Distributed Systems

Section # 1 Introduction to Distributed Systems and Characterization

Distributed Systems

- We define a distributed system as one in which hardware or software components located at networked computers that communicate and coordinate their actions only by passing messages. Computers that are connected by a network may be spatially separated by any distance.
- A distributed system is a collection of independent computers that appear to the users of the system as a single computer.

Consequences of Distributed Systems

1. **Concurrency**: The coordination of concurrently executing programs that share resources is a major challenge in the world of distributed systems. Usually this problem is addressed by message passing or synchronization.
2. **No Global Clock**: When programs need to cooperate, they coordinate their actions by exchanging messages. Close coordination often depends on a shared idea of the time at which the programs' actions occur. But it turns out that there are limits to the accuracy with which the computers in a network can synchronize their clocks – there is no single global notion of the correct time.
3. **Independent Failures**: Each component of the system can fail independently, leaving the others still running. And it is the responsibility of system designers to plan for the consequences of possible failures.
4. **Heterogeneity**: All the computers in a distributed system aren't similar, they may have different architectures, different OS etc.

Cluster

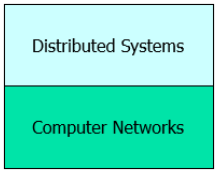
A type of parallel or distributed processing system, which consists of a collection of interconnected stand-alone computers cooperatively working together as a single, integrated computing resource.

Cloud

A type of parallel and distributed system consisting of a collection of interconnected and virtualised computers that are dynamically provisioned and presented as one or more unified computing resources based on service-level agreements established through negotiation between the service provider and consumers.

Network vs. Distributed Systems

- **Networks**: A media for interconnecting local and wide area computers and exchange messages based on protocols. Network entities are visible, and they are explicitly addressed (IP address). Networks focuses on packets, routing, etc., whereas distributed systems focus on applications.
- **Distributed System**: existence of multiple autonomous computers is transparent. Every distributed system relies on services provided by a computer network.



Reasons for Distributed Systems

- **Functional Separation**: Existence of Computers with different capabilities and purposes. These can be shared between computers using distributed systems e.g. printers etc.
- **Inherent Distribution**:
    - **Information**: Different information is stored and maintained by different computer and needs to be accessed by others
    - **People**: Computer Supported collaborative work
    - **Retail**: Inventory chains for companies
- **Power Imbalance and Load Variation**: Computational load is shared between computers
- **Reliability**: Data is backed up at different locations
- **Economies**: Shared resources reduce the overall cost of systems

Characteristics of Distributed Systems

- **Parallel Activities**: Autonomous components executing numerous tasks at the same time
- **Communication via Message Passing**: The components don't have a shared memory, they communicate via message passing
- **Resource Sharing**: Hardware and software resources can be shared between computers
- **No Global State**: No single process can have knowledge of the current global state of the system
- **No Global Clock**: Distributed systems have no global clock, only limited precession for processes to synchronize their clocks

Goals of Distributed Systems

- **Connecting Users and resources**: Providing users with access to shared resources
- **Transparency**: The users are not aware of the spatial separation between the distributed system components, they view it as a one whole.
- **Openness**: The system is open to different types of hardware and software resources and platforms.
- **Scalability**: Distributed systems can be scaled to incorporate more resources as per need.
- **Enhanced Availability**: In case of failure, users are provided with alternate resources (managing failures)

Challenges of Distributed Systems

- **Heterogeneity**: The underlying hardware and software of different computers in the system can be different. These components should be able to interoperate.
- **Distribution Transparency**: The users should not face any difference in server due to spatial difference between system components, they should be able to view it as one whole.
- **Concurrency**: Multiple users may want to access/ edit the same information at the same time
- **Fault Tolerance**: In case of failure, users are provided with alternate resources. Failure of a component (partial failure) should not result in failure of the whole system
- **Security**: Secure information from security breaches, intruders and viruses

- **Scalability**: The Distributed systems should be able to incorporate more resources as per need. And should be able to deal with large number of users.
- **Reuse/ Openness/ Interoperability**: The system should be able to incorporate different types of underlying architectures and operating systems. Openness in systems can be ensured by implementing protocols like W3C, ISO, IEEE, OMG, IETF etc.

Types of Heterogeneity

- Operating systems
- Hardware architectures
- Communication architectures
- Programming languages
- Software interfaces
- Security measures
- Information representation

Types of Distribution Transparency

- **Access Transparency**: Access from local or remote resources should be identical
- **Location Transparency**: Access without the knowledge of location
- **Failure Transparency**: Tasks can be completed despite failures
- **Replication Transparency**: Access to replicated resources as if there was just one. (Replicated resources enhance reliability)
- **Migration Transparency**: Allow the movement of resources and clients within a system without affecting the operation of users or applications.
- **Concurrency Transparency**: A process should not notice that there are other sharing the same resources
- **Performance Transparency**: Allows the system to be reconfigured to improve performance as loads vary
- **Scaling Transparency**: Allows the system and applications to expand in scale without changes in the system structure or the application algorithms
- **Application Level Transparencies**
    - **Persistence transparency**: Masks the deactivation and reactivation of an object
    - **Transaction transparency**: Hides the coordination required to satisfy the transactional properties of operations

Fault Tolerance

- **Failure**: an offered service no longer complies with its specification (e.g., no longer available or very slow to be usable)
- **Fault**: cause of a failure (e.g. crash of a component)
- **Fault tolerance**: no failure despite faults i.e., programmed to handle failures and hides them from users.

Fault Tolerance Mechanisms

- **Fault Detection**: Multiple mechanisms are implemented at software, communication and hardware level to detect faults and failures e.g. heartbeat, checksum etc.
- **Fault Masking**: There are multiple approaches to mask failures e.g. retransmitting data, redundancy etc.
- **Fault Toleration**: We implement mechanisms to handle/ capture failures and provide a graceful response in such cases e.g. Exception handling, timeouts etc.

- **Fault Recovery**: These are mechanisms used to recover from faults like Rollback mechanisms etc.

Challenges of Scalability

- **Cost of physical resources**: Cost should linearly increase with system size
- **Performance Loss**: For example, in hierarchically structure data, search performance loss due to data growth should not be beyond O(log n), where n is the size of data
- **Preventing software resources running out**: Numbers used to represent Internet addresses (32 bit->64bit), Y2K-like problems
- **Avoiding performance bottlenecks**: Use of decentralized algorithms (centralized DNS to decentralized)

Challenges of Concurrency

- **Fair scheduling**: Every user should get a chunk of the requested resources
- **Preserve dependencies** (e.g. distributed transactions -- buy a book using Credit card, make sure user has sufficient funds prior to finalizing order): Preserving resources before the actual activity to ensure the resources are available when needed
- **Avoid deadlocks**: The users end up in the deadlock since both are holding one resource and requesting the resource held by the other user to complete operation.

Challenges of Security

- **Confidentiality**: Protection against disclosure to unauthorized individual information
- **Integrity**: Protection against alteration or corruption
- **Availability**: Protection against interference targeting access to the resources e.g. DoS attacks
- **Non-Repudiation**: Proof of sending / receiving information e.g. digital signatures

Security Mechanisms

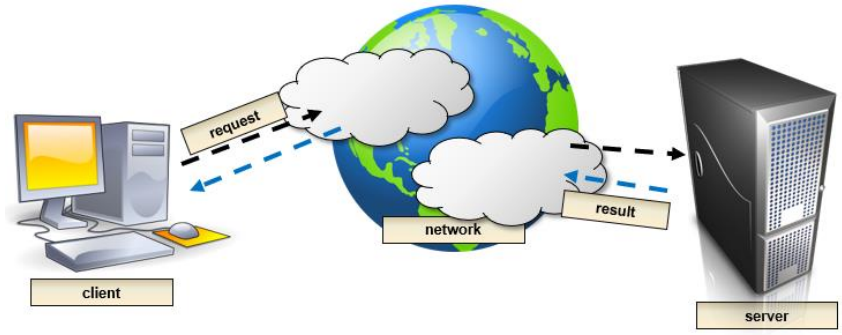
- **Encryption**: The data is encrypted using a key at the sender's end, the receiver has the key and decrypts the data upon reception
- **Authentication**: The access to data is protected using password etc.
- **Authorization**: Only certain users are authorized to access the data/ information.

Section # 2 Inter-Process Communication: Network Programming using TCP Java Sockets

Internet is growing rapidly and so are the opportunities presented by it. This means that the need for communication is growing as well. This can be achieved by using Sockets.

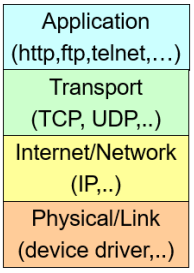
Elements of Client Server Communication

- **Client**: Any user that wishes to use a resource contained by another user or server. A network can have multiple clients.
- **Server**: The main program/ computer that holds the resources that the clients wish to access.
- **Underlying Network**: The communication between the client and server happens over a network that connects them.
- **Communication Protocols**: Rules that facilitate communication between the client and server over a network



Network Layers

- **Physical/ Link Layer**: Functionalities for transmission of signals representing a stream of data from one computer to another
- **Internet/ Network Layer**: IP (Internet Protocols) – a packet of data to be addressed to a remote computer and delivered
- **Transport Layer**: Functionalities for delivering data packets to a specific process on a remote computer
    - TCP (Transmission Control Protocol) – Connection Oriented protocol
    - UDP (User Datagram Protocol) – Connectionless Protocol
    - Programming Interface:
        - Sockets
- **Applications Layer**: Message exchange between standard or user applications



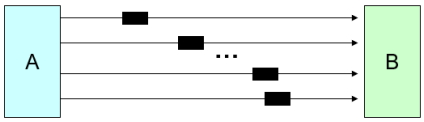
Transmission Control Protocol (TCP)

- Connection Oriented Protocol
- Provides Reliable Communication between two computers
- TCP messages have a fixed arrival order (first sent, first received)

User Datagram Protocol (UDP)

- Connectionless Protocol
- Sends independent packets of data called **Datagrams** from one computer to another
- There is no guarantee that the datagrams sent from one computer reached another computer
- Each datagram has the address of the sender and the receiver
- The datagrams can have variable arrival order

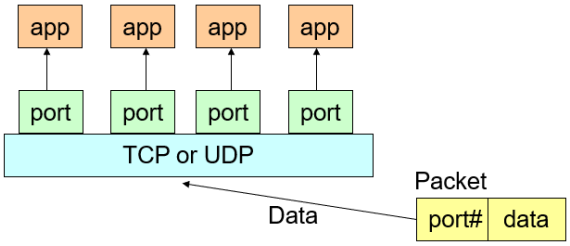
Connection-Oriented Communication



Connectionless Communication

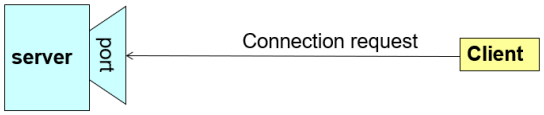
Ports

- Ports are receiving protocols (receiving channels), these are used to map incoming data to a process running on the computer. Each application is bind to a specific port and uses that port to listen for incoming data.
- Ports in a computer are represented by a positive integer.
- Some ports are reserved and cannot be used by any other services
- User level processes/ services generally used the port number 1024

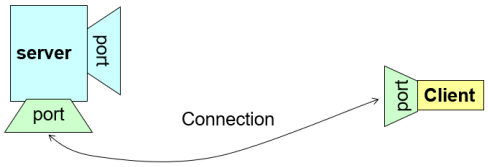


Socket

- A *socket* is one endpoint of a two-way communication link between two programs running on the network. A socket is bound to a port number so that the TCP layer can identify the application that data is destined to be sent to.
- Sockets provide an interface for programming networks at the transport layer
- Socket-based communication is programming language independent
- A server (program) runs on a specific computer and has a socket that is bound to a specific port. The server waits and listens to the socket for a client to make a connection request.



- If everything goes well, the server accepts the connection. Upon acceptance, the server gets a new socket bounds to a different port. It needs a new socket (consequently a different port number) so that it can continue to listen to the original socket for connection requests while serving the connected client.

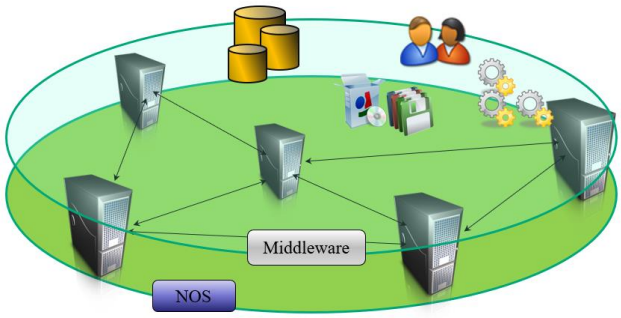


- Java's .net package provides two classes to implement Sockets:
    - Socket – for implementing a client
    - ServerSocket – for implementing a server

Section # 3 OS Support for Building Distributed Applications: Multithreaded Programming using Java Threads

Middleware

- **Middleware** is computer software that provides services to software applications beyond those available from the operating system.
- Middleware is computer software that connects software components or applications. The software consists of a set of services that allows multiple processes running on one or more machines to interact.
- Middleware layer lies between the NOS (Network Operating System and the user Applications)



Operating Systems

- An operating system (OS) is system software that manages computer hardware and software resources and provides common services for computer programs. All computer programs, excluding firmware, require an operating system to function.
- Time-sharing operating systems schedule tasks for efficient use of the system and may also include accounting software for cost allocation of processor time, mass storage, printing, and other resources.

Network Operating System

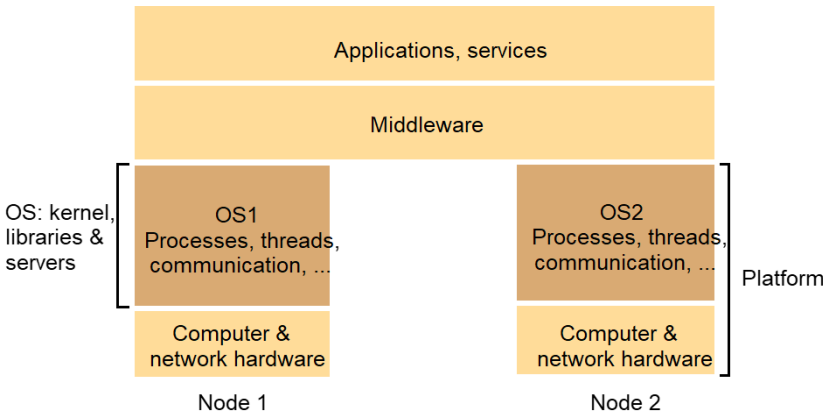
An operating system oriented to computer networking, to allow shared file and printer access among multiple computers in a network, to enable the sharing of data, users, groups, security, applications, and other networking functions, typically over a local area network (LAN), or private network. **Network Operating Systems are loosely coupled whereas, Distributed Operating Systems are tightly coupled.**

Features of Operating Systems

- **Encapsulation**: They should provide a useful service interface to their resources – that is, a set of operations that meet their clients' needs. Details such as management of memory and devices used to implement resources should be hidden from clients.
- **Protection**: Resources require protection from illegitimate accesses – for example, files are protected from being read by users without read permissions, and device registers are protected from application processes.
- **Concurrent processing**: Clients may share resources and access them concurrently. Resource managers are responsible for achieving concurrency transparency.

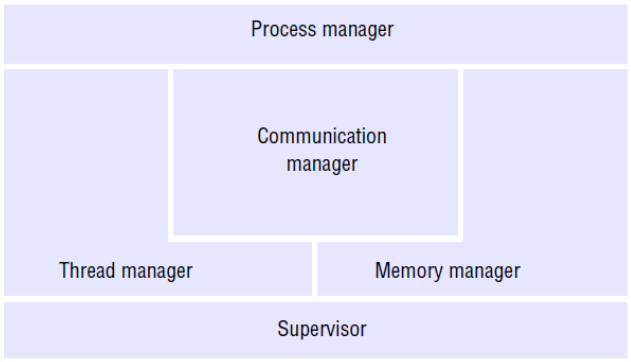
Building a Distributed System

- Middleware
  - High-level features for DS
    - Communication
    - Management
    - Application specific
  - Uniform layer where to build DS services
  - Runtime environment of applications
- Operating System
  - Low / medium level (core) features
    - Process / threads management
    - Local hardware (CPU, disk, memory)
    - Security (users, groups, domain, ACLs)
    - Basic networking



The core OS Components

- **Process manager**: Creation of and operations upon processes. A process is a unit of resource management, including an address space and one or more threads.
- **Thread manager**: Thread creation, synchronization and scheduling. Threads are schedulable activities attached to processes.
- **Communication manager**: Communication between threads attached to different processes on the same computer. Some kernels also support communication between threads in remote processes. Other kernels have no notion of other computers built into them, and an additional service is required for external communication.
- **Memory manager**: Management of physical and virtual memory.
- **Supervisor**: Dispatching of interrupts, system call traps and other exceptions; control of memory management unit and hardware caches; processor and floating-point unit register manipulations. This is known as the Hardware Abstraction Layer in Windows.



Threaded Applications

Modern Applications & Systems

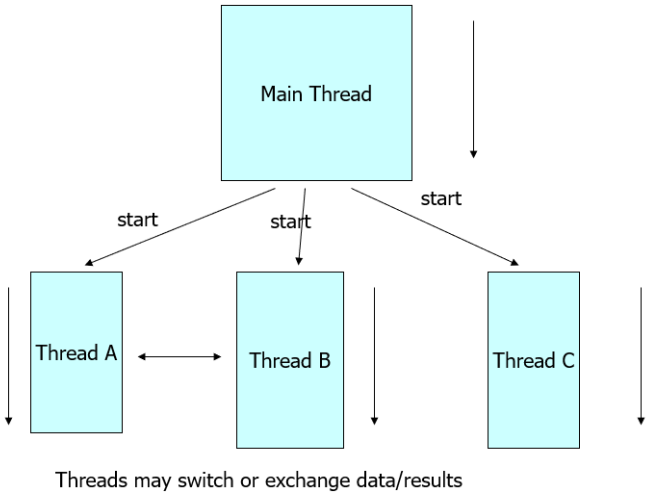
- **Operating System Level**
  - **Multitasking**: multiple independent applications running at once. Multiple processes running simultaneously on a machine.
- **Application Level**
  - **Multithreading**: multiple operations performed at the same time. Same application can perform multiple tasks at once.

Threads

- Threads are light-weight processes within a process
- A Thread is a piece of code that runs in concurrent with other threads.
- Each thread is a statically ordered sequence of instructions.
- Threads are used to express concurrency on both single and multiprocessors machines.
- A thread of execution is the smallest sequence of programmed instructions that can be managed independently by a scheduler, which is typically a part of the operating system.
- Multiple threads can exist within one process, executing concurrently and sharing resources such as memory, while different processes do not share these resources.
- Threads can also be termed as the flow of control in a program.
- Threads share a common address space
- **Single Threaded Programs**: Programs with only one flow of control

Multithreading

- Multithreading is the ability of a central processing unit (CPU) or a single core in a multi-core processor to execute multiple processes or threads concurrently, appropriately supported by the operating system.

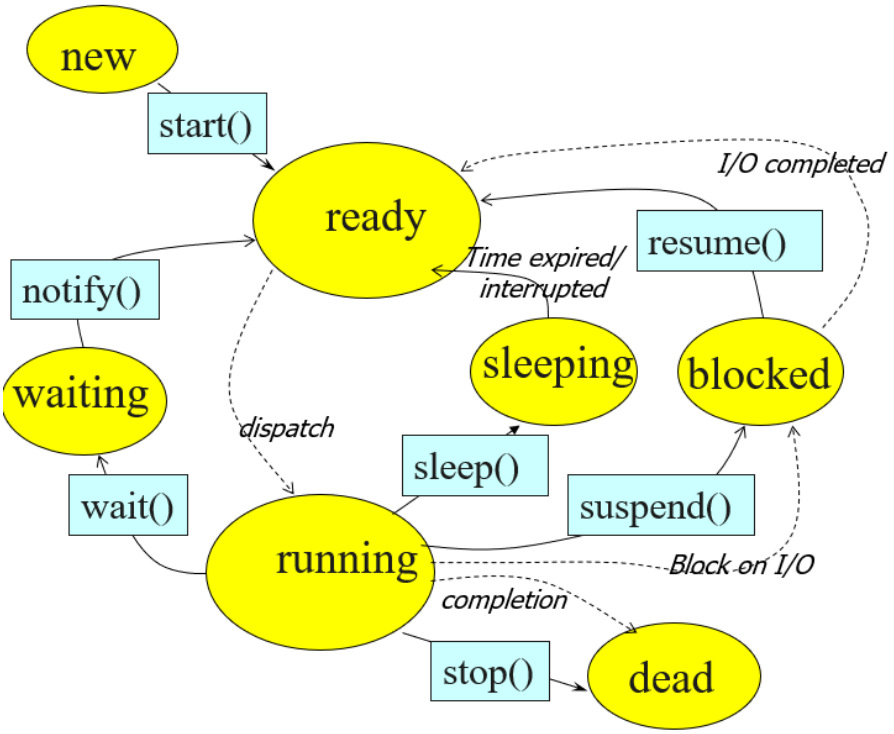


- Most Modern servers are multithreaded – they are able to serve multiple clients concurrently.

Applications of Threads

- Parallelism and concurrent execution of independent tasks / operations.
- Implementation of reactive user interfaces.
- Non-blocking I/O operations.
- Asynchronous behavior.
- Timer and alarms implementation.

Life Cycle of a Thread



Thread Priority

- In Java, each thread is assigned priority, which affects the order in which it is scheduled for running.

- The threads so far had same default priority (NORM_PRIORITY) and they are served using FCFS policy.
- Java allows users to change priority using “setPriority” method

Accessing Shared Resources

- Applications access to shared resources need to be coordinated. If one thread tries to read the data and other thread tries to update the same data, it leads to inconsistent state.
- This can be prevented by synchronising access to the data.
- This can be done by using “synchronized” method in Java

Multithreaded Servers Architectures

- **Worker pool**: In worker-pool architectures, the server creates **a fixed pool of worker threads** to process requests. The module “receipt and queuing” receives requests from sockets/ports and places them on a shared request queue for retrieval by the workers.
- **Thread-per-request**: IO Thread creates a new worker thread for each request and worker thread destroys itself after serving the request.
- **Thread-per-connection**: Server associates a Thread with each connection and destroys when client closes the connection. Client may make many requests over the connection.
- **Thread-per-object**: Associates Thread with each object. An IO thread receives request and queues them for workers, but this time there is a **per-object queue**.

Note – all thread operations are blocking except thread.start()

Section # 4 Distributed System Models

Functional Goal

The functional goal of a distributed system is that it can function correctly under all circumstances.

Structural Goal

- To cover the widest possible range of circumstances.
- To cope with possible difficulties and threats.
- To meet the current and possibly the future demands.

Goals of Distributed System Models

- classifying and understanding different implementations
- identifying their weaknesses and their strengths
- crafting new systems outs of pre-validated building blocks

Challenges of Distributed Systems

- Performance Issues
    - Responsiveness
        - Support interactive clients
            - Use caching and replication
    - Throughput
    - Load balancing and timeliness
- Quality of Service:
    - Reliability
    - Security
    - Adaptive performance.
- Dependability issues:
    - Correctness, security, and fault tolerance
    - Dependable applications continue to work in the presence of faults in hardware, software, and networks.

Architectural Models

An Architectural model of a distributed system is concerned with the placement of its parts and relationship between them. Architectural models provide us with the following point of views

- **A pragmatic starting point** – In terms of implementation models and basic blocks
- **A conceptual view** – In terms of logical view of the system, interaction flow, and components

Types of Architectural Models

- Client-Server Architecture
- Peer-to-peer Architecture

Goals of Architectural Models

- Simplifies and abstracts the functions of individual components
    - An initial simplification is achieved by classifying processes as:

        - Server processes
        - Client processes
        - Peer processes
            - Cooperate and communicate in a symmetric manner to perform a task.
- The placement of the components across a network of computers – define patterns for the distribution of data and workloads
- The interrelationship between the components – i.e., functional roles and the patterns of communication between them.

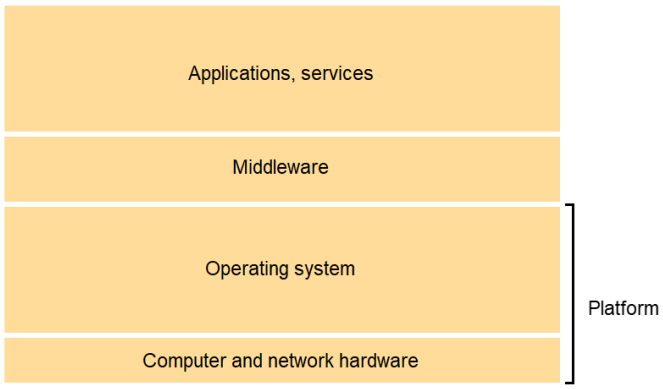
Software Architecture and Layers

Software architecture refers to the high-level structures of a software system, the discipline of creating such structures, and the documentation of these structures. These structures are needed to reason about the software system.

- Originally to the structure of software as *layers* or modules in a single computer.
- More recently in terms of *services* offered and requested between processes in the same or different computers
    - **Layer**: a group of related functional components
    - **Service**: functionality provided to the next layer.

Software and Hardware Service Layers in Distributed Systems

- **Platform**
    - The lowest hardware and software layers are often referred to as a platform for distributed systems and applications.
    - These low-level layers provide services to the layers above them, which are implemented independently in each computer.
    - Platform is the hardware and Operating system e.g. Windows/Intel x86, Mac OS etc.
- **Middleware**
    - A layer of software whose purpose is to mask heterogeneity present in distributed systems and to provide a convenient programming model to application developers e.g. Microsoft .NET, Microsoft Azure etc.



System Architecture

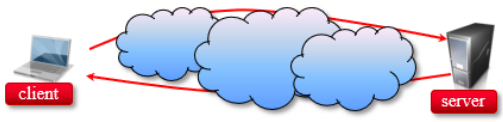
The most evident aspect of DS design is the division of responsibilities between system components (applications, servers, and other processes) and the placement of the components on computers in the network.

It has major implication for:

- Performance
- Reliability
- Security

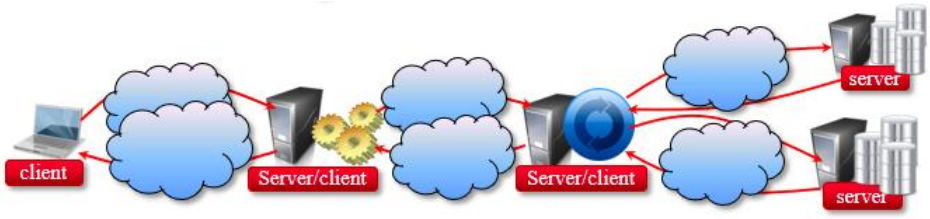
Client Server Architecture Types

- Two-tier model (classic)

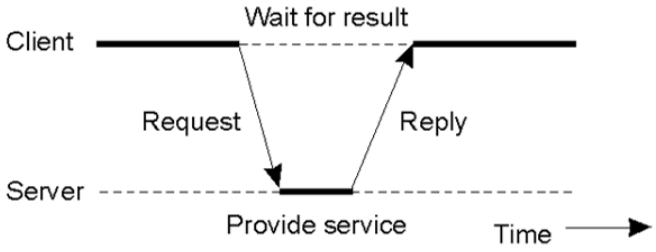


- Three-tier (when the server, becomes a client)

- Multi-tier (cascade model)



Interaction between Client and Server



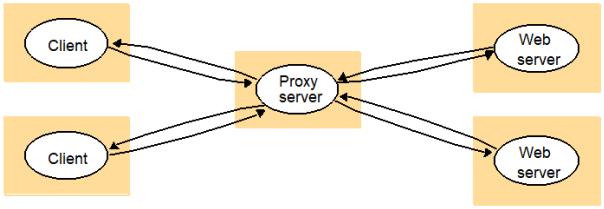
Challenges for Architectural Models

- **Widely varying models of use** – High variation of workload, partial disconnection of components, or poor connection.
    - **Solution** – The structure and the organization of systems allow for distribution of workloads, redundant services, and high availability.
- **Wide range of system environments –** Heterogeneous hardware, operating systems, network, and performance.
    - **Solution** – A flexible and modular structure allows for implementing different solutions for different hardware, OS, and networks.
- **Internal problems –** Non-synchronized clocks, conflicting updates, various hardware and software failures.
    - **Solution** – The relationship between components and the patterns of interaction can resolve concurrency issues, while structure and organization of component can support failover mechanisms.
- **External threats –** Attacks on data integrity, secrecy, and denial of service.

- **Solution** – Security must be built into the infrastructure and it is fundamental for shaping the relationship between components.

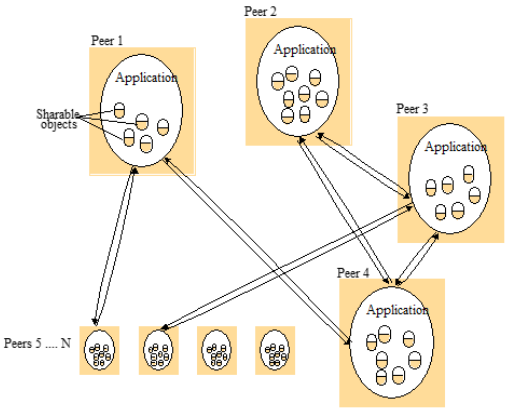
Proxy Servers & Cache

- Proxy Servers offer replication Transparency
- A cache is a store of recently used data



Peer to Peer Architectures

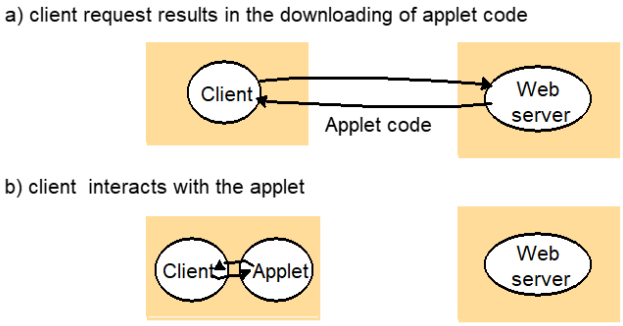
- Peer-to-peer architecture (P2P architecture) is a commonly used computer networking architecture in which each workstation, or node, has the same capabilities and responsibilities.
- All the processes play similar roles, interacting cooperatively as peers to perform distributed activities or computations without distinction between clients and servers.



- P2P architectures can
- **Pure** – where there is no centralized server and each peer acts as a server as well as a client (like in the picture above)
- **Hybrid** – The architecture has a centralized server which controls certain aspects e.g. Indexing, of the architecture, but the peers still act as servers and clients simultaneously

Variant of Client Server Model

Mobile Code and Web Applets



- Applets downloaded to clients give good interactive response
- Mobile codes such as Applets are potential security threat, so the browser gives applets limited access to local resources (e.g. NO access to local/user file system).

Mobile Agents

- A running program (code and data) that travels from one computer to another in a network carrying out an autonomous task, usually on behalf of some other process
    - advantages: flexibility, savings in communications cost
    - virtual markets, software maintain on the computers within an organization.
- Potential security threat to the resources in computers they visit. The environment receiving agent should decide which of the local resource to allow. (e.g., crawlers and web servers).
- Agents themselves can be vulnerable – they may not be able to complete task if they are refused access.

Thin Clients

- Thin clients are software agents
- **Network computer**: download OS and applications from the network and run on a desktop (solve up-gradation problem) at runtime.
- **Thin clients**: Windows-based UI on the user machine and application execution on a remote computer. E.g., X-11 system.
- A thin client is a lightweight computer that is purpose-built for remoting into a server (typically cloud or desktop virtualization environments). It depends heavily on another computer (its server) to fulfil its computational roles.

Physical Models

- A representation of the underlying H/W elements of a DS that abstracts away specific details of the computer/networking technologies.
- Baseline physical model – a small set of nodes.
- Three Generations of DSs (Distributed Systems):
    - **Early DSs [70-80s]**: LAN-based, 10-100 nodes

- **Internet-scale DSs [early 90-2005]**: Clusters, grids, P2P (with autonomous nodes)
- **Contemporary DSs**: dynamic nodes in mobile systems that offer location-aware services, Clouds with resource pools offering services on pay-as-you-go basis.

Fundamental Models

Fundamental Models are concerned with a formal description of the properties that are common in all the architectural models

Types of Fundamental Models

- **Interaction Model** – deals with performance and the difficulty of setting of time limits in a distributed system.
- **Failure Model** – specification of the faults that can be exhibited by processes
- **Security Model** – discusses possible threats to processes and communication channels.

Interaction Model

- Computation occurs within processes;
- The processes interact by passing messages, resulting in:
  - Communication (information flow)
  - Coordination (synchronization and ordering of activities) between processes.
- Two significant factors affecting interacting processes in a distributed system are:
  - Communication performance is often a limiting characteristic.
  - It is impossible to maintain a single global notion of time.

Performance of Communication Channel

The communication channel in an interaction model can be implemented in a variety of ways in

- Streams
- Simple message passing over a network etc.

Communication over a computer network has the following performance characteristics:

- **Latency:** A delay between the start of a message's transmission from one process to the beginning of reception by another.
- **Bandwidth:** the total amount of information that can be transmitted over in a given time. Communication channels using the same network, must share the available bandwidth.
- **Jitter:** The variation in the time taken to deliver a series of messages. It is very relevant to multimedia data.

Computer Clocks and Timing Events

- Each computer in a DS has its own internal clock, which can be used by local processes to obtain the value of the current time.
- Therefore, two processes running on different computers can associate timestamp with their events.
- However, even if two processes read their clocks at the same time, their local clocks may supply different time.
  - This is because computer clock drifts from perfect time and their drift rates differ from one another.
- Even if the clocks on all the computers in a DS are set to the same time initially, their clocks would eventually vary quite significantly unless corrections are applied.

- There are several techniques to correct time on computer clocks. For example, computers may use radio receivers to get readings from GPS (Global Positioning System) with an accuracy about 1 microsecond.

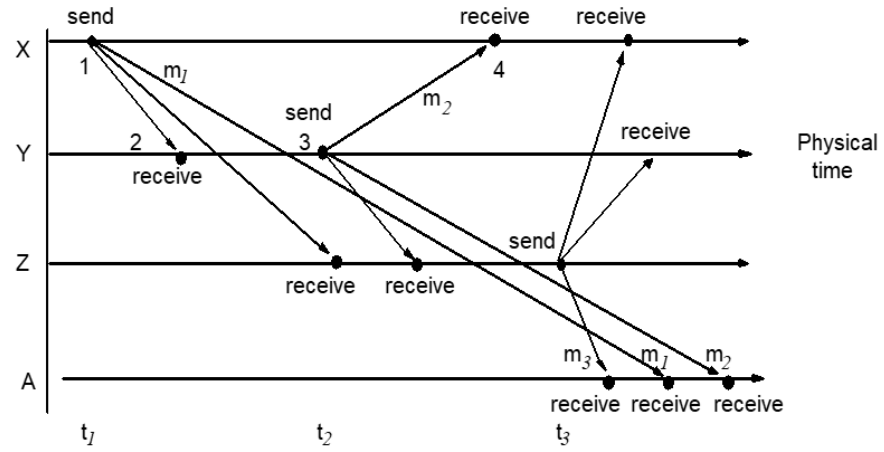
Variants of Interaction Model

- **Synchronous DS** – hard to achieve:
  - The time taken to execute a step of a process has known lower and upper bounds.
  - Each message transmitted over a channel is received within a known bounded time.
  - Each process has a local clock whose drift rate from real time has known bound.
  - A type of two-way communication with virtually no time delay, allowing participants to respond in real time.
  - Regularly occurring events in timed intervals are kept in step using some form of electronic clocking mechanism.
  - Synchronous distributed systems are hard to achieve (especially over the internet) do to their unpredictable nature
- **Asynchronous DS –** There is NO bounds on:
  - Process execution speeds
  - Message transmission delays
  - Clock drift rates.
  - A type of two-way communication that occurs with a time delay, allowing participants to respond at their own convenience.

Event Ordering

In many DS applications we are interested in knowing whether an event occurred before, after, or concurrently with another event at other processes.

- The execution of a system can be described in terms of events and their ordering despite the lack of accurate clocks.
- Due to independent delivery in message delivery, message may be delivered in different order.



Failure Model

In a DS, both processes and communication channels may fail

Types of failures:

- Omission Failure

- Arbitrary Failure
- Timing Failure

Omission Failure

Communication channel produces an omission failure if it does not transport a message from outgoing message buffer of the sender process to incoming message buffer of the receiver process. This is known as "dropping messages" and is generally caused by a lack of buffer space at the receiver or at gateway or by a network transmission error.

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,* but the message is not put in its outgoing message buffer.
Receive-omission	Process	A message is put in a process's incoming message buffer, but that process does not receive it.
Arbitrary (Byzantine)	Process or channel	Process/channel exhibits arbitrary behaviour: it may send/transmit arbitrary messages at arbitrary times, commit omissions; a process may stop or take an incorrect step.

Timing Failure

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.

Masking Failures

- It is possible to construct reliable services from components that exhibit failures.
    - For example, multiple servers that hold replicas of data can continue to provide a service when one of them crashes.
- A knowledge of failure characteristics of a component can enable a new service to be designed to mask the failure of the components on which it depends:
    - Checksums are used to mask corrupted messages.

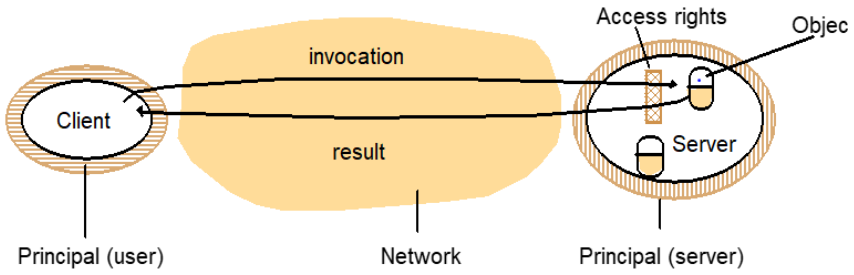
Security Model

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

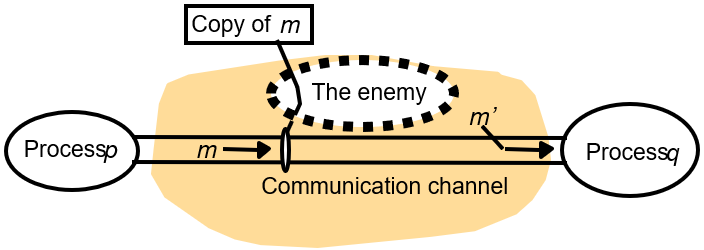
Protecting Objects

- Use "access rights" that define who is allowed to perform operation on a object.

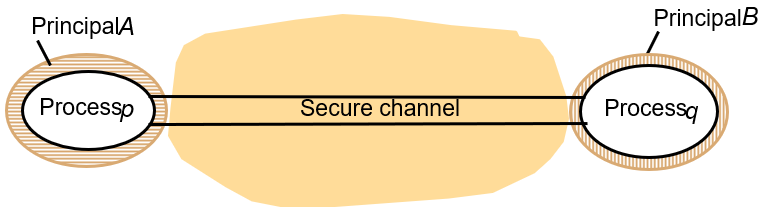
- The server should verify the identity of the principal (user) behind each operation and checking that they have sufficient access rights to perform the requested operation on the particular object, rejecting those who do not.



  - To model security threats, we postulate an enemy that is capable of sending any process or reading/copying message between a pair of processes
  - Threats form a potential enemy: threats to processes, threats to communication channels, and denial of service.



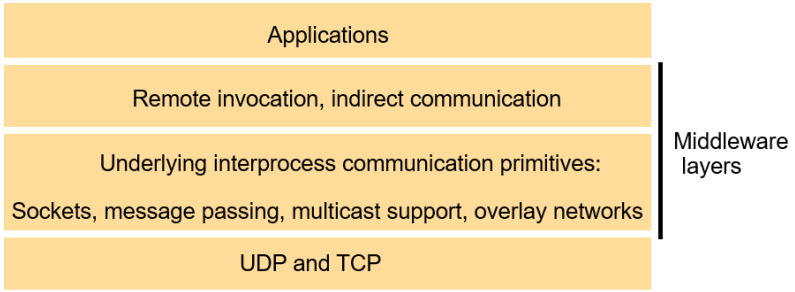
  - **Encryption** and **authentication** are used to build secure channels.
  - Each of the processes knows the identity of the principal on whose behalf the other process is executing and can check their access rights before performing an operation.



Section # 5 Remote Invocation

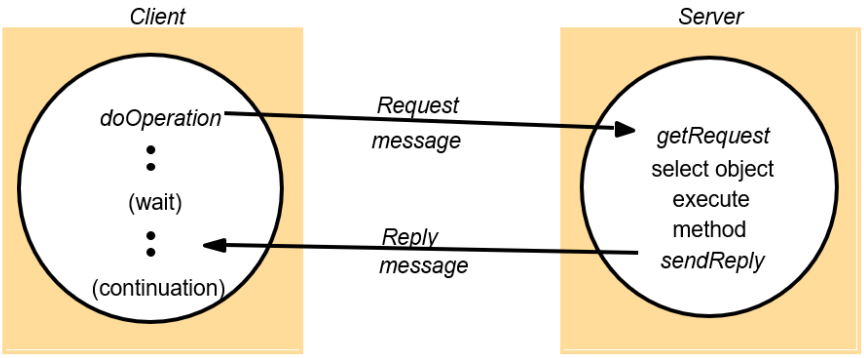
High Level Programming Models for Distributed Systems

- **Remote Procedure Call** - an extension of the conventional procedure call model
- **Remote Method Invocation** - an extension of the object-oriented programming model



Request Reply Protocol

- It is the protocol used for communication between a client and a server
- Exchange protocol for the implementation of remote invocation in a distributed system
- We discuss the protocol based on three abstract operations:
    - doOperation
    - getRequest
    - sendReply



doOperation

- Sends a request message to the remote server and returns the reply
- The arguments specify the remote server, the operation to be invoked and the arguments of that operation
- Takes 3 arguments
    - RemoteRef s – can be a data structure or an object, it contains the IP address and port number of the server. May also have remote object reference
    - int operationId – helps server identify the operation it has to execute
    - byte[] arguments – contains the input arguments

getRequest

Acquires a client request via the server port

sendReply

- Sends the reply message reply to the client at its Internet address and port

- It contains three arguments
    - byte[] reply – the reply
    - InetAddress clientHost – client IP address
    - int clientPort – Client port number

Message Format in Request Reply Protocol

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

- The arguments are usually in JSON or XML format

JSON Argument Format

{

"TYPE" : "QUERY"

"ARG" : "SYSTEM"

}

Request Reply Failure Model

- If UDP model is used
    - Lost messages
    - Out of order service
- Protocol can crash

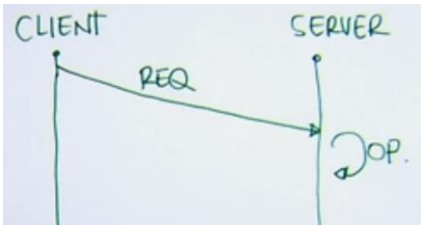
Request Reply Protocol Design Issues

- Fault tolerance measures we can consider include:
    - Timeouts
    - Discarding duplicate messages
    - Handling lost reply messages - strategy depends on whether the server operations are idemponent (an operation that can be performed repeatedly)
    - History - if servers have to send replies without re-execution, a history has to be maintained
- Three main design decisions related to implementations of the request/reply protocols are:
    - Strategy to retry request message
    - Mechanism to filter duplicates
    - Strategy for results retransmission

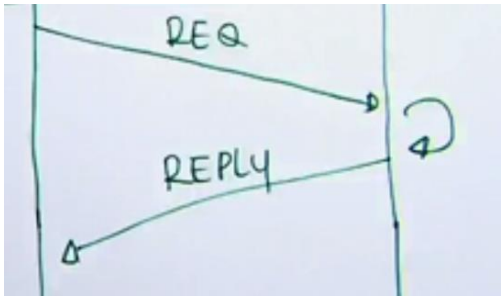
Request Reply Exchange Protocols

Three different types of protocols are typically used that address the design issues to a varying degree:

- The request (R) protocol



- The request-reply (RR) protocol



- The request-reply-acknowledge reply (RRA) protocol

Name	*Messages sent by*		
	Client	*Server*	*Client*
R	*Request*		
RR	*Request*	*Reply*	
RRA	*Request*	*Reply*	*Acknowledge reply*

Invocation Semantics

These are different types of guarantees provided by the middleware that is implementing remote invocation to the applications

- **Maybe**: The remote procedure call may be executed once or not at all. Unless the caller receives a result, it is unknown as to whether the remote procedure was called.
- **At-least-once**: Either the remote procedure was executed at least once, and the caller received a response, or the caller received an exception to indicate the remote procedure was not executed at all.
- **At-most-once**: The remote procedure call was either executed exactly once, in which case the caller received a response, or it was not executed at all and the caller receives an exception.

Java RMI supports at-most-one semantics.

Object Oriented Concepts

- **Objects**
  - Attributes + methods
  - Objects communicate with other objects by invoking methods, passing arguments and receiving results

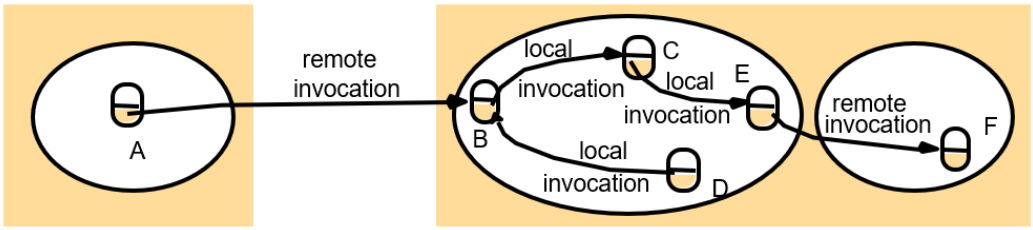
- **Object References**
  - Used to access objects
  - Object references can be assigned to variables, passed as arguments and returned as results
- **Interfaces**
  - Define the methods that are available to objects to invoke
  - Each method signature specifies the arguments and return values
- **Actions**
  - Objects performing a particular task on a target object. An action could result in:
    - The state of the object changed or queried
    - A new object created
    - Delegation of tasks to other objects
- **Exceptions**
  - Are thrown when an error occurs

Remote Object

- An object that can receive remote invocations
- Can also receive local invocations
- Can invoke methods in local objects as well as other remote objects

Remote Invocation – when an object from a process calls another object from another process.

Local Invocation – when an object from a process calls another object from the same process.



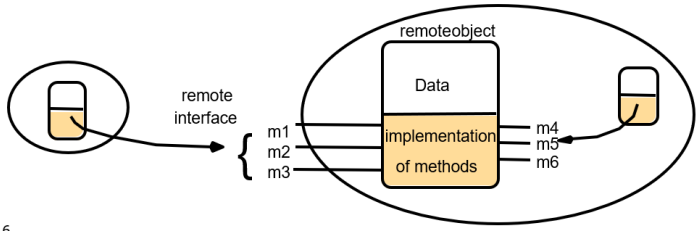
Remote Object Reference

- An identifier that can be used throughout a distributed system to refer to a unique remote object
- Other objects can invoke the methods of a remote object if they have access to its *remote object reference*

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

Remote Interface

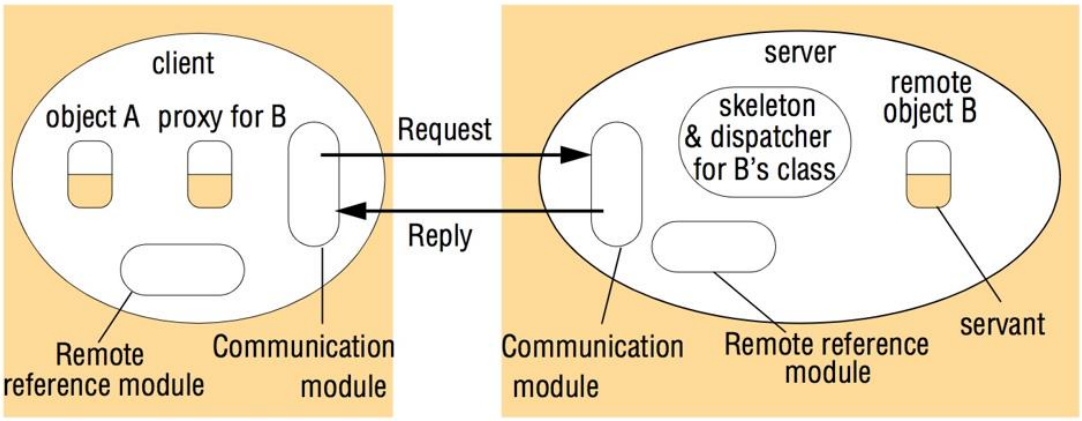
- Defines the methods that can be invoked by external processes.
- Remote objects implement the remote interface



Actions

- Actions can be performed on remote objects (objects in other processes)
- An action could be executing a remote method defined in the remote interface or creating a new object in the target process
- Actions are invoked using Remote Method Invocation (RMI)

Implementation of RMI



Communication Module

- The two cooperating communication modules carry out the request-reply protocol
- It uses three fields from the message:
  - Message type
  - Request ID
  - Remote object reference
- It is responsible for implementing the invocation semantics
- The server module queries the *remote reference module* to obtain the local reference of the object and passes the local reference to the *dispatcher* for the class

Remote Reference Module

- Creates remote object references
- Has a remote object table that records the correspondence between local object references in that process and remote object references (system-wide)
- The remote object table contains an entry for each:
  - Remote object reference held by the process
  - Local proxy
- Entries are added to the remote object table when:
  - A remote object reference is passed for the first time
  - When a remote object reference is received, and an entry is not present in the table

RMI Software

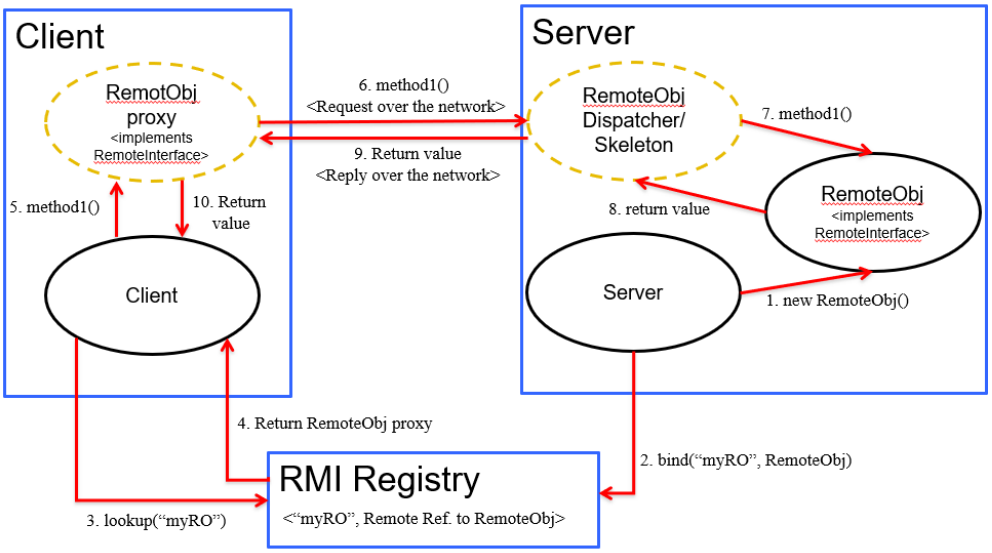
- Software layer that lies between the application and the communication and object reference modules
- **Proxy:** Plays the role of a local object to the invoking object. There is a proxy for each remote object which is responsible for:
    - Marshalling the reference of the target object, its own method id and the arguments and forwarding them to the communication module
    - Unmarshalling the results and forwarding them to the invoking object
- **Dispatcher:** There is one dispatcher for each remote object class. Is responsible for mapping to an appropriate method in the skeleton based on the method ID
- **Skeleton:** Is responsible for:
    - Unmarshalling the arguments in the request and forwarding them to the servant
    - Marshalling the results from the servant to be returned to the client

Implementation

- Remote Interface
    - Exposes the set of methods and properties available
    - Defines the contract between the client and the server
    - Constitutes the root for both stub and skeleton
- Servant component
    - Represents the remote object (skeleton)
    - Implements the remote interface
- Server component
    - Main driver that creates, initializes, and makes available the servant
    - Registers the servant with the *binder*
- Client component
    - Uses the *binder* to lookup for remote objects.
- Compile source and generate stubs
    - Client proxy/stub
    - Server dispatcher and skeleton

Java RMI

The binder in Java is called RMI Registry.



Transparency in RMI

- **Location**: We can access any object remotely as if it was present locally.
- **Access**: It should appear that the way in which we access remote resources is the same way we access local resources

However, full access transparency is not desired because

- Remote invocations are more vulnerable to failures
  - They involve a network, potentially another computer
  - There is always the chance that no result is received
  - Clients making remote invocations should be able to recover from this situation
- The latency of a remote invocation is much larger than that of a local one. Programs making remote invocations should be able to take this into account so that they can:
  - Minimize the number of remote invocations
  - Abort a remote invocation that is taking too long (timeout)

Java RMI Example

Specify Remote Interface

```
public interface IRemoteMath extends Remote {

        double add(double i, double j) throws RemoteException;

        double subtract(double i, double j) throws RemoteException;

}
```

Implement Servant Class

```
public class RemoteMathServant extends UnicastRemoteObject implements IRemoteMath {

   public double add ( double i, double j ) throws RemoteException {

      return (i+j);

   }

   public double subtract ( double i, double j ) throws RemoteException {
```

```
        return (i-j);
    }
}
```

Implement the Server

```
public class MathServer {
    public static void main(String args[]){
        System.setSecurityManager(new RMISecurityManager());
        try{
            IRemoteMath remoteMath = new RemoteMathServant();
            Registry registry = LocateRegistry.getRegistry();
            registry.bind("Compute", remoteMath );
                System.out.println("Math server ready");
        }catch(Exception e) {
            e.printStackTrace();
        }
    }
}
```

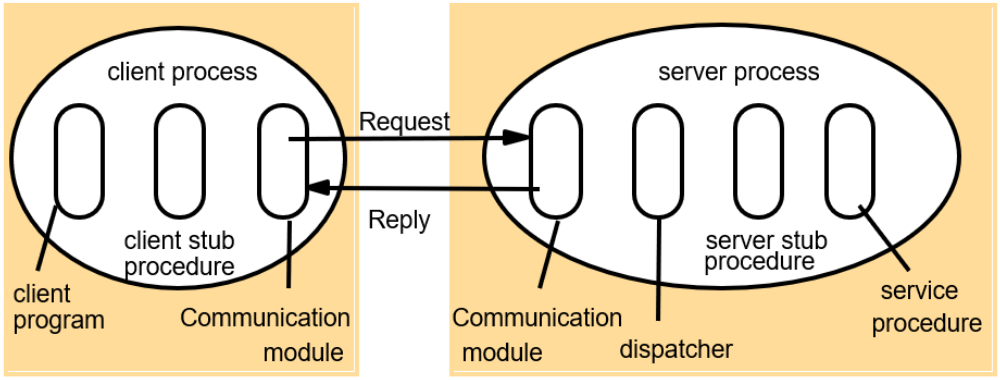
Implement the Client

```
public class MathClient {
    public static void main(String[] args) {
        try {
            if(System.getSecurityManager() == null)
                System.setSecurityManager( new RMISecurityManager() );
            LocateRegistry.getRegistry("localhost");
            IRemoteMath remoteMath = (IRemoteMath) registry.lookup("Compute");
            System.out.println( "1.7 + 2.8 = ” + math.add(1.7, 2.8) );
            System.out.println( "6.7 - 2.3 = ” + math.subtract(6.7, 2.3) );
        }
        catch( Exception e ) {
            System.out.println( e );
        }
    }
```

}

Remote Procedure Calls

RPCs enable clients to execute procedures in server processes based on a defined service interface.



- Communication Module
  - Implements the desired design choices in terms of retransmission of requests, dealing with duplicates and retransmission of results
- Client Stub Procedure
  - Behaves like a local procedure to the client. Marshals the procedure identifiers and arguments which is handed to the communication module
  - Unmarshalls the results in the reply
- Dispatcher
  - Selects the server stub based on the procedure identifier and forwards the request to the server stub
- Server stub procedure
  - Unmarshalls the arguments in the request message and forwards it to the service procedure
  - Marshalls the arguments in the result message and returns it to the client

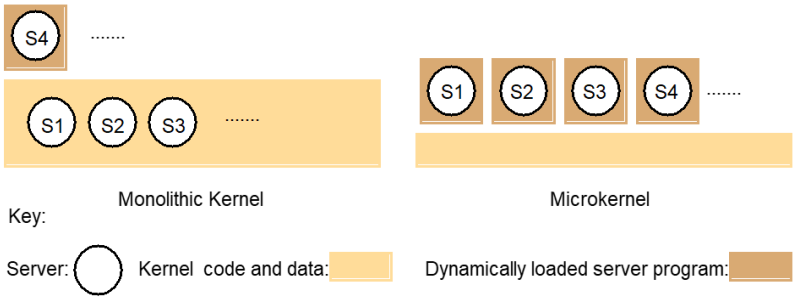
Section # 6 Operating System Architecture and Distributed Systems

Kernel Architectures

- Monolithic kernels
- Layered architecture-based kernels
- Micro-kernels – mostly suitable for distributed systems

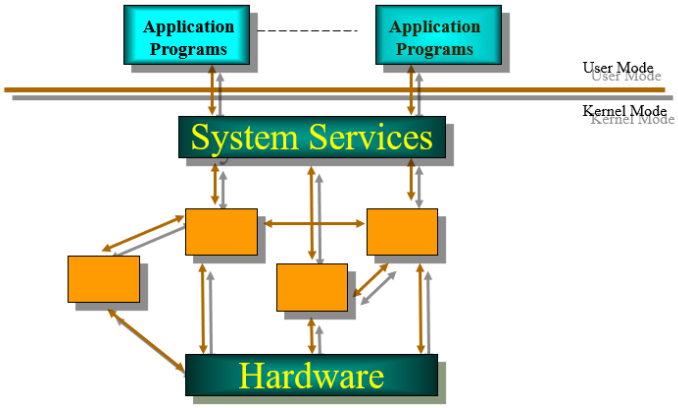
A key principles of DS are openness, simplicity, flexibility and performance. Monolithic and Layered architectures can be put under the category as monolithic.

The monolithic and microkernel architecture differ primarily in the decision as to what functionality belongs in the kernel and what is left to server processes that can be dynamically loaded to run on top of it.



- In Monolithic Kernel, most of the functionalities are part of the kernel mode. Whereas, in Microkernel, most activities are the part of user mode and the kernel has minimum functionalities.

Monolithic Operating Systems



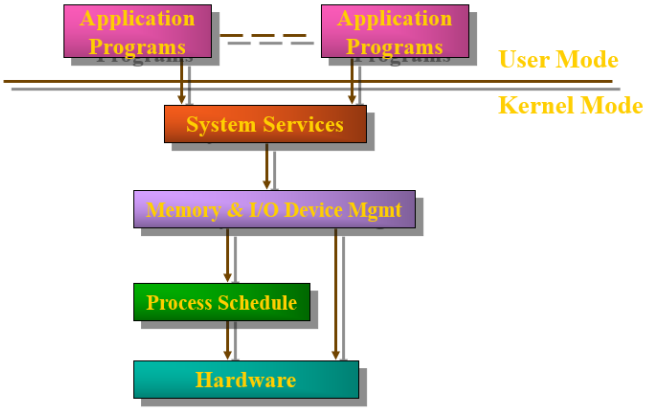
- Better application Performance
- Difficult to extend
- Kernel is very heavy
- E.g. MS DOS

Disadvantages of Monolithic OS

- **Massive**: It performs all basic OS functions and takes up in the order of megabytes of code and data

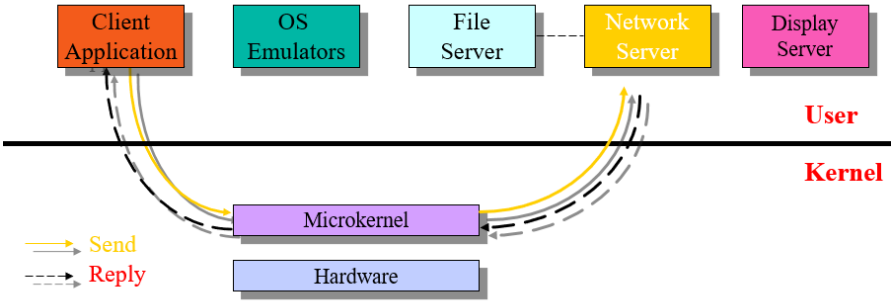
- **Undifferentiated**: It is coded in a non-modular way (traditionally) although modern ones are much more layered.
- **Intractable**: Altering any individual software component to adapt it to changing requirements is difficult.

Layered Operating System



- Easier to enhance due to layered nature
- Each layer of code access lower level interface
- Low-application performance due to layering since a request has to go through all the layers
- Kernel is very heavy
- E.g. Unix

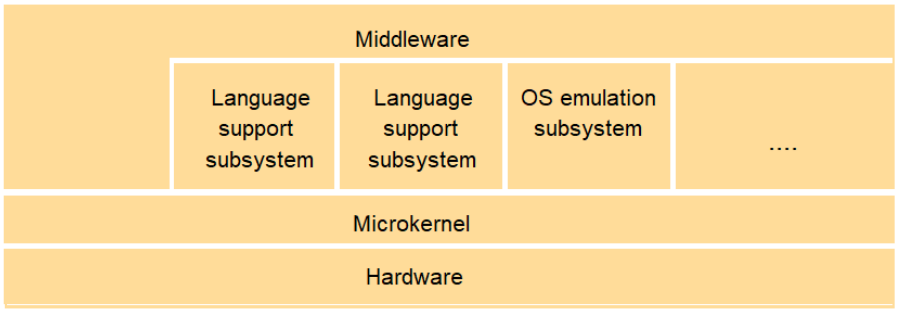
Microkernel Operating System



- Compared to monolithic, microkernel design provides only the most basic abstractions,
  - address space, threads and local IPC.
- All other system services are provided by servers that are dynamically loaded precisely on those computers in the DS that require them.
- Clients access these system services using the kernel's message-based invocation mechanisms.
- Tiny OS kernel providing basic primitive (process, memory, IPC)
- Traditional services become subsystems
- OS = Microkernel + User Subsystems
- E.g. Mach, PARAS, Chorus

The Role of Microkernel

MK appears as a layer between H/W and a layer of major system components (subsystems). If performance, rather than portability is goal, then middleware may use facilities of MK directly.



The microkernel supports middleware via subsystems

Monolithic vs. Microkernel Design

Advantages of a MK-based OS

- A relative small kernel is more likely to be free of bugs than one that is larger and complex.
- Extensibility and its ability to enforce modularity behind memory protection boundaries

Advantage of a monolithic OS

- Relative efficiency with which operations can be invoked is high because even invocation to a separate user-level address space on the same node is more costly.

Open DS and System Software

- An open DS should make it possible to:
    - Run only that system software at each computer that is necessary for its particular role in the system architecture. For example, system software needs of laptops and dedicated servers are different and loading redundant modules wastes memory resources.
    - Allow the software implementing any particular service to be changed independent of other facilities.
    - Allow for alternatives of the same services to be provided, when this is required to suit different users or applications.
    - Introduce new services without harming the integrity of existing ones.

Guiding Principle of OS Design

The separation of fixed resource management (RM) **mechanisms** from resource management **policies,** which vary from application to application and service to service.

Section # 7 Security

Object: Object in a security world is any resource

Principal: Principals can be users or processes

Attacks: Any attempt to sabotage the system

Enemy: Any external source trying to intrude in the system of alter it somehow

Threats

- To processes
- To channels
- Denial of Service

Properties

- Each process is sure of the identity of the other
- Data is private and protected against tampering
- Protection against repetition and reordering of data

Security Goal

Restrict access to information/resources to just to those entities that are authorized to access.

- Designers of secure distributed systems must cope with the exposed interfaces and insecure network in an environment where attackers are likely to have knowledge of the algorithms used to deploy computing resources.
- Cryptography provides the basis for the authentication of messages as well as their secrecy and integrity.

Security Attack Methods

Security attacks take various forms:

- Eavesdropping - A form of leakage
  - obtaining private or secret information or copies of messages without authority.
- Masquerading – A form of impersonating
  - assuming the identity of another user/principal – i.e, sending or receiving messages using the identity of another principal without their authority.
- Message tampering
  - altering the content of messages in transit
    - *man, in the middle attack (tampers with the secure channel mechanism)*
- Replaying
  - storing secure messages and sending them later
- Denial of service - Vandalism
  - flooding a channel or other resource, denying access to others
  - Deliberately excessive use of resources to the extent that they are not available to legitimate users
- Trojan horses and other viruses
  - Viruses can only enter computers when program code is imported.
    - But users often require new programs, for example:

        - *New software installation*
        - *Mobile code downloaded dynamically by existing software (e.g. Java applets)*
        - *Accidental execution of programs transmitted surreptitiously*
    - Defences: code authentication (signed code), code validation (type checking, proof), sandboxing.

Types of Security Threats

- **Leakage**: Acquisition of information by unauthorised recipients
- **Tampering**: Unauthorised alteration of information
- **Vandalism**: Interference with the proper operation of systems

Cryptography

Cryptographic concealment is based on:

- Confusion
- Diffusion

Designing Secure Distributed Systems

- Immense strides have been made in recent years in the development of cryptographic techniques and their applications, yet design of secure systems remains an inherently difficult task.
    - Aim: exclude *all* possible attacks and loop holes.
    - This looks like programmer aiming to *exclude* all bugs from his/her program.
- Security is about avoiding disasters and minimizing mishaps. When designing for security it is necessary to assume the worst.
- The design of security system is an exercise in balancing costs against threats:
    - A cost (computational and network usage) is incurred for their use.
    - Inappropriately specified security measures may exclude legitimate users from performing necessary actions.

Cryptography

- Cryptography is an art of encoding information in a format that only intended recipient can access.
- Cryptography can be used to provide a proof of authenticity of information in a manner analogous to the use of signature in conventional transactions.
- A cryptography key is a parameter used in an encryption algorithm in such a way that the encryption cannot be reversed without a knowledge of the key.

Classes of Cryptographic Algorithm

- **Shared Secret Keys:**
    - The sender and recipient share a knowledge of the key and it must not be revealed to anyone.
- **Public/Private Key Pair:**
    - The sender of a message uses a recipient's public key to encrypt the message.
    - The recipient uses a corresponding private key to decrypt the message.

Uses of Cryptography

- Secrecy
- Integrity
- Authentication
- Digital Signatures

Security Notations

K_A	Alice's secret key
K_B	Bob's secret key
K_{AB}	Secret key shared between Alice and Bob
K_{Apriv}	Alice's private key (known only to Alice)
K_{Apub}	Alice's public key (published by Alice for all to read)
$\{M\}_K$	Message M encrypted with key K
$[M]_K$	Message M signed with key K

Alice	First participant
Bob	Second participant
Carol	Participant in three- and four-party protocols
Dave	Participant in four-party protocols
Eve	Eavesdropper
Mallory	Malicious attacker
Sara	A server

Needham Schroeder Protocol

The term **Needham–Schroeder protocol** can refer to one of the two key transport protocols intended for use over an insecure network, both proposed by Roger Needham and Michael Schroeder. These are:

- The Needham–Schroeder Symmetric Key Protocol is based on a symmetric encryption algorithm. It forms the basis for the Kerberos protocol. This protocol aims to establish a session key between two parties on a network, typically to protect further communication.
- The Needham–Schroeder Public-Key Protocol, based on public-key cryptography. This protocol is intended to provide mutual authentication between two parties communicating on a network, but in its proposed form is insecure.

How Needham Schroeder Works

Bob is a file server; Sara is an authentication service. Sara shares secret key K_A with Alice and secret key K_B with Bob.

- Alice sends an (unencrypted) message to Sara stating her identity and requesting a *ticket* for access to Bob.
- Sara sends a response to Alice. $\{\{Ticket\}_{KB}, K_{AB}\}_{KA}$. It is encrypted in K_A and consists of a ticket (to be sent to Bob with each request for file access) encrypted in K_B and a new secret key K_{AB}.
- Alice uses K_A to decrypt the response.
- Alice sends Bob a request R to access a file: $\{Ticket\}_{KB}$, Alice, R.

- The ticket is actually $\{K_{AB}, \text{Alice}\}_{KB}$. Bob uses K_B to decrypt it, checks that Alice's name matches and then uses K_{AB} to encrypt responses to Alice.

Limitations of Needham Schroeder

- It depends upon prior knowledge by the authentication server Sara of Alice's and Bob's keys. This is feasible in a single organization where Sara run a physically secure computer and is managed by a trusted principal.
    - Not suitable in E-commerce or other wide area applications.
- Usefulness of challenges: They introduced the concept of a *cryptographic challenge*. That means in step 2 of our scenario, where Sara issues a ticket to Allice encrypted in Alice's secret key, KA.

Authenticated Communication with Public Keys

Bob has a public/private key pair $<K_{Bpub}, K_{Bpriv}>$ & establishes K_{AB} as follows:

1. Alice obtains a certificate that was signed by a trusted authority stating Bob's public key K_{Bpub}
2. Alice creates a new shared key K_{AB} , encrypts it using K_{Bpub} using a public-key algorithm and sends the result to Bob.
3. Bob uses the corresponding private key K_{Bpriv} to decrypt it.

(If they want to be sure that the message hasn't been tampered with, Alice can add an agreed value to it and Bob can check it.)

Digital Signatures with secure digest function

Alice wants to publish a document M in such a way that anyone can verify that it is from her.

1. Alice computes a fixed-length digest of the document Digest(M).
2. Alice encrypts the digest in her private key, appends it to M and makes the resulting signed document (M, $\{\text{Digest(M)}\}_{K_{Apriv}}$) available to the intended users.
3. Bob obtains the signed document, extracts M and computes Digest(M).
4. Bob uses Alice's public key to decrypt $\{\text{Digest(M)}\}_{K_{Apriv}}$ and compares it with his computed digest. If they match, Alice's signature is verified.

Cryptographic Algorithms

Symmetric (Secret Key)

E(K, M) = $\{M\}_K$ == D(K, E(K, M)) = M

Same key for Encryption and Decryption

Asymmetric (Public Key)

Separate encryption and decryption keys: K_e, K_d

$D(K_d. E(K_e, M)) = M$

Depends on the use of a *trap-door function* to make the keys. A message encrypted using Private key can be decrypted using public key and vice versa.

Hybrid protocols - used in SSL (now called TLS)

Uses asymmetric crypto to transmit the symmetric key that is then used to encrypt a session.

Public Key Infrastructure (PKI)

- PKI allows you to know that a given public key belongs to a given user
- PKI builds on asymmetric encryption:
  - Each entity has two keys: public and private
  - Data encrypted with one key can only be decrypted with other.
  - The private key is known only to the entity
- The public key is given to the world encapsulated in a X.509 certificate

X509 Format

Subject	Distinguished Name, Public Key
Issuer	Distinguished Name, Signature
Period of validity	Not Before Date, Not After Date
Administrative information	Version, Serial Number
Extended Information	

- It provides a public key to a named entity called the subject. The binding is in the signature, which is issued by another entity called issuer (CA, Certificate Authority).
- Certificates can act as *credentials*
  - Evidence for a principal's right to access a resource

Access Control

- Access control list (ACL) associated with each object
  - *E.g. Unix file access permissions*
- Capabilities associated with principals
  - *Like a key – allowing the holder access to certain operations on a specified resource.*
  - *Format: <resource id, permitted operations, authentication code>*

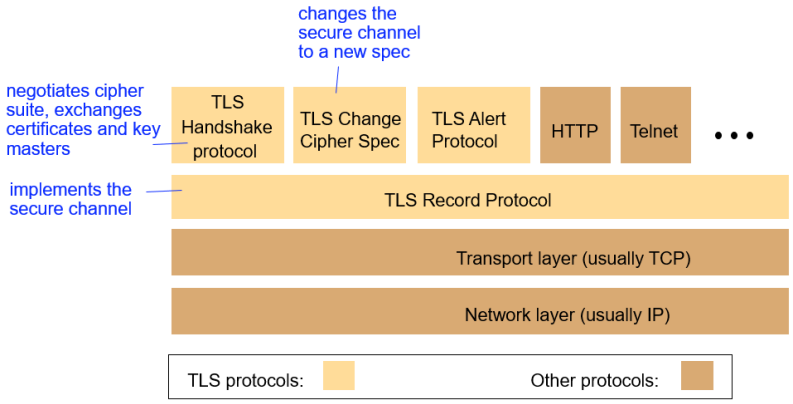
The secure Socket Layer

- Key distribution and secure channels for Internet commerce
  - Hybrid protocol; depends on public-key cryptography
  - Originally developed by Netscape Corporation (1994) and supported by most browsers and is widely used in Internet commerce.
  - Extended and adopted as an Internet standard with the name Transport Level Security (TLS) – RFC 2246
  - Provides the security in all web servers and browsers and in secure versions of Telnet, FTP and other network applications
- Key Feature
  - **Negotiable encryption and authentication algorithms**. In an open network we should NOT assume that all parties use the same client software, or all client/server software includes a particular encryption algorithms.
- Design requirements
  - Secure communication without prior negotiation or help from 3rd parties

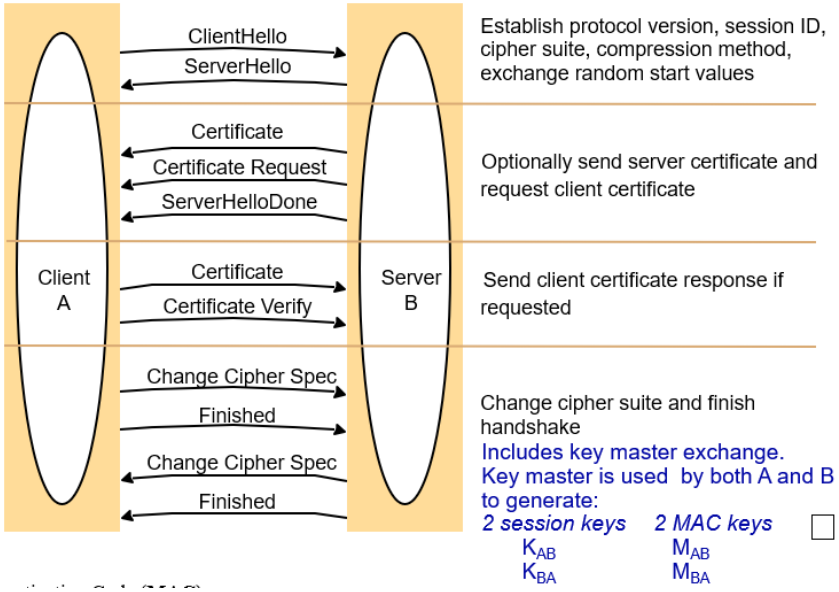
- Free choice of crypto algorithms by client and server
- communication in each direction can be authenticated, encrypted or both

Bootstrapped Secure Communication

- To meet the need for secure communication without previous negotiation/help from 3rd parties, the secure channel is established using a hybrid schemes.
- The secure channel is fully configurable.
- The details of TLS protocols are standardized and several software libraries and toolkits are available to support it [www.openssl.org]
- TLS consists of two layers:
  - **TLS Record Protocol Layer**: implements a secure channel, encrypting and authenticating messages transmitted through any connection oriented protocol. It is realized at session layer.
  - **Handshake Layer**: Containing Handshake protocol and two other related protocols that establish and maintain a TLS session (i.e., secure channel) between client and server.
  - Both are implemented by software libraries at application level in the client and the server.

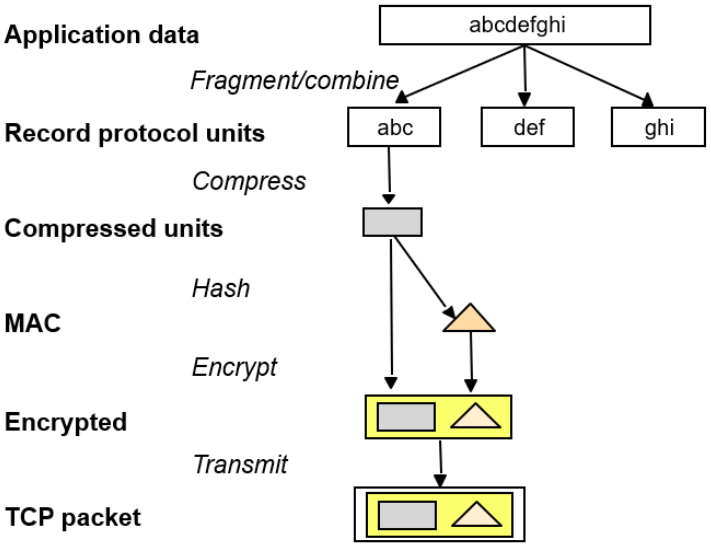


TSL/ SSL Handshake Protocol



Component	*Description*	*Example*
Key exchange method	the method to be used for exchange of a session key	RSA with public-key certificates
Cipher for data transfer	the block or stream cipher to be used for data	IDEA (International Data Encryption Algorithm)
Message digest function	for creating message authentication codes (MACs)	SHA (Secure Hash Algorithm)

TLS record protocol operation: a pipeline for data transformation



Section # 8 Distributed File System (DFS)

In first generation of distributed systems (1974-95), file systems (e.g. NFS) were the only networked storage systems.

With the advent of distributed object systems (CORBA, Java) and the web, the picture has become more complex.

Current focus is on large scale, scalable storage.

- Google File System
- Amazon S3 (**Simple Storage Service)**
- Cloud Storage (e.g., **DropBox**)

Properties of Storage Systems

	Sharing	*Persis-tence*	*Distributed cache/replicas*	*Consistency maintenance*	*Example*
Main memory	×	×	×	1	RAM
File system	×	✓	×	1	UNIX file system
Distributed file system	✓	✓	✓	✓	Sun NFS
Web	✓	✓	✓	×	Web server
Distributed shared memory	✓	×	✓	✓	Ivy (Ch. 16)
Remote objects (RMI/ORB)	✓	×	×	1	CORBA
Persistent object store	✓	✓	×	1	CORBA Persistent Object Service
Peer-to-peer storage store	✓	✓	✓	2	OceanStore

Types of consistency between copies: 1 - strict one-copy consistency
√ - approximate/slightly weaker guarantees
X - no automatic consistency
2 – considerably weaker guarantees

File Systems

- Persistent stored data sets
- Hierarchic name space visible to all processes
- API with the following characteristics:
  - access and update operations on persistently stored data sets
  - Sequential access model (with additional random facilities)
- Sharing of data between users, with access control
- Concurrent access:
  - For read-only access

Class Exercise

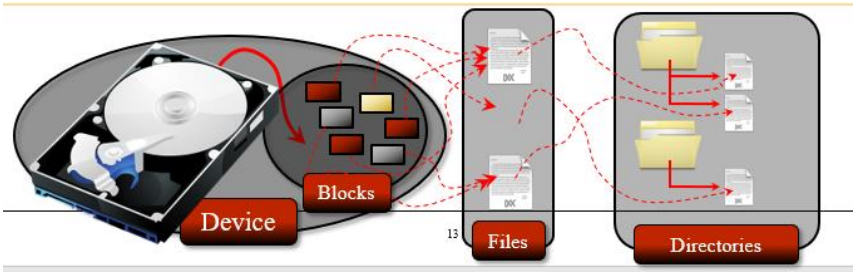
Write a simple C program to copy a file using the UNIX file system operations.

```
#define BUFSIZE 1024
#define READ 0
#define FILEMODE 0644
void copyfile(char* oldfile, char* newfile)
{       char buf[BUFSIZE]; int i,n=1, fdold, fdnew;

        if((fdold = open(oldfile, READ))>=0) {
                fdnew = creat(newfile, FILEMODE);
                while (n>0) {
                        n = read(fdold, buf, BUFSIZE);
                        if(write(fdnew, buf, n) < 0) break;
                }
        close(fdold);   close(fdnew);
        }
        else printf("Copyfile: couldn't open file: %s \n", oldfile);
}
main(int argc, char **argv)  {
        copyfile(argv[1], argv[2]);
}
```

File System Modules

Directory module:	relates file names to file IDs
File module:	relates file IDs to particular files
Access control module:	checks permission for operation requested
File access module:	reads or writes file data or attributes
Block module:	accesses and allocates disk blocks
Device module:	disk I/O and buffering



DFS Requirements

Transparency

- *Access*: Same operations (client programs are unaware of distribution of files)
- *Location*: Same name space after relocation of files or processes (client programs should see a uniform file name space)
- *Mobility*: Automatic relocation of files is possible (neither client programs nor system admin tables in client nodes need to be changed when files are moved).
- *Performance*: Satisfactory performance across a specified range of system loads
- *Scaling*: Service can be expanded to meet additional loads or growth.

Concurrency

- Changes to a file by one client should not interfere with the operation of other clients simultaneously accessing or changing the same file.
- Concurrency properties
    - Isolation
    - File-level or record-level locking
    - Other forms of concurrency control to minimise contention

Replication

- File service maintains multiple identical copies of files
- Load-sharing between servers makes service more scalable
    - Local access has better response (lower latency)
    - Fault tolerance
- Full replication is difficult to implement.
- Caching (of all or part of a file) gives most of the benefits (except fault tolerance)

Heterogeneity

- Service can be accessed by clients running on (almost) any OS or hardware platform.
- Design must be compatible with the file systems of different OSes
- Service interfaces must be *open* - precise specifications of APIs are published.

Fault Tolerance

- Service must continue to operate even when clients make errors or crash.
- Service must resume after a server machine crashes.
- If the service is replicated, it can continue to operate even during a server crash.

Consistency

- Unix offers one-copy update semantics for operations on local files - caching is completely transparent.
- Difficult to achieve the same for distributed file systems while maintaining good performance and scalability.

Security

- Must maintain access control and privacy as for local files.
    - based on identity of user making request
    - identities of remote users must be authenticated
    - privacy requires secure communication
- Service interfaces are open to all processes not excluded by a firewall.
    - vulnerable to impersonation and other attacks

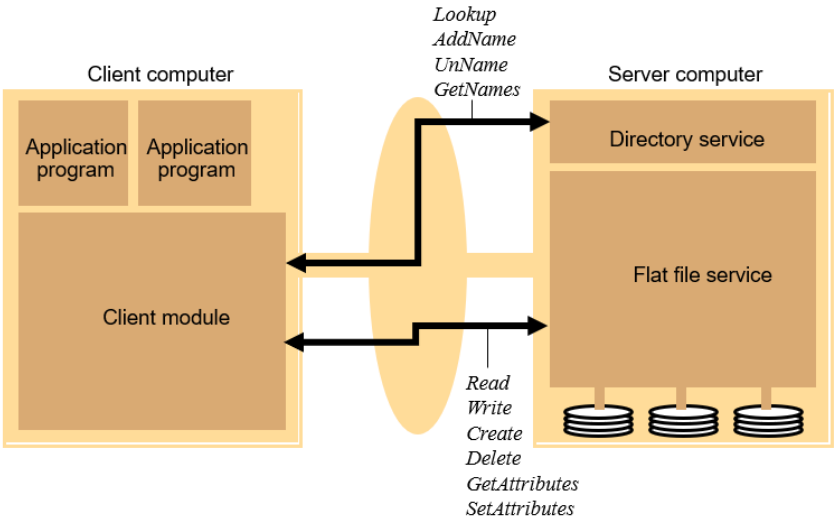
Efficiency

- Goal for distributed file systems is usually performance comparable to local file system.

File Server Architecture

- An architecture that offers a clear separation of the main concerns in providing access to files is obtained by structuring the file service as three components:
    - A flat file service – A file service on the server
    - A directory service – Directory with information of where the files are stored
    - A client module.
- The relevant modules and their relationship is (shown next).

- The Client module implements exported interfaces by flat file and directory services on server side.



- Flat file service:
  - Concerned with the implementation of operations on the contents of file. *Unique File Identifiers* (UFIDs) are used to refer to files in all requests for flat file service operations. UFIDs are long sequences of bits chosen so that each file has a unique among all of the files in a distributed system.
- Directory Service:
  - Provides mapping between *text names* for the files and their UFIDs. Clients may obtain the UFID of a file by quoting its text name to directory service. Directory service supports functions needed to generate directories and to add new files to directories.
- Client Module:
  - It runs on each computer and provides integrated service (flat file and directory) as a single API to application programs. For example, in UNIX hosts, a client module emulates the full set of Unix file operations.
  - It holds information about the network locations of flat-file and directory server processes; and achieve better performance through implementation of a cache of recently used file blocks at the client.

File Group

- A collection of files that can be located on any server or moved between servers while maintaining the same names.
  - Similar to a UNIX *filesystem*
  - Helps with distributing the load of file serving between several servers.
  - File groups have identifiers which are unique throughout the system (and hence for an open system, they must be globally unique).
    - *Used to refer to file groups and files*

Architectural Components (UNIX/ LINUX)

- Server:
  - nfsd: NFS server daemon that services requests from clients.
  - mountd: NFS mount daemon that carries out the mount request passed on by nfsd.

    - rpcbind: RPC port mapper used to locate the nfsd daemon.
    - /etc/exports: configuration file that defines which portion of the file systems are exported through NFS and how.
- Client:
    - mount: standard file system mount command.
    - /etc/fstab: file system table file.
    - nfsiod: (optional) local asynchronous NFS I/O server.

Mount Server

- Mount operation:
    - *mount(remotehost, remotedirectory, localdirectory)*
- Server maintains a table of clients who have mounted filesystems at that server
- Each client maintains a table of mounted file systems holding:
  < IP address, port number, file handle>

Hard Mount vs. Soft Mount

When performing an I/O operation on a remote file, a **Soft Mount** would try to perform the action a few times and return unsuccessful if the function cannot be performed. Whereas, a **Hard Mount** will keep trying till the action is successfully accomplished.

Automounter

- NFS client catches attempts to access 'empty' mount points and routes them to the Automounter
    - Automounter has a table of mount points and multiple candidate serves for each
    - it sends a probe message to each candidate server and then uses the mount service to mount the filesystem at the first server to respond
- Keeps the mount table small
- Provides a simple form of replication for read-only filesystems

Kerberized NFS

Issues with Kerbros Protocol

- Kerberos protocol is too costly to apply on each file access request
- Kerberos is used in the mount service:
    - to authenticate the user's identity
    - User's UserID and GroupID are stored at the server with the client's IP address
- For each file request:
    - The UserID and GroupID sent must match those stored at the server
    - IP addresses must also match
- This approach has some problems
    - can't accommodate multiple users sharing the same client computer
    - all remote filestores must be mounted each time a user logs in

New Design Approaches

- Distribute file data across several servers
    - Exploits high-speed networks (InfiniBand, Gigabit Ethernet)
    - Layered approach, lowest level is like a 'distributed virtual disk'
    - Achieves scalability even for a single heavily-used file
- 'Serverless' architecture
    - Exploits processing and disk resources in all available network nodes

- Service is distributed at the level of individual files

Section # 9 Name Services

In a Distributed System, a Naming Service is a specific service whose aim is to provide a consistent and uniform naming of resources, thus allowing other programs or services to localize them and obtain the required metadata for interacting with them.

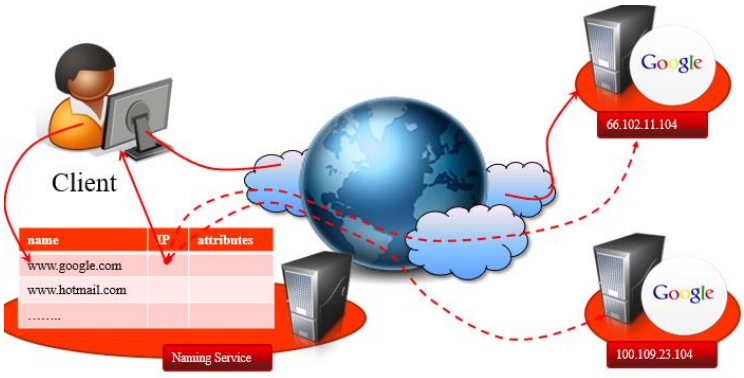
- In a distributed system, names are used to refer to a wide variety of resources such as:
  - Computers, services, remote objects, and files, as well as users.
- Naming is fundamental issue in DS design as it facilitates communication and resource sharing.
  - A name in the form of URL is needed to access a specific web page.
  - Processes cannot share particular resources managed by a computer system unless they can name them consistently
  - Users cannot communicate within one another via a DS unless they can name one another, with email address.
- Naming Services are not only useful to locate resources but also to gather additional information about them such as attributes

Advantages

- Resource localization
- Uniform naming
- Device independent address (e.g., you can move domain name/web site from one server to another server seamlessly).

Different Naming Services

- An URL facilitates the localization of a resource exposed on the Web.
  - *e.g., abc.net.au means it is likely to be an Australian entity?*
- A consistent and uniform naming helps processes in a distributed system to interoperate and manage resources.
  - *e.g., commercials use .com; non-profit organizations use .org*
- Users refers to each other by means of their names (i.e. email) rather than their system ids



Uniform Resource Identifier

Uniform Resource Identifiers (URI) offer a general solution for any type of resource. There two main classes:

- *URL – Uniform Resource Locator (URL)*
  - typed by the protocol field (http, ftp, nfs, etc.)

    - part of the name is service-specific
    - resources cannot be moved between domains
- *URN – Uniform Resource Name (URN)*
    - requires a universal resource name lookup service - a DNS-like system for all resources
    - *format: urn:<nameSpace>:<name-within-namespace>*
    - *examples:*
        - *urn:ISBN:021-61918-0*
        - *urn:cloudbus.unimelb.edu.au:TR2005-10*

Navigation

Navigation is the act of chaining multiple Naming Services in order to resolve a single name to the corresponding resource.

Iterative Navigation

- DNS: Client presents entire name to servers, starting at a local server, NS1. If NS1 has the requested name, it is resolved, else NS1 suggests contacting NS2 (a server for a domain that includes the requested name).
- NFS: Client segments pathnames (into 'simple names') and presents them one at a time to a server together with the filehandle of the directory that contains the simple name.

Reason for NFS iterative name resolution

This is because the file service may encounter a symbolic link (i.e. an *alias*) when resolving a name. A symbolic link must be interpreted in the client's file system name space because it may point to a file in a directory stored at another server. The client computer must determine which server this is, because only the client knows its mount points.

Server Controlled Navigation

- In an alternative model, name server coordinates naming resolution and returns the results to the client. It can be:
    - Recursive:
        - *it is performed by the naming server*
        - *the server becomes like a client for the next server*
        - *this is necessary in case of client connectivity constraints*
    - Non- recursive:
        - *it is performed by the client or the first server*
        - *the server bounces back the next hop to its client*

Domain Name Server

- A distributed naming database (specified in RFC 1034/1305)
- Name structure reflects administrative structure of the Internet
- Rapidly resolves domain names to IP addresses
    - exploits caching heavily
    - typical query time ~100 milliseconds
- Scales to millions of computers
    - partitioned database
    - caching
- Resilient to failure of a server
    - Replication

DNS Issues

- Name tables change infrequently, but when they do, caching can result in the delivery of stale data.
    - Clients are responsible for detecting this and recovering
- Its design makes changes to the structure of the name space difficult.