies protested that the restrictions were inhibiting the export of software such as browsers, and the export restrictions were eventually formulated in a form that allowed the export of code using keys of no more than 40 bits – hardly strong cryptography!

The export restrictions may have hindered the growth of electronic commerce, but they were not particularly effective in preventing the spread of cryptographic expertise or in keeping cryptographic software out of the hands of users in other countries, since many programmers inside and outside the US were eager and able to implement and distribute cryptographic code. The current position is that software that implements most of the major cryptographic algorithms has been available world-wide for several years, in print and online, in commercial and freeware versions.

An example is the program called PGP (Pretty Good Privacy), originally developed by Philip Zimmermann and carried forward by him and others. This is part of a technical and political campaign to ensure that the availability of cryptographic methods is not controlled by the US government. PGP has been developed and distributed with the aim of enabling all computer users to enjoy the level of privacy and integrity afforded by the use of public-key cryptography in their communications. PGP generates and manages public and secret keys on behalf of a user. It uses RSA public-key encryption for authentication and to transmit secret keys to the intended communication partner, and it uses the IDEA or 3DES secret-key encryption algorithms to encrypt mail messages and other documents. (At the time PGP was first developed, use of the DES algorithm was controlled by the US government.) PGP is widely available in both free and commercial versions. It is distributed via separate distribution sites for North American users and those in other parts of the world to circumvent (perfectly legally) the US export regulations.

The US government eventually recognized the futility of the NSA’s position and the harm that it was causing to the US computer industry (which was unable to market secure
versions of web browsers, distributed operating systems and many other products world-wide). In January 2000 the US government introduced a new policy intended to allow US software vendors to export software that incorporates strong encryption. But a legal bar was retained on delivery to certain countries and end-users. Of course, the US does not have a monopoly on the production or the publication of cryptographic software; open source implementations are available for all the well-known algorithms. The effect of the regulations is simply to hamper the marketing of some US-produced commercial software products.

Other political initiatives have aimed to maintain control over the use of cryptography by introducing legislation insisting on the inclusion of loopholes or trap doors available only to government law-enforcement and security agencies. Such proposals spring from the perception that secret communication channels can be very useful to criminals of all sorts. Before the advent of digital cryptography, governments always had the means to intercept and analyze communications between members of the public. Strong digital cryptography radically alters that situation. But these proposals to legislate to prevent the use of strong, uncompromised cryptography have been strongly resisted by citizens and civil liberties bodies, who are concerned about their impact on citizens’ privacy rights. So far, none of these legislative proposals has been adopted, but political efforts are continuing and the eventual introduction of a legal framework for the use of cryptography may be inevitable.

12.4. SUMMARY

In this unit we have understood the concept of digital signature, digital signing, secure digest functions, certificate standards authorities, Performance of cryptographic algorithms.

12.5. KEYWORDS

Digest functions: Digest functions are also called secure hash functions and denoted $H(M)$. They must be carefully designed to ensure that $H(M)$ is different from $H(M')$ for all likely pairs of messages $M$ and $M'$. 

239
11. Explain digital signature with public keys.
12. Discuss about low-cost signatures with a shared secret key.
13. Give X509 certificate format.

Answers: SEE

1. 12.2
2. 12.2
3. 12.2
4. 12.3

12.7 SUGGESTED READINGS

- DISTRIBUTED SYSTEMS Principles and Paradigm By: Andrew S. Tanenbaum, Maarten Van Steen.
UNIT 13: FILE SERVICE ARCHITECTURE

Structure:
13.0 Objectives
13.1 Introduction
13.2 File service architecture
13.3 Case studies
13.4 Enhancement and further development
13.5 Summary
13.6 Keywords
13.7 Unit-end exercises and answers
13.8 Suggested readings

13.0 OBJECTIVES

At the end of this unit you will be able to know:

- File service architecture
- Case study: Sun Network File System
- Enhancement and further development

13.1 INTRODUCTION

This unit is about the file services architecture. A distributed file system enables programs to store and access remote files exactly as they do local ones, allowing users to access files from any computer on a network. The performance and reliability experienced for access to files stored at a server should be comparable to that for files stored on local disks. Here, we define a simple architecture for file systems and describe two basic distributed file service implementations with contrasting designs that have been in widespread use for over two decades:

- The Sun Network File System, NFS;
- The Andrew File System, AFS.
Each emulates the UNIX file system interface, with differing degrees of scalability, fault
tolerance and deviation from the strict UNIX one-copy file update semantics.

It has been identified the sharing of resources as a key goal for distributed systems. The
sharing of stored information is perhaps the most important aspect of distributed resource
sharing. Mechanisms for data sharing take many forms. Web servers provide a restricted
form of data sharing in which files stored locally, in file systems at the server or in
servers on a local network, are made available to clients throughout the Internet. The
design of large-scale wide area read-write file storage systems poses problems of load
balancing, reliability, availability and security, whose resolution is the goal of the peer-
to-peer file storage systems focuses on replicated storage systems that are suitable for
applications requiring reliable access to data stored on systems where the availability of
individual hosts cannot be guaranteed.

The requirements for sharing within local networks and intranets lead to a need for a
different type of service – one that supports the persistent storage of data and programs of
all types on behalf of clients and the consistent distribution of up-to-date data. The
purpose of this unit is to describe the architecture and implementation of these basic
distributed file systems. We use the word ‘basic’ here to denote distributed file systems
whose primary purpose is to emulate the functionality of a non-distributed file system for
client programs running on multiple remote computers. They do not maintain multiple
persistent replicas of files, nor do they support the bandwidth and timing guarantees
required for multimedia data. Basic distributed file systems provide an essential
underpinning for organizational computing based on intranets.

File systems were originally developed for centralized computer systems and desktop
computers as an operating system facility providing a convenient programming interface
to disk storage. They subsequently acquired features such as access-control and file-
locking mechanisms that made them useful for the sharing of data and programs.
Distributed file systems support the sharing of information in the form of files and
hardware resources in the form of persistent storage throughout an intranet. A well
designed file service provides access to files stored at a server with performance and reliability similar to, and in some cases better than, files stored on local disks. Their design is adapted to the performance and reliability characteristics of local networks, and hence they are most effective in providing shared persistent storage for use in intranets.

A file service enables programs to store and access remote files exactly as they do local ones, allowing users to access their files from any computer in an intranet. The concentration of persistent storage at a few servers reduces the need for local disk storage and (more importantly) enables economies to be made in the management and archiving of the persistent data owned by an organization. Other services, such as the name service, the user authentication service and the print service, can be more easily implemented when they can call upon the file service to meet their needs for persistent storage. Web servers are reliant on filing systems for the storage of the web pages that they serve. In organizations that operate web servers for external and internal access via an intranet, the web servers often store and access the material from a local distributed file system.

With the advent of distributed object-oriented programming, a need arose for the persistent storage and distribution of shared objects. One way to achieve this is to serialize objects and to store and retrieve the serialized objects using files. But this method for achieving persistence and distribution is impractical for rapidly changing objects, so several more direct approaches have been developed. Java remote object invocation and CORBA ORBs provide access to remote, shared objects, but neither of these ensures the persistence of the objects, nor are the distributed objects replicated.

<table>
<thead>
<tr>
<th>Sharing Consistency</th>
<th>Persistence</th>
<th>Distributed</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Example</td>
<td>maintenance</td>
</tr>
<tr>
<td></td>
<td>cache/replicas</td>
<td></td>
</tr>
<tr>
<td>Main memory</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>File system</td>
<td>×</td>
<td>√</td>
</tr>
</tbody>
</table>
Distributed file system  √  √  √  √  Sun NFS
Web  √  √  √  ×  Web server
Distributed shared memory  √  ×  √  √  Ivy
Remote objects (RMI/ORB)  √  ×  ×  ×  1  CORBA
Persistent object store  √  √  ×  1  CORBA
Persistent

Peer-to-peer storage system  √  √  √  2  OceanStore

Types of consistency:
1: strict one-copy  √: slightly weaker guarantees  2: considerably weaker guarantees

Figure 13.1.1 Storage systems and their properties

Figure 13.1.1 provides an overview of types of storage system. The table includes distributed shared memory (DSM) systems and persistent object stores. It provides an emulation of a shared memory by the replication of memory pages or segments at each host, but it does not necessarily provide automatic persistence. Persistent object stores aim to provide persistence for distributed shared objects. Examples include the CORBA Persistent State Service and persistent extensions to Java. Some research projects have developed in platforms that support the automatic replication and persistent storage of objects. Peer-to-peer storage systems offer scalability to support client loads much larger than the systems, but they incur high performance costs in providing secure access control and consistency between updatable replicas.

The consistency column indicates whether mechanisms exist for the maintenance of consistency between multiple copies of data when updates occur. Virtually all storage systems rely on the use of caching to optimize the performance of programs. Caching was first applied to main memory and non-distributed file systems, and for those the
consistency is strict (denoted by a ‘1’, for one-copy consistency in Figure 4.1.1) – programs cannot observe any discrepancies between cached copies and stored data after an update. When distributed replicas are used, strict consistency is more difficult to achieve. Distributed file systems such as Sun NFS and the Andrew File System cache copies of portions of files at client computers, and they adopt specific consistency mechanisms to maintain an approximation to strict consistency – this is indicated by a tick (√) in the consistency column of Figure 4.1.1.

The Web uses caching extensively both at client computers and at proxy servers maintained by user organizations. The consistency between the copies stored at web proxies and client caches and the original server is only maintained by explicit user actions. Clients are not notified when a page stored at the original server is updated; they must perform explicit checks to keep their local copies up-to-date. This serves the purposes of web browsing adequately, but it does not support the development of cooperative applications such as a shared distributed whiteboard. Persistent object systems vary considerably in their approach to caching and consistency. The CORBA and Persistent Java schemes maintain single copies of persistent objects, and remote invocation is required to access them, so the only consistency issue is between the persistent copy of an object on disk and the active copy in memory, which is not visible to remote clients. The PerDiS and Khazana projects that we mentioned above maintain cached replicas of objects and employ quite elaborate consistency mechanisms to produce forms of consistency similar to those found in DSM systems.

---

Directory module: relates file names to file IDs
File module: relates file IDs to particular files
Access control module: checks permission for operation requested
File access module: reads or writes file data or attributes
Block module: accesses and allocates disk blocks
Device module: performs disk I/O and buffering

---

Figure 13.1.2: File system modules
Having introduced some wider issues relating to storage and distribution of persistent and non-persistent data, we now return to the main topic— the design of basic distributed file systems. We describe some relevant characteristics of (non-distributed) file systems.

**Characteristics of file systems**

File systems are responsible for the organization, storage, retrieval, naming, sharing and protection of files. They provide a programming interface that characterizes the file abstraction, freeing programmers from concern with the details of storage allocation and layout. Files are stored on disks or other non-volatile storage media.

<table>
<thead>
<tr>
<th>Characteristics</th>
</tr>
</thead>
<tbody>
<tr>
<td>File length</td>
</tr>
<tr>
<td>Creation timestamp</td>
</tr>
<tr>
<td>Read timestamp</td>
</tr>
<tr>
<td>Write timestamp</td>
</tr>
<tr>
<td>Attribute timestamp</td>
</tr>
<tr>
<td>Reference count</td>
</tr>
<tr>
<td>Owner</td>
</tr>
<tr>
<td>File type</td>
</tr>
<tr>
<td>Access control list</td>
</tr>
</tbody>
</table>

**Figure 13.1.3: File attribute record structure**

Files contain both *data* and *attributes*. The data consist of a sequence of data items (typically 8-bit bytes), accessible by operations to read and write any portion of the sequence. The attributes are held as a single record containing information such as the length of the file, timestamps, file type, owner’s identity and access control lists. A typical attribute record structure is illustrated in Figure 13.1.3. The shaded attributes are managed by the file system and are not normally updatable by user programs.
File systems are designed to store and manage large numbers of files, with facilities for creating, naming and deleting files. The naming of files is supported by the use of directories. A directory is a file, often of a special type, that provides a mapping from text names to internal file identifiers. Directories may include the names of other directories, leading to the familiar hierarchic file-naming scheme and the multi-part pathnames for files used in UNIX and other operating systems. File systems also take responsibility for the control of access to files, restricting access to files according to users’ authorizations and the type of access requested (reading, updating, executing and so on).

The term metadata is often used to refer to all of the extra information stored by a file system that is needed for the management of files. It includes file attributes, directories and all the other persistent information used by the file system. Figure 4.1.2 shows a typical layered module structure for the implementation of a non-distributed file system in a conventional operating system. Each layer depends only on the layers below it. The implementation of a distributed file service requires all of the components shown there, with additional components to deal with client-server communication and with the distributed naming and location of files.

---

```
filedes = open(name, mode)  # Opens an existing file with the given name.
filedes = creat(name, mode) # Creates a new file with the given name.

Both operations deliver a file descriptor referencing the openfile. The mode is read, write or both.

status = close(filedes)  # Closes the open file filedes.

count = read(filedes, buffer, n)  # Transfers n bytes from the file referenced by filedes to buffer

count = write(filedes, buffer, n)  # Transfers n bytes to the file referenced by filedes from buffer.

Both operations deliver the number of bytes actually transferred and advance the read-write pointer.
```
pos = lseek(filedes, offset, whence)  
    Moves the read-write pointer to offset (relative or absolute, depending on whence).

status = unlink(name)  
    Removes the file name from the directory structure. If the file has no other names, it is deleted.

status = link(name1, name2)  
    Adds a new name (name2) for a file (name1).

status = stat(name, buffer)  
    Puts the file attributes for file name into buffer.

---

**Figure 13.1.4: UNIX file system operations**

**File system operations:** Figure 4.1.4 summarizes the main operations on files that are available to applications in UNIX systems. These are the system calls implemented by the kernel; application programmers usually access them through procedure libraries such as the C Standard Input/Output Library or the Java file classes. We give the primitives here as an indication of the operations that file services are expected to support and for comparison with the file service interfaces that we shall introduce below.

The UNIX operations are based on a programming model in which some file state information is stored by the file system for each running program. This consists of a list of currently open files with a read-write pointer for each, giving the position within the file at which the next read or write operation will be applied.

The file system is responsible for applying access control for files. In local file systems such as UNIX, it does so when each file is opened, checking the rights allowed for the user’s identity in the access control list against the mode of access requested in the open system call. If the rights match the mode, the file is opened and the mode is recorded in the open file state information.

**Distributed file system requirements**

Many of the requirements and potential pitfalls in the design of distributed services were first observed in the early development of distributed file systems. Initially, they offered access transparency and location transparency; performance, scalability, concurrency control, fault tolerance and security requirements emerged and were met in subsequent
phases of development. We discuss these and related requirements in the following subsections.

**Transparency:** The file service is usually the most heavily loaded service in an intranet, so its functionality and performance are critical. The design of the file service should support many of the transparency requirements for distributed systems. The design must balance the flexibility and scalability that derive from transparency against software complexity and performance. The following forms of transparency are partially or wholly addressed by current file services:

**Access transparency:** Client programs should be unaware of the distribution of files. A single set of operations is provided for access to local and remote files. Programs written to operate on local files are able to access remote files without modification.

**Location transparency:** Client programs should see a uniform file name space. Files or groups of files may be relocated without changing their pathnames, and user programs see the same name space wherever they are executed.

**Mobility transparency:** Neither client programs nor system administration tables in client nodes need to be changed when files are moved. This allows file mobility – files or, more commonly, sets or volumes of files may be moved, either by system administrators or automatically.

**Performance transparency:** Client programs should continue to perform satisfactorily while the load on the service varies within a specified range.

**Scaling transparency:** The service can be expanded by incremental growth to deal with a wide range of loads and network sizes.

**Concurrent file updates:** Changes to a file by one client should not interfere with the operation of other clients simultaneously accessing or changing the same file. This is the
well-known issue of concurrency control. The need for concurrency control for access to shared data in many applications is widely accepted and techniques are known for its implementation, but they are costly. Most current file services follow modern UNIX standards in providing advisory or mandatory file- or record-level locking.

**File replication:** In a file service that supports replication, a file may be represented by several copies of its contents at different locations. This has two benefits – it enables multiple servers to share the load of providing a service to clients accessing the same set of files, enhancing the scalability of the service, and it enhances fault tolerance by enabling clients to locate another server that holds a copy of the file when one has failed. Few file services support replication fully, but most support the caching of files or portions of files locally, a limited form of replication.

**Hardware and operating system heterogeneity:** The service interfaces should be defined so that client and server software can be implemented for different operating systems and computers. This requirement is an important aspect of openness.

**Fault tolerance:** The central role of the file service in distributed systems makes it essential that the service continue to operate in the face of client and server failures. Fortunately, a moderately fault-tolerant design is straightforward for simple servers. To cope with transient communication failures, the design can be based on *at-most-once* invocation semantics; or it can use the simpler *at-least-once* semantics with a server protocol designed in terms of *idempotent* operations, ensuring that duplicated requests do not result in invalid updates to files. The servers can be *stateless*, so that they can be restarted and the service restored after a failure without any need to recover previous state. Tolerance of disconnection or server failures requires file replication, which is more difficult to achieve.

**Consistency:** Conventional file systems such as that provided in UNIX offer *one-copy update semantics*. This refers to a model for concurrent access to files in which the file contents seen by all of the processes accessing or updating a given file are those that they
would see if only a single copy of the file contents existed. When files are replicated or cached at different sites, there is an inevitable delay in the propagation of modifications made at one site to all of the other sites that hold copies, and this may result in some deviation from one-copy semantics.

**Security**: Virtually all file systems provide access-control mechanisms based on the use of access control lists. In distributed file systems, there is a need to authenticate client requests so that access control at the server is based on correct user identities and to protect the contents of request and reply messages with digital signatures and (optionally) encryption of secret data.

**Efficiency**: A distributed file service should offer facilities that are of at least the same power and generality as those found in conventional file systems and should achieve a comparable level of performance. Birrell and Needham expressed their design aims for the Cambridge File Server (CFS) in these terms:

We would wish to have a simple, low-level file server in order to share an expensive resource, namely a disk, whilst leaving us free to design the filing system most appropriate to a particular client, but we would wish also to have available a high-level system shared between clients.

The changed economics of disk storage have reduced the significance of their first goal, but their perception of the need for a range of services addressing the requirements of clients with different goals remains and can best be addressed by a modular architecture of the type outlined above.

The techniques used for the implementation of file services are an important part of the design of distributed systems. A distributed file system should provide a service that is comparable with, or better than, local file systems in performance and reliability. It must be convenient to administer, providing operations and tools that enable system administrators to install and operate the system conveniently.
We have constructed an abstract model for a file service to act as an introductory example, separating implementation concerns and providing a simplified model. We describe the Sun Network File System in some detail, drawing on our simpler abstract model to clarify its architecture.

**File service architecture:** This is an abstract architectural model that underpins both NFS and AFS. It is based upon a division of responsibilities between three modules – a client module that emulates a conventional file system interface for application programs, and server modules, that perform operations for clients on directories and on files. The architecture is designed to enable a stateless implementation of the server module.

**SUN NFS:** Sun Microsystems’s Network File System (NFS) has been widely adopted in industry and in academic environments since its introduction in 1985. The design and development of NFS were undertaken by staff at Sun Microsystems in 1984. Although several distributed file services had already been developed and used in universities and research laboratories, NFS was the first file service that was designed as a product. The design and implementation of NFS have achieved success both technically and commercially.

The client-server relationship is symmetrical: each computer in an NFS network can act as both a client and a server, and the files at every machine can be made available for remote access by other machines. Any computer can be a server, exporting some of its files, and a client, accessing files on other machines. But it is common practice to configure larger installations with some machines as dedicated servers and others as workstations.

An important goal of NFS is to achieve a high level of support for hardware and operating system heterogeneity. The design is operating system–independent: client and
server implementations exist for almost all operating systems and platforms, including all versions of Windows, Mac OS, Linux and every other version of UNIX. Implementations of NFS on high-performance multiprocessor hosts have been developed by several vendors, and these are widely used to meet storage requirements in intranets with many concurrent users.

**Andrew File System:** Andrew is a distributed computing environment developed at Carnegie Mellon University (CMU) for use as a campus computing and information system. The design of the Andrew File System (henceforth abbreviated AFS) reflects an intention to support information sharing on a large scale by minimizing client-server communication. This is achieved by transferring whole files between server and client computers and caching them at clients until the server receives a more up-to-date version.

AFS was initially implemented on a network of workstations and servers running BSD UNIX and the Mach operating system at CMU and was subsequently made available in commercial and public-domain versions. A public-domain implementation of AFS is available in the Linux operating system [Linux AFS]. AFS was adopted as the basis for the DCE/DFS file system in the Open Software Foundation’s Distributed Computing Environment (DCE) [www.opengroup.org]. The design of DCE/DFS went beyond AFS in several important respects.

### 13.3 FILE SERVICE ARCHITECTURE

An architecture that offers a clear separation of the main concerns in providing access to files is obtained by structuring the file service as three components – a *flat file service*, a *directory service* and a *client module*. The relevant modules and their relationships are shown in Figure 4.1.5. The flat file service and the directory service each export an interface for use by client programs, and their RPC interfaces, taken together, provide a comprehensive set of operations for access to files.

The division of responsibilities between the modules can be defined as follows:
**Flat file service:** The flat file service is concerned with implementing operations on the contents of files. *Unique file identifiers* (UFIDs) are used to refer to files in all requests for flat file service operations. The division of responsibilities between the file service and the directory service is based upon the use of UFIDs. UFIDs are long sequences of bits chosen so that each file has a UFID that is unique among all of the files in a distributed system. When the flat file service receives a request to create a file, it generates a new UFID for it and returns the UFID to the requester.

**Directory service:** The directory service provides a mapping between *text names* for files and their UFIDs. Clients may obtain the UFID of a file by quoting its text name to the directory service. The directory service provides the functions needed to generate directories, to add new file names to directories and to obtain UFIDs from directories. It is a client of the flat file service; its directory files are stored in files of the flat file service. When a hierarchic file-naming scheme is adopted, as in UNIX, directories hold references to other directories.

**Client module:** A client module runs in each client computer, integrating and extending the operations of the flat file service and the directory service under a single application programming interface that is available to user-level programs in client computers. For example, in UNIX hosts, a client module would be provided that emulates the full set of UNIX file operations, interpreting UNIX multi-part file names by iterative requests to the
directory service. The client module also holds information about the network locations of the flat file server and directory server processes. Finally, the client module can play an important role in achieving satisfactory performance through the implementation of a cache of recently used file blocks at the client.

**Flat file service interface:** Figure 4.1.6 contains a definition of the interface to a flat file service. This is the RPC interface used by client modules. It is not normally used directly by user-level programs. A *FileId* is invalid if the file that it refers to is not present in the server processing the request or if its access permissions are inappropriate for the operation requested. All of the procedures in the interface except *Create* throw exceptions if the *FileId* argument contains an invalid UFID or the user doesn’t have sufficient access rights. These exceptions are omitted from the definition for clarity. The most important operations are those for reading and writing. Both the *Read* and the *Write* operation require a parameter *i* specifying a position in the file. The *Read* operation copies the sequence of *n* data items beginning at item *i* from the specified file into *Data*, which is then returned to the client. The *Write* operation copies the sequence of data items in *Data* into the specified file beginning at item *i*, replacing the previous contents of the file at the corresponding position and extending the file if necessary. *Create* creates a new, empty file and returns the UFID that is generated. *Delete* removes the specified file.

\[
\begin{align*}
\text{Read}(\text{FileId}, i, n) & \rightarrow \text{Data} \quad \text{if } 1 \leq i \leq \text{Length}(	ext{File}): \text{Reads a} \\
& \quad \text{sequence of up to } n \text{ items} \\
& \quad \text{from a file starting at item } i \text{ and} \\
& \quad \text{returns it in } \text{Data}, \\
\text{Write}(\text{FileId}, i, \text{Data}) & \quad \text{if } 1 \leq i \leq \text{Length}(	ext{File}) + 1: \text{Write a} \\
& \quad \text{sequence of } \text{Data} \text{ to a} \\
& \quad \text{file, starting at item } i, \\
& \quad \text{extending the file if necessary.} \\
\text{Create}() & \rightarrow \text{FileId} \quad \text{Creates a new file of length0} \\
\text{and delivers a UFID for it.} \\
\text{Delete}(\text{FileId}) & \quad \text{Removes the file from the} \\
\text{file store.} \\
\text{GetAttributes}(\text{FileId}) & \rightarrow \text{Attributes} \quad \text{Returns the file attributes for the} \\
\text{file.} \\
\text{SetAttributes}(\text{FileId}, \text{Attributes}) & \quad \text{Sets the file attributes (only} \\
& \quad \text{those attributes that are not} \\
& \quad \text{shaded in Figure 13.1.6 Flat file service operations}
\end{align*}
\]
GetAttributes and SetAttributes enable clients to access the attribute record. GetAttributes is normally available to any client that is allowed to read the file. Access to the SetAttributes operation would normally be restricted to the directory service that provides access to the file. The values of the length and timestamp portions of the attribute record are not affected by SetAttributes; they are maintained separately by the flat file service itself.

Comparison with UNIX: Our interface and the UNIX file system primitives are functionally equivalent. It is a simple matter to construct a client module that emulates the UNIX system calls in terms of our flat file service and the directory service operations described in the next section.

In comparison with the UNIX interface, our flat file service has no open and close operations – files can be accessed immediately by quoting the appropriate UFID. The Read and Write requests in our interface include a parameter specifying a starting point within the file for each transfer, whereas the equivalent UNIX operations do not. In UNIX, each read or write operation starts at the current position of the read-write pointer, and the read-write pointer is advanced by the number of bytes transferred after each read or write. A seek operation is provided to enable the read-write pointer to be explicitly repositioned.

The interface to our flat file service differs from the UNIX file system interface mainly for reasons of fault tolerance:

**Repetable operations:** With the exception of Create, the operations are idempotent, allowing the use of at-least-once RPC semantics – clients may repeat calls to which they receive no reply. Repeated execution of Create produces a different new file for each call.
**Stateless servers:** The interface is suitable for implementation by *stateless* servers. Stateless servers can be restarted after a failure and resume operation without any need for clients or the server to restore any state.

The UNIX file operations are neither idempotent nor consistent with the requirement for a stateless implementation. A read-write pointer is generated by the UNIX file system whenever a file is opened, and it is retained, together with the results of access-control checks, until the file is closed. The UNIX *read* and *write* operations are not idempotent; if an operation is accidentally repeated, the automatic advance of the read-write pointer results in access to a different portion of the file in the repeated operation. The read-write pointer is a hidden, client-related state variable. To mimic it in a file server, *open* and *close* operations would be needed, and the read-write pointer’s value would have to be retained by the server as long as the relevant file is open. By eliminating the read-write pointer, we have eliminated most of the need for the file server to retain state information on behalf of specific clients.

**Access control:** In the UNIX file system, the user’s access rights are checked against the access mode (read or write) requested in the *open* call and the file is opened only if the user has the necessary rights. The user identity (UID) used in the access rights check is retrieved during the user’s earlier authenticated login and cannot be tampered with in non-distributed implementations. The resulting access rights are retained until the file is closed, and no further checks are required when subsequent operations on the same file are requested.

In distributed implementations, access rights checks have to be performed at the server because the server RPC interface is an otherwise unprotected point of access to files. A user identity has to be passed with requests, and the server is vulnerable to forged identities. Furthermore, if the results of an access rights check were retained at the server and used for future accesses, the server would no longer be stateless. Two alternative approaches to the latter problem can be adopted:
• An access check is made whenever a file name is converted to a UFID, and the results are encoded in the form of a capability, which is returned to the client for submission with subsequent requests.

• A user identity is submitted with every client request, and access checks are performed by the server for every file operation.

Both methods enable stateless server implementation, and both have been used in distributed file systems. The second is more common; it is used in both NFS and AFS. Neither of these approaches overcomes the security problem concerning forged user identities, but this can be addressed by the use of digital signatures. Kerberos is an effective authentication scheme that has been applied to both NFS and AFS.

In our abstract model, we make no assumption about the method by which access control is implemented. The user identity is passed as an implicit parameter and can be used whenever it is needed.

**Directory service interface:** Figure 4.1.7 contains a definition of the RPC interface to a directory service. The primary purpose of the directory service is to provide a service for translating text names to UFIDs. In order to do so, it maintains directory files containing the mappings between text names for files and UFIDs. Each directory is stored as a conventional file with a UFID, so the directory service is a client of the file service.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LocateName (Dir, Name)</td>
<td>Locates the text name in the directory and returns the relevant UFID. If Name is not in the directory, returns an exception.</td>
</tr>
<tr>
<td>AddName (Dir, Name, File)</td>
<td>Adds Name to the directory and updates the file’s attribute record. If Name is already in the directory, throws an exception.</td>
</tr>
<tr>
<td>ListName (Dir, Name)</td>
<td>Returns all the text names in the directory that match the expression Pattern. If Name is not in the directory, throws an exception.</td>
</tr>
</tbody>
</table>

**Figure 13.1.7 Directory service operations**
We define only operations on individual directories. For each operation, a UFID for the file containing the directory is required (in the Dir parameter). The Lookup operation in the basic directory service performs a single Name $\rightarrow$ UFID translation. It is a building block for use in other services or in the client module to perform more complex translations, such as the hierarchic name interpretation found in UNIX. As before, exceptions caused by inadequate access rights are omitted from the definitions.

There are two operations for altering directories: AddName and UnName. AddName adds an entry to a directory and increments the reference count field in the file’s attribute record.

UnName removes an entry from a directory and decrements the reference count. If this causes the reference count to reach zero, the file is removed. GetNames is provided to enable clients to examine the contents of directories and to implement pattern-matching operations on file names such as those found in the UNIX shell. It returns all or a subset of the names stored in a given directory. The names are selected by pattern matching against a regular expression supplied by the client.

The provision of pattern matching in the GetNames operation enables users to determine the names of one or more files by giving an incomplete specification of the characters in the names. A regular expression is a specification for a class of strings in the form of an expression containing a combination of literal substrings and symbols denoting variable characters or repeated occurrences of characters or substrings.

Hierarchic file system: A hierarchic file system such as the one that UNIX provides consists of a number of directories arranged in a tree structure. Each directory holds the names of the files and other directories that are accessible from it. Any file or directory can be referenced using a pathname – a multi-part name that represents a path through the tree. The root has a distinguished name, and each file or directory has a name in a directory. The UNIX file-naming scheme is not a strict hierarchy – files can have several
names, and they can be in the same or different directories. This is implemented by a *link* operation, which adds a new name for a file to a specified directory.

A UNIX-like file-naming system can be implemented by the client module using the flat file and directory services that we have defined. A tree-structured network of directories is constructed with files at the leaves and directories at the other nodes of the tree. The root of the tree is a directory with a ‘well-known’ UFID. Multiple names for files can be supported using the *AddName* operation and the reference count field in the attribute record.

A function can be provided in the client module that gets the UFID of a file given its pathname. The function interprets the pathname starting from the root, using *Lookup* to obtain the UFID of each directory in the path. In a hierarchic directory service, the file attributes associated with files should include a type field that distinguishes between ordinary files and directories. This is used when following a path to ensure that each part of the name, except the last, refers to a directory.

**File groups:** A *file group* is a collection of files located on a given server. A server may hold several file groups, and groups can be moved between servers, but a file cannot change the group to which it belongs. A similar construct called a *filesystem* is used in UNIX and in most other operating systems. (Terminology note: the single word *filesystem* refers to the set of files held in a storage device or partition, whereas the words *file system* refers to a software component that provides access to files.) File groups were originally introduced to support facilities for moving collections of files stored on removable media between computers. In a distributed file service, file groups support the allocation of files to file servers in larger logical units and enable the service to be implemented with files stored on several servers. In a distributed file system that supports file groups, the representation of UFIDs includes a file group identifier component, enabling the client module in each client computer to take responsibility for dispatching requests to the server that holds the relevant file group.
File group identifiers must be unique throughout a distributed system. Since file groups can be moved and distributed systems that are initially separate can be merged to form a single system, the only way to ensure that file group identifiers will always be distinct in a given system is to generate them with an algorithm that ensures global uniqueness. For example, whenever a new file group is created, a unique identifier can be generated by concatenating the 32-bit IP address of the host creating the new group with a 16-bit integer derived from the date, producing a unique 48-bit integer:

<table>
<thead>
<tr>
<th>32 bits</th>
<th>16 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>file</td>
<td>group</td>
</tr>
<tr>
<td>IP address</td>
<td>Date</td>
</tr>
</tbody>
</table>

identifier:

Note that the IP address cannot be used for the purpose of locating the file group, since it may be moved to another server. Instead, a mapping between group identifiers and servers should be maintained by the file service.

### 13.4 ENHANCEMENTS AND FURTHER DEVELOPMENTS

Several advances have been made in the design of distributed file systems since the emergence of NFS and AFS. In this section, we describe advances that enhance the performance, availability and scalability of conventional distributed file systems. More radical advances are described elsewhere in the book, including the maintenance of consistency in replicated read-write file systems to support disconnected operation and high availability in the Bayou and Coda systems and a highly scalable architecture for the delivery of streams of real-time data with quality guarantees in the Tiger video file server.

**NFS enhancements:** Several research projects have addressed the need for one-copy update semantics by extending the NFS protocol to include open and close operations and adding a callback mechanism to enable the server to notify clients of the need to invalidate cache entries. We describe two such efforts here; their results seem to indicate
that these enhancements can be accommodated without undue complexity or extra communication costs.

Some recent efforts by Sun and other NFS developers have been directed at making NFS servers more accessible and useful in wide-area networks. While the HTTP protocol supported by web servers offers an effective and highly scalable method for making whole files available to clients throughout the Internet, it is less useful to application programs that require access to portions of large files or those that update portions of files. The WebNFS development (described below) makes it possible for application programs to become clients of NFS servers anywhere in the Internet (using the NFS protocol directly instead of indirectly through a kernel module). This, together with appropriate libraries for Java and other network programming languages, should offer the possibility of implementing Internet applications that share data directly, such as multi-user games or clients of large dynamic databases.

**Achieving one-copy update semantics:** The stateless server architecture of NFS brought great advantages in terms of robustness and ease of implementation, but it precluded the achievement of precise one-copy update semantics (the effects of concurrent writes by different clients to the same file are not guaranteed to be the same as they would be in a single UNIX system when multiple processes write to a local file). It also prevents the use of callbacks notifying clients of changes to files, and this results in frequent `getattr` requests from clients to check for file modification.

Clients’ modules must send an `open` operation whenever a local user-level process opens a file that is on the server. The parameters of the Sprite `open` operation specify a mode (read, write or both) and include counts of the number of local processes that currently have the file open for reading and for writing. Similarly, when a local process closes a remote file, a `close` operation is sent to the server with updated counts of readers and writers. The server records these numbers in an `open files table` with the IP address and port number of the client.
When the server receives an open, it checks the open files table for other clients that have the same file open and sends callback messages to those clients instructing them to modify their caching strategy. If the open specifies write mode, then it will fail if any other client has the file open for writing. Other clients that have the file open for reading will be instructed to invalidate any locally cached portions of the file.

For open operations that specify read mode, the server sends a callback message to any client that is writing, instructing it to stop caching (i.e., to use a strictly writethrough mode of operation), and it instructs all clients that are reading to cease caching the file (so that all local read calls result in a request to the server).

These measures result in a file service that maintains the UNIX one-copy update semantics at the expense of carrying some client-related state at the server. They also enable some efficiency gains in the handling of cached writes. If the client-related state is held in volatile memory at the server, it is vulnerable to server crashes. Spritely NFS implements a recovery protocol that interrogates a list of clients that have recently opened files on the server to recover the full open files table. The list of clients is stored on disk, is updated relatively infrequently and is ‘pessimistic’ – it may safely include more clients than those that had files open at the time of a crash. Failed clients may also result in excess entries in the open files table, but these entries will be removed when the clients restart.

**NQNFS:** The NQNFS (Not Quite NFS) project [Macklem 1994] had similar aims to Spritely NFS – to add more precise cache consistency to the NFS protocol and to improve performance through better use of caching. An NQNFS server maintains similar client-related state concerning open files, but it uses leases to aid recovery after a server crash. The server sets an upper bound on the time for which a client may hold a lease on an open file. If the client wishes to continue beyond that time, it must renew the lease. Callbacks are used in a similar manner to Spritely NFS to request clients to flush their caches when a write request occurs, but if the clients don’t reply, the server simply waits until their leases expire before responding to the new write request.
**WebNFS:** The advent of the Web and Java applets led to the recognition by the NFS development team and others that some Internet applications could benefit from direct access to NFS servers without many of the overheads associated with the emulation of UNIX file operations included in standard NFS clients.

The aim of WebNFS is to enable web browsers and other applications to access files on an NFS server that ‘publishes’ them using a public file handle relative to a public root directory. This mode of use bypasses the mount service and the port mapper service. WebNFS clients interact with an NFS server at a well-known port number (2049). To access files by pathname, they issue lookup requests using a public file handle. The public file handle has a well-known value that is interpreted specially by the virtual file system at the server. Because of the high latency of wide-area networks, a multicomponent variant of the lookup operation is used to look up a multi-part pathname in a single request.

Thus WebNFS enables clients to be written that access portions of files stored in NFS servers at remote sites with minimal setup overheads. There is provision for access control and authentication, but in many cases the client will require only read access to public files, and in that case the authentication option can be turned off. To read a portion of a single file located on an NFS server that supports WebNFS requires the establishment of a TCP connection and two RPC calls – a multicomponent lookup and a read operation. The size of the block of data read is not limited by the NFS protocol.

**NFS version 4:** A new version of the NFS protocol was introduced in 2000. The goals of NFS version 4 are described in RFC 2624. Like WebNFS, it aims to make it practical to use NFS in wide-area networks and Internet applications. It includes the features of WebNFS, but the introduction of a new protocol also offers an opportunity to make more radical enhancements.
AFS enhancements: We have mentioned that DCE/DFS, the distributed file system included in the Open Software Foundation’s Distributed Computing Environment, was based on the Andrew File System. The design of DCE/DFS goes beyond AFS, particularly in its approach to cache consistency. In AFS, callbacks are generated only when the server receives a close operation for a file that has been updated. DFS adopted a similar strategy to Spritely NFS and NQNFS to generating callbacks as soon as a file is updated. In order to update a file, a client must obtain a write token from the server, specifying a range of bytes in the file that the client is permitted to update. When a write token is requested, clients holding copies of the same file for reading receive revocation callbacks. Tokens of other types are used to achieve consistency for cached file attributes and other metadata. All tokens have an associated lifetime, and clients must renew them after their lifetime has expired.

**Improvements in storage organization:** There has been considerable progress in the organization of file data stored on disks. The impetus for much of this work arose from the increased loads and greater reliability that distributed file systems need to support, and they have resulted in file systems with substantially improved performance. The principal results of this work are:

**Redundant Arrays of Inexpensive Disks (RAID):** This is a mode of storage in which data blocks are segmented into fixed-size chunks and stored in ‘stripes’ across several disks, along with redundant error-correcting codes that enable the data blocks to be reconstructed completely and operation to continue normally in the event of disk failures. RAID also produces considerably better performance than a single disk, because the stripes that make up a block are read and written concurrently.

**Log-structured file storage (LFS):** Like Spritely NFS, this technique originated in the Sprite distributed operating system project at Berkeley. The authors observed that as larger amounts of main memory became available for caching in file servers, an increased level of cache hits resulted in excellent read performance, but write performance remained mediocre. This arose from the high latencies associated with writing individual
data blocks to disk and associated updates to metadata blocks (that is, the blocks known as *i-nodes* that hold file attributes and a vector of pointers to the blocks in a file).

The LFS solution is to accumulate a set of writes in memory and then commit them to disk in large, contiguous, fixed-sized segments. These are called *log segments* because the data and metadata blocks are stored strictly in the order in which they were updated. A log segment is 1 Mbyte or larger in size and is stored in a single disk track, removing the disk head latencies associated with writing individual blocks. Fresh copies of updated data and metadata blocks are always written, requiring the maintenance of a dynamic map (in memory with a persistent backup) pointing to the i-node blocks. Garbage collection of stale blocks is also required, with compaction of ‘live’ blocks to leave contiguous areas of storage free for the storage of log segments. The latter is a fairly complex process; it is carried out as a background activity by a component called the *cleaner*. Some sophisticated cleaner algorithms have been developed for it based on the results of simulations. Despite these extra costs, the overall performance gain is outstanding.

**New design approaches:** The availability of high-performance switched networks (such as ATM and switched high-speed Ethernet) have prompted several efforts to provide persistent storage systems that distribute file data in a highly scalable and fault tolerant manner among many nodes on an intranet, separating the responsibilities for reading and writing data from the responsibilities for managing the metadata and servicing client requests. In the following, we outline two such developments.

These approaches scale better than the more centralized servers that we have described in the preceding sections. They generally demand a high level of trust among the computers that cooperate to provide the service, because they include a fairly low level protocol for communication with the nodes holding data (somewhat analogous to a ‘virtual disk’ API). Hence their scope is likely to be limited to a single local network.
**xFs:** A group at the University of California, Berkeley, proposed a serverless network file system architecture and developed a prototype implementation called. Their approach was motivated by three factors:

1. The opportunity provided by fast switched LANs for multiple file servers in a local network to transfer bulk data to clients concurrently;
2. Increased demand for access to shared data;
3. The fundamental limitations of systems based on central file servers.

Concerning (3), they refer to the facts that the construction of high-performance NFS servers requires relatively costly hardware with multiple CPUs, disks and network controllers, and that there are limits to the process of partitioning the file space – i.e., placing shared files in separate file systems mounted on different servers. They also point to the fact that a central server represents a single point of failure.

**Frangipani:** Frangipani is a highly scalable distributed file system developed and deployed at the Digital Systems Research Center (now Compaq Systems Research Center). Its goals are very similar to those of xFS, and like xFS, it approaches them with a design that separates persistent storage responsibilities from other file service actions. But Frangipani’s service is structured as two totally independent layers. The lower layer is provided by the Petal distributed virtual disk System.

**13.6. SUMMARY**

In this unit we introduced file service architecture, case studies, enhancement and further development. Distributed file systems are very heavily employed in organizational computing, and their performance has been the subject of much tuning. NFS has a simple stateless protocol, but it has maintained its early position as the dominant distributed file system technology with the help of some relatively minor enhancements to the protocol, tuned implementations and high-performance hardware support. AFS demonstrated the feasibility of a relatively simple architecture using server state to reduce the cost of maintaining coherent client caches. AFS outperforms NFS in many situations.
13.7. **KEYWORDS**

**File groups:** A file group is a collection of files located on a given server.

**Hierarchic file system:** A hierarchic file system such as the one that UNIX provides consists of a number of directories arranged in a tree structure.

**Stateless servers:** Stateless servers can be restarted after a failure and resume operation without any need for clients or the server to restore any state.

**Flat file service:** The flat file service is concerned with implementing operations on the contents of files.

13.8. **UNIT-END EXERCISES AND ANSWERS**

1. What are Distributed file system requirements?
2. What is transparency? Explain various forms of transparencies.
3. Explain SUN NFS and Andrew File System.

**Answers:** SEE

11. 13.2
12. 13.2
13. 13.3

13.9 **SUGGESTED READINGS**

- DISTRIBUTED SYSTEMS Principles and Paradigm By: [Andrew S. Tanenbaum](http://example.com), Maarten Van Steen.
UNIT 14: NAME SERVICES

Structure:
14.0 Objectives
14.1 Introduction
14.2 Domain name system
14.3 Directory services
14.4 Global name services
14.5 Summary
14.6 Keywords
14.7 Unit-end exercises and answers
14.8 Suggested readings

14.0 OBJECTIVES

At the end of this unit you will be able to know:

- Domain name system
- Directory services
- Global name services

14.1 INTRODUCTION

This unit introduces the name service as a distinct service that is used by client processes to obtain attributes such as the addresses of resources or objects when given their names. The entities named can be of many types, and they may be managed by different services. Basic design issues for name services, such as the structure and management of the space of names recognized by the service and the operations that the name service supports, are outlined and illustrated in the context of the Internet Domain Name System (DNS).

In a distributed system, names are used to refer to a wide variety of resources such as computers, services, remote objects and files, as well as to users. Names facilitate communication and resource sharing. A name is needed to request a computer system to act upon a specific resource chosen out of many. Names are not the only useful means of identification: descriptive attributes are another.
Names, addresses and other attributes

Any process that requires access to a specific resource must possess a name or an identifier for it. Examples of human-readable names are file names such as `/etc/passwd`, URLs such as `http://www.cdk5.net/` and Internet domain names such as `www.cdk5.net`. The term *identifier* is sometimes used to refer to names that are interpreted only by programs. Remote object references and NFS file handles are examples of identifiers. Identifiers are chosen for the efficiency with which they can be looked up and stored by software.

We say that a name is *resolved* when it is translated into data about the named resource or object, often in order to invoke an action upon it. The association between a name and an object is called a *binding*. In general, names are bound to *attributes* of the named objects, rather than the implementation of the objects themselves. An attribute is the value of a property associated with an object. A key attribute of an entity that is usually relevant in a distributed system is its address. For example:

- The DNS maps domain names to the attributes of a host computer: its IP address, the type of entry (for example, a reference to a mail server or another host) and, for example, the length of time the host’s entry will remain valid.
- The X500 directory service can be used to map a person’s name onto attributes including their email address and telephone number.
- The CORBA Naming Service and Trading Service: The Naming Service maps the name of a remote object onto its remote object reference, whereas the Trading Service maps the name of a remote object onto its remote object reference, together with an arbitrary number of attributes describing the object in terms understandable by human users.
Figure 4.2.1 shows the domain name portion of a URL resolved first via the DNS into an IP address and then, at the final hop of Internet routing, via ARP to an Ethernet address for the web server. The last part of the URL is resolved by the file system on the web server to locate the relevant file.

**Names and services:** Many of the names used in a distributed system are specific to some particular service. Also, a client may use a service-specific name when requesting a service to perform an operation upon a named object or resource that it manages. For example, a file name is given to the file service when requesting that the file be deleted, and a process identifier is presented to the process management service when requesting that it be sent a signal. These names are used only in the context of the service that manages the objects named, except when clients communicate about shared objects.

Names are also sometimes needed to refer to entities in a distributed system that is beyond the scope of any single service. The major examples of these entities are users.
Uniform Resource Identifiers: Uniform Resource Identifiers (URIs) came about from the need to identify resources on the Web, and other Internet resources such as electronic mailboxes. An important goal was to identify resources in a coherent way, so that they could all be processed by common software such as browsers. URIs are ‘uniform’ in that their syntax incorporates that of indefinitely many individual types of resource identifiers (that is, URI schemes), and there are procedures for managing the global namespace of schemes. The advantage of uniformity is that it eases the process of introducing new types of identifier, as well as using existing types of identifier in new contexts, without disrupting existing usage.

Uniform Resource Locators: Some URIs contain information that can be used to locate and access a resource; others are pure resource names. The familiar term Uniform Resource Locator (URL) is often used for URIs that provide location information and specify the method for accessing the resource. For example, http://www.cdk5.net/ identifies a web page at the given path (‘/’) on the host www.cdk5.net, and specifies that the HTTP protocol be used to access it. Another example is a ‘mailto’ URL, such as mailto:fred@flintstone.org, which identifies the mailbox at the given address.

URLs are efficient identifiers for accessing resources. But they suffer from the disadvantage that if a resource is deleted or if it moves, say from one web site to another, there may be dangling links to the resource containing the old URL. If a user clicks on a dangling link to a web resource, then the web server will either respond that the resource is not found or – worse, perhaps – supply a different resource that now occupies the same location.

Uniform Resource Names: Uniform Resource Names (URNs) are URIs that is used as pure resource names rather than locators. For example, the URI: mid:0E4FC272-5C02-11D9-B115-000A95B55BC8@hpl.hp.com is a URN that identifies the email message
containing it in its ‘Message-Id’ field. The URI distinguishes that message from any other email message. But it does not provide the message’s address in any store, so a lookup operation is needed to find it.

A special sub-tree of URIs beginning with urn: has been reserved for URNs – although, as the mid: example shows, not all URNs are urn: URIs. The latter urn-prefixed URIs are all of the form urn:nameSpace:nameSpace-specificName.

14.2 THE DOMAIN NAME SYSTEM

A name service stores information about a collection of textual names, in the form of bindings between the names and the attributes of the entities they denote, such as users, computers, services and objects. The collection is often subdivided into one or more naming contexts: individual subsets of the bindings that are managed as a unit. The major operation that a name service supports is to resolve a name – that is, to look up attributes from a given name. Operations are also required for creating new bindings, deleting bindings and listing bound names, and adding and deleting contexts.

The Domain Name System is a name service design whose main naming database is used across the Internet. It was devised principally by Mockapetris and specified in RFC 1034 and RFC 1035. DNS replaced the original Internet naming scheme, in which all host names and addresses were held in a single central master file and downloaded by FTP to all computers that required them.

This original scheme was soon seen to suffer from three major shortcomings:

- It did not scale to large numbers of computers.
- Local organizations wished to administer their own naming systems.
- A general name service was needed – not one that serves only for looking up computer addresses.
The objects named by the DNS are primarily computers – for which mainly IP addresses are stored as attributes.

**Domain names:** The DNS is designed for use in multiple implementations, each of which may have its own name space. In practice, however, only one is in widespread use, and that is the one used for naming across the Internet. The Internet DNS name space is partitioned both organizationally and according to geography. The names are written with the highest-level domain on the right. The original top-level organizational domains (also called *generic domains*) in use across the Internet were:

- **com** – Commercial organizations
- **edu** – Universities and other educational institutions
- **gov** – US governmental agencies
- **mil** – US military organizations
- **net** – Major network support centres
- **org** – Organizations not mentioned above
- **int** – International organizations

New top-level domains such as *biz* and *mobi* have been added since the early 2000s.

In addition, every country has its own domains:

- **us** – United States
- **uk** – United Kingdom
- **fr** – France
- ...

Countries, particularly those other than the US often use their own subdomains to distinguish their organizations. The UK, for example, has domains *co.uk* and *ac.uk*, which correspond to *com* and *edu* respectively.
**DNS queries:** The Internet DNS is primarily used for simple host name resolution and for looking up electronic mail hosts, as follows:

*Host name resolution:* In general, applications use the DNS to resolve host names into IP addresses. For example, when a web browser is given a URL containing the domain name `www.dcs.qmul.ac.uk`, it makes a DNS enquiry and obtains the corresponding IP address.

*Mail host location:* Electronic mail software uses the DNS to resolve domain names into the IP addresses of mail hosts – i.e., computers that will accept mail for those domains. For example, when the address `tom@dcs.rnx.ac.uk` is to be resolved, the DNS is queried with the address `dcs.rnx.ac.uk` and the type designation ‘mail’. It returns a list of domain names of hosts that can accept mail for `dcs.rnx.ac.uk`, if such exist (and, optionally, the corresponding IP addresses). The DNS may return more than one domain name so that the mail software can try alternatives if the main mail host is unreachable for some reason.

In principle, the DNS can be used to store arbitrary attributes. A query is specified by a domain name, class and type. For domain names in the Internet, the class is IN. The type of query specifies whether an IP address, a mail host, a name server or some other type of information is required.

**DNS name servers:** The problems of scale are treated by a combination of partitioning the naming database and replicating and caching parts of it close to the points of need. The DNS database is distributed across a logical network of servers. Each server holds part of the naming database – primarily data for the local domain. Queries concerning computers in the local domain are satisfied by servers within that domain. However, each server records the domain names and addresses of other name servers, so that queries pertaining to objects outside the domain can be satisfied.

The DNS naming data are divided into zones. A zone contains the following data:
• Attribute data for names in a domain, less any subdomains administered by lower level authorities.
• The names and addresses of at least two name servers that provide authoritative data for the zone. These are versions of zone data that can be relied upon as being reasonably up-to-date.
• The names of name servers that hold authoritative data for delegated subdomains; and ‘glue’ data giving the IP addresses of these servers.
• Zone-management parameters, such as those governing the caching and replication of zone data.

A server may hold authoritative data for zero or more zones. So that naming data are available even when a single server fails, the DNS architecture specifies that each zone must be replicated authoritatively in at least two servers.

**Navigation and query processing:** A DNS client is called a *resolver*. It is normally implemented as library software. It accepts queries, formats them into messages in the form expected under the DNS protocol and communicates with one or more name servers in order to satisfy the queries. A simple request-reply protocol is used, typically using UDP packets on the Internet (DNS servers use a well-known port number). The resolver times out and resends its query if necessary. The resolver can be configured to contact a list of initial name servers in order of preference in case one or more are unavailable.

<table>
<thead>
<tr>
<th>Record type</th>
<th>Meaning</th>
<th>Main contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>A computer address</td>
<td>IP number</td>
</tr>
<tr>
<td>NS</td>
<td>An authoritative name server</td>
<td>Domain name for server</td>
</tr>
<tr>
<td>CNAME</td>
<td>The canonical name for an alias</td>
<td>Domain name for alias</td>
</tr>
<tr>
<td>SOA</td>
<td>Marks the start of data for a zone</td>
<td>Parameters governing the zone</td>
</tr>
<tr>
<td>WKS</td>
<td>A well-known service description, e.g., hostnames, or hostnames and protocols</td>
<td>List of service names and protocols</td>
</tr>
<tr>
<td>PTR</td>
<td>Domain name pointer (reverse lookups)</td>
<td>Domain name</td>
</tr>
<tr>
<td>HINFO</td>
<td>Host information</td>
<td>Machine architecture and operating system</td>
</tr>
<tr>
<td>MX</td>
<td>Mail exchange</td>
<td>List of preference, hosts</td>
</tr>
<tr>
<td>TXT</td>
<td>Text string</td>
<td>Arbitrary text</td>
</tr>
</tbody>
</table>

**Figure 14.2.2:** DNS resource records
The DNS architecture allows for recursive navigation as well as iterative navigation. The resolver specifies which type of navigation is required when contacting a name server. However, name servers are not bound to implement recursive navigation.

In order to save on network communication, the DNS protocol allows for multiple queries to be packed into the same request message and for name servers correspondingly to send multiple replies in their response messages.

**Resource records:** Zone data are stored by name servers in files in one of several fixed types of resource record. For the Internet database, these include the types given in Figure 4.2.3. Each record refers to a domain name, which is not shown. The data for a zone starts with an SOA-type record, which contains the zone parameters that specify, for example, the version number and how often secondaries should refresh their copies. This is followed by a list of records of type NS specifying the name servers for the domain and a list of records of type MX giving the domain names of mail hosts, each prefixed by a number expressing its preference.

Further records of type A later in the database give the IP addresses for the two name servers *dns0* and *dns1*. The IP addresses of the mail hosts and the third name server are given in the databases corresponding to their domains.

<table>
<thead>
<tr>
<th>domain</th>
<th>name</th>
<th>time to live</th>
<th>class</th>
<th>type</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>dcs.qmul.ac.uk</td>
<td>IN</td>
<td>NS</td>
<td>dns0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>dcs.qmul.ac.uk</td>
<td>IN</td>
<td>NS</td>
<td>dns1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>dcs.qmul.ac.uk</td>
<td>IN</td>
<td>MX</td>
<td>mail1.qmul.ac.uk</td>
<td></td>
<td></td>
</tr>
<tr>
<td>dcs.qmul.ac.uk</td>
<td>IN</td>
<td>MX</td>
<td>mail2.qmul.ac.uk</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Figure 14.2.3:** DNS zone data records
The majority of the remainder of the records in a lower-level zone like `dcs.qmul.ac.uk` will be of type A and map the domain name of a computer onto its IP address. They may contain some aliases for the well-known services, for example:

<table>
<thead>
<tr>
<th>domain name</th>
<th>time to live</th>
<th>class</th>
<th>type</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>www</td>
<td>1D</td>
<td>IN</td>
<td>CNAME</td>
<td>traffic</td>
</tr>
<tr>
<td>traffic</td>
<td>1D</td>
<td>IN</td>
<td>A</td>
<td>138.37.95.150</td>
</tr>
</tbody>
</table>

If the domain has any sub-domains, there will be further records of type NS specifying their name servers, which will also have individual A entries. For example, at one point the database for `qmul.ac.uk` contained the following records for the name servers in its sub-domain `dcs.qmul.ac.uk`:

<table>
<thead>
<tr>
<th>domain name</th>
<th>time to live</th>
<th>class</th>
<th>type</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>dcs</td>
<td>1D</td>
<td>IN</td>
<td>NS</td>
<td>dns0.dcs</td>
</tr>
<tr>
<td>dns0.dcs</td>
<td>1D</td>
<td>IN</td>
<td>A</td>
<td>138.37.88.249</td>
</tr>
<tr>
<td>dcs</td>
<td>1D</td>
<td>IN</td>
<td>NS</td>
<td>dns1.dcs</td>
</tr>
<tr>
<td>dns1.dcs</td>
<td>1D</td>
<td>IN</td>
<td>A</td>
<td>138.37.94.248</td>
</tr>
</tbody>
</table>

**Load sharing by name servers:** At some sites, heavily used services such as the Web and FTP are supported by a group of computers on the same network. In this case, the same domain name is used for each member of the group. When a domain name is shared by several computers, there is one record for each computer in the group, giving its IP address. By default, the name server responds to queries for which multiple records match the requested name by returning the IP addresses according to a round-robin schedule. Successive clients are given access to different servers so that the servers can share the workload. Caching has a potential for spoiling this scheme, for once a non-authoritative name server or a client has the server’s address in its cache it will continue to use it. To counteract this effect, the records are given a short time to live.
The BIND implementation of the DNS: The Berkeley Internet Name Domain (BIND) is an implementation of the DNS for computers running UNIX. Client programs link in library software as the resolver. DNS name server computers run the named daemon.

BIND allows for three categories of name server: primary servers, secondary servers and caching-only servers. The named program implements just one of these types, according to the contents of a configuration file. The first two categories are described as above. Caching-only servers read in from a configuration file sufficient names and addresses of authoritative servers to resolve any name. Thereafter, they only store this data and data that they learn by resolving names for clients.

14.3 DIRECTORY SERVICES

We have described how name services store collections of <name, attribute> pairs, and how the attributes are looked up from a name. It is natural to consider the dual of this arrangement, in which attributes are used as values to be looked up. In these services, textual names can be considered to be just another attribute. Sometimes users wish to find a particular person or resource, but they do not know its name, only some of its other attributes.

A service that stores collections of bindings between names and attributes and that looks up entries that match attribute-based specifications is called a directory service. Examples are Microsoft’s Active Directory Services, X.500 and its cousin LDAP, Univers and Profile.

Directory services are sometimes called yellow pages services, and conventional name services are correspondingly called white pages services, in an analogy with the traditional types of telephone directory. Directory services are also sometimes known as attribute-based name services.

UDDI aside, the term discovery service normally denotes the special case of a directory service for services provided by devices in a spontaneous networking environment.
Devices in spontaneous networks are liable to connect and disconnect unpredictably. One core difference between a discovery service and other directory services is that the address of a directory service is normally well known and preconfigured in clients, whereas a device entering a spontaneous networking environment has to resort to multicast navigation, at least the first time it accesses the local discovery service.

Attributes are clearly more powerful than names as designators of objects: programs can be written to select objects according to precise attribute specifications where names might not be known. Another advantage of attributes is that they do not expose the structure of organizations to the outside world, as do organizationally partitioned names. However, the relative simplicity of use of textual names makes them unlikely to be replaced by attribute-based naming in many applications.

14.4 CASE STUDY: THE GLOBAL NAME SERVICE

A Global Name Service (GNS) was designed and implemented by Lampson and colleagues at the DEC Systems Research Center to provide facilities for resource location, mail addressing and authentication. The design goals of the GNS reflect the fact that a name service for use in an internetwork must support a naming database that may extend to include the names of millions of computers and (eventually) email addresses for billions of users. The designers of the GNS also recognized that the naming database is likely to have a long lifetime and that it must continue to operate effectively while it grows from small to large scale and while the network on which it is based evolves. The structure of the name space may change during that time to reflect changes in organizational structures.

The service should accommodate changes in the names of the individuals, organizations and groups that it holds, and changes in the naming structure.

The potentially large naming database and the scale of the distributed environment in which the GNS is intended to operate make the use of caching essential and render it
extremely difficult to maintain complete consistency between all copies of a database entry. The GNS manages a naming database that is composed of a tree of directories holding names and values. Each directory is also assigned an integer, which serves as a unique directory identifier (DI).

A directory contains a list of names and references. The values stored at the leaves of the directory tree are organized into value trees, so that the attributes associated with names can be structured values. Names in the GNS have two parts: <directory name, value name>. The first part identifies a directory; the second refers to a value tree, or some portion of a value tree. For example, see Figure 4.2.4, in which the DIs are illustrated as small integers.

The directory tree is partitioned and stored in many servers, with each partition replicated in several servers. The consistency of the tree is maintained in the face of two or more concurrent updates – for example, two users may simultaneously attempt to create entries with the same name, and only one should succeed. Replicated directories present a second consistency problem; this is addressed by an asynchronous update distribution algorithm that ensures eventual consistency, but with no guarantee that all copies are always current.

![GNS directory tree and value tree for user Peter.Smith](image)

**Figure 14.2.4: GNS directory tree and value tree for user Peter.Smith**
Accommodating change: We now turn to the aspects of the design that are concerned with accommodating growth and change in the structure of the naming database. At the level of clients and administrators, growth is accommodated through extension of the directory tree in the usual manner. But we may wish to integrate the naming trees of two previously separate GNS services.

![Diagram of directory structure](image)

**Figure 4.2.5** Merging trees under a new root

The working root for each program must be identified as part of its execution environment (much as is done for a program’s working directory).

The GNS also supports the restructuring of the database to accommodate organizational change. Figure 4.2.6 shows the new directory tree.

![Diagram of directory structure](image)

**Figure 4.2.6:** Restructuring the directory
14.5. SUMMARY

In this unit we studied domain name system, directory services and global name services.

14.6. KEYWORDS

DNS: Domain Name Service
URI: Uniform Resource Identifier
URL: Uniform Resource Locator
URN: Uniform resource name

14.7. UNIT-END EXERCISES AND ANSWERS

15. Explain Names, addresses and other attributes.
17. What is a Domain Name System (DNS)? Explain Domain names
18. Write a note DNS name servers.
19. Write a short on Global Name Service (GNS).

Answers: SEE

14. 2.1
15. 2.1
16. 2.2
17. 2.2
18. 2.4

14.8  SUGGESTED READINGS

- DISTRIBUTED SYSTEMS Principles and Paradigm By: Andrew S. Tanenbaum, Maarten Van Steen.
UNIT 15: PEER-TO-PEER SYSTEMS

Structure:
15.0 Objectives
15.1 Introduction
15.2 Peer-to-Peer systems
15.3 Routing overlays
15.4 Overlays case studies
15.5 Application case studies
15.6 Summary
15.7 Keywords
15.8 Unit-end exercises and answers
15.9 Suggested readings

15.0 OBJECTIVES

At the end of this unit you will be able to know:

- Peer-to-Peer systems
- Routing overlays
- Overlays case studies
- Application case studies

15.1 INTRODUCTION

In this unit we study peer-to-peer systems. Peer-to-peer systems represent a paradigm for the construction of distributed systems and applications in which data and computational resources are contributed by many hosts on the Internet, all of which participate in the provision of a uniform service. Their emergence is a consequence of the very rapid growth of the Internet, embracing many millions of computers and similar numbers of users requiring access to shared resources.
Peer-to-peer middleware systems are emerging that have the capacity to share computing resources, storage and data present in computers ‘at the edges of the Internet’ on a global scale. They exploit existing naming, routing, data replication and security techniques in new ways to build a reliable resource-sharing layer over an unreliable and untrusted collection of computers and networks.

Peer-to-peer applications have been used to provide file sharing, web caching, information distribution and other services, exploiting the resources of tens of thousands of machines across the Internet.

15.2 PEER-TO-PEER SYSTEMS

The demand for services in the Internet can be expected to grow to a scale that is limited only by the size of the world’s population. The goal of peer-to-peer systems is to enable the sharing of data and resources on a very large scale by eliminating any requirement for separately managed servers and their associated infrastructure.

Peer-to-peer systems aim to support useful distributed services and applications using data and computing resources available in the personal computers and workstations that are present in the Internet and other networks in ever-increasing numbers. This is increasingly attractive as the performance difference between desktop and server machines narrows and broadband network connections proliferate. But there is another, broader aim: one author has defined peer-to-peer applications as ‘applications that exploit resources available at the edges of the Internet – storage, cycles, content, human presence’. Each type of resource sharing mentioned in that definition is already represented by distributed applications available for most types of personal computer. The purpose of this unit is to describe some general techniques that simplify the construction of peer-to-peer applications and enhance their scalability, reliability and security.
Traditional client-server systems manage and provide access to resources such as files, web pages or other information objects located on a single server computer or a small cluster of tightly coupled servers. With such centralized designs, few decisions are required about the placement of the resources or the management of server hardware resources, but the scale of the service is limited by the server hardware capacity and network connectivity. Peer-to-peer systems provide access to information resources located on computers throughout a network (whether it be the Internet or a corporate network). Algorithms for the placement and subsequent retrieval of information objects are a key aspect of the system design. The aim is to deliver a service that is fully decentralized and self-organizing, dynamically balancing the storage and processing loads between all the participating computers as computers join and leave the service.

Peer-to-peer systems share these characteristics:

- Their design ensures that each user contributes resources to the system.
- Although they may differ in the resources that they contribute, all the nodes in a peer-to-peer system have the same functional capabilities and responsibilities.
- Their correct operation does not depend on the existence of any centrally administered systems.
- They can be designed to offer a limited degree of anonymity to the providers and users of resources.
- A key issue for their efficient operation is the choice of an algorithm for the placement of data across many hosts and subsequent access to it in a manner that balances the workload and ensures availability without adding undue overheads.

Computers and network connections owned and managed by a multitude of different users and organizations are necessarily volatile resources; their owners do not guarantee to keep them switched on, connected and fault-free. So the availability of the processes and computers participating in peer-to-peer systems is unpredictable. Peer-to-peer services therefore cannot rely on guaranteed access to individual resources, although they can be designed to make the probability of failure to access a copy of a replicated object arbitrarily small. It is worth noting that this weakness of peer-to-peer systems can be
turned into strength if the replication of resources that it calls for is exploited to achieve a degree of resistance to tampering by malicious nodes.

But the potential for the deployment of peer-to-peer services using resources at the edges of the Internet emerged only when a significant number of users had acquired always-on, broadband connections to the network, making their desktop computers suitable platforms for resource sharing.

**Peer-to-peer middleware:** The third generation is characterized by the emergence of middleware layers for the application-independent management of distributed resources on a global scale. Several research teams have now completed the development, evaluation and refinement of peer-to-peer middleware platforms and demonstrated or deployed them in a range of application services. The best-known and most fully developed examples include Pastry, Tapestry, CAN, Chord and Kademlia.

These platforms are designed to place resources (data objects, files) on a set of computers that are widely distributed throughout the Internet and to route messages to them on behalf of clients, relieving clients of any need to make decisions about placing resources and to hold information about the whereabouts of the resources they require.

Unlike the second-generation systems, they provide guarantees of delivery for requests in a bounded number of network hops. They place replicas of resources on available host computers in a structured manner, taking account of their volatile availability, their variable trustworthiness and requirements for load balancing and locality of information storage and use.

Resources are identified by globally unique identifiers (GUIDs), usually derived as a secure hash from some or all of the resource’s state. The use of a secure hash makes a resource ‘self-certifying’ – clients receiving a resource can check the validity of the hash. This protects it against tampering by untrusted nodes on which it may be stored, but this technique requires that the states of resources are immutable, since a change to the
state would result in a different hash value. Hence peer-to-peer storage systems are inherently best suited to the storage of immutable objects (such as music or video files). Their use for objects with changing values is more challenging, but this can be accommodated by the addition of trusted servers to manage a sequence of versions and identify the current version.

<table>
<thead>
<tr>
<th>Node</th>
<th>IP</th>
<th>Application-level routing layer</th>
</tr>
</thead>
<tbody>
<tr>
<td>IPv4 is unicast addressed, the IPv6 name space is much larger and non-overlapping. Each identifier is hierarchically structured such that the space is logically divided into administrative requirements.</td>
<td>Peer-to-peer systems can address objects. The GUID name space is very large (32 bytes), allowing it to be multi-byte occupied.</td>
<td></td>
</tr>
<tr>
<td>Load balancing</td>
<td>Loads on routers are determined by the network topology and associated traffic patterns.</td>
<td>Object locations can be randomized and hence traffic patterns are divorced from the network topology.</td>
</tr>
<tr>
<td>Network dynamics (addition/deletion of nodes)</td>
<td>IP routing tables are updated dynamically. Routing tables can be calculated in an asynchronous fashion every second.</td>
<td>Routers see each other as replicated peers, ensuring resilience against failures.</td>
</tr>
<tr>
<td>Fault tolerance</td>
<td>Redundancy is introduced on a network-by-network basis, ensuring tolerance of multiple failures.</td>
<td>Routers and object management operations are not connected and failures are isolated.</td>
</tr>
<tr>
<td>Target identifiability</td>
<td>Each IP address maps to exactly one target, ensuring uniqueness.</td>
<td>Security can be achieved in environments where anonymity is not achievable.</td>
</tr>
<tr>
<td>Security anonymity</td>
<td>Addressing is only known to those with knowledge of the protocol.</td>
<td>Security can be achieved in environments where anonymity is not achievable.</td>
</tr>
</tbody>
</table>

Figure 15.3.1: Distinctions between IP and overlay routing for peer-to-peer applications

The use of peer-to-peer systems for applications that demand a high level of availability for the objects stored requires careful application design to avoid situations in which all of the replicas of an object are simultaneously unavailable. There is a risk of this for objects stored on computers with the same ownership, geographic location, administration, network connectivity, country or jurisdiction. The use of randomly distributed GUIDs assists by distributing the object replicas to randomly located nodes in the underlying network. If the underlying network spans many organizations across the globe, then the risk of simultaneous unavailability is much reduced.

Overlay routing versus IP routing: At first sight, routing overlays share many characteristics with the IP packet routing infrastructure that constitutes the primary communication mechanism of the Internet. It is therefore legitimate to ask why an additional application-level routing mechanism is required in peer-to-peer systems. The
answer lies in several distinctions that are identified in Figure 4.3.1. It may be argued that some of these distinctions arise from the ‘legacy’ nature of IP as the Internet’s primary protocol, but the legacy’s impact is too strong for it to be overcome in order to support peer-to-peer applications more directly.

**Distributed computation:** The exploitation of spare computing power on end-user computers has long been a subject of interest and experiment. Work with the first personal computers at Xerox PARC showed the feasibility of performing loosely coupled compute-intensive tasks by running background processes on ~100 personal computers linked by a local network. More recently, much larger numbers of computers have been put to use to perform several scientific calculations that require almost unlimited quantities of computing power.

### 15.3 ROUTING OVERLAYS

The development of middleware that meets the functional and non-functional requirements have already emerged.

In peer-to-peer systems a distributed algorithm known as a *routing overlay* takes responsibility for locating nodes and objects. The name denotes the fact that the middleware takes the form of a layer that is responsible for routing requests from any client to a host that holds the object to which the request is addressed. The objects of interest may be placed at and subsequently relocated to any node in the network without client involvement. It is termed an overlay since it implements a routing mechanism in the application layer that is quite separate from any other routing mechanisms deployed at the network level such as IP routing. This approach to the management and location of replicated objects was first analyzed and shown to be effective for networks involving sufficiently many nodes in a groundbreaking paper by Plaxton.

The routing overlay ensures that any node can access any object by routing each request through a sequence of nodes, exploiting knowledge at each of them to locate the destination object. Peer-to-peer systems usually store multiple replicas of objects to
ensure availability. In that case, the routing overlay maintains knowledge of the location of all the available replicas and delivers requests to the nearest ‘live’ node (i.e. one that has not failed) that has a copy of the relevant object.

The GUIDs used to identify nodes and objects are an example of the ‘pure’. These are also known as opaque identifiers, since they reveal nothing about the locations of the objects to which they refer.

The main task of a routing overlay is the following:

**Routing of requests to objects:** A client wishing to invoke an operation on an object submits a request including the object’s GUID to the routing overlay, which routes the request to a node at which a replica of the object resides.

But the routing overlay must also perform some other tasks:

**Insertion of objects:** A node wishing to make a new object available to a peer-to-peer service computes a GUID for the object and announces it to the routing overlay, which then ensures that the object is reachable by all other clients.

**Deletion of objects:** When clients request the removal of objects from the service the routing overlay must make them unavailable.

**Node addition and removal:** Nodes (i.e., computers) may join and leave the service. When a node joins the service, the routing overlay arranges for it to assume some of the responsibilities of other nodes. When a node leaves (either voluntarily or as a result of a system or network fault), its responsibilities are distributed amongst the other nodes.

An object’s GUID is computed from all or part of the state of the object using a function that delivers a value that is, with very high probability, unique. Uniqueness is verified by searching for another object with the same GUID. A hash is used to generate the GUID.
from the object’s value. Overlay routing systems are sometimes described as distributed hash tables (DHT).

A slightly more flexible form of API is provided by a distributed object location and routing (DOLR) layer, as shown in Figure 4.3.3. With this interface objects can be stored anywhere and the DOLR layer is responsible for maintaining a mapping between object identifiers (GUIDs) and the addresses of the nodes at which replicas of the objects are located. Objects may be replicated and stored with the same GUID at different hosts, and the routing overlay takes responsibility for routing requests to the nearest available replica.

```plaintext
put(GUID, data)
The data is stored in replicas at all nodes responsible for the object identified by GUID.
remove(GUID)
Deletes all references to GUID and the associated data.
value = get(GUID)
The data associated with GUID is retrieved from one of the nodes responsible it.
```

Figure 15.3.2: Basic programming interface for a distributed hash table (DHT) as implemented by the PAST API over Pastry

The interfaces in Figures 4.3.2 and 4.3.2 are based on a set of abstract representations proposed by Dabek to show that most peer-to-peer routing overlay implementations developed to date provide very similar functionality.

We now describe Pastry, which implements a distributed hash table API similar to the one presented in Figure 4.3.2, and Tapestry, which implements an API similar to that shown in Figure 4.3.3. Both Pastry and Tapestry employ a routing mechanism known as prefix routing to determine routes for the delivery of messages based on the values of the GUIDs to which they are addressed.
GUIDs are not human-readable, so client applications must obtain the GUIDs for resources of interest through some form of indexing service using human-readable names or search requests.

```
publish(GUID )
GUID can be computed from the object (or some part of it, e.g. its name). This function makes the node performing a publish operation the host for the object corresponding to GUID.
unpublish(GUID)
Makes the object corresponding to GUID inaccessible.
sendToObj(msg, GUID, [n])
Following the object-oriented paradigm, an invocation message is sent to an object in order to access it. This might be a request to open a TCP connection for data transfer or to return a message containing all or part of the object’s state. The final optional parameter [n], if present, requests the delivery of the same message to n replicas of the object.
```

Figure 15.3.3: Basic programming interface for distributed object location and routing (DOLR) as implemented by Tapestry

15.4 OVERLAY CASE STUDIES: PASTRY, TAPESTRY

The prefix routing approach is adopted by both Pastry and Tapestry. Pastry is the message routing infrastructure deployed in several applications including PAST, an archival (immutable) file storage system.

Tapestry is the basis for the OceanStore storage system. It has a more complex architecture than Pastry because it aims to support a wider range of locality approaches.
Pastry
Pastry is a routing overlay. All the nodes and objects that can be accessed through Pastry are assigned 128-bit GUIDs. For nodes, these are computed by applying a secure hash function to the public key with which each node is provided. For objects such as files, the GUID is computed by applying a secure hash function to the object’s name or to some part of the object’s stored state. The resulting GUIDs have the usual properties of secure hash values – that is, they are randomly distributed in the range 0 to \(2^{128}-1\). They provide no clues as to the value from which they were computed, and clashes between GUIDs for different nodes or objects are extremely unlikely. (If a clash occurs, Pastry detects it and takes remedial action.)

In a network with \(N\) participating nodes, the Pastry routing algorithm will correctly route a message addressed to any GUID in \(O(\log N)\) steps. If the GUID identifies a node that is currently active, the message is delivered to that node; otherwise, the message is delivered to the active node whose GUID is numerically closest to it. Active nodes take responsibility for processing requests addressed to all objects in their numerical neighborhood.

**Routing algorithm:** The full routing algorithm involves the use of a routing table at each node to route messages efficiently, but for the purposes of explanation, we describe the routing algorithm in two stages. The first stage describes a simplified form of the algorithm that routes messages correctly but inefficiently without a routing table, and the second stage describes the full routing algorithm, which routes a request to any node in \(O(\log N)\) messages:

**Stage 1:** Each active node stores a *leaf set* – a vector \(L\) (of size \(2l\)) containing the GUIDs and IP addresses of the nodes whose GUIDs are numerically closest on either side of its own. Leaf sets are maintained by Pastry as nodes join and leave. Even after a node failure, they will be corrected within a short time.
The GUID space is treated as circular: GUID 0’s lower neighbour is $2^{128} - 1$. Figure 4.3.4 gives a view of active nodes distributed in this circular address space. Since every leaf set includes the GUIDs and IP addresses of the current node’s immediate neighbours, a Pastry system with correct leaf sets of size at least 2 can route messages to any GUID. Figure 4.3.4 illustrates this for a Pastry system with $l = 4$.

**Stage II:** The second part of our explanation describes the full Pastry algorithm and shows how efficient routing is achieved with the aid of routing tables.

Each Pastry node maintains a tree-structured routing table giving GUIDs and IP addresses for a set of nodes spread throughout the entire range of $2^{128}$ possible GUID values, with increased density of coverage for GUIDs numerically close to its own.

![Pastry routing table](Figure 15.3.4: First four rows of a Pastry routing table)
Figure 4.3.6 shows the structure of the routing table for a specific node, and Figure 4.3.5 illustrates the actions of the routing algorithm. The routing table is structured as follows: GUIDs are viewed as hexadecimal values and the table classifies GUIDs based on their hexadecimal prefixes. The table has as many rows as there are hexadecimal digits in a GUID, so for the prototype Pastry system that we are describing, there are $128/4 = 32$ rows. Any row $n$ contains 15 entries – one for each possible value of the $n$th hexadecimal digit, excluding the value in the local node’s GUID.

The routing process at any node $A$ uses the information in its routing table $R$ and leaf set $L$ to handle each request from an application and each incoming message from another node according to the **Pastry’s routing algorithm** shown below.

**Pastry’s routing algorithm**

To handle a message $M$ addressed to a node $D$ (where $R[p,i]$ is the element at column $i$, row $p$ of the routing table):

1. If $(L_i < D < L_{i+1})$ \{ // the destination is within the leaf set or is the current node.}
2. Forward $M$ to the element $L_i$ of the leaf set with GUID closest to $D$ or the current node $A$.

3. } else { // use the routing table to despatch $M$ to a node with a closer GUID

4. Find $p$, the length of the longest common prefix of $D$ and $A$, and $i$, the $(p+1)^{th}$ hexadecimal digit of $D$.

5. If ($R[p,i] \neq null$) forward $M$ to $R[p,i]$ // route $M$ to a node with a longer common prefix.

6. else { // there is no entry in the routing table.

7. Forward $M$ to any node in $L$ or $R$ with a common prefix of length $p$ but a GUID that is numerically closer.

} } } 

We can be sure that the algorithm will succeed in delivering $M$ to its destination because lines 1, 2 and 7 perform the actions described in Stage I of our description above, and we have shown this to be a complete, although inefficient, routing algorithm. The remaining steps are designed to use the routing table to improve the algorithm’s performance by reducing the number of hops required.

Lines 4–5 come into play whenever $D$ does not fall within the numeric range of the current node’s leaf set and relevant routing table entries are available. The selection of a destination for the next hop involves comparing the hexadecimal digits of $D$ with those of $A$ (the GUID of the current node) from left to right to discover the length, $p$, of their longest common prefix. This length is then used as a row offset, together with the first non-matching digit of $D$ as a column offset, to access the required element of the routing table. The construction of the table ensures that this element (if not empty) contains the IP address of a node whose GUID has $p+1$ prefix digits in common with $D$.

Line 7 is used when $D$ falls outside the numeric range of the leaf set and there isn’t a relevant routing table entry. This case is rare; it arises only when nodes have recently failed and the table hasn’t yet been updated. The routing algorithm is able to proceed by
scanning both the leaf set and the routing table and forwarding $M$ to another node whose GUID has $p$ matching prefix digits but is numerically closer to $D$.

**Host integration:** New nodes use a joining protocol in order to acquire their routing table and leaf set contents and notify other nodes of changes they must make to their tables. First, the new node computes a suitable GUID, then it makes contact with a nearby Pastry node. Suppose that the new node’s GUID is $X$ and the nearby node it contacts has GUID $A$. Node $X$ sends a special *join* request message to $A$, giving $X$ as its destination. $A$ dispatches the *join* message via Pastry in the normal way. Pastry will route the *join* message to the existing node whose GUID is numerically closest to $X$; let us call this destination node $Z$.

$A$, $Z$ and all the nodes ($B$, $C$,...) through which the *join* message is routed on its way to $Z$ add additional steps to the normal Pastry routing algorithm, which result in the transmission of the contents of the relevant parts of their routing tables and leaf sets to $X$. $X$ examines them and constructs its own routing table and leaf set from them, requesting some additional information from other nodes if necessary.

To see how $X$ builds its routing table, note that the first row of the table depends on the value of $X$’s GUID, and to minimize routing distances, the table should be constructed to route messages via neighboring nodes whenever possible. $A$ is a neighbor of $X$, so the first row of $A$’s table is a good initial choice for the first row of $X$’s table, $X0$. On the other hand, $A$’s table is probably not relevant for the second row, $X1$, because $X$’s and $A$’s GUIDs may not share the same first hexadecimal digit. The routing algorithm ensures that $X$’s and $B$’s GUIDs do share the same first digit, though, which implies that the second row of $B$’s routing table, $B1$, is a suitable initial value for $X1$. Similarly, $C2$ is suitable for $X2$, and so on.

Finally, once $X$ has constructed its leaf set and routing table in the manner outlined above, it sends their contents to all the nodes identified in the leaf set and the routing table and they adjust their own tables to incorporate the new node. The entire task of
incorporating a new node into the Pastry infrastructure requires the transmission of $O(\log N)$ messages.

**Host failure or departure:** Nodes in the Pastry infrastructure may fail or depart without warning. A Pastry node is considered failed when its immediate neighbors (in GUID space) can no longer communicate with it. When this occurs, it is necessary to repair the leaf sets that contain the failed node’s GUID.

Nearest neighbor algorithm: The new node should have the address of at least one existing Pastry node, but it might not be nearby. To ensure that nearby nodes are known Pastry includes a ‘nearest neighbor’ algorithm to find a nearby node by recursively measuring the round-trip delay for a probe message sent periodically to each member of the leaf set of the nearest currently known Pastry node.

To repair its leaf set $L$, the node that discovers the failure looks for a live node close to the failed node in $L$ and requests a copy of that node’s leaf set, $L’$. $L’$ will contain a sequence of GUIDs that partly overlap those in $L$, including one with an appropriate value to replace the failed node. Other neighbouring nodes are then informed of the failure and they perform a similar procedure. This repair procedure guarantees that leaf sets will be repaired unless $l$ adjacentely numbered nodes fail simultaneously.

Repairs to routing tables are made on a ‘when discovered’ basis. The routing of messages can proceed with some routing table entries that are no longer live – failed routing attempts result in the use of a different entry from the same row of a routing table.

**Locality:** The Pastry routing structure is highly redundant. There are many routes between each pair of nodes. The construction of the routing tables aims to take advantage of this redundancy to reduce actual message transmission times by exploiting the locality properties of nodes in the underlying transport network.
**Fault tolerance:** The Pastry routing algorithm assumes that all entries in routing tables and leaf sets refer to live, correctly functioning nodes. All nodes send ‘*heartbeat*’ messages (i.e., messages sent at fixed time intervals to indicate that the sender is alive) to neighboring nodes in their leaf sets, but information about failed nodes detected in this manner may not be disseminated sufficiently rapidly to eliminate routing errors. Nor does it account for malicious nodes that may attempt to interfere with correct routing. To overcome these problems, clients that depend upon reliable message delivery are expected to employ an *at-least-once* delivery mechanism and repeat their requests several times in the absence of a response. This will allow Pastry a longer time window to detect and repair node failures.

**Dependability:** The authors of Pastry have developed an updated version called MSPastry that uses the same routing algorithm and similar host management methods, but also includes some additional dependability measures and some performance optimizations in the host management algorithms.

Dependability measures include the use of acknowledgements at each hop in the routing algorithm. If the sending host does not receive an acknowledgement after a specified timeout, it selects an alternative route and retransmits the message. The node that failed to send an acknowledgement is then noted as a suspected failure.

To detect failed nodes each Pastry node periodically sends a heartbeat message to its immediate neighbor to the left (i.e., with a lower GUID) in the leaf set. Each node also records the time of the last heartbeat message received from its immediate neighbor on the right (with a higher GUID). If the interval since the last heartbeat exceeds a timeout threshold, the detecting node starts a repair procedure that involves contacting the remaining nodes in the leaf set with a notification about the failed node and a request for suggested replacements.
**Evaluation work:** Castro and his colleagues have carried out an exhaustive performance evaluation of MSPastry, aimed at determining the impact on performance and dependability of the host join/leave rate and the associated dependability mechanisms.

The evaluation was performed by running the MSPastry system under control of a simulator running on a single machine that simulates a large network of hosts, with message passing replaced by simulated transmission delays. The simulation realistically modeled the join/leave behavior of hosts and IP transmission delays based on parameters from real installations.

Overall these results show that overlay routing layers can be constructed that achieve good performance and high dependability with thousands of nodes operating in realistic environments.

**Tapestry**

Tapestry implements a distributed hash table and routes messages to nodes based on GUIDs associated with resources using prefix routing in a manner similar to Pastry. But Tapestry’s API conceals the distributed hash table from applications behind a DOLR interface like the one shown in Figure 4.3.3. Nodes that hold resources use `publish (GUID)` primitive to make them known to Tapestry, and the holders of resources remain responsible for storing them. Replicated resources are published with the same GUID by each node that holds a replica, resulting in multiple entries in the Tapestry routing structure.

This gives Tapestry applications additional flexibility: they can place replicas close (in network distance) to frequent users of resources in order to reduce latency and minimize network load or to ensure tolerance of network and host failures. But this distinction between Pastry and Tapestry is not fundamental: Pastry applications can achieve similar flexibility by making the objects associated with GUIDs simply act as proxies for more complex application-level objects and Tapestry can be used to implement a distributed hash table in terms of its DOLR API.
The routing overlay layers described in the preceding section have been exploited in several application experiments and the resulting applications have been extensively evaluated. We have chosen three of them for further study, the Squirrel web caching service based on Pastry, and the OceanStore and Ivy file stores.

**Squirrel web cache**
The authors of Pastry have developed the Squirrel peer-to-peer web caching service for use in local networks of personal computers. In medium and large local networks web caching is typically performed using a dedicated server computer or cluster. The Squirrel system performs the same task by exploiting storage and computing resources already available on desktop computers in the local network. We first give a brief general description of the operation of a web caching service, then we outline the design of Squirrel and review its effectiveness.

**OceanStore file store**
The developers of Tapestry have designed and built a prototype for a peer-to-peer file store. Unlike PAST, it supports the storage of mutable files. The OceanStore design aims to provide a very large scale, incrementally scalable persistent storage facility for mutable data objects with long-term persistence and reliability in an environment of constantly changing network and computing resources. OceanStore is intended for use in a variety of applications including the implementation of an NFS-like file service, electronic mail hosting, databases and other applications involving the sharing and persistent storage of large numbers of data objects.

The design includes provision for the replicated storage of both mutable and immutable data objects. The mechanism for maintaining consistency between replicas can be tailored to application needs in a manner that was inspired by the Bayou system. Privacy and integrity are achieved through the encryption of data and the use of a Byzantine
agreement protocol for updates to replicated objects. This is needed because the trustworthiness of individual hosts cannot be assumed.

**Ivy file system**

Like OceanStore, Ivy is a read/write file system supporting multiple readers and writers implemented over an overlay routing layer and a distributed hash-addressed data store. Unlike OceanStore, the Ivy file system emulates a Sun NFS server. Ivy stores the state of files as logs of the file update requests issued by Ivy clients and reconstructs the files by scanning the logs whenever it is unable to satisfy an access request from its local cache. The log records are held in the DHash distributed hash-addressed storage service.

15.6. **SUMMARY**

In this unit we introduced Peer-to-Peer systems, Routing overlays, Overlays case studies and Application case studies

15.7. **KEYWORDS**

**Peer-to-peer systems:** represent a paradigm for the construction of distributed systems and applications in which data and computational resources are contributed by many hosts on the Internet, all of which participate in the provision of a uniform service.

15.8. **UNIT-END EXERCISES AND ANSWERS**

1. What is the goal of peer-to-peer systems?
2. What characteristics peer-to-peer systems share?
3. Compare Overlay routing versus IP routing.
4. Explain the tasks of a routing overlay.
5. Write Pastry’s routing algorithm.
6. Explain any one overlay case study.
Answers: SEE

1. 15.2
2. 15.2
3. 15.2
4. 15.3
5. 15.4
6. 15.5

15.9 SUGGESTED READINGS

- DISTRIBUTED SYSTEMS Principles and Paradigm By: Andrew S. Tanenbaum, Maarten Van Steen.
UNIT 16:  TIME ANDCLOCKS

Structure:
16.0 Objectives
16.1 Introduction
16.2 Clocks, events, and process states.
16.3 Synchronizing physical clocks
16.4 Global states
16.5 Summary
16.6 Keywords
16.7 Unit-end exercises and answers
16.8 Suggested readings

16.0 OBJECTIVES
At the end of this unit you will be able to know:

- Clocks, events, and process states.
- Synchronizing physical clocks
- Time and global states

16.1 INTRODUCTION

In this unit, we introduce some topics related to the issue of time in distributed systems. Time is an important practical issue. For example, we require computers around the world to timestamp electronic commerce transactions consistently. But time is problematic in distributed systems. Each computer may have its own physical clock, but the clocks typically deviate, and we cannot synchronize them perfectly. We examine algorithms for synchronizing physical clocks approximately and then go on to explain logical clocks, including vector clocks, which are a tool for ordering events without knowing precisely when they occurred.

The absence of global physical time makes it difficult to find out the state of our distributed programs as they execute. We often need to know what state process A is in
when process B is in a certain state, but we cannot rely on physical clocks to know what is true at the same time.

In this unit we introduce fundamental concepts and algorithms related to monitoring distributed systems as their execution unfolds, and to timing the events that occur in their executions.

Time is an important and interesting issue in distributed systems, for several reasons. First, time is a quantity we often want to measure accurately. In order to know at what time of day a particular event occurred at a particular computer it is necessary to synchronize its clock with an authoritative, external source of time.

Second, algorithms that depend upon clock synchronization have been developed for several problems in distribution. These include maintaining the consistency of distributed data, checking the authenticity of a request sent to a server and eliminating the processing of duplicate updates.

Measuring time can be problematic due to the existence of multiple frames of reference.

### 16.2 CLOCKS, EVENTS AND PROCESS STATES

Here we to understand how to characterize the system’s evolution as it executes, and how to timestamp the events in a system’s execution that interest users.

We begin by considering how to order and timestamp the events that occur at a single process.

We take a distributed system to consist of a collection $P$ of $N$ processes $p_i$, $i = 1, 2, \ldots, N$. Each process executes on a single processor, and the processors do not share memory. Each process $p_i$ in $P$ has a state $S_i$ that, in general, it transforms as it executes. The process’s state includes the values of all the variables within it. Its state may also include the values of any objects in its local operating system environment that it affects, such as
files. We assume that processes cannot communicate with one another in any way except by sending messages through the network. So, for example, if the processes operate robot arms connected to their respective nodes in the system, then they are not allowed to communicate by shaking one another’s robot hands!

As each process \( P_i \) executes it takes a series of actions, each of which is either a message send or receive operation, or an operation that transforms \( P_i \)’s state – one that changes one or more of the values in \( S_i \). In practice, we may choose to use a high-level description of the actions, according to the application. We define an event to be the occurrence of a single action that a process carries out as it executes – a communication action or a state-transforming action. The sequence of events within a single process \( P_i \) can be placed in a single, total ordering, which we denote by the relation \( \rightarrow_i \) between the events. That is, \( e \rightarrow_i e’ \) if and only if the event \( e \) occurs before \( e’ \) at \( P_i \). This ordering is well defined, whether or not the process is multithreaded, since we have assumed that the process executes on a single processor.

Now we can define the history of process \( P_i \) to be the series of events that take place within it, ordered as we have described by the relation \( \rightarrow_i \):

\[
\text{history}(P_i) = h_i = \langle e_i^0 \ e_i^1 \ e_i^2 \ldots \rangle
\]

**Clocks:** We have seen how to order the events at a process, but not how to timestamp them – i.e., to assign to them a date and time of day. Computers each contain their own physical clocks. These clocks are electronic devices that count oscillations occurring in standard for elapsed real time, known as *International Atomic Time*. Since 1967, the standard second has been defined as 9,192,631,770 periods of transition between the two hyperfine levels of the ground state of Caesium-133 (Cs133).

**Coordinated Universal Time** – abbreviated as UTC (from the French equivalent) – is an international standard for timekeeping. It is based on atomic time, but a so-called ‘leap second’ is inserted – or, more rarely, deleted – occasionally to keep it in step with
astronomical time. UTC signals are synchronized and broadcast regularly from land-based radio stations and satellites covering many parts of the world.

16.3 SYNCHRONIZING PHYSICAL CLOCKS

In order to know at what time of day events occur at the processes in our distributed system P – for example, for accountancy purposes – it is necessary to synchronize the processes’ clocks, \( C_i \), with an authoritative, external source of time. This is external synchronization. And if the clocks \( C_i \) are synchronized with one another to a known degree of accuracy, then we can measure the interval between two events occurring at different computers by appealing to their local clocks, even though they are not necessarily synchronized to an external source of time. This is internal synchronization. We define these two modes of synchronization more closely as follows, over an interval of real time \( I \):

**External synchronization:** For a synchronization bound \( D > 0 \), and for a source \( S \) of UTC time, \( |S(t) - C_i(t)| < D \), for \( i = 1, 2, \ldots, N \) and for all real times \( t \) in \( I \). Another way of saying this is that the clocks \( C_i \) are accurate to within the bound \( D \).

**Internal synchronization:** For a synchronization bound \( D > 0 \), \( |C_i(t) - C_j(t)| < D \) for \( i, j = 1, 2, \ldots, N \), and for all real times \( t \) in \( I \). Another way of saying this is that the clocks \( C_i \) agree within the bound \( D \).

Clocks that are internally synchronized are not necessarily externally synchronized, since they may drift collectively from an external source of time even though they agree with one another. However, it follows from the definitions that if the system \( P \) is externally synchronized with a bound \( D \), then the same system is internally synchronized with a bound of \( 2D \).

Various notions of *correctness* for clocks have been suggested. It is common to define a hardware clock \( H \) to be correct if its drift rate falls within a known bound \( \rho > 0 \). This
means that the error in measuring the interval between real times \( t \) and \( t' \) \((t' > t)\) is bounded:

\[
(1 - \rho) (t' - t) \leq H(t') - H(t) \leq (1 + \rho) (t' - t)
\]

This condition forbids jumps in the value of hardware clocks (during normal operation). Sometimes we also require our software clocks to obey the condition but a weaker condition of \textit{monotonicity} may suffice. Monotonicity is the condition that a clock \( C \) only ever advances:

\[
T' > t \implies C(t') > C(t)
\]

We can achieve monotonicity despite the fact that a clock is found to be running fast. We need only change the rate at which updates are made to the time as given to applications. This can be achieved in software without changing the rate at which the underlying hardware clock ticks – recall that \( C_i(t) = \alpha H_i(t) + \beta \), where we are free to choose the values of \( \alpha \) and \( \beta \).

A hybrid correctness condition that is sometimes applied is to require that a clock obeys the monotonicity condition, and that its drift rate is bounded between synchronization points, but to allow the clock value to jump ahead at synchronization points.

A clock that does not keep to whatever correctness conditions apply is defined to be \textit{faulty}. A clock’s \textit{crash failure} is said to occur when the clock stops ticking altogether; any other clock failure is an \textit{arbitrary failure}.

Note that clocks do not have to be accurate to be correct, according to the definitions. Since the goal may be internal rather than external synchronization, the criteria for correctness are only concerned with the proper functioning of the clock’s ‘mechanism’, not its absolute setting.
Synchronization in a synchronous system

We begin by considering the simplest possible case: of internal synchronization between two processes in a synchronous distributed system. In a synchronous system, bounds are known for the drift rate of clocks, the maximum message transmission delay, and the time required to execute each step of a process.

One process sends the time $t$ on its local clock to the other in a message $m$. In principle, the receiving process could set its clock to the time $t + T_{\text{trans}}$, where $T_{\text{trans}}$ is the time taken to transmit $m$ between them. The two clocks would then agree.

Unfortunately, $T_{\text{trans}}$ is subject to variation and is unknown. In general, other processes are competing for resources with the processes to be synchronized at their respective nodes, and other messages compete with $m$ for the network resources. Nonetheless, there is always a minimum transmission time, $\text{min}$, that would be obtained if no other processes executed and no other network traffic existed; $\text{min}$ can be measured or conservatively estimated.

In a synchronous system, by definition, there is also an upper bound $\text{max}$ on the time taken to transmit any message. Let the uncertainty in the message transmission time be $u$, so that $u = (\text{max} - \text{min})$. If the receiver sets its clock to be $t + \text{min}$, then the clock skew may be as much as $u$, since the message may in fact have taken time $\text{max}$ to arrive. Similarly, if it sets its clock to $t + \text{max}$, the skew may again be as large as $u$. If, however, it sets its clock to the halfway point, $t + (\text{max} + \text{min})/2$, then the skew is at most $u/2$. In general, for a synchronous system, the optimum bound that can be achieved on clock skew when synchronizing $N$ clocks is $u(1 - 1/N)$.

Most distributed systems found in practice are asynchronous: the factors leading to message delays are not bounded in their effect, and there is no upper bound $\text{max}$ on message transmission delays. This is particularly so for the Internet. For an asynchronous system, we may say only that $T_{\text{trans}} = \text{min} + x$, where $x \geq 0$. The value of $x$ is not known in
a particular case, although a distribution of values may be measurable for a particular installation.

Cristian’s method for synchronizing clocks

Cristian suggested the use of a time server, connected to a device that receives signals from a source of UTC, to synchronize computers externally. Upon request, the server process $S$ supplies the time according to its clock, as shown in Figure 4.4.1.

![Diagram of clock synchronization using a time server.]

Figure 16.4.1: Clock synchronization using a time server.

Cristian observed that while there is no upper bound on message transmission delays in an asynchronous system, the round-trip times for messages exchanged between pairs of processes are often reasonably short—a small fraction of a second. He describes the algorithm as **probabilistic**: the method achieves synchronization only if the observed round-trip times between client and server are sufficiently short compared with the required accuracy.

A process $p$ requests the time in a message $m_r$, and receives the time value $t$ in a message $m_t$ ($t$ is inserted in $m_t$ at the last possible point before transmission from $S$’s computer). Process $p$ records the total round-trip time $T_{round}$ taken to send the request $m_r$ and receive the reply $m_t$. It can measure this time with reasonable accuracy if its rate of clock drift is small. For example, the round-trip time should be on the order of 1–10 milliseconds on a LAN, over which time a clock with a drift rate of $10^{-6}$ seconds/second varies by at most $10^{-5}$ milliseconds.

A simple estimate of the time to which $p$ should set its clock is $t + T_{round} / 2$, which assumes that the elapsed time is split equally before and after $S$ placed $t$ in $m_t$. This is
normally a reasonably accurate assumption, unless the two messages are transmitted over different networks. If the value of the minimum transmission time \( min \) is known or can be conservatively estimated, then we can determine the accuracy of this result as follows.

The earliest point at which \( S \) could have placed the time in \( m_t \) was \( min \) after \( p \) dispatched \( m_t \). The latest point at which it could have done this was \( min \) before \( m_t \) arrived at \( p \). The time by \( S \)’s clock when the reply message arrives is therefore in the range \([t + min, t + T_{\text{round}} - min]\). The width of this range is \( T_{\text{round}} - 2min \), so the accuracy is \( \pm(T_{\text{round}}/2 - min) \).

Variability can be dealt with to some extent by making several requests to \( S \) and taking the minimum value of \( T_{\text{round}} \) to give the most accurate estimate. The greater the accuracy required, the smaller the probability of achieving it. This is because the most accurate results are those in which both messages are transmitted in a time close to \( min \) – an unlikely event in a busy network.

**Discussion of Cristian’s algorithm:** Cristian’s method suffers from the problem associated with all services implemented by a single server: that the single time server might fail and thus render synchronization temporarily impossible.

**The Berkeley algorithm**

Gusella and Zatti describe an algorithm for internal synchronization that they developed for collections of computers running Berkeley UNIX. In it, a coordinator computer is chosen to act as the *master*. Unlike in Cristian’s protocol, this computer periodically polls the other computers whose clocks are to be synchronized, called *slaves*. The slaves send back their clock values to it. The master estimates their local clock times by observing the round-trip times (similarly to Cristian’s technique), and it averages the values obtained (including its own clock’s reading). The balance of probabilities is that this average cancels out the individual clocks’ tendencies to run fast or slow. The accuracy of the protocol depends upon a nominal maximum round-trip time between the master and the
slaves. The master eliminates any occasional readings associated with larger times than this maximum.

Instead of sending the updated current time back to the other computers – which would introduce further uncertainty due to the message transmission time – the master sends the amount by which each individual slave’s clock requires adjustment. This can be a positive or negative value.

The Berkeley algorithm eliminates readings from faulty clocks. Such clocks could have a significant adverse effect if an ordinary average was taken so instead the master takes a fault-tolerant average. That is, a subset is chosen of clocks that do not differ from one another by more than a specified amount, and the average is taken of readings from only these clocks.

Should the master fail, then another can be elected to take over and function exactly as its predecessor.

**The Network Time Protocol**

Cristian’s method and the Berkeley algorithm are intended primarily for use within intranets. The Network Time Protocol (NTP) defines an architecture for a time service and a protocol to distribute time information over the Internet. NTP’s chief design aims and features are as follows:

*To provide a service enabling clients across the Internet to be synchronized accurately to UTC:* Although large and variable message delays are encountered in Internet communication, NTP employs statistical techniques for the filtering of timing data and it discriminates between the quality of timing data from different servers.

*To provide a reliable service that can survive lengthy losses of connectivity:* There are redundant servers and redundant paths between the servers. The servers can reconfigure so as to continue to provide the service if one of them becomes unreachable.
To enable clients to resynchronize sufficiently frequently to offset the rates of drift found in most computers: The service is designed to scale to large numbers of clients and servers.

To provide protection against interference with the time service, whether malicious or accidental: The time service uses authentication techniques to check that timing data originate from the claimed trusted sources. It also validates the return addresses of messages sent to it.

The NTP service is provided by a network of servers located across the Internet. Primary servers are connected directly to a time source such as a radio clock receiving UTC; secondary servers are synchronized, ultimately, with primary servers. The servers are connected in a logical hierarchy called a synchronization subnet (see Figure 4.4.2) whose levels are called strata. Primary servers occupy stratum 1: they are at the root. Stratum 2 servers are secondary servers that are synchronized directly with the primary servers; stratum 3 servers are synchronized with stratum 2 servers, and so on. The lowest-level (leaf) servers execute in users’ workstations.

![Diagram of synchronization subnet](image)

Note: Arrows denote synchronization control, numbers denote strata.

Figure 16.4.2: An example synchronization subnet in an NTP implementation
The clocks belonging to servers with high stratum numbers are liable to be less accurate than those with low stratum numbers, because errors are introduced at each level of synchronization. NTP also takes into account the total message round-trip delays to the root in assessing the quality of timekeeping data held by a particular server.

The synchronization subnet can reconfigure as servers become unreachable or failures occur. If, for example, a primary server’s UTC source fails, then it can become a stratum 2 secondary server. If a secondary server’s normal source of synchronization fails or becomes unreachable, then it may synchronize with another server.

NTP servers synchronize with one another in one of three modes: multicast, procedure-call and symmetric mode. Multicast mode is intended for use on a high-speed LAN. One or more servers periodically multicasts the time to the servers running in other computers connected by the LAN, which set their clocks assuming a small delay. This mode can achieve only relatively low accuracies, but ones that nonetheless are considered sufficient for many purposes.

Procedure-call mode is similar to the operation of Cristian’s algorithm. In this mode, one server accepts requests from other computers, which it processes by replying with its timestamp (current clock reading). This mode is suitable where higher accuracies are required than can be achieved with multicast, or where multicast is not supported in hardware. For example, file servers on the same or a neighboring LAN that need to keep accurate timing information for file accesses could contact a local server in procedure-call mode.

Finally, symmetric mode is intended for use by the servers that supply time information in LANs and by the higher levels (lower strata) of the synchronization subnet, where the highest accuracies are to be achieved. A pair of servers operating in symmetric mode exchanges messages bearing timing information. Timing data are retained as part of an association between the servers that is maintained in order to improve the accuracy of their synchronization over time.
In all modes, messages are delivered unreliably, using the standard UDP Internet transport protocol. In procedure-call mode and symmetric mode, processes exchange pairs of messages. Each message bears timestamps of recent message events: the local times when the previous NTP message between the pair was sent and received, and the local time when the current message was transmitted. The recipient of the NTP message notes the local time when it receives the message. The four times $T_{i-3}$, $T_{i-2}$, $T_{i-1}$ and $T_i$ are shown in Figure 4.4.3 for the messages $m$ and $m'$ sent between servers $A$ and $B$. Note that in symmetric mode, unlike in Cristian’s algorithm, there can be a nonnegligible delay between the arrival of one message and the dispatch of the next. Also, messages may be lost, but the three timestamps carried by each message are nonetheless valid.

![Figure 16.4.3: Messages exchanged between a pair of NTP peers](image)

For each pair of messages sent between two servers the NTP calculates an offset $o_i$, which is an estimate of the actual offset between the two clocks, and a delay $d_i$, which is the total transmission time for the two messages. If the true offset of the clock at $B$ relative to that at $A$ is $o$, and if the actual transmission times for $m$ and $m'$ are $t$ and $t'$, respectively, then we have:

$$T_{i-2} = T_{i-3} + t + o \text{ and } T_i = T_{i-1} + t' - o$$

This leads to:

$$d_i = t + t' = T_{i-2} - T_{i-3} + T_i - T_{i-1}$$
and:

\[
o = o_i + (t' - t)/2, \text{ where } o_i = (T_{t-2} - T_{t-3} + T_{t-1} - T_t)/2
\]

Using the fact that \( t, t' \geq 0 \), it can be shown that \( o_i - d_i/2 \leq o \leq o_i + d_i/2 \). Thus \( o_i \) is an estimate of the offset, and \( d_i \) is a measure of the accuracy of this estimate.

NTP servers apply a data filtering algorithm to successive pairs \( <o_i, d_i> \), which estimates the offset \( o \) and calculates the quality of this estimate as a statistical quantity called the *filter dispersion*. A relatively high filter dispersion represents relatively unreliable data. The eight most recent pairs \( <o_i, d_i> \) are retained. As with Cristian’s algorithm, the value of \( o_j \) that corresponds to the minimum value \( d_j \) is chosen to estimate \( o \).

The value of the offset derived from communication with a single source is not necessarily used by itself to control the local clock, however. In general, an NTP server engages in message exchanges with several of its peers. In addition to data filtering applied to exchanges with each single peer, NTP applies a peer-selection algorithm. This examines the values obtained from exchanges with each of several peers, looking for relatively unreliable values. The output from this algorithm may cause a server to change the peer that it primarily uses for synchronization.

Peers with lower stratum numbers are more favored than those in higher strata because they are ‘closer’ to the primary time sources. Also, those with the lowest *synchronization dispersion* are relatively favored. This is the sum of the filter dispersions measured between the server and the root of the synchronization subnet.

NTP employs a phase lock loop model, which modifies the local clock’s update frequency in accordance with observations of its drift rate. To take a simple example, if a clock is discovered always to gain time at the rate of, say, four seconds per hour, then its frequency can be reduced slightly (in software or hardware) to compensate for this. The clock’s drift in the intervals between synchronization is thus reduced.
16.4 GLOBAL STATES

We examine the problem of finding out whether a particular property is true of a distributed system as it executes. We begin by giving the examples of distributed garbage collection, deadlock detection, termination detection and debugging:

**Distributed garbage collection:** An object is considered to be garbage if there are no longer any references to it anywhere in the distributed system. The memory taken up by that object can be reclaimed once it is known to be garbage. To check that an object is garbage, we must verify that there are no references to it anywhere in the system. In Figure 4.4.4(a), process \( p_1 \) has two objects that both have references – one has a reference within \( p_1 \) itself, and \( p_2 \) has a reference to the other. Process \( p_2 \) has one garbage object, with no references to it anywhere in the system. It also has an object for which neither \( p_1 \) nor \( p_2 \) has a reference, but there is a reference to it in a message that is in transit between the processes. This shows that when we consider properties of a system, we must include the state of communication channels as well as the state of the processes.

**Distributed deadlock detection:** A distributed deadlock occurs when each of a collection of processes waits for another process to send it a message, and where there is a cycle in
the graph of this ‘waits-for’ relationship. Figure 4.4.4(b) shows that processes $p_1$ and $p_2$ are each waiting for a message from the other, so this system will never make progress.

**Distributed termination detection:** The problem here is how to detect that a distributed algorithm has terminated. Detecting termination is a problem that sounds deceptively easy to solve: it seems at first only necessary to test whether each process has halted. To see that this is not so, consider a distributed algorithm executed by two processes $p_1$ and $p_2$, each of which may request values from the other. Instantaneously, we may find that a process is either active or passive – a passive process is not engaged in any activity of its own but is prepared to respond with a value requested by the other. Suppose we discover that $p_1$ is passive and that $p_2$ is passive (Figure 4.4.4c). To see that we may not conclude that the algorithm has terminated, consider the following scenario: when we tested $p_1$ for passivity, a message was on its way from $p_2$, which became passive immediately after sending it. On receipt of the message, $p_1$ became active again – after we had found it to be passive. The algorithm had not terminated.

The phenomena of termination and deadlock are similar in some ways, but they are different problems. First, a deadlock may affect only a subset of the processes in a system, whereas all processes must have terminated. Second, process passivity is not the same as waiting in a deadlock cycle: a deadlocked process is attempting to perform a further action, for which another process waits; a passive process is not engaged in any activity.

**Distributed debugging:** Distributed systems are complex to debug, and care needs to be taken in establishing what occurred during the execution. For example, suppose Smith has written an application in which each process $p_i$ contains a variable $x_i$ ($i = 1, 2 \ldots, N$). The variables change as the program executes, but they are required always to be within a value of one another.

Unfortunately, there is a bug in the program, and Smith suspects that under certain circumstances $|x_i - x_j|$ for some $i$ and $j$, breaking her consistency constraints. Her
problem is that this relationship must be evaluated for values of the variables that occur at the same time.

Each of the problems above has specific solutions tailored to it; but they all illustrate the need to observe a global state, and so motivate a general approach.

**Global states**

It is possible in principle to observe the succession of states of an individual process, but the question of how to ascertain a global state of the system – the state of the collection of processes – is much harder to address.

The essential problem is the absence of global time. If all processes had perfectly synchronized clocks, then we could agree on a time at which each process would record its state – the result would be an actual global state of the system. From the collection of process states we could tell, for example, whether the processes were deadlocked. But we cannot achieve perfect clock synchronization, so this method is not available to us.

So we might ask whether we can assemble a meaningful global state from local states recorded at different real times. The answer is a qualified ‘yes’.

---

**SUMMARY**

In this unit we introduced Clocks, events, and process states, Synchronizing physical clocks and Global states.

**KEYWORDS**

UTC: Coordinated Universal Time

Monotonicity: is the condition that a clock $C$ only ever advances.

NTP: The Network Time Protocol
16.7. UNIT-END EXERCISES AND ANSWERS

1. Define the *history* of a process. 4.1
2. Explain clocks and events. 4.2
3. Explain briefly synchronizing physical clocks. 4.3
4. Differentiate between external *synchronization* and *internal synchronization*. 4.3
5. Explain Cristian’s method for synchronizing clocks 4.3
6. Describe Berkeley algorithm for internal synchronization. 4.3
7. List and explain NTP’s chief design aims and features. 4.3

Answers: SEE

1. 4.1
2. 4.2
3. 4.3
4. 4.3
5. 4.3
6. 4.3
7. 4.3

16.8 SUGGESTED READINGS


- DISTRIBUTED SYSTEMS Principles and Paradigm By: Andrew S. Tanenbaum, Maarten Van Steen.