Distributed Systems Summary

1. Definition:

Distributed Systems: A system in which hardware or software components located at the networked computers communicate and coordinate their actions only by message passing.

Cluster vs Cloud:

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

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. However, networks focus on packets whereas distributed systems focus on applications.

Issues of DS: Concurrency, heterogeneity, no global lock and independent failures.

Characteristics of DS: Parallel activities, communication via message passing, resource sharing, no global state and no global lock.

Goals of DS: Connecting users and resources, transparency, openness, scalability, enhanced availability.

Challenges of DS:

- Heterogeneity
  - Heterogeneous components must be able to interoperate
- Distribution transparency
  - Distribution should be hidden from the user as much as possible
- Fault tolerance
  - Failure of a component (partial failure) should not result in failure of the whole system
- Scalability
  - System should work efficiently with an increasing number of users
  - System performance should increase with inclusion of additional resources
- Concurrency
  - Shared access to resources must be possible
- Openness
  - Interfaces should be publicly available to ease inclusion of new components
- Security
  - The system should only be used in the way intended

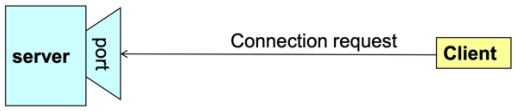
2. Socket

What: Socket provides an interface for programming networks at the transport layer. The communication is socket is similar to performing file I/O. Socket-based communication is programming language

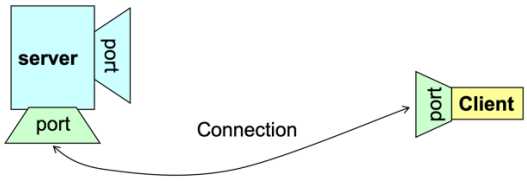
independent. A socket is bound to a port number so that TCP layer can identify the application that data destinated to be sent.

Socket Communication:

- 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.



Middleware: A layer of software between applications and operating system powering the nodes of a distributed system. Building distributed systems need both middleware and operating system. The purpose of middleware is to mask heterogeneity present in distributed systems. RPC, RMI is the examples of middleware.

Middleware vs Operating System: Middleware provides higher-level features for DS, including communication, management and application specific, while operating system provides lower-level features such as process/thread management, local hardware management, security and some basic networking.

Why Network OS is widely used:

- Many DOS (Distributed OS) have been investigated, but there are none in general/wide use. But NOS are in wide use for various reasons both technical and non-technical.
  - Users have much invested in their application software; they will not adopt a new OS that will not run their applications.
  - Users tend to prefer to have a degree of autonomy of their machines, even in a closely knit organisation.
- A combination of middleware and NOSs provides an acceptable balance between the requirement of autonomy and network transparency.
  - NOS allows users to run their favorite word processor.
  - Middleware enables users to take advantage of services that become available in their distributed systems.

Multitasking: Multiple applications running at the same time.

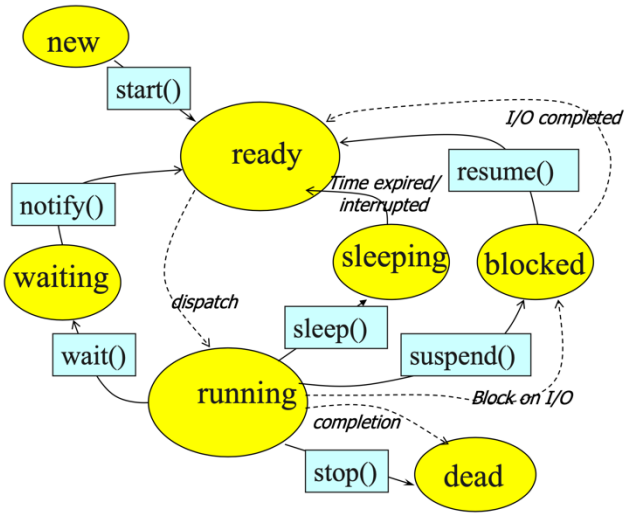
Multithreading: Multiple operations running at the same time.

Thread: A piece of code that runs in concurrent with other threads.

Thread Applications:

- Applications – Threads are used to perform:
  - Parallelism and concurrent execution of independent tasks / operations.
  - Implementation of reactive user interfaces.
  - Non blocking I/O operations.
  - Asynchronous behavior.
  - Timer and alarms implementation.

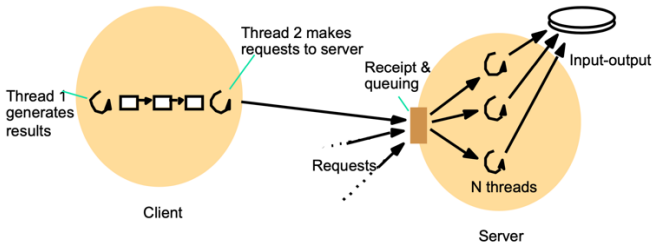
Java Threads: To create a thread, one way is to extend Thread class or implement Runnable interface.



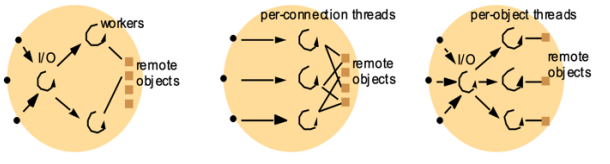
The thread can be assigned priority, which affects the order in which it is scheduled for running.

Synchronization: When multiple threads are accessing the shared resources, it should be synchronized to prevent they end up with the inconsistent state.

Architecture for Multithreaded Servers: Worker pool, thread-per-request, thread-per-connection, thread-per-object.



- 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.



a. Thread-per-request | b. Thread-per-connection | c. Thread-per-object

IO Thread creates a new worker thread for each request and worker thread destroys itself after serving the request.

Server associates a Thread with each connection and destroys when client closes the connection. Client may make many requests over the connection.

Associates Thread with each object. An IO thread receives request and queues them for workers, but this time there is a **per-object queue**.

3. Distributed System Models

Why: Distributed systems should be designed to function correctly in all circumstances. The distributed system models help in classifying and understanding different implementations, identifying their weaknesses and strengths, crafting new systems out of pre-validated building blocks.

Goals:

The structure and the organization of systems and the relationship among their components should be designed with the following goals in mind:

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

Challenges:

- Widely varying models of use
  - High variation of workload, partial disconnection of components, or poor connection.
- Wide range of system environments
  - Heterogeneous hardware, operating systems, network, and performance.
- Internal problems
  - Non synchronized clocks, conflicting updates, various hardware and software failures.
- External threats
  - Attacks on data integrity, secrecy, and denial of service.

Solutions:

- Widely varying models of use
  - The structure and the organization of systems allow for distribution of workloads, redundant services, and high availability.
- Wide range of system environments
  - A flexible and modular structure allows for implementing different solutions for different hardware, OS, and networks.
- Internal problems
  - 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
  - Security has to be built into the infrastructure and it is fundamental for shaping the relationship between components.

Physical Models:

- **Definition:**

A representation of the underlying H/W elements of a DS that abstracts away specific details of the computer/networking technologies.

- **Three Generations**:

- Early DSs [70-80s]: LAN-based, 10-100 nodes
- Internet-scale DSs [early 90-2010]: 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, and **Internet of Things (IoT)** (seamless interaction between physical and cyber world for smart * applications such as Smart Health and Smart Cities)

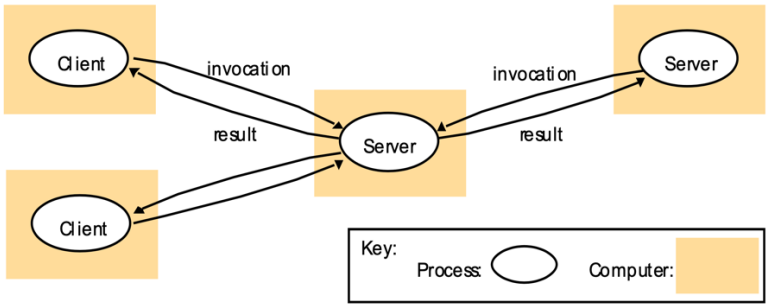
Architecture Models:

- **Definition**:

- Simplifies and abstracts the functions of individual components
- The placement of the components across a network of computers – define patterns for the distribution of data and workloads
- The interrelationship between the components – ie., functional roles and the patterns of communication between them.

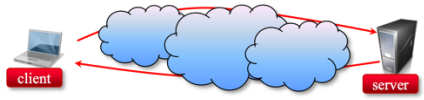
- Goals: Meet present and future demands.
- Concerns: Make the system reliable, manageable, adaptable and cost-effective.
- Platform: The lowest hardware and software.
- Sever-Client model:

  (i) Clients interact with individual server



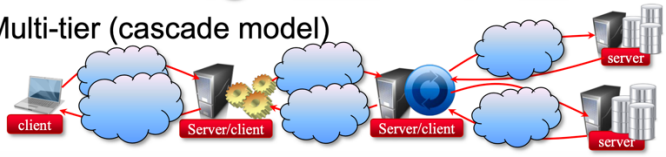
- Client processes interact with individual server processes in a separate computer in order to access data or resource. The server in turn may use services of other servers.

- Two-tier model (classic)

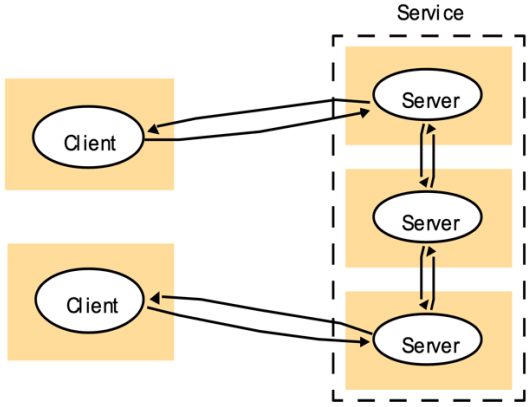


- Three-tier (when the server, becomes a client)

- Multi-tier (cascade model)

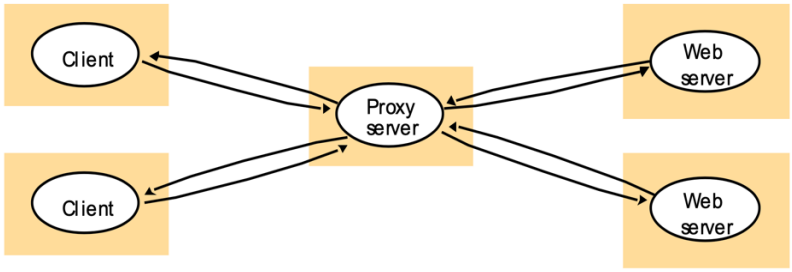


(ii) Clients interact with multiple servers

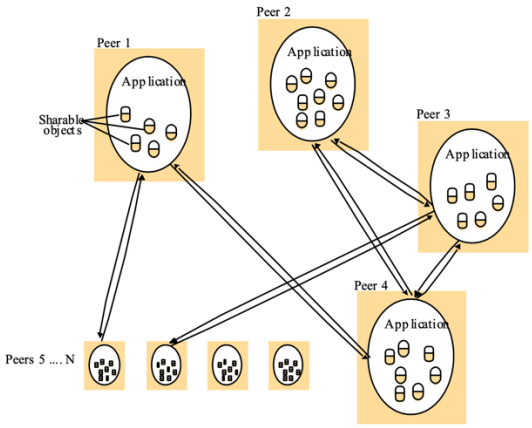


Services may be implemented as several server processes in separate host computers.

(iii) Proxy servers: Use cache to store recently used data.



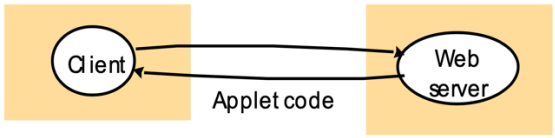
- Peer model:



All of the processes play similar roles, interacting cooperatively as peers to perform distributed activities or computations without distinction between clients and servers. E.g., music sharing systems Napster, Gnutella, Kaza, BitTorrent.

- Mobile code and web applet:

a) client request results in the downloading of applet code



b) client interacts with the applet

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).

- Model agent:

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

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.

- Network computer: Downloaded computer and applications from the network and run on a desktop.
- Thin clients: Windows-based UI on the user machine and application execution on a remote computer.
- Design requirements:

- 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.

Fundamental Model

Definition:

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

Interactive Model: It deals with performance and the difficulty of setting time limits in a distributed system. The processes interact by passing messages, resulting in communication and coordination. The communication performance is often a limiting characteristic, and the global notion of time is hard to maintain.

- Performance of communication channel: The three basic characteristics of a computer network are latency, bandwidth and jitter.
- Clock time drift: Even two processes on different computers read their clocks at the same time, their local clocks may supply different time, because the drift rate (drift from perfect time) differs from computer to computer. Even the clocks on all computers in DS share the same time initially, their clocks would eventually vary quite significantly unless correlations are applied.
- Synchronous DS and Asynchronous DS:

- 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.
- Asynchronous DS: There is NO bounds on:
  - Process execution speeds
  - Message transmission delays
  - Clock drift rates.

- Event ordering:

- The execution of a system can be described in terms of events and their ordering despite the lack of accurate clocks.

Failure Model: Specification of the faults that can be exhibited by processes.

- Omission failure:

  - Communication channel produces an omission failure if it does not transport a message from "p"s outgoing message buffer to "q"'s incoming message buffer. 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.

- Arbitrary failure:

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 failure:

  - 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 object:

  - 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.

- Security threat:

  - 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.

- Defeating methods:

- Encryption and authentication are use 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.

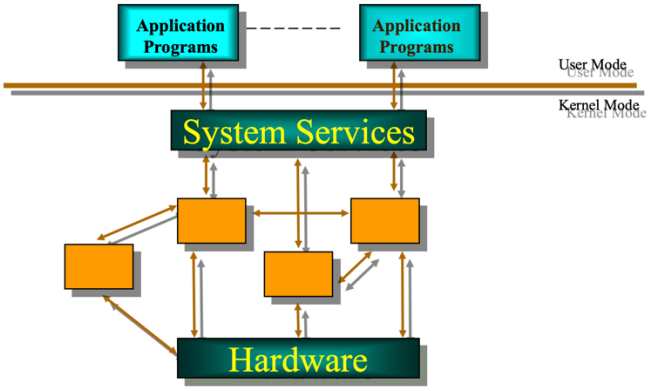
4. Operating System

Open Distributed System:

- A open DS should make it possible to:
  - Run only that ("specific" components of) 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 service to be provided, when this is required to suit different users or applications.
  - Introduce new services without harming the integrity of existing ones.

Kernel Design: The design of kernels differs in the decision as to what functionality belongs in the kernel and what is left to server processes.

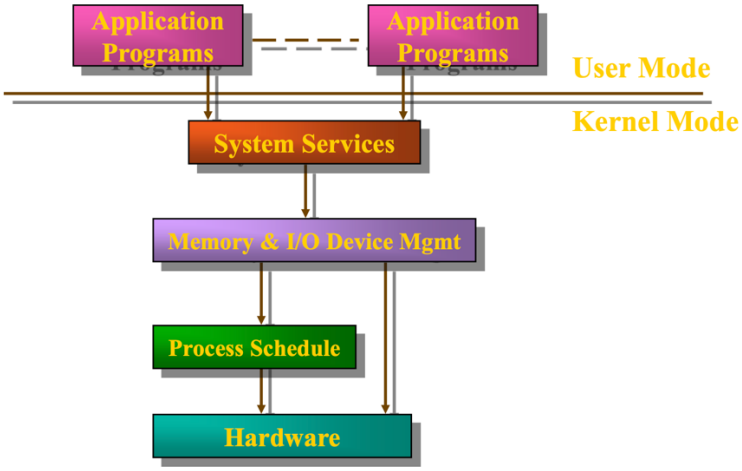
- Monolithic OS:



Disadvantages:

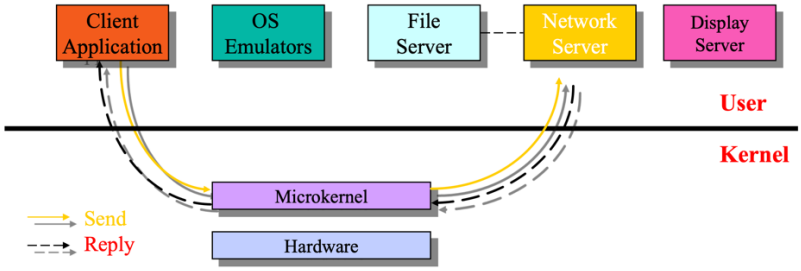
- It is massive:
  - It performs all basic OS functions and takes up in the order of megabytes of code and data
- It is undifferentiated:
  - It is coded in a non-modular way (traditionally) although modern ones are much more layered.
- It is intractable:
  - Altering any individual software component to adapt it to changing requirements is difficult.

- Layered OS:



- Micro-kernel OS: Appears as a layer between hardware and a layer of major system components.

- 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.



Monolithic vs Micro-Kernel OS:

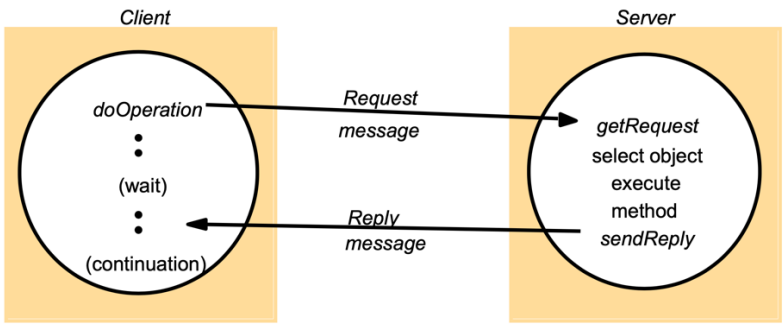
- The main advantages of a MK-based OS:
  - A relative small kernel is more likely to be free of bugs than one that is larger and complex.
  - Extensibility and its ability to enforce modularity behind memory protection boundaries
- The advantage of a monolithic OS:
  - Relative efficiency with which operations can be invoked is high because even invocation to a separate user-level address space on the same node is more costly.

Hybrid Approaches: Change from user-level address space to kernel address space.

5. Remove Method Invocation

Two Basic Models: Remote Procedure Call and Remote Method Invocation.

Request-Reply Protocol: Consists of doOperation on client side, getRequest and sendReply on server side.



Invocation Semantics:

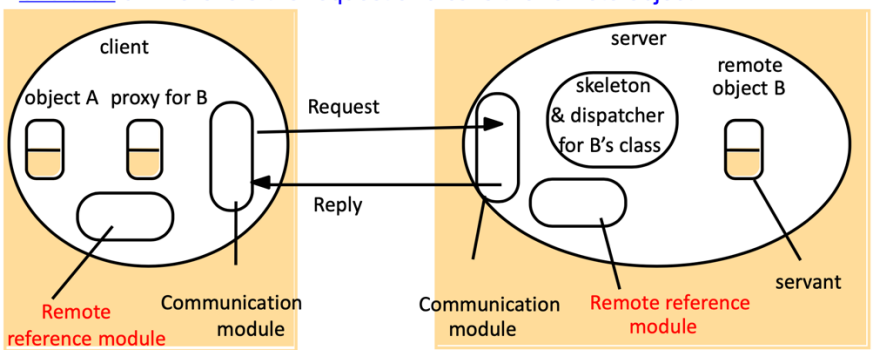
- **Maybe**: The remote procedure call may be executed once or not at all. Unless the caller receives a result, it is unknown as to whether the remote procedure was called.
- **At-least-once**: Either the remote procedure was executed at least once, and the caller received a response, or the caller received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

RMI supports at-most-once invocation, while RPC supports at-least-once invocation.

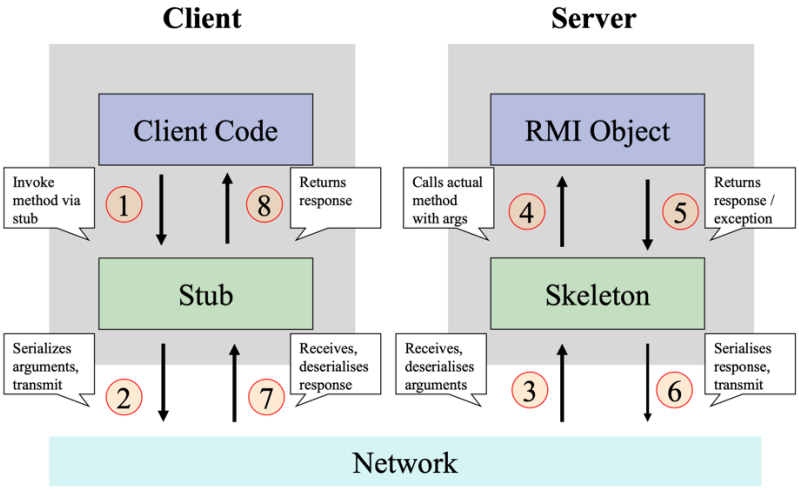
Distributed Objects: Each process contains a collection of objects, some of which can receive both local and remote invocations.

RMI Architecture:

- Remote reference module (at client & server) is responsible for providing addressing to the proxy (stub) object
- Proxy is used to implement a stub and provide transparency to the client. It is invoked directly by the client (as if the proxy itself was the remote object), and then marshal the invocation into a request
- Communication module is responsible for networking
- Dispatcher selects the proper skeleton and forward message to it
- Skeleton un-marshals the request and calls the remote object

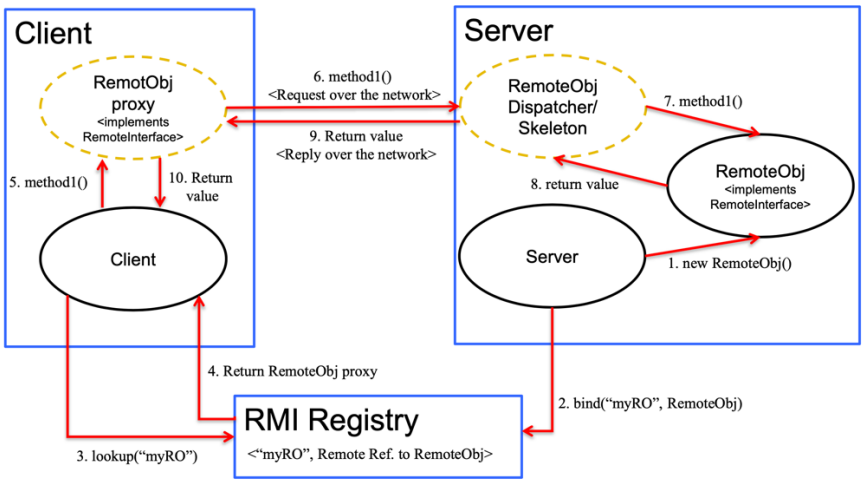


RMI Lifecycle:



Implementation of RMI:

- 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 makes available the servant
  - It usually registers with the naming service
- Client component



Security Manager: For access control. In Java, we use java.security to specify them. The security manager in RMI ensures that the operations performed by downloaded code go through a set of security checks. There are two ways to declare security manager. One is use system property, another is to declare in the source code.

Exceptions: The only exception that could be thrown out RemoteException, and all RMI remote method must throw this exception.

Passing Object:

- All the fields in a serializable object must be also serializable
- Primitives are serializable
- System related features (e.g. *Thread*, *File*) are non-serializable

Dynamic Class Loading:

- Ability to download bytecode (classes) from Remote JVM
- New types can be introduced into a remote virtual machine without informing the client
  - Extend the behavior of an application dynamically
  - Removes the need to deploy stubs manually
- Explicit set property to support dynamic class load
  - Specify system property *java.rmi.server.codebase* to tell the program where to download classes

6. Security

Security Goal: Restrict access to resources to just those entities that are authorized.

Security Threats:

- Security Threats - Three broad Classes:
  - Leakage: Acquisition of information by unauthorised recipients
  - Tampering: Unauthorised alteration of information
  - Vandalism: Interference with the proper operation of systems

Method of Attacks:

- 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 at a later date
- Denial of service - Vandalism
  - flooding a channel or other resource, denying access to others

15

Design Secure Systems: Although the development of cryptography has made in recent years, designing a secure system is still a hard problem. There are some worst case assumptions:

- Interfaces are exposed
  - DSs made up of processes with open interfaces
- Networks are insecure
  - Messages sources can be falsified.
- Limit the lifetime and scope of each secret
  - Passwords and keys validity – needs to be time restricted.
- Algorithms and code are available to hackers
- Attackers may have access to large resources
- Minimise the trusted base.

Cryptography:

- Two main classes:

- **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.*

- Use of cryptography: secrecy, integrity, authentication, digital signature.
- Scenario

  Scenario 1: secrete communication with a shared secrete key.

  Alice wishes to send some information secretly. Alice and Bob share a secret key K_{AB}.

  1. Alice uses K_{AB} and an agreed encryption function $E(K_{AB}, M)$ to encrypt and send any number of messages $\{M_i\}_{K_{AB}}$ to Bob.
  2. Bob reads the encrypted messages using the corresponding decryption function $D(K_{AB}, M)$.

  Alice and Bob can go on using K_{AB} as long as it is safe to assume that K_{AB} has not been *compromised*.

  Issues: key distribution, authentication.

  Scenario 2: authenticated communication with a server.

  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.

  1. Alice sends an (unencrypted) message to Sara stating her identity and requesting a *ticket* for access to Bob.
  2. Sara sends a response to Alice. $\{\{Ticket\}_{K_B}, K_{AB}\}_{K_A}$. 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}.
  3. Alice uses K_A to decrypt the response.
  4. Alice sends Bob a request R to access a file: $\{Ticket\}_{K_B}$, Alice, R.
  5. The ticket is actually $\{K_{AB}, Alice\}_{K_B}$. Bob uses K_B to decrypt it, checks that Alice's name matches and then uses K_{AB} to encrypt responses to Alice.

  Issue: timing and replay attack, not scalable.

  Scenario 3: 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.)

  Issues: someone may intercept Alice request for Bob's public key and replaced it with his public key, and then intercept all the subsequent messages.

  Scenario 4: digital signatures with a 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 $(M, \{Digest(M)\}_{K_{Apriv}})$ 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 $\{Digest(M)\}_{K_{Apriv}}$ and compares it with his computed digest. If they match, Alice's signature is verified.

- Cryptography Algorithms:

- Symmetric (secret key)
  $E(K, M) = \{M\}_K$ $D(K, E(K, M)) = M$
  Same key for E and D
  M must be hard (infeasible) to compute if K is not known.
  Usual form of attack is brute-force: try all possible key values for a known pair M, $\{M\}_K$. Resisted by making K sufficiently large ~ 128 bits

- 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. E has high computational cost. Very large keys > 512 bits

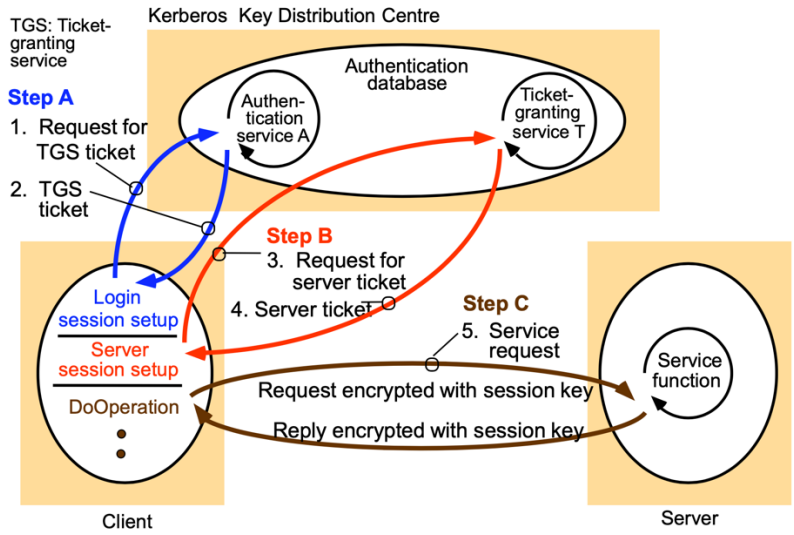
- Hybrid protocols - used in SSL (now called TLS)
  Uses asymmetric crypto to transmit the symmetric key that is then used to encrypt a session.

Public Key Infrastructure: Allows you to know that a given public key belongs to a given user.

- Certificate: A statement that signed by an appropriate authority. It has a standard format, a chain of trust, expire dates, and so on.
- Certificate Authority: A authority that certificates the identity of the holder of private key by signing a digital signature attached to the public key.

Access Control: Use access control list associated with each object; Capabilities associated with principles.

Kerberos Key Distribution:



- Kerberos protocol is too costly to apply on each NFS operation
- 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:
  - UserID and GroupID are sent encrypted in the shared session key
  - The UserID and GroupID 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

Secure Socket Layer: Use hybrid model, which uses public key to exchange shared secrete key for encryption and decryption.

- 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.

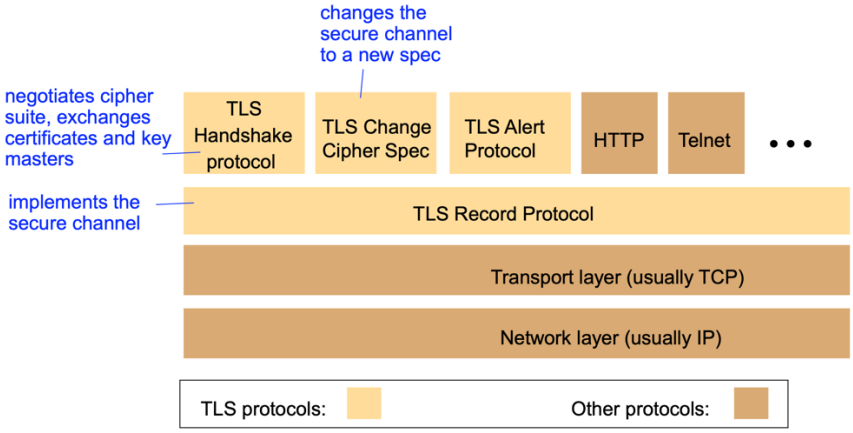
- Design Requirements:

– 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

- Transport Level Security:

– **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.

- TLS Protocol Stack:



7. Distributed File System

Why: The primary use of DFS is to connect users and resources.

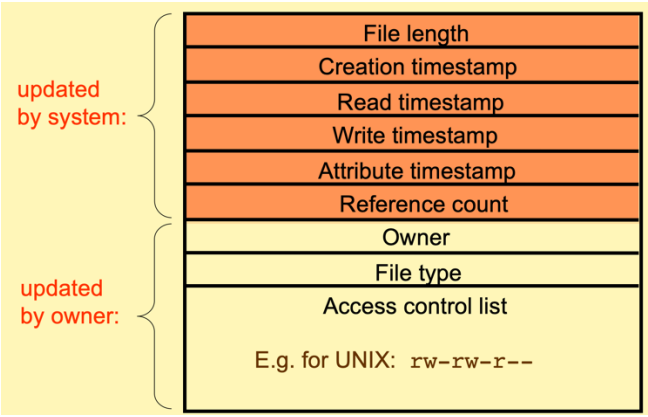
Storage Systems Properties:

	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

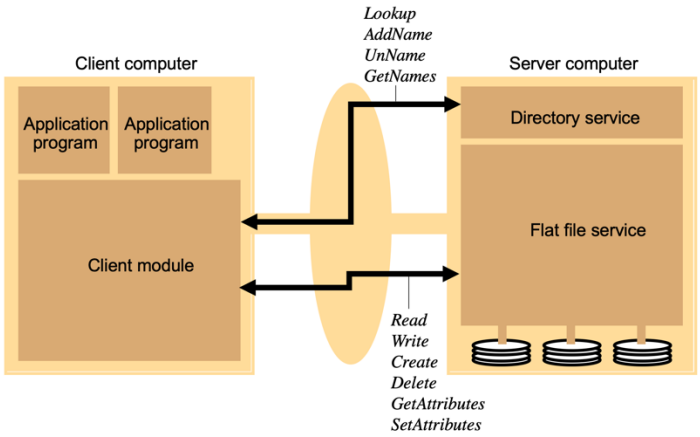
File System: A persistent stored dataset, which has a hierarchical name space visible to all processes.

- **Module**: dictionary module relates file name to file IDs; file module relates file IDs to particular files; access control module checks permission of operation requested; file access model reads or writes file data or attributes; block module accesses and allocates disk blocks; device module handle disk I/O and buffering.



Distributed File System

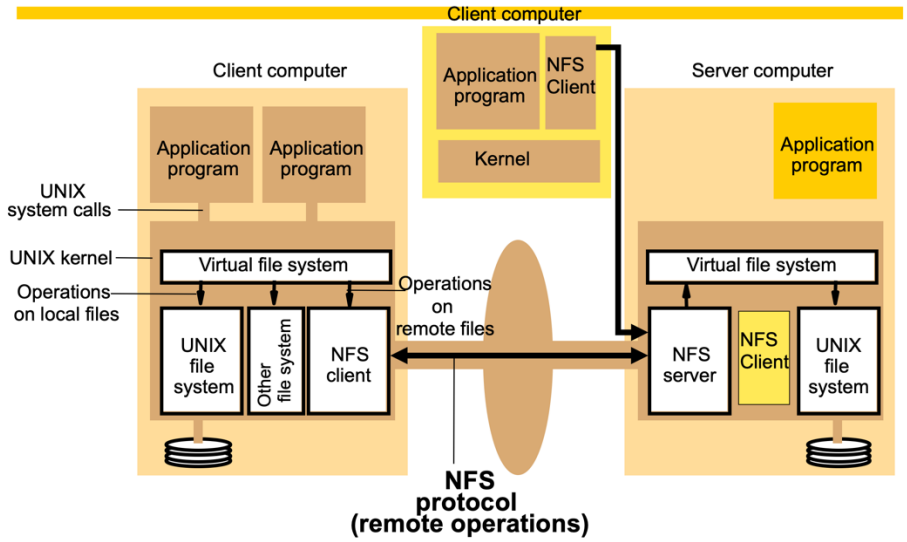
- Requirements: transparency, concurrency, replication, heterogeneity, fault tolerance, consistency, security, efficiency.
- File service architecture: flat file service, directory service, client module



- 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. It has identifiers which are unique throughout the system.

Network File System: An industry standard file sharing on local networks, but it has the limited achievement of concurrency, replication, consistency and security.



The implementation of NFS is not in the system kernel but on the application level.

- Access control: NFS is a stateless server, so user's identity must be checked by the server on each request, which is accomplished by userID and groupID. However, they must be protected by encryption to avoid imposter attack. Kerbero provides a security solution.
- Components:

- 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 Service:

- 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>

Automounter: NFS catches attempts to access empty mount points and routes them to the automounter. Automounter has a table of mount points and multiple candidates serves for each. It will send a probe message to each candidate server, and mount the file system at the first server to respond. It also provides a simple form of read-only replication file systems.

8. Naming Service

Name and Code: A meaningful name is easier to remember than codes or numbers, but codes and numbers are more useful for structuring data and locating resources by a program.

Aim: To provide a service where clients are used to obtain attributes, such as the address of resources (computers, services, remote objects and files) or objects when given a name. It facilitates the communication and resource sharing. Names are not the only identifiers, descriptive attributes are another. URL facilitates the localization of the resources on the web, so that a consistent and uniform naming helps processes in a distributed system to interoperate and manage resources.

Naming 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.

The benefits of naming service are it provides resource localization, uniform naming and device independent address.

Role of Names and Naming Services

- Resources are accessed using *identifier* or *reference*
  - An identifier can be stored in variables and retrieved from tables quickly
  - Identifier includes or can be transformed to an address for an object
    - *E.g. NFS file handle, CORBA remote object reference*
  - A *name* is human-readable value (usually a string) that can be *resolved* to an identifier or address
    - *Internet domain name, file pathname, process number*
    - *E.g ./etc/passwd, http://www.cdk5.net/*
- For many purposes, names are preferable to identifiers
  - because the binding of the named resource to a physical location is deferred and can be changed
  - because they are more meaningful to users
- Resource names are *resolved* by name services
  - to give identifiers and other useful attributes

Requirements of Name Spaces:

- Allow simple but meaningful names to be used
- Potentially infinite number of names
- Structured
  - to allow similar subnames without clashes
  - to group related names
- Allow re-structuring of name trees
  - for some types of change, old programs should continue to work
- Management of trust

URI:

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

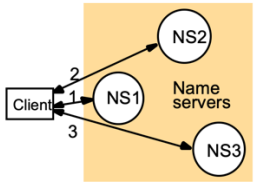
URN *Uniform Resource Name (URN)*
- requires a universal resource name lookup service - a DNS-like system for all resources

14

Navigation: An act of chaining multiple Naming Services in order to resolve a single name to the

corresponding resource.

- Iterative navigation:



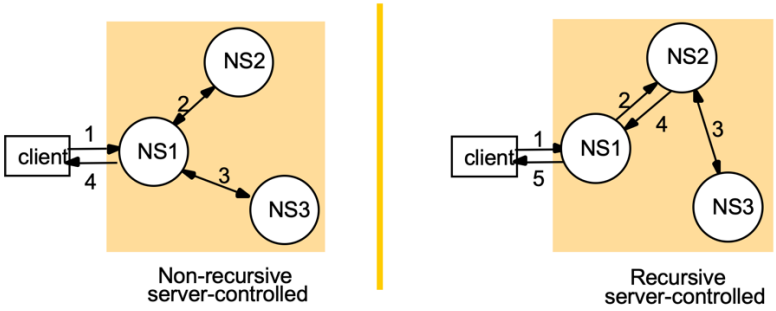
A client iteratively contacts name servers NS1–NS3 in order to resolve a name

Used in:

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.

- Server-controlled navigation:



A name server NS1 communicates with other name servers on behalf of a client

– 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*

Non-recursive is the standard technique, unless we want to limit client access to their DNS information for security reasons.

DNS: A server that is responsible to resolve a name to a resource location. DNS is a distributed database, and it heavily caches the results and uses replication method to avoid a failure node.

Basic DNS algorithm for name resolution (domain name -> IP number)

- Look for the name in the local cache
- Try a superior DNS server, which responds with:
  – another recommended DNS server
  – the IP address (which may not be entirely up to date)

Other functions:

– get *mail host* for a domain
– reverse resolution - get domain name from IP address
– Host information - type of hardware and OS
– Well-known services - a list of well-known services offered by a host
– Other attributes can be included (optional)

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. For example:
  - merging previously separate domain trees under a new root
  - moving subtrees to a different part of the structure (e.g. if Scotland became a separate country, its domains should all be moved to a new country-level domain.)

Directory Service: Store collections of bindings and attributes and also looks up entries that match attribute-based specifications. It retrieves the set of names that satisfy a given description.

Discovery Service:

- is automatically updated as the network configuration changes
- meets the needs of clients in spontaneous networks
- discovers services required by a client (who may be mobile) within the current *scope*, for example, to find the most suitable printing service for image files after arriving at a hotel.