DISTRIBUTED SYATEM

Er. Dilip Kumar Shrestha

GANDAKI COLLEGE OF ENGINEERING AND SCIENCE

Lamachaur, Pokhara

Table of Contents

Introduction to Distributed System

1. Definitions

A computing system composed of large numbers of CPUs connected by a network is known as the distributed system. Or, a distributed system can be defined as a collection of autonomous computers linked by a network with software designed to produce an integrated computing facility. In other words, a distributed system consists of a collection of autonomous computers linked by a computer network and equipped with distributed system software. Distributed system enables computers to coordinate their activities and share the resources of the system-hardware, software and data. The structure of the distributed system varies in a size from few workstations interconnected by a LAN to the Internet (WAN) and their applications range from the provision of general-purpose computing facilities for a group of users to automated banking, online reservations, communication systems, etc.

> *A distributed system is a collection of independent computers that appears to its users as a single coherent system.*

This definition has two aspects.

- Hardware: the machines are autonomous
- Software: the user think they are dealing with a single system

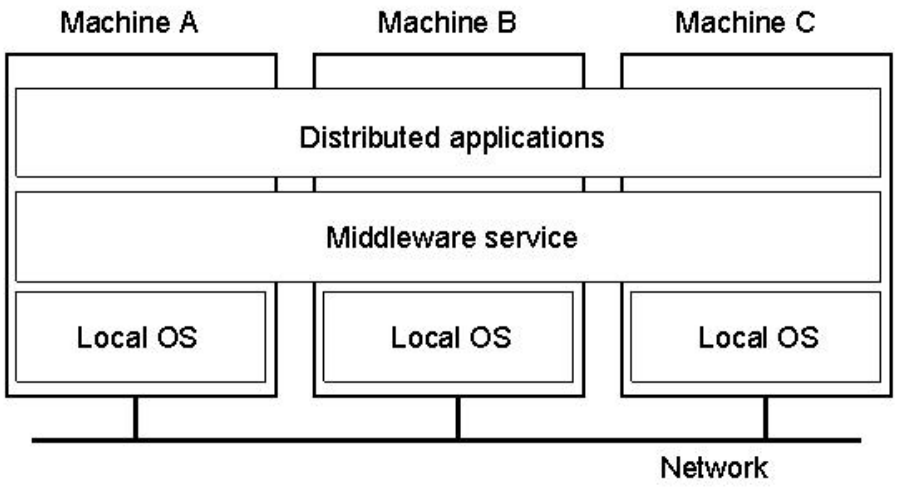


Figure 1: A distributed system organized as middleware. The middleware layer extends over multiple machines, and offers each application the same interface.

> *Distributed system is a system in which hardware or software components located at networked computers communicate and coordinate their actions only by passing messages.*

Computers that are connected by a network may be spatially separated by any distance. They may be on separate continents, in the same building or in the same room. This definition of distributed systems has the following significant characteristics:

- Concurrency: Several clients will attempt to access shared resources at the same time.
- No global clock: When programs need to cooperate, they coordinate their actions by exchanging messages.
- Independent failures: Each component of the system can fail independently, leaving the others still running.

Distributed systems are groups of networked computers, which have the same goal for their work. The terms "concurrent computing", "parallel computing", and "distributed computing" have a lot of overlap, and no clear distinction exists between them. The same system may be characterized both as "parallel" and "distributed"; the processors in a typical distributed system run concurrently in parallel. Parallel computing may be seen as a particular tightly coupled form of distributed computing, and distributed computing may be seen as a loosely coupled form of parallel computing. Nevertheless, it is possible to roughly classify concurrent systems as "parallel" or "distributed" using the following criteria:

- In parallel computing, all processors may have access to a shared memory to exchange information between processors.
- In distributed computing, each processor has its own private memory (distributed memory). Information is exchanged by passing messages between the processors.

Few terminologies

A computer program that runs in a distributed system is called a **distributed program**, and **distributed programming** is the process of writing such programs. There are many alternatives for the message passing mechanism, including RPC-like connectors and message queues. An important goal and challenge of distributed systems is location transparency.

Distributed computing also refers to the use of distributed systems to solve computational problems. In distributed computing, a problem is divided into many tasks, each of which is solved by one or more computers, which communicate with each other by message passing.

2. Key characteristics of Distributed System

1. **Resource sharing**: It is the fundamental characteristic of distributed systems. Resources may be data, software and hardware components.
    a. Data sharing: allows many users to access to a common file and database.
    b. Device sharing: allows users to share the hardware peripherals like storage devices, printers, etc.

   c. Software sharing: using the network based application software like online banking, airline reservation, etc.
2. **Flexibility**: The distributed system can be extended or contracted in various ways. Such as the addition or the removal of various hardware peripherals is possible as per the requirement or the operation of the organization. Similarly, the operating system features, communication protocols and resource-sharing services can also be customized. These all are possible without disruption of or duplication of an existing system.
3. **Concurrency**: Many server processes run concurrently, each responding to different requests from client processes or many users simultaneously invoke commands or interact with application programs.
4. **Reliability**: If one computer of a distributed system crashes, the system can still operate smoothly without interruption.
   Matrix of reliability:
   a. POCOF: Probability of Failure on Demand
   b. ROCOF: Rate of Occurrence of Failure
   c. MTTF: Mean Time to Failure
   d. Avail: Availability
5. **Fault Tolerance**: Computer systems sometimes fail. When fault occurs in a system, programs may produce incorrect results or the operation may get interrupted. The fault tolerant computer system is designed on two basic approaches:
   a. Hardware redundancy: if any hardware peripheral fails, then a spare device replaces the faulty one. For instance, in RAID-5, if one of the hard disk drive fails, then a spare disk is activated and data is automatically transferred into it.
   b. Software recovery: Applications which recover back the data, files or folders if the fault persists in the application program or the file system.
6. **Speed**: A distributed system does have more total computer power than a mainframe due to workload sharing among the computers.

3. Goals

1. **Resource sharing**: The main goal of distributed system is to make it easy for users to access remote resource and to share them with other users in a controlled way. Resources may be data, software and hardware components.
   a. Data sharing: allows many users to access to a common file and database.
   b. Device sharing: allows users to share the hardware peripherals like storage devices, printers, etc.
   c. Software sharing: using the network based application software like online banking, airline reservation, etc.
2. **Transparency**: An important goal of distributed system is to present itself as a coherent system i.e. to hide the fact that its processes and resources are physically distributed across multiple computers. A distributed system that is able to present itself to users and applications as if it were only a single computer system is said to be transparent.
   a. Access: Hide differences in data representation and how a resource is accessed
   b. Location: Hide where a resource is located
   c. Migration: Hide that a resource may move to another location
   d. Relocation: Hide that a resource may be moved to another location while in use
   e. Replication: Hide that a resource is replicated

f. Concurrency: Hide that a resource may be shared by several competitive users
g. Failure: Hide the failure and recovery of resource

3. **Openness**: An open distributed system is a system that offers services according to standard rules that describe the syntax and semantics of those services. Services are generally specified through interfaces which are often described in an Interface Definition Language (IDL). Interface definitions written in an IDL nearly always capture only the syntax of services. If properly specified, an interface definition allows an arbitrary process that needs a certain interface to talk to another process that provides that interface. It also allows two independent parties to build completely different implementations of those interfaces, leading to two separate distributed systems that operate in exactly the same way. Proper specifications are complete and neutral. Complete means that everything that is necessary to make an implementation has indeed been specified. However, many interface definitions are not at all complete. So, that it is necessary for a developer to add implementation-specific details. Just as important is the fact that specifications do not prescribe what an implementation should look like: they should be neutral. Completeness and neutrality are important for **interoperability** and **portability**. Interoperability characterizes the extent by which two implementations of systems or components from different manufacturers can co-exist and work together by merely relying on each other's services as specified by a common standard. Portability characterizes to what extent an application developed for a distributed system A can be executed without modification, on a different distributed system B that implements the same interfaces as A.
4. **Scalability**: Scalability of a system can be measured along at least three different dimensions. First, a system can be scalable with respect to its size, meaning that we can easily add more users and resources to the system. Second, a geographically scalable system is one in which the users and resources may lie far apart. Third, a system can be administratively scalable, meaning that it can still be easy to manage even if it spans many independent administrative organizations. Unfortunately, a system that is scalable in one or more of these dimensions often exhibits some loss of performance as the system scales up.
    a. Scalability Problems

Concept	Example
Centralized services	A single server for all users
Centralized data	A single on-line telephone book
Centralized algorithms	Doing routing based on complete information

Table 1: Examples of scalability limitations

    b. Scaling Techniques
        i. Hiding communication latencies
        ii. Distribution
        iii. Replication

4. Design Issues

1. **Naming**: Distributed systems are based on the sharing of resources. A process that requires access to a resource must possess a name or an identifier. The names assigned to resources or objects must have global meanings that are independent of the locations of the object, and must be supported by a name interpretation system that can translate names in order to enable programs to access named resources (name needs to be resolved into a form understandable to communication network). A name is resolved when it's translated into a form in which it can be used to invoke an action on the resource or object to which it refers. In distributed systems, a resolved name is termed as a communication identifier along with other attributes that maybe useful for communication.

   To locate a host in internet 32-bits IP-address (IPv4) is used e.g. 175.196.2.25

   To identify specific service/process on the host an integer value (0-65535) called as port number is attached with IP-address i.e. IPaddress:port

   e.g. a web server running in the host with IPaddress 175.196.2.25 can be accessed by following identifier 175.196.2.25:80
   Port 0~1024 are reserved for specific application
   80 – http
   25 – smtp
   20 – telnet
   23 – DNS

   DNS (Domain Name Server) is responsible to convert name into IP address and vice-versa.

2. **Communication:** The components of a distributed system are both logical and physically separated; they must communicate in order to interact. Distributed systems and applications are composed of separate software components that interact in order to perform tasks. Communication between a pair of processes involves operations of sending and receiving that together result in:
   a. The transfer of data from the sender to the receiver
   b. The synchronization of the receiving activity with the sending activity

3. **Software Structure:** In centralized computer systems, the operating system is the main system software layer. It manages all of the basic resources in the systems and provides essential services to application programs and users. These include:
   a. Basic resource management:
      i. Memory allocation and protection
      ii. Process creation and processor scheduling
      iii. Peripheral device handling
   b. User and application services:
      i. User authentication and access control
      ii. file management

In distributed systems, apart from those above ones, other services are to be provided to application programs in a manner that enables new services to be conveniently added. Hence the kernel's function is restricted to basic resource management:

- Memory allocation and protection
- Process creation and processor scheduling
- Peripheral device handling
- Inter-process communication

And a new class of software components is introduced called open services to provide all other shared resources and services.

4. **Openness:** Openness refers to the fact that distributed systems can be configured as per the requirement of a group of users or set of applications. Principally, by the introduction of open system, computers with different architectures and operating systems can be integrated and new applications can run in cooperation with one another. Moreover, systems are also called open when hardware and software components supplied by different companies can work together in the system. Openness is achieved the design and construction of software components with well-defined interfaces

5. **Workload allocation:** Good performance is a requirement of almost all engineered products and is a major concern for programmers and system designers. Hence one of the prime design issues for distributed systems is how to deploy the resources in a network to optimum effect in the processing of changing workload.

   In **workstation-server model**, the processor performance and memory capacity of a workstation determines the size of the largest task that can be performed on behalf of its user.

   In **processor pool model**, the workstation-server architecture is modified to include processors that can be allocated to user's task dynamically. A user who performs a task involving several processes is able to access more computing power than a single workstation can offer. For instance, in case of compiling a multi -segment program in a language such as C, each of the segments can be compiled independently to produce separate relocatable object code files. In a processor pool system, a program with n segments can be compiled using n processors producing a potential n-fold speed-up of that phase of the task. A distributed system that supports the processor pool model consists of a workstation-server system in which one or more processor pools have been integrated. A processor pool consists of a collection of low-cost computers, each consisting of little more than a processor, memory and network interface card.

6. **Consistency maintenance:**
   - *Update Consistency*: consistency issues arise when several processes access and update the data concurrently. The modification of related set of data values is not possible instantaneously, but the changes by a given process should appear to all other process as though it was instantaneous.
   - *Replication Consistency*: If data which is derived from a single source have been copied to several computers and subsequently modified at one or more of them, then there is a possibility of inconsistencies between the values of data items at different machines.

- *Cache Consistency*: It refers to the problem that arises when data values that have been cached by one client are updated by another.
- *Failure Consistency*: When a centralize computer system fails, all of the applications which are running on it fail simultaneously. But in case of distributed system, when one component fails, the others should continue to operate normally.
- *User interface consistency*: Whenever a user performs an input action such as key press or mouse click, in an interactive program, the screen becomes temporarily inconsistent while the processing of input goes on. This time delay should be acceptable.
- *Clock consistency*: Many of the algorithms used in applications and system software depend on the time stamps. In distributed system, clocks of the system can be synchronized by network communication.

5. Design Requirements

1. **Performance Issues**:
2. **Quality of Service**:
3. **Use of Caching and Replication**:
4. **Dependability Issues**:

6. Challenges

1. **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 – different networks like mobile, LAN, Intranet
   - computer hardware – different hardware represents data types differently
   - operating systems – different OS provide different programming interface
   - programming languages – different representation for characters and data structure
   - implementations by different developers - Programs written by different developers cannot communicate with one another unless they use common standards

Approaches helping to adapt heterogeneity

   - Middleware: The term middleware applies to a software layer that provides a programming abstraction as well as masking the heterogeneity of the underlying networks, hardware, operating systems and programming languages
   - Mobile code: The term mobile code is used to refer to program code that can be transferred from one computer to another and run at the destination – Java applets are an example
2. **Openness:** The openness of a computer system is the characteristic that determines whether the system can be extended and 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.
   - 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. But the conformance of each component to the published standard must be carefully tested and verified if the system is to work correctly.

3. **Security:** Many of the information resources that are made available and maintained in distributed systems have a high intrinsic value to their users and distributed system has inherited security issues. 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
   - Availability – protection against interference with the means to access the resources

   Different approaches to maintain security

   - Firewall
   - Checksum
   - Encryption techniques

   Two security challenges not yet fully met

   - Denial of service attacks
   - Security of mobile code
4. **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.
   Challenges while designing scalable distributed systems:
   - Controlling the cost of physical resources
   - Controlling the performance loss
   - Preventing software resources running out
   - Avoiding performance bottlenecks
5. **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 techniques to deal with failures
   - Detecting failures
   - Masking failures
   - Tolerating failures
   - Recovery from failures
   - Redundancy

Few false assumptions that everyone makes when developing a distributed application for the first time:

1. The network is reliable.
2. The network is secure.
3. The network is homogeneous.
4. The topology does not change.
5. Latency is zero.
6. Bandwidth is infinite.
7. Transport cost is zero.
8. There is one administrator.

7. Advantages and Disadvantages

7.1. Advantages

a. **Reflects organizational structure:** Many organizations are naturally distributed over several locations. For example, a bank has many offices in different cities. It is natural for databases used in such an application to be distributed over these locations. A bank may keep a database at each branch office containing details such things as the staff that work at that location, the account information of customers etc.
The staff at a branch office will make local inquiries of the database. The company headquarters may wish to make global inquiries involving the access of data at all or a number of branches.

b. **Improved share ability and local autonomy:** The geographical distribution of an organization can be reflected in the distribution of the data; users at one site can access data stored at other sites. Data can be placed at the site close to the users who normally use that data. In this way, users have local control of the data, and they can consequently establish and enforce local policies regarding the use of this data. A global database administrator (DBA) is responsible for the entire system. Generally, part of this responsibility is assigned the local level, so that the local DBA can manage the local DBMS.

c. **Improved availability:** In a centralized DBMS, a computer failure terminates the applications of the DBMS. However, a failure at one site of a DDBMS, or a failure of a communication link making\ some sites inaccessible, does not make the entire system in opera bite. Distributed DBMSs are designed to continue to function despite such failures. If a single node fails, the system may be able to reroute the failed node's requests to another site.

d. **Improved reliability:** As data may be replicated so that it exists at more than one site, the failure of a node or a communication link does not necessarily make the data inaccessible.

e. **Improved Performance:** As the data is located near the site of 'greatest demand', and given the inherent parallelism of distributed DBMSs, speed of database access may be better than that achievable from a remote centralized database. Furthermore, since each site handles only a part of the entire database, there may not be the same contention for CPU and I/O services as characterized by a centralized DBMS.

f. **Economics:** It is now generally accepted that it costs much less to create a system of smaller computers with the equivalent power of a single large computer. This makes it more cost effective for corporate divisions and departments to obtain separate computers. It is also much more cost-effective to add workstations· to a network than to update a mainframe system.

The second potential cost saving occurs where database are geographically remote and the applications require access to distributed data. In such cases, owing to the relative expense of data being transmitted across the network as opposed to the cost of local access, it may be much more economical to partition the application and perform the processing locally at each site.

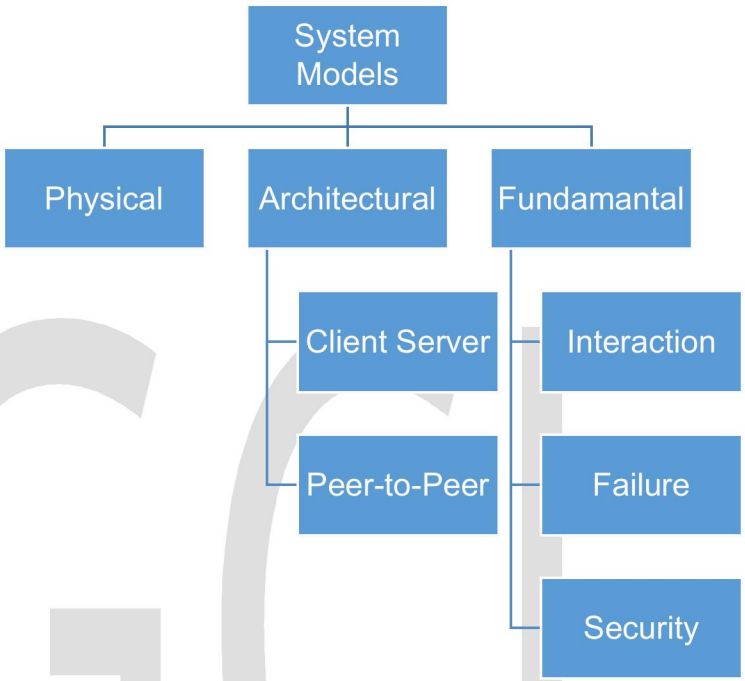
g. **Modular growth:** In a distributed environment, it is much easier to handle expansion. New sites can be added to the network without affecting the operations of other sites. This flexibility allows an organization to expand relatively easily. Adding processing and storage power to the network can usually handle the increase in database size. In a centralized DBMS, growth may entail changes to both hardware (the procurement of a more powerful system) and software (the procurement of a more powerful or more configurable DBMS).

7.2. Disadvantages

a. **Complexity:** A distributed DBMS that hides the distributed nature from the user and provides an acceptable level of performance, reliability, availability is inherently more complex than a centralized DBMS. The fact that data can be replicated also adds an extra level of complexity to the distributed DBMS. If the software does not handle data replication adequately, there wi1l be degradation in availability, reliability and performance compared with the centralized system, and the advantages we cite above will become disadvantages.
b. **Cost:** Increased complexity means that we can expect the procurement and maintenance costs for a DDBMS to be higher than those for a centralized DBMS. Furthermore, a distributed DBMS requires additional hardware to establish a network between sites. There are ongoing communication costs incurred with the use of this network. There are also additional labor costs to manage and maintain the local DBMSs and the underlying network.
c. **Security:** In a centralized system, access to the data can be easily controlled. However, in a distributed DBMS not only does access to replicated data have to be controlled in multiple locations but also the network itself has to be made secure. In the past, networks were regarded as an insecure communication medium. Although this is still partially true, significant developments have been made to make networks more secure.
d. **Integrity control more difficult:** Database integrity refers to the validity and consistency of stored data. Integrity is usually expressed in terms of constraints, which are consistency rules that the database is not permitted to violate. Enforcing integrity constraints generally requires access to a large amount of data that defines the constraints. In a distributed DBMS, the communication and processing costs that are required to enforce integrity constraints are high as compared to centralized system.
e. **Lack of Standards:** Although distributed DBMSs depend on effective communication, we are only now starting to see the appearance of standard communication and data access protocols. This lack of standards has significantly limited the potential of distributed DBMSs. There are also no tools or methodologies to help users convert a centralized DBMS into a distributed DBMS
f. **Lack of experience:** General-purpose distributed DBMSs have not been widely accepted, although many of the protocols and problems are well understood. Consequently, we do not yet have the same level of experience in industry as we have with centralized DBMSs. For a prospective adopter of this technology, this may be a significant deterrent.
g. **Networking:** If the network gets saturated then problems with transmission will surface.
h. **Software:** There is currently very little less software support for Distributed system.

i. **Troubleshooting:** Troubleshooting and diagnosing problems in a distributed system can also become more difficult, because the analysis may require connecting to remote nodes or inspecting communication between nodes.

8. Distributed System models



8.1. Physical models

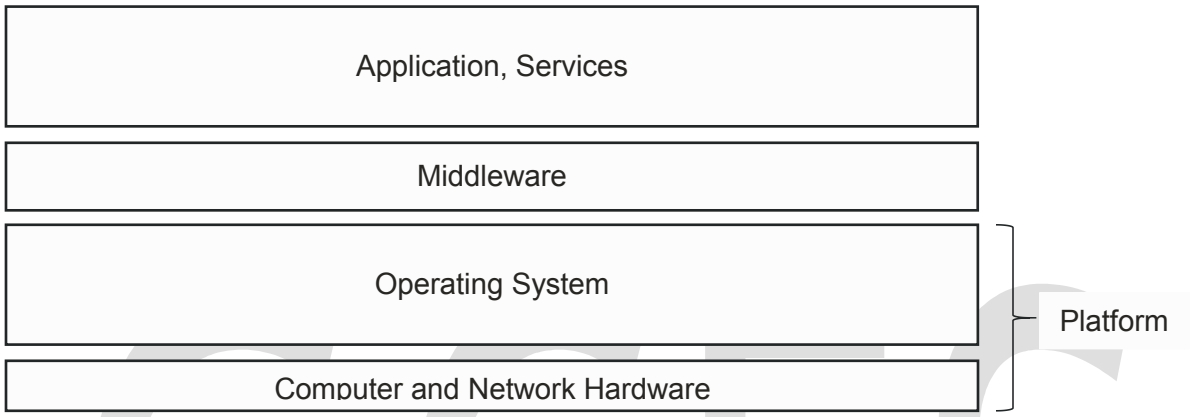
Physical models consider the types of computers and devices that constitute a system and their interconnectivity, without details of specific technologies. These models are the most explicit way in which to describe a system; they capture the hardware composition of a system in terms of the computers (and other devices, such as mobile phones) and their interconnecting networks. So 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.

8.2. Architectural models

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.

8.2.1. Software Layers

Architecture originally refers to structuring of software as layers or modules in a single computer and more recently in terms of services offered and requested between processes located in the same of different computers



Platform: It is the lowest level hardware and software layers.Examples: Intel x86/Windows, Intel x86/Solaris, PowerPC/MAC OS, Intel x86/Linux

Middleware: It is a layer of software which masks heterogeneity and provides a convenient programming model to application services. Object-oriented middleware standards and products include COBRA, Java RMI, web services, etc.

8.2.2. 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 paradigm is used?
- What (potentially changing) roles and responsibilities do they have in the overall architecture?
- How are they mapped on to the physical distributed infrastructure (what is their placement)?

8.2.3. Communicating entities

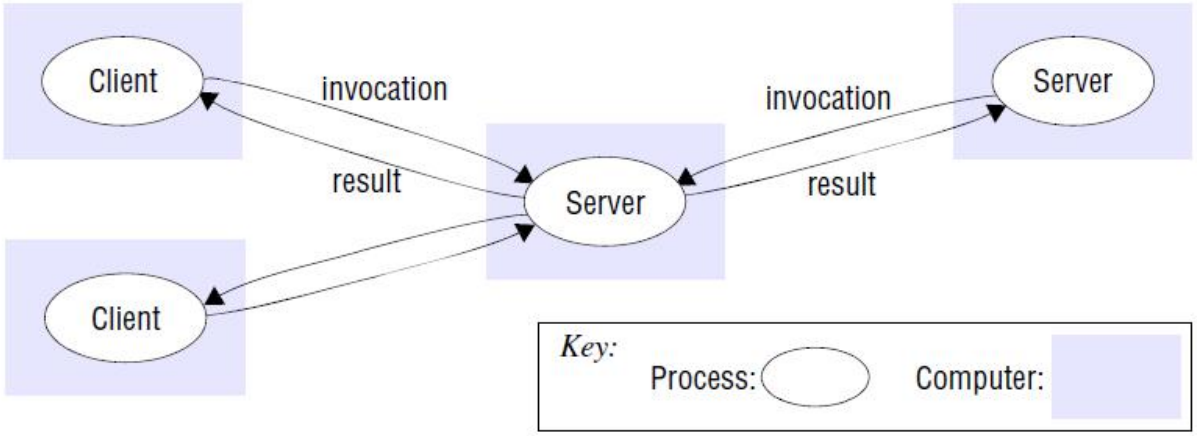
- objects
- Components
- Web Services

8.2.4. Communication paradigms

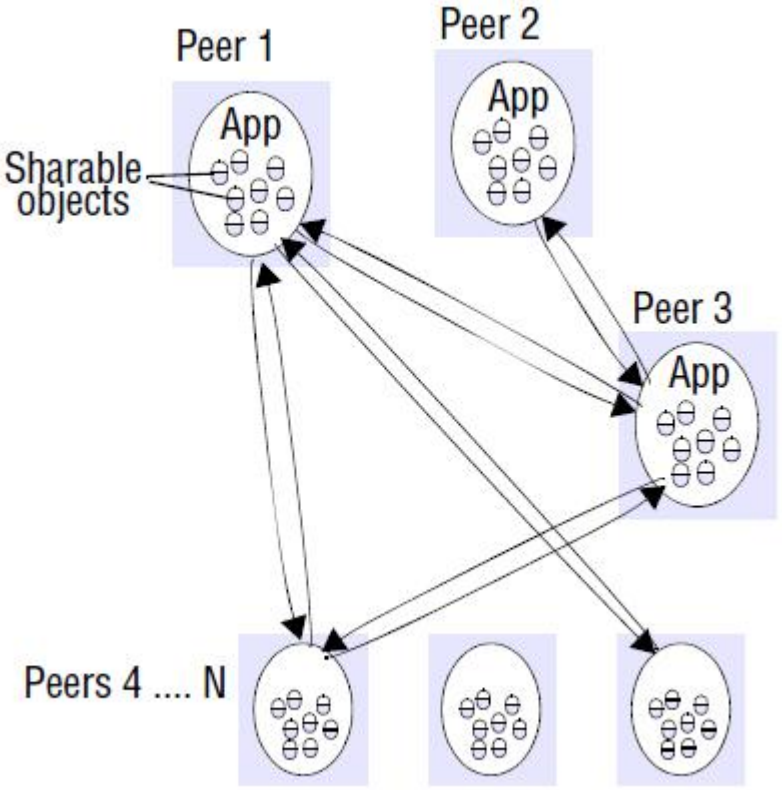
- Inter-process communication refers to the relatively low-level support for communication between processes in distributed systems, including message-passing primitives, direct access to the API offered by Internet protocols (socket programming) and support for multicast communication.
- Remote invocation represents the most common communication paradigm in distributed systems, covering a range of techniques based on a two-way exchange between communicating entities in a distributed system and resulting in the calling of a remote operation, procedure or method. This includes Request-reply protocols, Remote procedure calls and Remote method invocation
- Indirect communication
    - Group communication: Group communication is concerned with the delivery of messages to a set of recipients and hence is a multiparty communication paradigm supporting one-to-many communication.
    - Publish-subscribe systems: A communication system used where large number of producers (or publishers) distribute information items of interest (events) to a similarly large number of consumers (or subscribers).
    - Message queues: Whereas publish-subscribe systems offer a one-to-many style of communication, message queues offer a point-to-point service whereby producer processes can send messages to a specified queue and consumer processes can receive messages from the queue or be notified of the arrival of new messages in the queue.
    - Tuple spaces: Tuple spaces offer a further indirect communication service by supporting a model whereby processes can place arbitrary items of structured data, called tuples, in a persistent tuple space and other processes can either read or remove such tuples from the tuple space by specifying patterns of interest. Since the tuple space is persistent, readers and writers do not need to exist at the same time.
    - Distributed shared memory: Distributed shared memory (DSM) systems provide an abstraction for sharing data between processes that do not share physical memory.

8.2.5. Two distinct architectural models

1. **Client Server**



2. **Peer-to-Peer**



8.2.6. Variations

1. Services provided by multiple servers
2. Proxy servers and caches
3. Mobile code
4. Mobile agent

5. Network computer
6. Thin clients
7. Mobile devices and spontaneous interoperation

8.3. Fundamental models

All the models of a systems share some fundamental properties. In particular, all of them are composed of processes that communicate with one another by sending messages over a computer network. All of the models share the design requirements of achieving the performance and reliability characteristics of processes and networks and ensuring the security of the resources in the system. Fundamental models represent models based on the fundamental properties that allow us to be more specific about their characteristics and the failures and security risks they might exhibit.

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:

- The interaction model deals with performance and with the difficulty of setting time limits in a distributed system, for example for message delivery.
- The failure model attempts to give a precise specification of the faults that can be exhibited by processes and communication channels. It defines reliable communication and correct processes.
- 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.

8.4. Interaction Model

The interaction model must reflect the facts that communication takes place with delays that are often of considerable duration, and that the accuracy with which independent processes can be coordinated is limited by these delays and by the difficulty of maintaining the same notion of time across all the computers in a distributed system.

Two variants of the Interaction Model

- Synchronous distributed system
    - time to execute each step of a computation within a process has known lower and upper bounds
    - message delivery times are bound to a known value
    - each process has a clock whose drift rate from real time is bounded by a known value
- Asynchronous distributed system

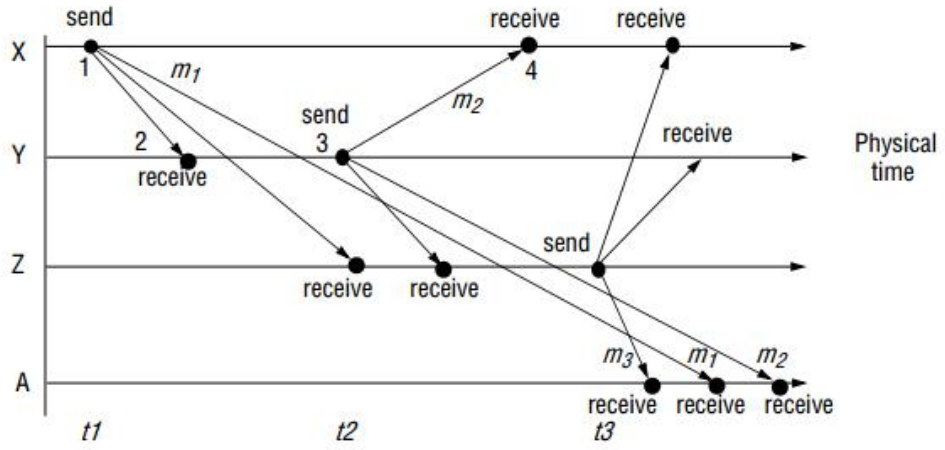
- no bound-on process execution times
- no bound-on message delivery times
- no bound-on clock drift rate

Note:

- synchronous distributed systems are easier to handle, but determining realistic bounds can be hard or impossible
- asynchronous distributed systems are more abstract and general: a distributed algorithm executing on one system is likely to also work on another one

Event Ordering

- In a distributed system, it is impossible for any process to have a view on the current global state of the system
- Possible to record timing information locally, and abstract from real time (logical clocks)
- event ordering rules:
  - if e1 and e2 happen in the same process and e1 happens before e2 then e1 → e2
  - if e1 is the sending of a message m and e2 is the receiving of the same message m then e1 → e2
  - hence, → describes a partial ordering relation on the set of events in the distributed system



Performance Characteristics of Communication Channels

- **Latency**: delay between sending and receipt of message
  - network access time (e.g. Ethernet transmission delay)

    - Time for first bit to travel from sender's network interface to receiver's network interface.
- **Throughput**: number of units (e.g. packets) delivered per unit of time
- **Bandwidth:** amount of information transmitted per time unit
- **Delay jitter**: variation in delay between different messages of the same type, (e.g., video frames)

8.5. Failure Model

The failure model defines the way in which failure may occur in order to provide an understanding of the effects of failure.

8.5.1. Types of Failure

1. Omission Failures
   The omission failures refer to cases when a process or communication channel fails to perform actions that it is supposed to do.
   - process omission failures
     - detection with timeouts
     - crash is fail-stop if other processes can detect with certainty that process has crashed
   - communication omission failures: message is not being delivered — dropping of messages
   - possible causes:
     - network transmission error
     - receiver incoming message buffer overflow
2. Arbitrary Failures
   The worst possible failure semantics in which any type of error may occur.
   - process: omit intended processing steps or carry out unintended ones
   - Communication channel: corruption or duplication etc.
3. Timing Failures
   They are applicable in synchronous distributed systems where time limits are set on process execution time, message delivery time and clock drift rate.

8.5.2. Masking/Hiding Failures

- A service may mask an error by hiding it entirely or, converting it into a more acceptable type of error
- A reliable protocol can be built upon an unreliable protocol by requesting retransmission of dropped messages
- Message sequence numbers can be used to ensure no message is delivered twice, particularly when used with a guaranteed delivery protocol.
- Parity bits or checksums can be used to detect an error and thereby turn an arbitrary failure into an omission failure.

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

Reliability of one-to-one communication

The term reliable communication is defined in terms of validity and integrity as follows:

Validity: Any message in the outgoing message buffer is eventually delivered to the incoming message buffer.

Integrity: The message received is identical to one sent, and no messages are delivered twice

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

8.6. Security

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

- Two related problems:
  - We wish to make sure only the intended recipient(s) can receive a message
  - Additionally, messages (for example invocation requests) should be authenticated so that we know from whom they originated
- These can be largely mitigated against with the use of modern cryptographic algorithms. However, their use incurs some cost which we may hope to minimize

Security means protection and protection is defined in terms of objects, although the concept applies equally well to resources of all types

8.6.1. Protecting objects

The server is responsible for verifying the identity of the principal behind each invocation and checking that they have sufficient access rights to perform the requested operation on the particular object invoked, rejecting those that do not. The client may check the identity of the principal behind the server to ensure that the result comes from the required server.

8.6.2. Securing processes and their interactions

Processes interact by sending messages. The messages are exposed to attack because the network and the communication service that they use are open, to enable any pair of processes to interact. Servers and peer processes expose their interfaces, enabling invocations to be sent to them by any other process.

8.6.3. The enemy

To model security threats, we postulate an enemy (sometimes also known as the adversary) that is capable of sending any message to any process and reading or copying any message sent between a pair of processes. The threats from a potential enemy include threats to processes and threats to communication channels.

Threats to processes: A process that is designed to handle incoming requests may receive a message from any other process in the distributed system, and it cannot necessarily determine the identity of the sender.

- *Servers*: Since a server can receive invocations from many different clients, it cannot necessarily determine the identity of the principal behind any particular invocation.

- *Clients:* When a client receives the result of an invocation from a server, it cannot necessarily tell whether the source of the result message is from the intended server or from an enemy, perhaps 'spoofing' the mail server.

Threats to communication channels: An enemy can copy, alter or inject messages as they travel across the network and its intervening gateways. Such attacks present a threat to the privacy and integrity of information as it travels over the network and to the integrity of the system.

8.6.4. Defeating security threats

Cryptography and shared secrets: Cryptography is the science of keeping messages secure, and encryption is the process of scrambling a message in such a way as to hide its contents. Modern cryptography is based on encryption algorithms that use secret keys – large numbers that are difficult to guess – to transform data in a manner that can only be reversed with knowledge of the corresponding decryption key.

Authentication: The use of shared secrets and encryption provides the basis for the authentication of messages – proving the identities supplied by their senders

Secure channels: Encryption and authentication are used to build secure channels as a service layer on top of existing communication services. A secure channel is a communication channel connecting a pair of processes, each of which acts on behalf of a principal.

8.6.5. Other possible threats from an enemy

Denial of service

- generating debilitating network or server load so that services become the equivalent of unavailable

Mobile Code:

- requires excitability privileges on target machine
- code may be malicious

Distributed Objects and Remote Communication

1. NETWORKING AND INTERNETWORKING

1.1. Introduction

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 collection of hardware and software components that provide the communication facilities for a distributed system are known 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.

1.1.1. Networking issues for distributed systems

Performance: The performance parameters affecting the speed with which individual messages can be transferred between two interconnected computers 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

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 time required for a network to transfer a message containing length bits between two computers is: *Message transmission time = latency + length／data transfer rate*

The total system bandwidth of a network is a measure of throughput – the total volume of traffic that can be transferred across the network in a given time.

Scalability: The network technologies on which Internet is based were not designed to cope with even its 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. 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 detection of communication errors and their correction is often best performed by application-level software.

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.

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.

Quality of service: The ability to meet deadlines when transmitting and processing streams of real-time multimedia data. This imposes major new requirements on computer networks.

Multicasting: Most communication in distributed systems is between pairs of processes, but there often is also a need for one-to-many communication. Many network technologies support the simultaneous transmission of messages to several recipients.

1.2. Types of 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.

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.

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.

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 (Wi-Fi), offering bandwidths of 10–100 Mbps over ranges up to 1.5 kilometers.

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 Wi-Fi networks in some applications.

Figure 3.1 Network performance

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

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. 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. 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.

Network errors: The reliability of the underlying data transmission media is very high in all types except wireless networks, where packets are frequently lost due to external interference. But packets may be lost in all types of network due to processing delays and buffer overflow at switches and at the destination node. This is by far the most common cause of packet loss. Packets may also be delivered in an order different from that in which they were transmitted. This arises only in networks where separate packets are individually routed – principally wide area network.

1.3. Network Principles

The basis for all computer networks is the packet-switching technique first developed in the 1960s. This enables data packets addressed to different destinations to share a single communications link, unlike the circuit-switching technology that underlies conventional telephony. Packets are queued in a buffer and transmitted when the link is available. Communication is asynchronous – messages arrive at their destination after a delay that varies depending upon the time that packets take to travel through the network.

1.3.1. Packet transmission

The requirement in a computer network is for the transmission of logical units of information, or messages – sequences of data items of arbitrary length. But before a message is transmitted it is subdivided into packets. The simplest form of packet is a sequence of binary data of restricted length together with addressing information sufficient to identify the source and destination computers.

1.3.2. Data streaming

The transmission and display of audio and video in real time is referred to as streaming. It requires much higher bandwidths than most other forms of communication in distributed systems. The timely delivery of audio and video streams depends upon the availability of connections with adequate quality of service – bandwidth, latency and reliability must all be considered. Ideally, adequate quality of service should be guaranteed. In general the Internet does not offer that capability, and the quality of real-time video streams sometimes reflects that, but in proprietary intranets such as those operated by media companies, guarantees are sometimes achieved.

1.3.3. Switching schemes

To transmit information between two arbitrary nodes, a switching system is required

Broadcast: Broadcasting is a transmission technique that involves no switching. Everything is transmitted to every node, and it is up to potential receivers to notice transmissions addressed to them. Some LAN technologies, including Ethernet, are based on broadcasting.

Circuit switching: This system is sometimes referred to as the plain old telephone system, or POTS. It is a typical circuit-switching network.

Packet switching: Instead of making and breaking connections to build circuits, a store-and-forward network just forwards packets from their source to their destination. There is a computer at each switching node. Each packet arriving at a node is first stored in memory at the node and then processed by a program that transmits it on an outgoing circuit, which transfers the packet to another node that is closer to its ultimate destination.

Frame relay: The frame relay switching method brings some of the advantages of circuit switching to packet-switching networks. They overcome the delay problems by switching small packets (called frames) on the fly. The switching nodes route frames based on the examination of their first few bits; frames as a whole are not stored at nodes but pass through them as short streams of bits. ATM networks are a prime example.

1.3.4. Protocols

The term protocol is used to refer to a well-known set of rules and formats to be used for communication between processes in order to perform a given task. The definition of a protocol has two important parts:

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

Protocol layers: Network software is arranged in a hierarchy of layers. Each layer presents an interface to the layers above it that extends the properties of the underlying communication system. A layer is represented by a module in every computer connected to the network. Each module appears to communicate directly with a module at the same level in another computer in the network, but in reality data is not transmitted directly between the protocol modules at each level. Instead, each layer of network software communicates by local procedure calls with the layers above and below it. On the sending side, each layer (except the topmost, or application layer) accepts items of data in a specified format from the layer above it and applies transformations to encapsulate the data in the format specified for that layer before passing it to the layer below for further processing.

Figure 3.5 OSI protocol summary

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

Protocol suites: A complete set of protocol layers is referred to as a protocol suite or a protocol stack, reflecting the layered structure.

Packet assembly: The task of dividing messages into packets before transmission and reassembling them at the receiving computer is usually performed in the transport layer. The network-layer protocol packets consist of a header and a data field. In most network technologies, the data field is variable in length, with the maximum length called the maximum transfer unit (MTU). If the length of a message exceeds the MTU of the underlying network layer, it must be fragmented into chunks of the appropriate size, with sequence numbers for use on reassembly, and transmitted in multiple packets.

Ports: The transport layer's task is to provide a network-independent message transport service between pairs of network ports. Ports are software-defined destination points at a host computer. They are attached to processes, enabling data transmission to be addressed to a specific process at a destination node.

Addressing: The transport layer is responsible for delivering messages to destinations with transport addresses that are composed of the network address of a host computer and a port number. A network address is a numeric identifier that uniquely identifies a host computer and enables it to be located by nodes that are responsible for routing data to it.

Port numbers below 1023 are defined as well-known ports whose use is restricted to privileged processes in most operating systems. The ports between 1024 and 49151 are registered ports for which (the Internet Assigned Numbers Authority) IANA holds service descriptions, and the remaining ports up to 65535 are available for private purposes. In practice, all of the ports above 1023 can be used for private purposes, but computers using them for private purposes cannot simultaneously access the corresponding registered services.

A fixed port number allocation does not provide an adequate basis for the development of distributed systems which often include a multiplicity of servers including dynamically allocated ones.

Packet delivery: There are two approaches to the delivery of packets by the network layer:

Datagram packet delivery: The essential feature of datagram networks is that the delivery of each packet is a 'one-shot' process; no setup is required, and once the packet is delivered the network retains no information about it. In a datagram network a sequence of packets transmitted by a single host to a single destination may follow different routes and when this occurs they may arrive out of sequence.

Virtual circuit packet delivery: Network-level services implement packet transmission in a manner that is analogous to a telephone network. A virtual circuit must be set up before packets can pass from a source host A to destination host B. The establishment of a virtual circuit involves the identification of a route from the source to the destination, possibly passing through several intermediate nodes. At each node along the route a table entry is made, indicating which link should be used for the next stage of the route.

1.3.5. Routing

Routing is a function that is required in all networks except those LANs, such as Ethernets, that provide direct connections between all pairs of attached hosts. In large networks, adaptive routing is employed: the best route for communication between two points in the network is re-evaluated periodically, taking into account the current traffic in the network and any faults such as broken connections or routers.

The determination of routes for the transmission of packets to their destinations is the responsibility of a routing algorithm implemented by a program in the network. A routing algorithm has two parts:

- It must make decisions that determine the route taken by each packet as it travels through the network. In circuit-switched network layers such as X.25 and frame relay networks such as ATM, the route is determined whenever a virtual circuit or connection is established. In packet-switched network layers such as IP it is determined separately for each packet, and the algorithm must be particularly simple and efficient if it is not to degrade network performance.

- It must dynamically update its knowledge of the network based on traffic monitoring and the detection of configuration changes or failures. This activity is less time-critical; slower and more computation-intensive techniques can be used.

Pseudo-code for RIP routing algorithm

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

1.3.6. Congestion control

The capacity of a network is limited by the performance of its communication links and switching nodes. When the load at any particular link or node approaches its capacity, queues will build up at hosts trying to send packets and at intermediate nodes holding packets whose onward transmission is blocked by other traffic. If the load continues at the same high level, the queues will continue to grow until they reach the limit of available buffer space. Once this state is reached at a node, the node has no option but to drop further incoming packets. Instead of allowing packets to travel through the network until they reach over congested nodes, where they will have to be dropped, it would be better to hold them at earlier nodes until the congestion is reduced. This will result in increased delays for packets but will not significantly degrade the total throughput of the network. Congestion control is the name given to techniques that are designed to achieve this.

In general, congestion control is achieved by informing nodes along a route that congestion has occurred and that their rate of packet transmission should therefore be reduced. Congestion information may be supplied to the sending node by explicit transmission of special messages (called choke packets) requesting a reduction in transmission rate, by the implementation of a specific transmission control protocol or by observing the occurrence of dropped packets.

1.3.7. Internetworking

To build an integrated network (an internetwork) we must integrate many subnets, each of which is based on one of these network technologies. To make this possible, the following are needed:

- a unified internetwork addressing scheme that enables packets to be addressed to any host connected to any subnet
- a protocol defining the format of internetwork packets and giving rules according to which they are handled
- interconnecting components that route packets to their destinations in terms of internetwork addresses, transmitting the packets using subnets with a variety of network technologies

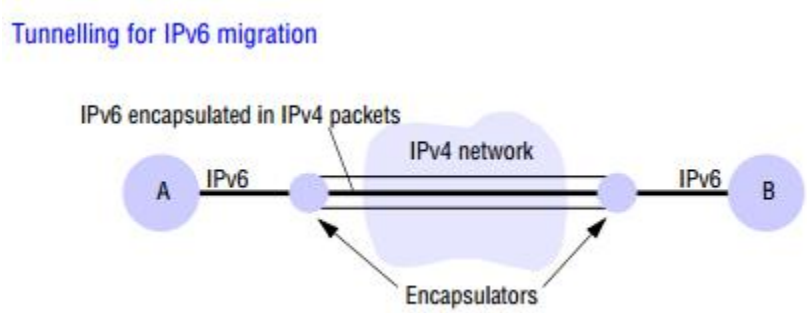
Routers: In an internetwork, the routers may be linked by direct connections, or they may be interconnected through subnets. The routers are responsible for forwarding the internetwork packets that arrive on any connection to the correct outgoing connection. They maintain routing tables for that purpose.

Bridges: Bridges link networks of different types. Some bridges link several networks, and these are referred to as bridge/routers because they also perform routing functions.

Hubs: Hubs are simply a convenient means of connecting hosts and extending segments of Ethernet and other broadcast local network technologies. They have a number of sockets (typically 4–64), to each of which a host computer can be connected.

Switches: Switches perform a similar function to routers, but for local networks (normally Ethernets) only. That is, they interconnect several separate Ethernets, routing the incoming packets to the appropriate outgoing network. The advantage of switches over hubs is that they separate the incoming traffic and transmit it only on the relevant outgoing network, reducing congestion on the other networks to which they are connected.

Tunneling: Bridges and routers transmit internetwork packets over a variety of underlying networks by translating between their network-layer protocols and an internetwork protocol, but there is one situation in which the underlying network protocol can be hidden from the layers above it without the use of an internetwork protocol. A pair of nodes connected to separate networks of the same type can communicate through another type of network by constructing a protocol 'tunnel'. A protocol tunnel is a software layer that transmits packets through an alien network environment.



1.4. Internet Protocols

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 [Leiner et al. 1997]. 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. Many application services and application-level protocols (shown in parentheses in the following list) now exist based on TCP/IP, including the Web (HTTP), email (SMTP, POP), Netnews (NNTP), file transfer (FTP) and Telnet (telnet).

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 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.

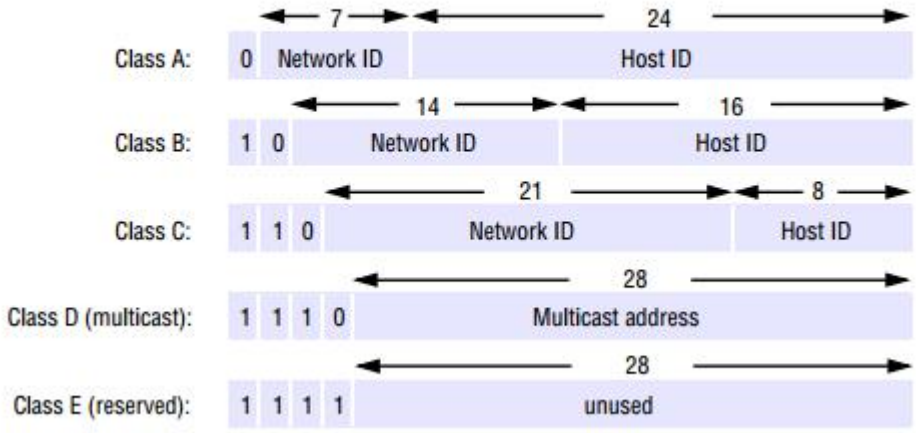
1.4.1. IP addressing

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 232 or approximately 4 billion addressable hosts (about the same as the population of the world at that time) was considered adequate.

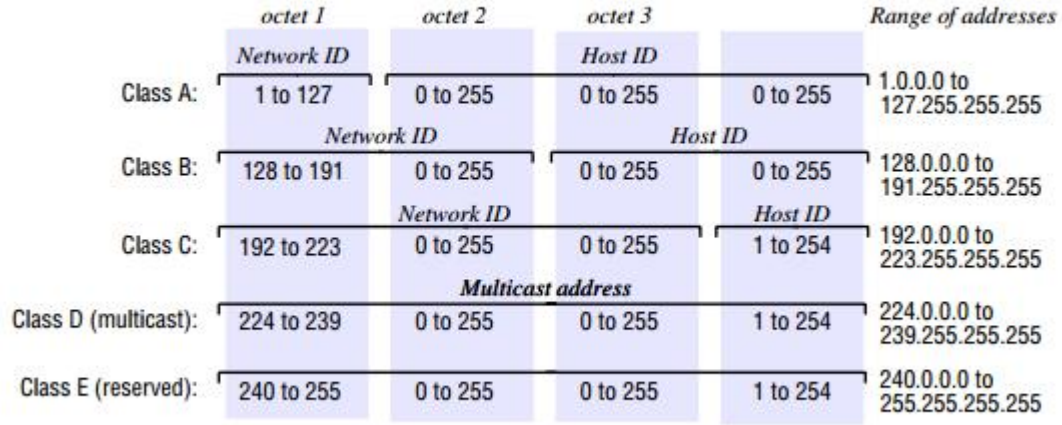
- The addressing scheme must lend itself to the development of a flexible and efficient routing scheme, but the addresses themselves cannot contain very much of the information needed to route a packet to its destination.

Figure 3.15 Internet address structure, showing field sizes in bits



The design adopted for the Internet address space is shown in Figure 3.15. 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.

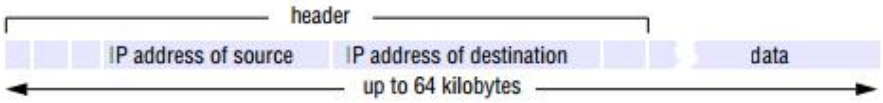
Figure 3.16 Decimal representation of Internet addresses



1.4.2. 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 the figure shows the main components. There are several header fields, not shown in the diagram, that are used by the transmission and routing algorithms.

Figure 3.17 IP packet layout



Address resolution: The address resolution module is responsible for converting Internet addresses to network addresses for a specific underlying network (sometimes called physical addresses).

IP spoofing: 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.

1.4.3. 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.

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 [Malkin 1993]) was developed from it to accommodate several additional requirements, including classless inter-domain 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 [1959] and has been shown to converge more rapidly than the RIP algorithm.

Default routes: The accuracy of routing information can be relaxed for most routers as long as some key routers (those closest to the backbone links) have relatively complete routing tables. The relaxation takes the form of a default destination entry in routing tables. The default entry specifies a route to be used for all IP packets whose destinations are not included in the routing table. 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.

Classless inter-domain routing (CIDR): The shortage of IP addresses 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 subnet.

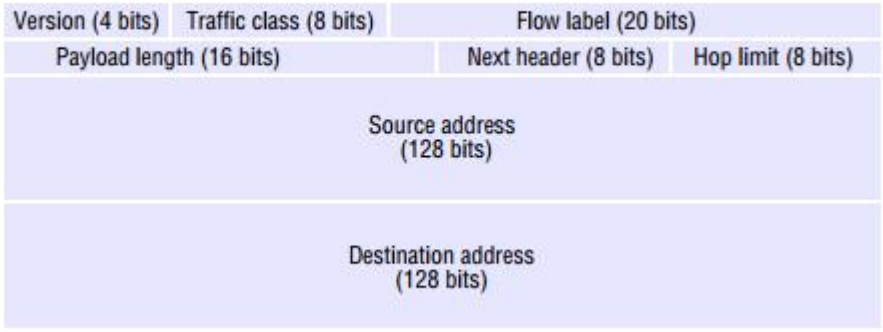
Unregistered addresses and Network Address Translation (NAT): Not all of the computers and devices that access the Internet need to be assigned globally unique IP addresses. Computers that are attached to a local network and access to the Internet through a NAT-enabled router can rely upon the router to redirect incoming UDP and TCP packets for them. NAT is described in RFC 1631 [Egevang and Francis 1994] and extended in RFC 2663 [Srisuresh and Holdrege 1999]. 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.

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. IPv6 must be seen as the next step, enabling full Internet participation for all computers and portable devices.

1.4.4. IP version 6

A more permanent solution to the addressing limitations of IPv4 was also pursued, and this led to the development and adoption of a new version of the IP protocol with substantially larger addresses. The IETF noticed the potential problems arising from the 32-bit addresses of IPv4 as early as 1990 and initiated a project to develop a new version of the IP protocol. IPv6 was adopted by the IETF in 1994 and a strategy for migration to it was recommended.

Figure 3.19 IPv6 header layout



Address space: IPv6 addresses are 128 bits (16 bytes) long. This provides for a truly astronomical number of addressable entities: 2128, or approximately 3 * 10^38.

Routing speed: The complexity of the basic IPv6 header and the processing required at each node are reduced. No checksum is applied to the packet content (payload), and no fragmentation can occur once a packet has begun its journey.

Real-time and other special services: The traffic class and flow label fields are concerned with this. Multimedia streams and other sequences of real-time data elements can be transmitted as part of an identified flow. The first 6 bits of the traffic class field can be used with the flow label or independently to enable specific packets to be handled more rapidly or with higher reliability than others. Traffic class values 0 through 8 are for transmissions that can be slowed without disastrous effects on the application. Other values are reserved for packets whose delivery is time-dependent.

Future evolution: The key to the provision for future evolution is the next header field. If non-zero, it defines the type of an extension header that is included in the packet. There are currently extension header types that provide additional data for special services of the following types: information for routers, route definition, fragment handling, authentication, encryption and destination handling.

Multicast and any-cast: Both IPv4 and IPv6 include support for the transmission of IP packets to multiple hosts using a single address (one that is in the range reserved for the purpose). The IP routers are then responsible for routing the packet to all of the hosts that have subscribed to the group identified by the relevant address. In addition, IPv6 supports a new mode of transmission called any-cast. This service delivers a packet to at least one of the hosts that subscribes to the relevant address.

Security: Security in IPv6 is implemented through the authentication and encrypted security payload extension header types. These implement features equivalent to the secure channel concept. The payload is encrypted and/or digitally signed as required.

Migration from IPv4: The consequences for the existing Internet infrastructure of a change in its basic protocol are profound. IP is processed in the TCP/IP protocol stack at every host and in

the software of every router. IP addresses are handled in many application and utility programs. All of these require upgrading to support the new version of IP, but the change is made inevitable by the forthcoming exhaustion of the address space provided by IPv4. The IETF working group responsible for IPv6 has defined a migration strategy – essentially it involves the implementation of 'islands' of IPv6 routers and hosts communicating with other IPv6 islands via tunnels and gradually merging into larger islands.

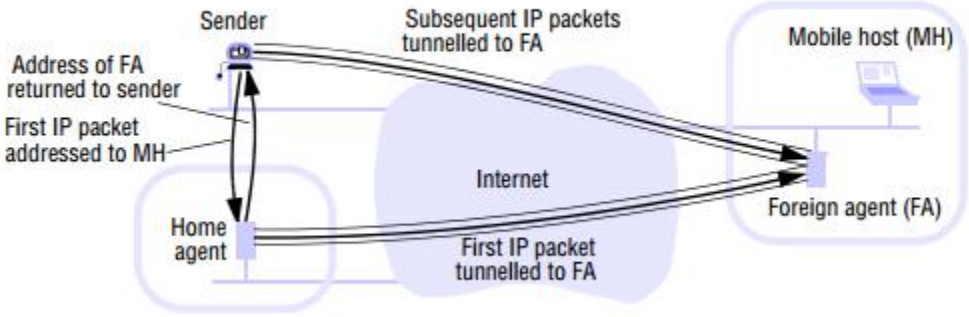
1.4.5. Mobile-IP

Mobile computers such as laptops and tablets are connected to the Internet at different locations as they migrate. In its owner's office a laptop may be connected to a local Ethernet connected to the Internet through a router, it may be connected via a mobile phone while it is in transit by car or train, then it may be attached to an Ethernet at another site. The user will wish to access services such as email and the Web at any of these locations.

If a mobile computer is to remain accessible to clients and resource sharing applications when it moves between local networks and wireless networks, it must retain a single IP number, but IP routing is subnet-based. Subnets are at fixed locations, and the correct routing of packets to them depends upon their position on the network. Mobile-IP is a solution for the latter problem. The solution is implemented transparently, so IP communication continues normally when a mobile host computer moves between subnets at different locations. It is based upon the permanent allocation of a normal IP address to each mobile host on a subnet in its 'home' domain.

When an IP packet addressed to the mobile host's home address is received at the home network, it is routed to the HA (home agent). The HA then encapsulates the IP packet in a MobileIP packet and sends it to the FA (foreign agent). The FA unpacks the original IP packet and delivers it to the mobile host via the local network to which it is currently attached.

Figure 3.20 The MobileIP routing mechanism



1.4.6. TCP and UDP

TCP and UDP provide the communication capabilities of the Internet in a form that is useful for application programs.

Use of ports • the first characteristic to note is that, whereas IP supports communication between pairs of computers (identified by their IP addresses), TCP and UDP, as transport protocols, must provide process-to-process communication. This is accomplished by the use of ports. Port numbers are used for addressing messages to processes within a particular computer and are valid only within that computer. A port number is a 16-bit integer. Once an IP packet has been delivered to the destination host, the TCP- or UDP-layer software dispatches it to a process via a specific port at that host.

UDP features • UDP is almost a transport-level replica of IP. A UDP datagram is encapsulated inside an IP packet. It has a short header that includes the source and destination port numbers (the corresponding host addresses are present in the IP header), a length field and a checksum. UDP offers no guarantee of delivery.

TCP features • TCP provides a much more sophisticated transport service. It provides reliable delivery of arbitrarily long sequences of bytes via stream-based programming abstraction.

Sequencing: A TCP sending process divides the stream into a sequence of data segments and transmits them as IP packets. A sequence number is attached to each TCP segment.

Flow control: The sender takes care not to overwhelm the receiver or the intervening nodes. This is achieved by a system of segment acknowledgements.

Retransmission: The sender records the sequence numbers of the segments that it sends. When it receives an acknowledgement it notes that the segments were successfully received, and it may then delete them from its outgoing buffers. If any segment is not acknowledged within a specified timeout, the sender retransmits it.

Buffering: The incoming buffer at the receiver is used to balance the flow between the sender and the receiver. If the receiving process issues receive operations more slowly than the sender issues send operations, the quantity of data in the buffer will grow.

Checksum: Each segment carries a checksum covering the header and the data in the segment. If a received segment does not match its checksum, the segment is dropped.

1.4.7. Domain names

The Internet supports a scheme for the use of symbolic names for hosts and networks, such as binkley.cs.mcgill.ca or essex.ac.uk. The named entities are organized into a naming hierarchy. The named entities are called domains and the symbolic names are called domain names. Domains are organized in a hierarchy that is intended to reflect their organizational structure. The naming hierarchy is entirely independent of the physical layout of the networks that constitute the Internet. Domain names are convenient for human users, but they must be translated to Internet (IP) addresses before they can be used as communication identifiers. This is the responsibility of a specific service, the DNS. Application programs pass requests to the DNS to convert the domain names that users specify into Internet addresses.

1.4.8. Firewalls

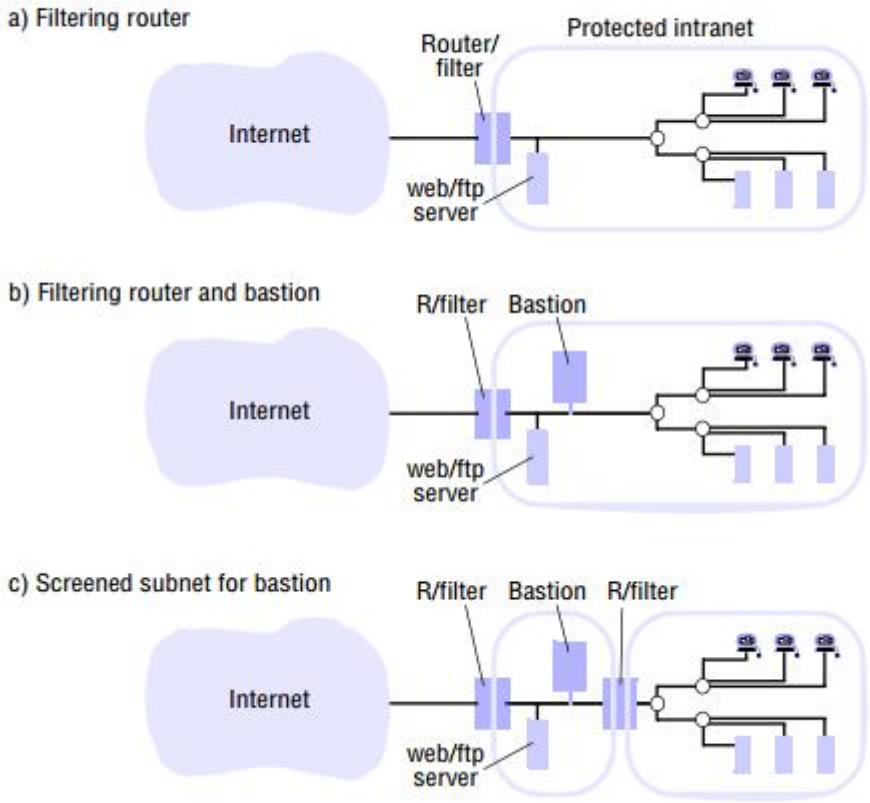
The purpose of a firewall is to monitor and control all communication into and out of an intranet. A firewall is implemented by a set of processes that act as a gateway to an intranet (Figure 3.21a), applying a security policy determined by the organization. The aims of a firewall security policy may include any or all of the following:

- Service control: To determine which services on internal hosts are accessible for external access and to reject all other incoming service requests. Outgoing service requests and the responses to them may also be controlled
- Behavior control: To prevent behavior that infringes the organization's policies, is antisocial or has no discernible legitimate purpose and is hence suspected of forming part of an attack.
- User control: The organization may wish to discriminate between its users, allowing some to access external services but inhibiting others from doing so.

The policy has to be expressed in terms of filtering operations that are performed by filtering processes operating at several different levels:

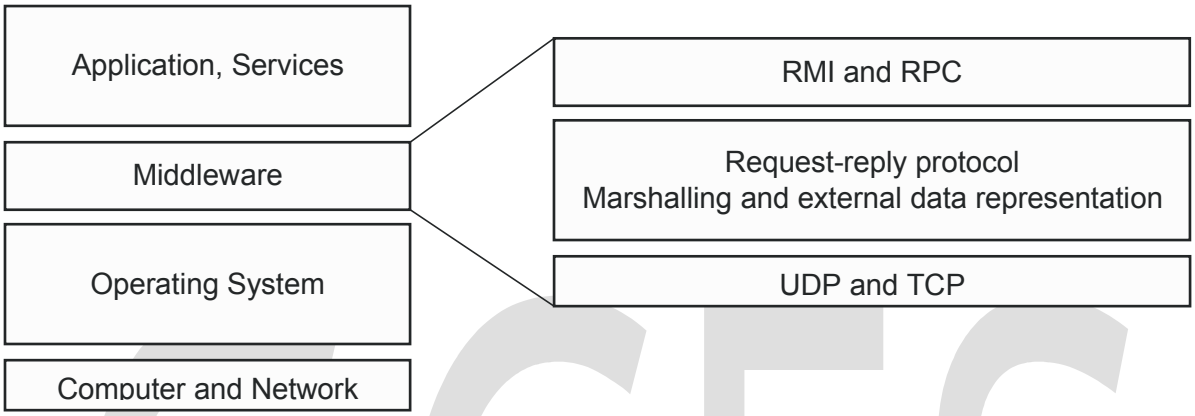
- IP packet filtering: This is a filter process examining individual IP packets. It may make decisions based on the destination and source addresses.
- TCP gateway: A TCP gateway process checks all TCP connection requests and segment transmissions.
- Application-level gateway: An application-level gateway process acts as a proxy for an application process.

Figure 3.21 Firewall configurations



Virtual private networks • Virtual private networks (VPNs) extend the firewall protection boundary beyond the local intranet by the use of cryptographically protected secure channels at the IP level. In Section 3.4.4, we outlined the IP security extensions available in IPv6 and IPv4 with IPSec tunneling [Thayer 1998]. These are the basis for the implementation of VPNs. They may be used for individual external users or to implement secure connections between intranets located at different sites using public Internet links.

Distributed Objects and Remote Communication



In distributed system middleware is all about providing communication platform for distributed application.

1. The API for the Internet Protocol

TCP and UDP provides API to Internet Protocol for high level application and services.

1.1. 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.

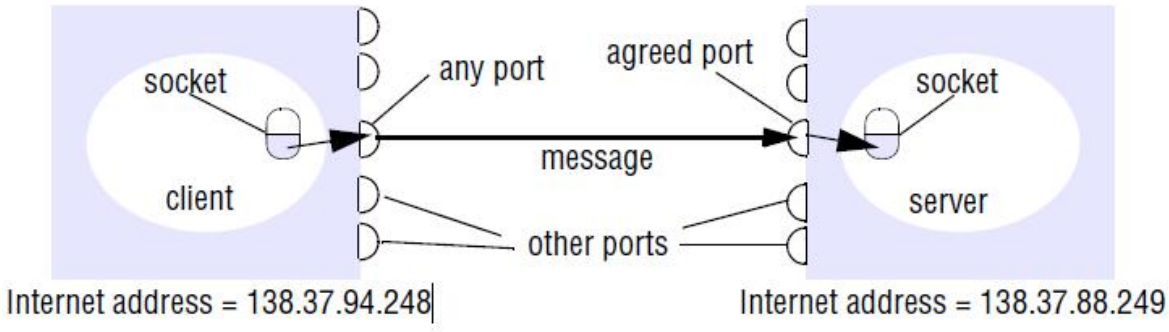
Synchronous and asynchronous communication: A queue is associated with each message destination. Sending processes cause messages to be added to remote queues and receiving processes remove messages from local queues. Communication between the sending and receiving processes may be either synchronous 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.

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 can be defined in 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.



Both forms of communication (UDP and TCP) use the *socket* abstraction, which provides an endpoint for communication between processes.

1.1.1. UDP Datagram Protocol

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.

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 2^{16} 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.

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 can be 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.

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. Section 5.2 discusses how reliable request-reply protocols for client-server communication may be built over UDP.

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.

1.1.2. TCP Stream Communication

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.

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 [Comer 2006] 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 establish 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 *accept*s 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 do 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; it is discussed in Section 5.2.

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.

1.2. External data representation and marshalling

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.

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 following

- 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
- 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.

Example

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

Let us consider Person struct with value: {'Smith', 'London', 1984}.

CORBA CDR

index in sequence of bytes	←4 bytes→	*notes on representation*
0–3	5	*length of string*
4–7	"Smit"	*'Smith'*
8–11	"h___"	
12–15	6	*length of string*
16–19	"Lond"	*'London'*
20–23	"on__"	
24–27	1984	*unsigned long*

Java object serialization

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

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

Extensible Markup Language (XML)

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

1.3. Remote Object Reference

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. Remote object references can also passed as arguments and returned as results of remote method invocations. 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.

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 4.13. 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.

Representation of a remote object reference

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

1.4. Multicast communication

The pairwise exchange of messages is not the best model for communication from one process to a group of other processes, which may be necessary, for example, when a service is implemented as a number of different processes in different computers, perhaps to provide fault tolerance or to enhance availability. A *multicast operation* is more appropriate – this is an operation that sends a single message from one process to each of the members of a group of processes, usually in such a way that the membership of the group is transparent to the sender. There is a range of possibilities in the desired behavior of a multicast. The simplest multicast protocol provides no guarantees about message delivery or ordering.

Multicast messages provide a useful infrastructure for constructing distributed systems with the following characteristics:

1. *Fault tolerance based on replicated services*: A replicated service consists of a group of servers. Client requests are multicast to all the members of the group, each of which performs an identical operation. Even when some of the members fail, clients can still be served.
2. *Discovering services in spontaneous networking*: Multicast messages can be used by servers and clients to locate available discovery services in order to register their interfaces or to look up the interfaces of other services in the distributed system.
3. *Better performance through replicated data*: Data are replicated to increase the performance of a service – in some cases replicas of the data are placed in users' computers. Each time the data changes, the new value is multicast to the processes managing the replicas.
4. *Propagation of event notifications*: Multicast to a group may be used to notify processes when something happens. For example, in Facebook, when someone changes their status, all their friends receive notifications. Similarly, publishsubscribe protocols may make use of group multicast to disseminate events to subscribers.

Time in Distributed System

We take a distributed system to consist of a collection § of N processes p_i, i = 1,2,…N. Each process executes on a single processor, and the processors do not share. Each process p_i in § 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. For example, if the processes in § are engaged in an ecommerce application, then the actions may be ones such as 'client dispatched order message' or 'merchant server recorded transaction to log'.

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 →i:

history(p_i) hi $<e_i^0 , e_i^1 , e_i^2 ,\ldots>$

1. Physical Clocks

Nearly all computers have a circuit for keeping track of time. Despite the widespread use of the word "clock" to refer to these devices, they are not actually clocks in the usual sense. Timer is perhaps a better word. A computer timer is usually a precisely machined quartz crystal. When kept under tension, quartz crystals oscillate at a well-defined frequency that depends on the kind of crystal, how it is cut, and the amount of tension. Associated with each crystal are two registers, a counter and a holding register. Each oscillation of the crystal decrements the counter by one. When the counter gets to zero, an interrupt is generated and the counter is reloaded from the holding register. In this way, it is possible to program a timer to generate an interrupt 60 times a second, or at any other desired frequency. Each interrupt is called one clock tick.

When the system is booted, it usually asks the user to enter the date and time, which is then converted to the number of ticks after some known starting date and stored in memory. Most computers have a special battery-backed up CMOS RAM so that the date and time need not be entered on subsequent boots. At every clock tick, the interrupt service procedure adds one to the time stored in memory.

The operating system reads the node's hardware clock value, $H_i(t)$, scales it and adds an offset so as to produce a software clock $C_i(t) = \alpha H_i(t) + \beta$ that approximately measures real, physical time t for process pi . In other words, when the real time in an absolute frame of reference is t, $C_i(t)$ is the reading on the software clock. For example, $C_i(t)$ could be the 64-bit value of the number of nanoseconds that have elapsed at time t since a convenient reference time. In general, the clock is not completely accurate, so $C_i(t)$ will differ from t. Nonetheless, if Ci behaves sufficiently well (we shall examine the notion of clock correctness shortly), we can use its value to timestamp any event at p_i. Note that successive events will correspond to different timestamps only if the clock resolution – the period between updates of the clock value – is smaller than the time interval between successive events. The rate at which events occur depends on such factors as the length of the processor instruction cycle.

Clock skew and clock drift: Computer clocks, like any others, tend not to be in perfect agreement. The instantaneous difference between the readings of any two clocks is called their skew. Also, the crystal-based clocks used in computers are, like any other clocks, subject to clock drift, which means that they count time at different rates, and so diverge. The underlying oscillators are subject to physical variations, with the consequence that their frequencies of oscillation differ. Moreover, even the same clock's frequency varies with temperature. Designs exist that attempt to compensate for this variation, but they cannot eliminate it. The difference in the oscillation period between two clocks might be extremely small, but the difference accumulated over many oscillations leads to an observable difference in the counters registered by two clocks, no matter how accurately they were initialized to the same value. A clock's drift rate is the change in the offset (difference in reading) between the clock and a nominal perfect reference clock per unit of time measured by the reference clock. For ordinary clocks based on a quartz crystal this is about 10^{-6} seconds/second, giving a difference of 1 second every 1,000,000 seconds, or 11.6 days. The drift rate of 'high-precision' quartz clocks is about 10–7 or 10–8.

Coordinated Universal Time: Computer clocks can be synchronized to external sources of highly accurate time. The most accurate physical clocks use atomic oscillators, whose drift rate is about one part in 10^{13}. The output of these atomic clocks is used as the 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 (Cs^{133}).

Seconds and years and other time units that we use are rooted in astronomical time. They were originally defined in terms of the rotation of the Earth on its axis and its rotation about the Sun. However, the period of the Earth's rotation about its axis is gradually getting longer, primarily because of tidal friction; atmospheric effects and convection currents within the Earth's core also cause short-term increases and decreases in the period. So astronomical time and atomic time have a tendency to get out of step.

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. For example, in the USA, the radio station WWV broadcasts time signals on several shortwave frequencies. Satellite sources include the Global Positioning System (GPS).

Receivers are available commercially. Compared with 'perfect' UTC, the signals received from land-based stations have an accuracy on the order of 0.1–10 milliseconds, depending on the station used. Signals received from GPS satellites are accurate to about 1 microsecond. Computers with receivers attached can synchronize their clocks with these timing signals.

2. Clock Synchronization

In a centralized system, time is unambiguous. When a process wants to know the time, it makes a system call and the kernel tells it. If process A asks for the time and then a little later process B asks for the time, the value that B gets will be higher than (or possibly equal to) the value A got. It will certainly not be lower. In a distributed system, achieving agreement on time is not trivial.

Just think, for a moment, about the implications of the lack of global time on the UNIX *make* program, as a single example. Normally, in UNIX, large programs are split up into multiple source files, so that a change to one source file only requires one file to be recompiled, not all the files. If a program consists of 100 files, not having to recompile everything because one file has been changed greatly increases the speed at which programmers can work.

The way *make* normally works is simple. When the programmer has finished changing all the source files, he runs *make*, which examines the times at which all the source and object files were last modified. If the source file *input.c* has time 2151 and the corresponding object file *input.o* has time 2150, make knows that *input.c* has been changed since *input.o* was created, and thus *input.c* must be recompiled. On the other hand, if *output.c* has time 2144 and *output.o* has time 2145, no compilation is needed. Thus *make* goes through all the source files to find out which ones need to be recompiled and calls the compiler to recompile them.

Now imagine what could happen in a distributed system in which there were no global agreement on time. Suppose that *output.o* has time 2144 as above, and shortly thereafter *output.c* is modified but is assigned time 2143 because the clock on its machine is slightly behind. *Make* will not call the compiler. The resulting executable binary program will then contain a mixture of object files from the old sources and the new sources. It will probably crash and the programmer will go crazy trying to understand what is wrong with the code.

2.1. Physical Clock Synchronization

In order to know at what time of day events occur at the processes in our distributed system § – 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 Ci 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,…N and for all real times t in I. Another way of saying this is that the clocks Ci 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,…N , and for all real times t in I. Another way of saying this is that the clocks Ci 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 ¢ 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 ρ>0 (a value derived from one supplied by the manufacturer, such as 10–6 seconds/second). This means that the error in measuring the interval between real times t and t' (t' > t) is bounded:

$$(1 – \rho)(t' – t) \leq H(t') – H(t) Ç (1 + \rho)(t' – t)$$

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 monotonicity may suffice. Monotonicity is the condition that a clock C only ever advances:

$$t' > t => C(t') > C(t)$$

For example, the UNIX make facility is a tool that is used to compile only those source files that have been modified since they were last compiled. The modification dates of each corresponding pair of source and object files are compared to determine this condition. If a computer whose clock was running fast set its clock back after compiling a source file but before the file was changed, the source file might appear to have been modified prior to the compilation. Erroneously, make will not recompile the source file.

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 $C_i(t) = \alpha Hi(t) + \beta$, where we are free to choose the values of α and β.

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 faulty. A clock's crash failure is said to occur when the clock stops ticking altogether; any other clock failure is an arbitrary failure. A historical example of an arbitrary failure is that of a clock with the 'Y2K bug', which broke the monotonicity condition by registering the date after 31 December 1999 as 1 January 1900 instead of 2000; another example is a clock whose batteries are very low and whose drift rate suddenly becomes very large.

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. We now describe algorithms for external synchronization and for internal synchronization.

2.2. 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_{trans}, where T_{trans} is the time taken to transmit m between them. The two clocks would then agree (since the aim is internal synchronization, it does not matter whether the sending process's clock is accurate).

Unfortunately, T_{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, min, that would be obtained if no other processes executed and no other network traffic existed; min can be measured or conservatively estimated.

In a synchronous system, by definition, there is also an upper bound max on the time taken to transmit any message. Let the uncertainty in the message transmission time be u, so that u = (max – min). If the receiver sets its clock to be t + min, then the clock skew may be as much as u, since the message may in fact have taken time max to arrive. Similarly, if it sets its clock to t + max, the skew may again be as large as u. If, however, it sets its clock to the halfway point, t + (max + 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 max on message transmission delays. This is particularly so for the Internet. For an asynchronous system, we

may say only that T_{trans} = min + x, where x ≥ 0. The value of x is not known in a particular case, although a distribution of values may be measurable for a particular installation.

2.3. Cristian's method for synchronizing clocks
2.4. The Berkeley algorithm
2.5. The Network Time Protocol
3. Logical Time
3.1. Logical Clocks
3.2. Vector Clock

Note: Casual ordering of event is obtained through the use of logical clock or vector clock

---

For above topics please go through book. Every line there are important

---

4. Global State

There are many situations where global state are required. Some of them are as follows:

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 (a) below, process p1 has two objects that both have references – one has a reference within p1 itself, and p2 has a reference to the other. Process p2 has one garbage object, with no references to it anywhere in the system. It also has an object for which neither p1 nor p2 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.

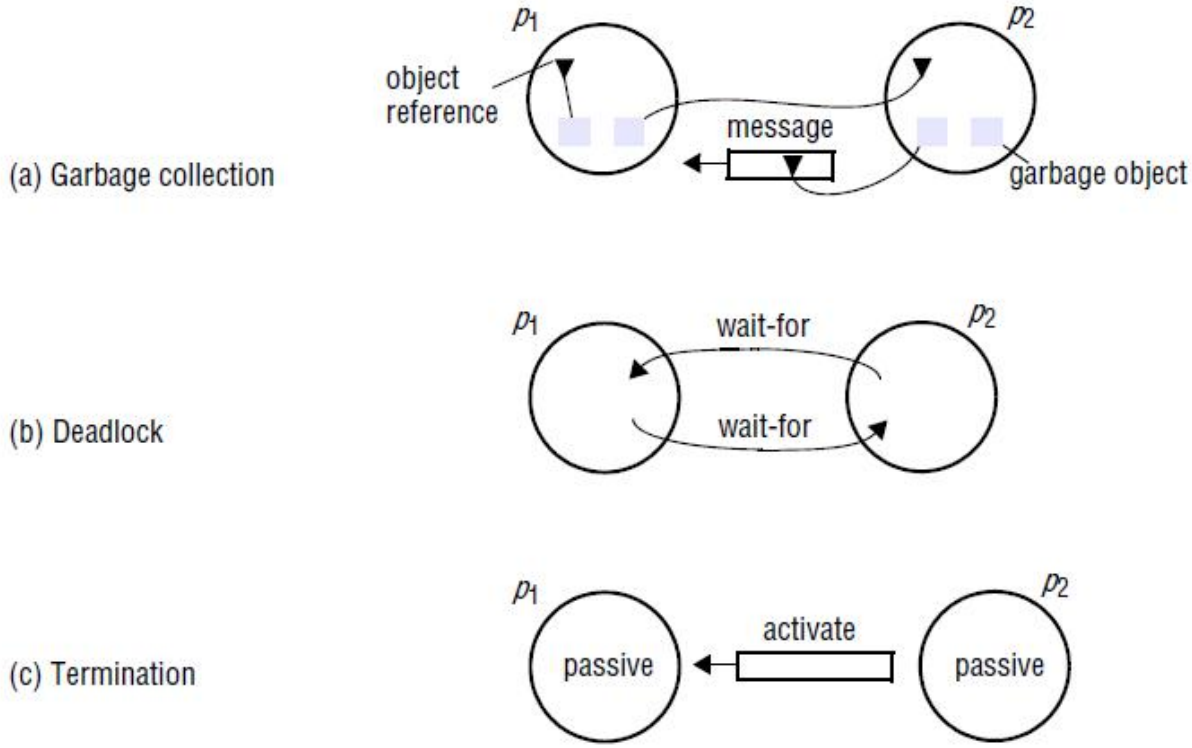
Distributed deadlock detection: A distributed deadlock occurs when each of a collection of processes waits for another process to send it a message, and where there is a cycle in the graph of this 'waits-for' relationship. Figure (b) shows that processes p1 and p2 are each waiting for a message from 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 p1 and p2, 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 p1 is passive and that p2 is passive (Figure c). To see that we may not conclude that the algorithm has terminated,

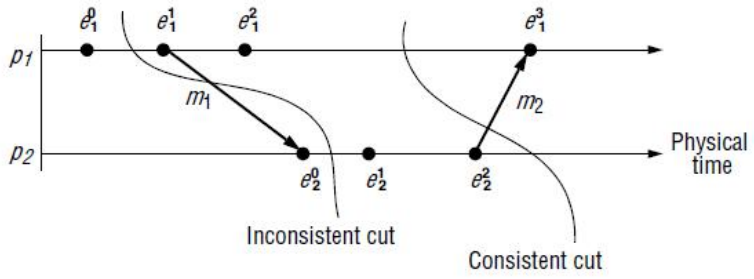
consider the following scenario: when we tested p1 for passivity, a message was on its way from p2, which became passive immediately after sending it. On receipt of the message, p1 became active again – after we had found it to be passive. The algorithm had not terminated.

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 pi contains a variable x_i (i = 1, 2, … 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 |xi – xj| > 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.



4.1. Global states and consistent cuts



Go through book for more

4.2. The 'snapshot' algorithm of Chandy and Lamport

Chandy and Lamport [1985] describe a 'snapshot' algorithm for determining global states of distributed systems. The goal of the algorithm is to record a set of process and channel states (a 'snapshot') for a set of processes p_i (i = 1, 2, … N) such that, even though the combination of recorded states may never have occurred at the same time, the recorded global state is consistent.

The algorithm records state locally at processes; it does not give a method for gathering the global state at one site. An obvious method for gathering the state is for all processes to send the state they recorded to a designated collector process, but we shall not address this issue further here.

The algorithm assumes that:

- Neither channels nor processes fail – communication is reliable so that every message sent is eventually received intact, exactly once.
- Channels are unidirectional and provide FIFO-ordered message delivery.
- The graph of processes and channels is strongly connected (there is a path between any two processes).
- Any process may initiate a global snapshot at any time.
- The processes may continue their execution and send and receive normal messages while the snapshot takes place.

For each process p_i, let the incoming channels be those at pi over which other processes send it messages; similarly, the outgoing channels of pi are those on which it sends messages to other processes. The essential idea of the algorithm is as follows. Each process records its state and also, for each incoming channel, a set of messages sent to it. The process records, for each channel, any messages that arrived after it recorded its state and before the sender recorded its own state. This arrangement allows us to record the states of processes at different times but to account for the differentials between process states in terms of messages transmitted but not

yet received. If process pi has sent a message m to process p_j, but p_j has not received it, then we account for m as belonging to the state of the channel between them.

The algorithm proceeds through use of special marker messages, which are distinct from any other messages the processes send and which the processes may send and receive while they proceed with their normal execution. The marker has a dual role: as a prompt for the receiver to save its own state, if it has not already done so; and as a means of determining which messages to include in the channel state.

The algorithm is defined through two rules, the marker receiving rule and the marker sending rule. The marker sending rule obligates processes to send a marker after they have recorded their state, but before they send any other messages.

The marker receiving rule obligates a process that has not recorded its state to do so. In that case, this is the first marker that it has received. It notes which messages subsequently arrive on the other incoming channels. When a process that has already saved its state receives a marker (on another channel), it records the state of that channel as the set of messages it has received on it since it saved its state.

Any process may begin the algorithm at any time. It acts as though it has received a marker (over a nonexistent channel) and follows the marker receiving rule. Thus it records its state and begins to record messages arriving over all its incoming channels. Several processes may initiate recording concurrently in this way (as long as the markers they use can be distinguished).

Marker receiving rule for process p_i

On receipt of a *marker* message at p_i over channel c:

if (p_i has not yet recorded its state) it

records its process state now;

records the state of c as the empty set;

turns on recording of messages arriving over other incoming channels;

else

p_i records the state of c as the set of messages it has received over c since it saved its state.

end if

Marker sending rule for process p_i

After p_i has recorded its state, for each outgoing channel c:

p_i sends one marker message over c

(before it sends any other message over c).

Agreement in Distributed System

1. Election Algorithm

An algorithm for choosing a unique process to play a particular role is called an election algorithm. For example, in a variant of our central-server algorithm for mutual exclusion, the 'server' is chosen from among the processes p_i, (i = 1, 2…N) that need to use the critical section. An election algorithm is needed for this choice. It is essential that all the processes agree on the choice. Afterwards, if the process that plays the role of server wishes to retire then another election is required to choose a replacement.

We say that a process calls the election if it takes an action that initiates a particular run of the election algorithm. An individual process does not call more than one election at a time, but in principle the N processes could call N concurrent elections.

At any point in time, a process pi is either a participant – meaning that it is engaged in some run of the election algorithm – or a non-participant – meaning that it is not currently engaged in any election.

An important requirement is for the choice of elected process to be unique, even if several processes call elections concurrently. For example, two processes could decide independently that a coordinator process has failed, and both call elections.

Without loss of generality, we require that the elected process be chosen as the one with the largest identifier. The 'identifier' may be any useful value, as long as the identifiers are unique and totally ordered. For example, we could elect the process with the lowest computational load by having each process use <1 / load, i > as its identifier, where load > 0 and the process index i is used to order identifiers with the same load.

Each process p_i, (i = 1, 2…N) has a variable $elected_i$, which will contain the identifier of the elected process. When the process first becomes a participant in an election it sets this variable

to the special value '⊥' to denote that it is not yet defined.

Our requirements are that, during any particular run of the algorithm:

E1: (safety) A participant process p_i has $elected_i = \perp$ or $elected_i = P$, where P is chosen as the

non-crashed process at the end of the run with the largest identifier.

E2: (liveness) All processes pi participate and eventually either set $elected_i \neq \perp$ – or crash.

Note that there may be processes p_j that are not yet participants, which record in $elected_j$ the identifier of the previous elected process.

We measure the performance of an election algorithm by its total network bandwidth utilization (which is proportional to the total number of messages sent), and by the turnaround time for the algorithm: the number of serialized message transmission times between the initiation and termination of a single run.

1.1. A Ring-Based Election Algorithm

The algorithm of Chang and Roberts [1979] is suitable for a collection of processes arranged in a logical ring. Each process pi has a communication channel to the next process in the ring, $p_{(i + 1) \bmod N}$, and all messages are sent clockwise around the ring. We assume that no failures occur, and that the system is asynchronous. The goal of this algorithm is to elect a single process called the coordinator, which is the process with the largest identifier.

Initially, every process is marked as a non-participant in an election. Any process can begin an election. It proceeds by marking itself as a participant, placing its identifier in an election message and sending it to its clockwise neighbour.

When a process receives an election message, it compares the identifier in the message with its own. If the arrived identifier is greater, then it forwards the message to its neighbour. If the arrived identifier is smaller and the receiver is not a participant, then it substitutes its own identifier in the message and forwards it; but it does not forward the message if it is already a participant. On forwarding an election message in any case, the process marks itself as a participant.

If, however, the received identifier is that of the receiver itself, then this process's identifier must be the greatest, and it becomes the coordinator. The coordinator marks itself as a non-participant once more and sends an elected message to its neighbour, announcing its election and enclosing its identity.

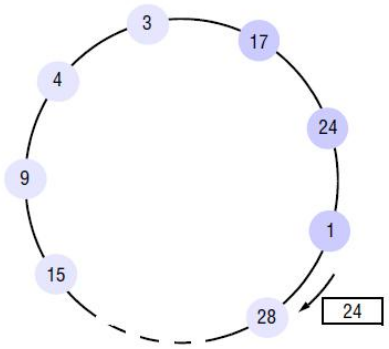
When a process pi receives an elected message, it marks itself as a nonparticipant, sets its variable $elected_i$ to the identifier in the message and, unless it is the new coordinator, forwards the message to its neighbour.

It is easy to see that condition E1 is met. All identifiers are compared, since a process must receive its own identifier back before sending an elected message. For any two processes, the one with the larger identifier will not pass on the other's identifier. It is therefore impossible that both should receive their own identifier back.

Condition E2 follows immediately from the guaranteed traversals of the ring (there are no failures). Note how the non-participant and participant states are used so that duplicate messages arising when two processes start an election at the same time are extinguished as soon as possible, and always before the 'winning' election result has been announced.

If only a single process starts an election, then the worst-performing case is when its anti-clockwise neighbour has the highest identifier. A total of N – 1 messages are then required to reach this neighbour, which will not announce its election until its identifier has completed another circuit, taking a further N messages. The elected message is then sent N times, making 3N – 1 messages in all. The turnaround time is also 3N – 1 , since these messages are sent sequentially.

An example of a ring-based election in progress is shown in figure below. The election message currently contains 24, but process 28 will replace this with its identifier when the message reaches it.



While the ring-based algorithm is useful for understanding the properties of election algorithms in general, the fact that it tolerates no failures makes it of limited practical value. However, with a reliable failure detector it is in principle possible to reconstitute the ring when a process crashes.

1.2. The Bully Algorithm

The bully algorithm [Garcia-Molina 1982] allows processes to crash during an election, although it assumes that message delivery between processes is reliable. Unlike the ring-based algorithm, this algorithm assumes that the system is synchronous: it uses timeouts to detect a process failure. Another difference is that the ring-based algorithm assumed that processes have minimal a priori knowledge of one another: each knows only how to communicate with its neighbour, and none knows the identifiers of the other processes. The bully algorithm, on the other hand, assumes that each process knows which processes have higher identifiers, and that it can communicate with all such processes.

There are three types of message in this algorithm: an election message is sent to announce an election; an answer message is sent in response to an election message and a coordinator message is sent to announce the identity of the elected process – the new 'coordinator'. A

process begins an election when it notices, through timeouts, that the coordinator has failed. Several processes may discover this concurrently.

Since the system is synchronous, we can construct a reliable failure detector. There is a maximum message transmission delay, T_{trans}, and a maximum delay for processing a message $T_{process}$. Therefore, we can calculate a time $T = 2T_{trans} + T_{process}$ that is an upper bound on the time that can elapse between sending a message to another process and receiving a response. If no response arrives within time T, then the local failure detector can report that the intended recipient of the request has failed.

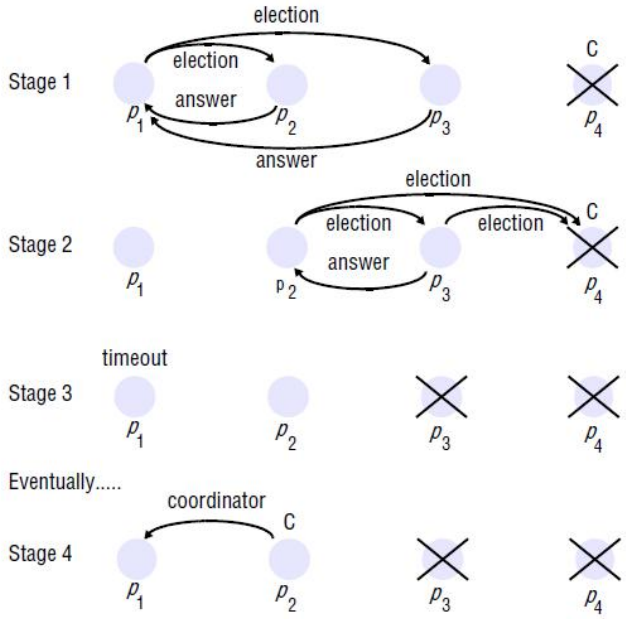
The process that knows it has the highest identifier can elect itself as the coordinator simply by sending a coordinator message to all processes with lower identifiers. On the other hand, a process with a lower identifier can begin an election by sending an election message to those processes that have a higher identifier and awaiting answer messages in response. If none arrives within time T, the process considers itself the coordinator and sends a coordinator message to all processes with lower identifiers announcing this. Otherwise, the process waits a further period T' for a coordinator message to arrive from the new coordinator. If none arrives, it begins another election.

If a process pi receives a coordinator message, it sets its variable $elected_i$ to the identifier of the coordinator contained within it and treats that process as the coordinator.

If a process receives an election message, it sends back an answer message and begins another election – unless it has begun one already.

When a process is started to replace a crashed process, it begins an election. If it has the highest process identifier, then it will decide that it is the coordinator and announce this to the other processes. Thus it will become the coordinator, even though the current coordinator is functioning. It is for this reason that the algorithm is called the 'bully' algorithm.

Example



The election of coordinator p_2, after the failure of p_4 and then p_3

There are four processes, p_1 –p_4. Process p_1 detects the failure of the coordinator p_4 and announces an election (stage 1 in the figure). On receiving an election message from p_1, processes p_2 and p_3 send answer messages to p_1 and begin their own elections; p_3 sends an answer message to p_2 , but p_3 receives no answer message from the failed process p_4 (stage

2). It therefore decides that it is the coordinator. But before it can send out the coordinator message, it too fails (stage 3). When p_1's timeout period T' expires (which we assume occurs before p_2's timeout expires), it deduces the absence of a coordinator message and begins another election. Eventually, p_2 is elected coordinator (stage 4).

This algorithm clearly meets the liveness condition E2, by the assumption of reliable message delivery. And if no process is replaced, then the algorithm meets condition E1. It is impossible for two processes to decide that they are the coordinator, since the process with the lower identifier will discover that the other exists and defer to it.

But the algorithm is not guaranteed to meet the safety condition E1 if processes that have crashed are replaced by processes with the same identifiers. A process that replaces a crashed process p may decide that it has the highest identifier just as another process (which has detected p's crash) decides that it has the highest identifier. Two processes will therefore announce themselves as the coordinator concurrently.

Unfortunately, there are no guarantees on message delivery order, and the recipients of these messages may reach different conclusions on which is the coordinator process.

Furthermore, condition E1 may be broken if the assumed timeout values turn out to be inaccurate – that is, if the processes' failure detector is unreliable.

Taking the example just given, suppose that either p_3 had not failed but was just running unusually slowly (that is, that the assumption that the system is synchronous is incorrect), or that p_3 had failed but was then replaced. Just as p_2 sends its coordinator message, p_3 (or its replacement) does the same. p_2 receives p_3's coordinator message after it has sent its own and so sets $elected_2 = p_3$. Due to variable message transmission delays, p_1 receives p_2's coordinator message after p_3's and so eventually sets $elected_1 = p_2$. Condition E1 has been broken.

With regard to the performance of the algorithm, in the best case the process with the second-highest identifier notices the coordinator's failure. Then it can immediately elect itself and send N – 2 coordinator messages. The turnaround time is one message. The bully algorithm requires $O(N^2)$ messages in the worst case – that is, when the process with the lowest identifier first detects the coordinator's failure. For then N – 1 processes altogether begin elections, each sending messages to processes with higher identifiers.

2. Mutual Exclusion

Fundamental to distributed systems is the concurrency and collaboration among multiple processes. In many cases, this also means that processes will need to simultaneously access the same resources. To prevent that such concurrent accesses corrupt the resource, or make it inconsistent, solutions are needed to grant mutual exclusive access by processes. In this section, we take a look at some of the more important distributed algorithms that have been proposed. A recent survey of distributed algorithms for mutual exclusion is provided by Saxena and Rai (2003). Older, but still relevant is Velazquez (1993).

We consider a system of N processes p_i, i = 1, 2, … N, that do not share variables. The processes access common resources, but they do so in a critical section. For the sake of simplicity, we assume that there is only one critical section. It is straightforward to extend the algorithms we present to more than one critical section.

We assume that the system is asynchronous, that processes do not fail and that message delivery is reliable, so that any message sent is eventually delivered intact, exactly once.

The application-level protocol for executing a critical section is as follows:

```
enter()                 // enter critical section – block if necessary
resourceAccesses()      // access shared resources in critical section
exit()                  // leave critical section – other processes may now enter
```

Our essential requirements for mutual exclusion are as follows:

ME1: (safety) At most one process may execute in the critical section (CS) at a time.

ME2: (liveness) Requests to enter and exit the critical section eventually succeed.

Condition ME2 implies freedom from both deadlock and starvation. A deadlock would involve two or more of the processes becoming stuck indefinitely while attempting to enter or exit the critical section, by virtue of their mutual interdependence. But even without a deadlock, a poor algorithm might lead to starvation: the indefinite postponement of entry for a process that has requested it.

The absence of starvation is a fairness condition. Another fairness issue is the order in which processes enter the critical section. It is not possible to order entry to the critical section by the times that the processes requested it, because of the absence of global clocks. But a useful fairness requirement that is sometimes made makes use of the happened-before ordering (between messages that request entry to the critical section:

ME3: (→ ordering) If one request to enter the CS happened-before another, then entry to the CS is granted in that order.

If a solution grants entry to the critical section in happened-before order, and if all requests are related by happened-before, then it is not possible for a process to enter the critical section more than once while another waits to enter. This ordering also allows processes to coordinate their accesses to the critical section. A multi-threaded process may continue with other processing while a thread waits to be granted entry to a critical section. During this time, it might send a message to another process, which consequently also tries to enter the critical section. ME3 specifies that the first process be granted access before the second.

We evaluate the performance of algorithms for mutual exclusion according to the following criteria:

- The bandwidth consumed, which is proportional to the number of messages sent in each entry and exit operation;
- The client delay incurred by a process at each entry and exit operation;
- The algorithm's effect upon the throughput of the system. This is the rate at which the collection of processes as a whole can access the critical section, given that some communication is necessary between successive processes. We measure the effect using the synchronization delay between one process exiting the critical section and the next process entering it; the throughput is greater when the synchronization delay is shorter.

Distributed mutual exclusion algorithms can be classified into two different categories. In token-based solutions mutual exclusion is achieved by passing a special message between the processes, known as a token. There is only one token available and whoever has that token is allowed to access the shared resource. When finished, the token is passed on to a next process. If a process having the token is not interested in accessing the resource, it simply passes it on.

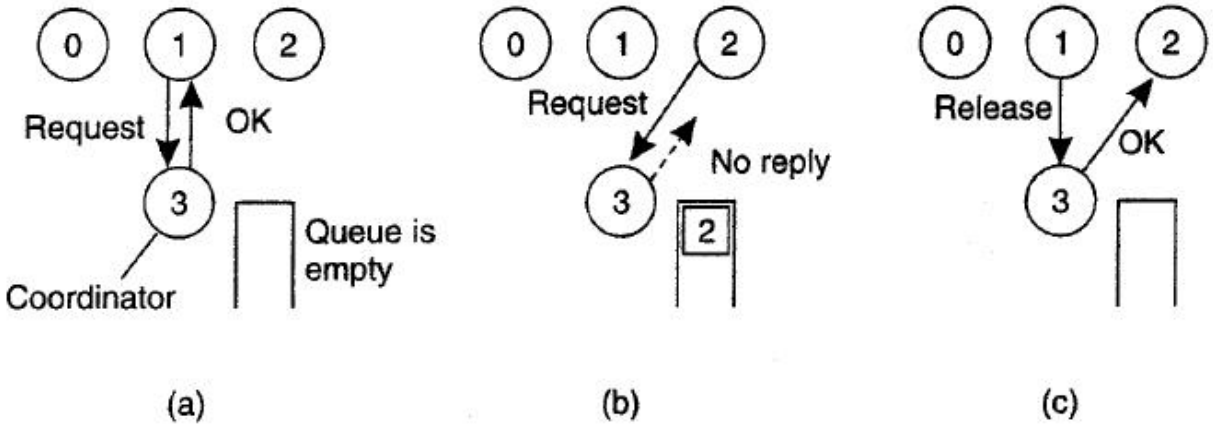
Token-based solutions have a few important properties. First, depending on the how the processes are organized, they can fairly easily ensure that every process will get a chance at accessing the resource. In other words, they avoid starvation. Second, deadlocks by which several processes are waiting for each other to proceed, can easily be avoided, contributing to their simplicity. Unfortunately, the main drawback of token-based solutions is a rather serious

one: when the token is lost (e.g., because the process holding it crashed), an intricate distributed procedure needs to be started to ensure that a new token is created, but above all, that it is also the only token.

As an alternative, many distributed mutual exclusion algorithms follow a permission-based approach. In this case, a process wanting to access the re source first requires the permission of other processes. There are many different ways toward granting such a permission and in the sections that follow we will consider a few of them.

2.1. A Centralized Algorithm

The most straightforward way to achieve mutual exclusion in a distributed system is to simulate how it is done in a one-processor system. One process is elected as the coordinator. Whenever a process wants to access a shared resource, it sends a request message to the coordinator stating which resource it wants to access and asking for permission. If no other process is currently accessing that resource, the coordinator sends back a reply granting permission, as shown in figure (a) below. When the reply arrives, the requesting process can go ahead.



Now suppose that another process, 2 in figure (b), asks for permission to access the resource. The coordinator knows that a different process is already at the resource, so it cannot grant permission. The exact method used to deny permission is system dependent. In figure (b), the coordinator just refrains from replying, thus blocking process 2, which is waiting for a reply. Alternatively, it could send a reply saying "permission denied." Either way, it queues the request from 2 for the time being and waits for more messages.

When process 1 is finished with the resource, it sends a message to the coordinator releasing its exclusive access, as shown in figure (c). The coordinator takes the first item off the queue of deferred requests and sends that process a grant message. If the process was still blocked (i.e., this is the first message to it), it unblocks and accesses the resource. If an explicit message has already been sent denying permission, the process will have to poll for incoming traffic or block later. Either way, when it sees the grant, it can go ahead as well.

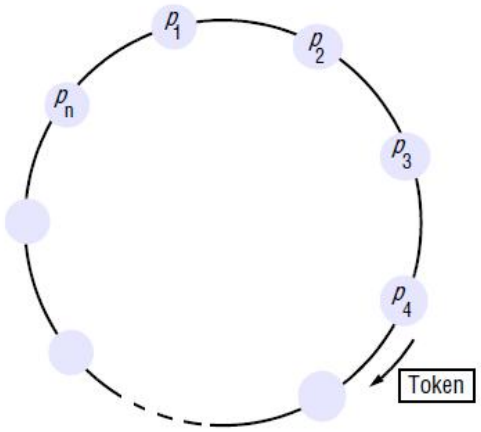
It is easy to see that the algorithm guarantees mutual exclusion: the coordinator only lets one process at a time to the resource. It is also fair, since requests are granted in the order in which they are received. No process ever waits forever (no starvation). The scheme is easy to implement, too, and requires only three messages per use of resource (request, grant, release). Its simplicity makes an attractive solution for many practical situations.

The centralized approach also has shortcomings. The coordinator is a single point of failure, so if it crashes, the entire system may go down. If processes normally block after making a request, they cannot distinguish a dead coordinator from "permission denied" since in both cases no message comes back. In addition, in a large system, a single coordinator can become a performance bottleneck. Nevertheless, the benefits coming from its simplicity outweigh in many cases the potential drawbacks.

2.2. A Token Ring Algorithm

One of the simplest ways to arrange mutual exclusion between the N processes without requiring an additional process is to arrange them in a logical ring. This requires only that each process pi has a communication channel to the next process in the ring, $p_{(i + 1) \bmod N}$. The idea is that exclusion is conferred by obtaining a token in the form of a message passed from process to process in a single direction clockwise, say – around the ring. The ring topology may be unrelated to the physical interconnections between the underlying computers.

If a process does not require to enter the critical section when it receives the token, then it immediately forwards the token to its neighbour. A process that requires the token waits until it receives it, but retains it. To exit the critical section, the process sends the token on to its neighbour.



The arrangement of processes is shown in figure above. It is straightforward to verify that the conditions ME1 and ME2 are met by this algorithm, but that the token is not necessarily obtained in happened-before order. (Recall that the processes may exchange messages independently of the rotation of the token.)

This algorithm continuously consumes network bandwidth (except when a process is inside the critical section): the processes send messages around the ring even when no process requires entry to the critical section. The delay experienced by a process requesting entry to the critical section is between 0 messages (when it has just received the token) and N messages (when it has just passed on the token). To exit the critical section requires only one message. The synchronization delay between one process's exit from the critical section and the next process's entry is anywhere from 1 to N message transmissions.

Algorithm has other problems to. If the token is ever lost, it must be regenerated. In fact, detecting that it is lost is difficult, since the amount of time between successive appearances of the token on the network is unbounded. The fact that the token has not been spotted for an hour does not mean that it has been lost; somebody may still be using it.

The algorithm also runs into trouble if a process crashes, but recovery is easier than in the other cases. If we require a process receiving the token to acknowledge receipt, a dead process will be detected when its neighbor tries to give it the token and fails. At that point the dead process can be removed from the group, and the token holder can throw the token over the head of the dead process to the next member down the line, or the one after that, if necessary. Of course, doing so requires that everyone maintain the current ring configuration.

2.2.1. Ricart and Agrawala Algorithm

Ricart and Agrawala [1981] developed an algorithm to implement mutual exclusion between N peer processes that is based upon multicast. The basic idea is that processes that require entry to a critical section multicast a request message, and can enter it only when all the other processes have replied to this message. The conditions under which a process replies to a request are designed to ensure that conditions ME1–ME3 are met.

Ricart and Agrawala's algorithm

```
On initialization
    state := RELEASED;

To enter the section
    state := WANTED;
    Multicast request to all processes;          } Request processing deferred here
    T := request's timestamp;
    Wait until (number of replies received = (N – 1));
    state := HELD;

On receipt of a request <Ti, pi> at pj (i≠ j)
    if (state = HELD or (state = WANTED and (T, pj) < (Ti, pi)))
    then
            queue request from pi without replying;
    else
            reply immediately to pi;
    end if

To exit the critical section
    state := RELEASED;
    reply to any queued requests;
```

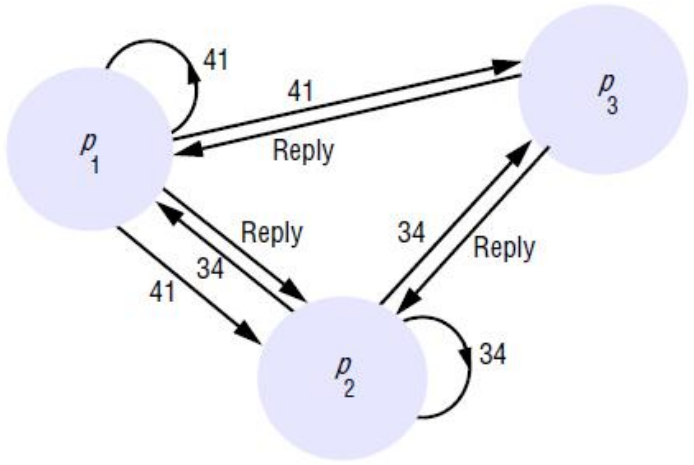
The processes p_1, p_2 … p_N bear distinct numeric identifiers. They are assumed to possess communication channels to one another, and each process p_i keeps a Lamport clock, updated according to the rules LC1 and LC2 of Lamport clock. Messages requesting entry are of the form <T, p_i>, where T is the sender's timestamp and pi is the sender's identifier.

Each process records its state of being outside the critical section (RELEASED), wanting entry (WANTED) or being in the critical section (HELD) in a variable state.

If a process requests entry and the state of all other processes is RELEASED, then all processes will reply immediately to the request and the requester will obtain entry. If some process is in the state HELD, then that process will not reply to requests until it has finished with the critical section, and so the requester cannot gain entry in the meantime. If two or more processes request entry at the same time, then whichever process's request bears the lowest timestamp will be the first to collect N – 1 replies, granting it entry next. If the requests bear equal Lamport timestamps, the requests are ordered according to the processes' corresponding identifiers. Note that, when a process requests entry, it defers processing requests from other processes until its own request has been sent and it has recorded the timestamp T of the request. This is so that processes make consistent decisions when processing requests.

This algorithm achieves the safety property ME1. If it were possible for two processes p_i and p_j ($i \neq j$) to enter the critical section at the same time, then both of those processes would have to have replied to the other. But since the pairs <T_i, p_i> are totally ordered, this is impossible. We leave the reader to verify that the algorithm also meets requirements ME2 and ME3.

To illustrate the algorithm, consider a situation involving three processes, p_1, p_2 and p_3, shown in figure



Let us assume that p_3 is not interested in entering the critical section, and that p_1 and p_2 request entry concurrently. The timestamp of p_1's request is 41, and that of p_2 is 34. When p_3 receives their requests, it replies immediately. When p_2 receives p_1's request, it finds that its own request has the lower timestamp and so does not reply, holding p_1 off. However, p1 finds that p_2's request has a lower timestamp than that of its own request and so replies immediately. On

receiving this second reply, p_2 can enter the critical section. When p_2 exits the critical section, it will reply to p_1's request and so grant it entry.

Gaining entry takes 2(N – 1) messages in this algorithm: N – 1 to multicast the request, followed by N – 1 replies. Or, if there is hardware support for multicast, only one message is required for the request; the total is then N messages. It is thus a more expensive algorithm, in terms of bandwidth consumption, than the algorithms just described. However, the client delay in requesting entry is again a round-trip time (ignoring any delay incurred in multicasting the request message).

The advantage of this algorithm is that its synchronization delay is only one message transmission time. Both the previous algorithms incurred a round-trip synchronization delay.

Lab Works

1. Lab #1
    a. TCP implementation
2. Lab #2
    a. UDP implementation
3. Lab #3
    a. Multicast using UPD
4. Lab #4
    a. Java Object Serialization
5. Lab #5
    a. RMI implementation
6. Lab #6
    a. CORBA implementation
7. Lab #7
8. Lab #
    a. Multi threads and Lock

Assignments

1. Assignment 1

Construct a reliable point-to-point basic file transfer tool using UDP/IP.

1. The assignment:

Using the unreliable UDP/IP protocol, you should write a tool that reliably copies a file to another machine.

The tool consists of two programs:

netcpy - the sender process.

netrcv - the receiver process.

In order to perform a file transfer operation, a receiver process (netrcv) should be run on the target machine.

A sender process (netcpy) should be run on the source machine using the following interface:

netcpy <source_file_name> <dest_file_name> <comp_ip> <comp_port> where <comp_ip> and <comp_port> are the ip and port of the computer where the server runs .

A receiver process (netrcv) should handle an UNLIMITED number of file-transfer operations, but it is allowed to handle one operation at a time (sequence them). If a sender comes along while the receiver is busy, the sender should be blocked until the receiver completes the current transfer. The sender should know about this and report this.

The receiver process has the following interface:

netrcv <comp_port> where <comp_port> is port where server will receive package.

A sender process (netcpy) handles one operation and then terminates. You can assume that the source file name represents a specific single file.

Both the sender (netcpy) and the receiver (netrcv) programs should report two statistics every 50Mbytes of data sent/received IN ORDER (all the data from the beginning of the file to that point was received with no gaps):

1) The total amount of data (in Mbytes) successfully transferred by that time.
2) The average transfer rate of the last 50Mbytes sent/received (in Mbits/sec).

At the end of the transfer, both sender and receiver programs should report the size of the file transferred, the amount of time required for the transfer, and the average rate at which the communication occurred (in Mbits/sec).

Moreover sender(netcpy) should show one more statistics along with above – total no. of failed packages.

2. Performance:

Using the reliable TCP/IP protocol, implement the corresponding t_netcpy and t_netrcv. These should report the amount of data transferred and the rate, the same way as netcpy and netrcv programs.

Compare the performance of (netcpy netrcv) to (t_netcpy t_netrcv) by measuring the time it takes to copy a 500 Mbytes file from one computer to another computer on the network. (this will evaluate your protocol when COMPETING with TCP)

3. Submission:

Two separate submissions are required. These submissions should be maintained in one of the online source controls (github, gitlab, bitbucket). Final submission will be taken from the source control. Your commits and contribution in source control will be considered for individual marking.

1) A paper submission of your draft design. Due **Thursday April 7** at the beginning of class. All the submission documents should be computer printed (not hand-written !)

A design document should be a well-written, complete design of your netcpy and netrcv programs and the protocols used to make it reliable. This includes the internal data structure in each program, the type and structure of the messages used, and the protocol (algorithm) description. It should be about 2-3 pages long with diagrams as needed.

2) A final design document is required with your final submission. The complete submission should include your final design. It should also include a discussion of the performance results you achieved and discuss the comparison of how the protocol worked under different percentages of loss. The programs should be complete, sound, and well documented.

You should work on it in groups of exactly 2 students. Contact us if you cannot find a partner and we will assign a partner for you.

Failure to comply with submission instructions will result in a lower grade. Issues such as correctness, design, documentation, and, of course, efficiency will be considered.

Final Submission date is Wednesday, April 22, 2016, 11:00pm.

2. Assignment 2

Simulate a Distributed file system using java RMI.

1. The Assignment

Using java RMI you should write a system to simulate Distributed file system where we access files of remote server in local client. The system will have two programs.

- nfs_server: file server
- nfs_client: file user

In order to simulate distributed file system, a fileserver process (nfs_server) should be run on the remote machine containing the files. The nfs_server points to a static directory and serves it as a drive for nfs_client program. nfs_server is responsible for accepting a certain set of bash commands, execute them and reply the result to nfs_client.

On the other hand nfs_client is client which browses the drive provided by nfs_server. nfs_client will run with the following interface.

nfs <command_name> <options> <arguments>

Following commands are expected to be implemented in the system

```
ls -a = list all files and folders
ls <folderName> = list files in
folder ls -lh = Detailed list, Human
readable ls -l *.jpg = list jpeg
files only ls -lh <fileName> =
Result for file only

cd <folderName> = change
directory if folder name has
spaces use “ ” cd / = go to
root cd .. = go up one folder

mkdir = create new
folder(s) mkdir myStuff
mkdir myStuff1 mystuff2

 cp image.jpg newimage.jpg = copy and rename a
file cp image.jpg <folderName>/ = copy to
folder cp image.jpg folder/sameImageNewName.jpg
cp -R stuff otherStuff = copy and rename a
folder cp *.txt stuff/ = copy all of *<file
type> to folder
```

```
rm <fileName> .. = delete file (s)

rm -i <fileName> .. = ask for confirmation each
file rm -f <fileName> = force deletion of a
file rm -r <foldername>/ = delete folder
```

2. Submission

Two separate submissions are required. These submissions should be maintained in one of the online source controls-Gitlab. Final submission will be taken from the source control. Your commits and contribution in source control will be considered for individual marking.

1) A paper submission of your draft design. Due **Friday May 13** at the beginning of class. All the submission documents should be computer printed (not hand-written!)

A design document should be include a complete design of all your class and an interaction diagram with explanation. It should be about 2-3 pages long with diagrams as needed.

2) A final design document is required with your final submission. The complete submission should include your final design. The programs should be complete, sound, and well documented. This is not a team project so an individual will be a team.

Failure to comply with submission instructions will result in a lower grade. Issues such as correctness, design, documentation, and, of course, efficiency will be considered.

Final Submission date is Wednesday, May 22, 2016, 11:00pm.