CSC454

Distributed Networking (DN) Handbook

compiled by
Abhishek Tamrakar
Linus Dhakal
Pranesh Dhunju Shrestha
Rojesh Tamrakar

Unit 1

Protocols

Separate machines are connected via a physical network, and computers pass messages to each other using an appropriate protocol. Networking operating system services interface to the network via the kernel and contains complex algorithms to deal with transparency, scheduling, security, fault tolerance, etc. They provide peer-to-peer communication services and user applications directly access the services, or the middleware.

Middleware is a high level of OS services. Protocol is a "well-known set of rules and formats to be sued for communications between processes in order to perform a given task". Protocols specify the sequence and format of messages to be exchanged.

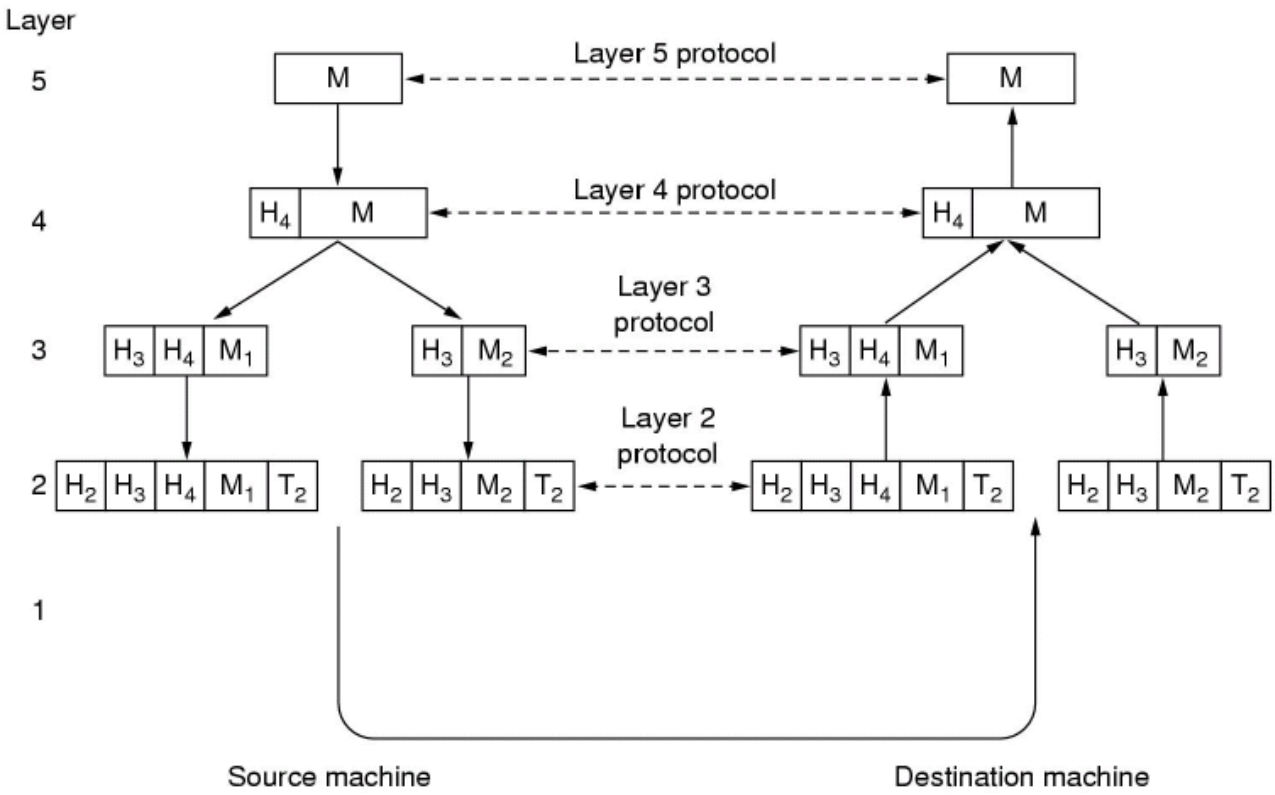
Protocol layers exist to reduce design complexity and improve portability and support for change. Networks are organized as series of layers or levels each built on the one below. The purpose of each layer is to offer services required by higher levels and to shield higher layers from the implementation details of lower layers.

Each layer has an associated protocol, which has two interfaces: a service interface - operations on this protocol, callable by the level above and a peer-to-peer interface, messages exchanged with the peer at the same level.

No data is directly transferred from layer *n* on one machine to layer *n* on another machine. Data and control pass from higher to lower layers, across the physical medium and then back up the network stack on the other machine.

As a message is passed down the stack, each layer adds a header (and possibly a tail such as a checksum, although this is most common at the bottom of the stack) and passes it to the next layer. As a message is received up the stack, the headers are removed and the message routed accordingly.

As you move down the stack, layers may have maximum packet sizes, so some layers may have to split up the packet and add a header on each packet before passing each individual packet to the lower layers.



Some protocol design issues include identification (multiple applications are generating many messages and each layer needs to be able to uniquely identify the sender and intended recipient of each message), data transfer alternatives (simplex, half and full duplex), EDC error control (error detection and correction - depends on environment e.g., SNR and application requirements), message order preservation and swamping of a slow receiver by a fast sender (babbling idiot problem, when a failed node swamps a databus with nonsense).

We can also consider two types of protocols, connection vs. connectionless. Connection orrientated services are where a connection is created and then messages are received in the order they are sent, a real world analogy is the POTS. Connectionless services are where data is sent independently. It is dispatched before the route is known and it may not arrive in the order sent, a real world analogy here is the postal system.

Connectionless and Connection Oriented Protocols

- **Connection-oriented** Requires a session connection (analogous to a phone call) be established before any data can be sent. This method is often called a "reliable" network service. It can guarantee that data will arrive in the same order. Connection-oriented services set up virtual links between end systems through a network, as shown in Figure 1. Note that the packet on the left is assigned the virtual circuit number 01. As it moves through the network, routers quickly send it through virtual circuit 01.

- **Connectionless** Does not require a session connection between sender and receiver. The sender simply starts sending packets (called datagrams) to the destination. This service does not have the reliability of the connection-oriented method, but it is useful for periodic burst transfers. Neither system must maintain state information for the systems that they send transmission to or receive transmission from. A connectionless network provides minimal services.

Protocol Stacks

OSI Reference Model

The ISO Open Systems Interconnections (OSI) reference model deals with connecting open networked systems. The key principles of the OSI are:

- layers are created when different levels of abstraction are needed
- the layer should perform a well-defined function
- the layer function should be chosen with international standards in mind
- layer boundaries should be chosen to minimize information flow between layers
- the number of layers should be: large enough to prevent distinct functions being thrown together, but small enough to prevent the whole architecture being unwieldy

Examples

HTTP

SSL

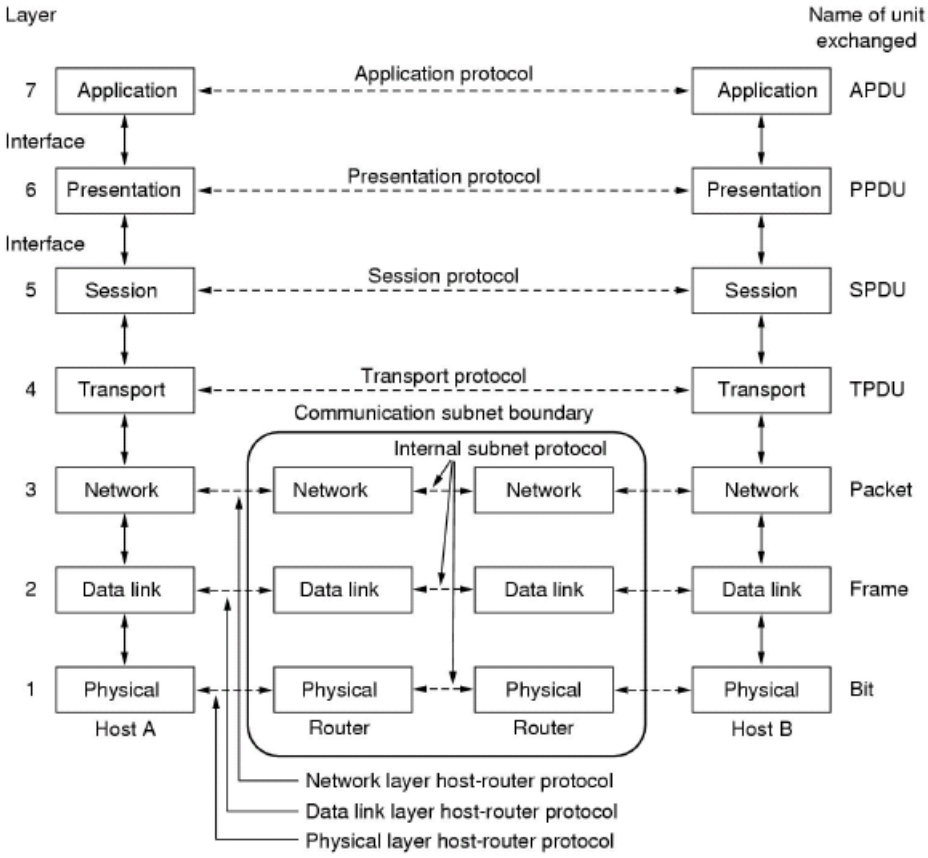
Fault tolerance

TCP/IP

ATM/VC

Ethernet MAC

2.4 GHz Radio



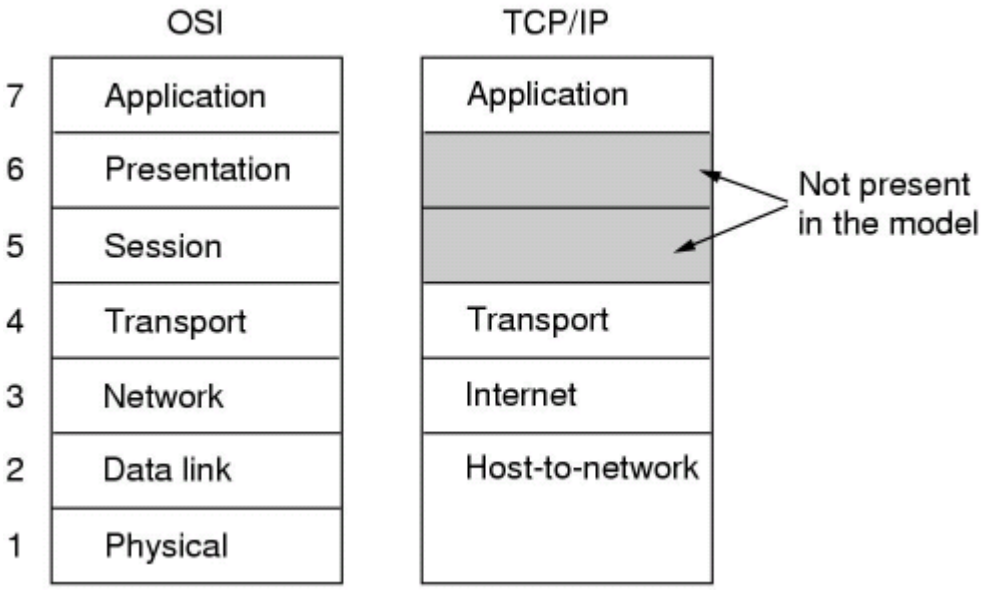
Layer	Description	Examples
Application (APP)	Protocols that are designed to meet the communication requirements of specific applications, often defining the interface to a service.	HTTP, FTP, SMTP, CORBA IIOP
Presentation (SECURITY)	Protocols at this level transmit data in a network representation that is independent of the representations used in individual computers, which may differ.	SSL, CORBA Data Representation
Session (ERRORS)	At this level, reliability and adaptation are performed, such as detection of failures and automatic recovery.	
Transport (TRANS)	This is the lowest level at which messages (rather than packets) are handled. Messages are addressed to communication ports attached to processes. Protocols in this layer be connection-orientated or connectionless.	TCP, UDP
Network (ROUTING)	Transfers data packets between computers in a specific network. In a WAN or an inter-network this involves generation of a route passing through routers. In a single LAN, no routing is required.	IP, ATM virtual circuits

Data link (DATA)	Responsible for transmission of packets between nodes that are directly connected by a physical link. In a WAN, transmission is between pairs of routers or between routers and hosts. In a LAN it is between a pair of hosts.	Ethernet MAC, ATM cell transfer, PPP
Physical (WIRES)	The circuits and hardware that drive a network. It transmits sequences of binary data by analogue signaling, using AM or FM of electrical signals (on cable circuits), light signals (on fiber optic circuits) or other EM signals (on radio or microwave circuits).	Ethernet based-band signaling, ISDN

Messages and headers combine to form packets - the bits that actually appear on the network.

TCP/IP Reference Model

The TCP/IP reference model originated from research in 1974 using the ARPANET (predecessor to the modern Internet). TCP/IP does not have session or presentation layers, which are only really needed in applications with specific requirements, and the data link and physical layers are combined to form one layer.



Protocols within the TCP/IP model include Telnet, FTP, SMTP and DNS at the application layers, TCP and UDP in the transport layer, IP in the network/Internet layer and ARPANET, SATNET, AX.25 and Ethernet in the physical/data-link layer.

The OSI and the TCP/IP models both have advantages and disadvantages. For example, the OSI model is well defined with the model before the protocols, whereas the TCP/IP model defined protocols first and then retrofitted a model on top. The number of layers is also different, although both have the network, transport and application layers the same. There are also different levels of

support for higher-level services - these are the key differences. OSI supports connectionless and connection-orientated communication in the network layer and connection orientated only in the transport layer, whereas TCP/IP supports connectionless in the network layer and both in the transport layer, so users are given an important choice.

OSI suffers from the standard criticism that can be applied to all ISO standards, which is that it was delivered too late, it is over complicated and developed from the viewpoint of telecoms. Additionally, the implementations of it are poor when compared to TCP/IP as implemented on UNIX. Additionally, politics were poor, as it was forced on the world, and was resisted in favor of UNIX and TCP/IP.

The TCP/IP model has the problem of only being able to support TCP/IP protocols, whilst the OSI model can include new protocols. There is very little to distinguish between the data link and physical layers, which do very different jobs. TCP/IP is the result of hacking by graduate students, so there is little design behind it.

Internetworking- bridges and routers

Bridges

Bridges act at the data-link layer and connect two or more network segments or LANs (however, they must be of the same type). They have an accept-and-forward strategy, and perform a basic level of routing. They usually do not alter a packet header.

Bridges do have the problem of not scaling, as the spanning tree algorithm does not scale, and having a single designated bridge can be a bottleneck. Additionally, it does not accommodate heterogeneity (bridges make use of frame headers, so they can only support networks with the same format for addresses). Additionally, different MTUs can cause problems.

Routers

Routers are capable of providing interconnects between different sorts of LANs and WANs.

Internet design and evolution:

The Internet is a global system of interconnected computer networks that use the standard Internet protocol suite (often called TCP/IP, although not all applications use TCP) to serve billions of users worldwide. It is a network of networks that consists of millions of private, public, academic, business, and government networks, of local to global scope, that are linked by a broad array of electronic, wireless and optical networking technologies. The Internet carries an extensive range of information resources and services, such as the interlinked hypertext documents of the World Wide Web (WWW) and the infrastructure to support email.

Internet is a short form of the technical term **internetwork**, the result of interconnecting computer networks with special gateways or routers. The Internet is also often referred to as the Net.

The terms Internet and World Wide Web are often used in everyday speech without much distinction. However, the Internet and the World Wide Web are not one and the same. The Internet establishes a global data communications system between computers. In contrast, the Web is one of the services communicated via the Internet. It is a collection of interconnected documents and other resources, linked by hyperlinks and URLs.

Unit 2

Network Design Styles

Distributed systems are often complex pieces of software of which the com-ponents are by definition dispersed across multiple machines. To master their complexity, it is crucial that these systems are properly organized. There are dif-ferent ways on how to view the organization of a distributed system, but an obvi-ous one is to make a distinction between the logical organization of the collection of software components and on the other hand the actual physical realization. The organization of distributed systems is mostly about the software com-ponents that constitute the system. These software architectures tell us how the various software components are to be organized and how they should interact. In this chapter we will first pay attention to some commonly applied approaches toward organizing (distributed) computer systems.

Several styles have by now been identified, of which the most important ones for distributed systems are:

- Layered architectures
- Object-based architectures
- Data-centered architectures
- Event-based architectures

The basic idea for the **layered style** is simple: components are organized in a layered fashion where a component at layer *L;* is allowed to call components at the underlying layer *Li:«,* but not the other way around, as shown in Fig. 2-I(a). A key observation is that control generally flows from layer to layer: requests go down the hierarchy whereas the results flow upward.

A far looser organization is followed in **object-based architectures**, which are illustrated in Fig. 2-1(b). In essence, each object corresponds to what we have defined as a component, and these components are connected through a (remote) procedure call mechanism. Not surprisingly, this software architecture matches the client-server system architecture we described above.

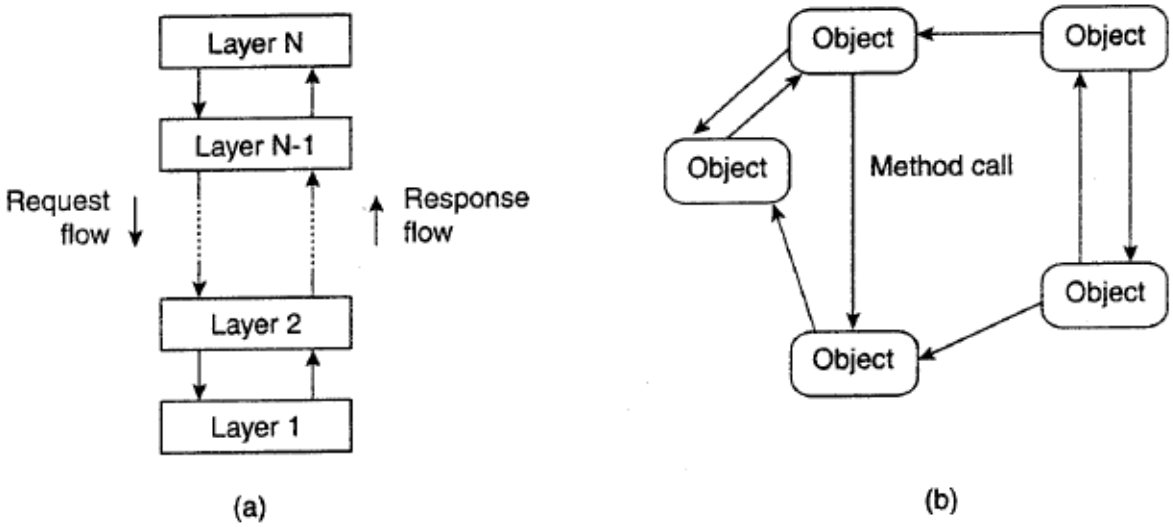


Figure 2-1. The (a) layered and (b) object-based architectural style.

Data-centered architectures evolve around the idea that processes communicate through a common (passive or active) repository. It can be argued that for distributed systems these architectures are as important as the layered and objectbased architectures. For example, a wealth of networked applications have been developed that rely on a shared distributed file system in which virtually all communication takes place through files.

In **event-based architectures**, processes essentially communicate through the propagation of events, which optionally also carry data, as shown in Fig. 2-2(a). For distributed systems, event propagation has generally been associated with what are known as publish/subscribe systems. The basicidea is that processes publish events after which the middleware ensures that onlythose processes that subscribed to those events will receive them. The main advantage of event-based systems is that processes are loosely coupled. In principle, they need not explicitly refer to each other.

Event-based architectures can be combined with data-centered architectures, yielding what is also known as shared data spaces. The essence of shared data spaces is that processes are now also decoupled in time: they need not both be active when communication takes place. What makes these software architectures important for distributed systems is that they all aim at achieving (at a reasonable level) distribution transparency. However, as we have argued, distribution transparency requires making trade-offs between performance, fault tolerance, ease-of-programming, and so on.

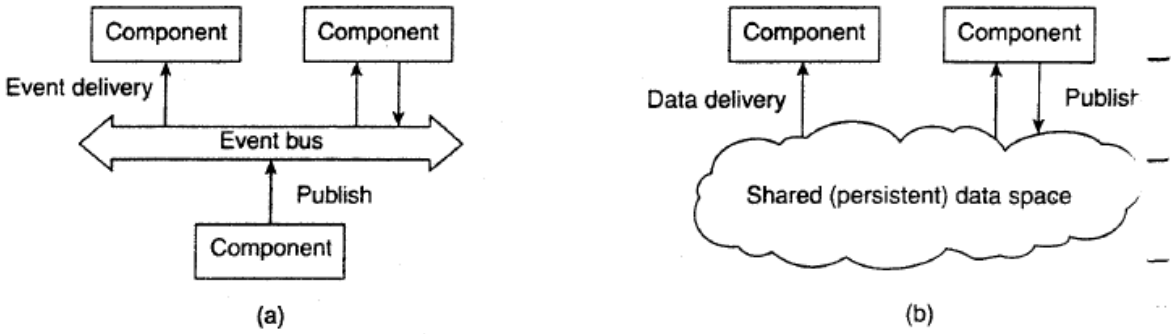


Figure 2-2. The (a) event-based and (b) shared data-space architectural style.

Network Designs

Centralized Designs

Despite the lack of consensus on many distributed systems issues, there is one issue that many researchers and practitioners agree upon: thinking in terms of *cli-ents* that request services from *servers* helps us understand and manage the com-plexity of distributed systems and that is a good thing.

In the basic client-server model, processes in a distributed system are divided into two (possibly overlapping) groups. A server is a process implementing a specific service, for example, a file system service or a database service. A client is a process that requests a service from a server by sending it a request and subsequently waiting for the server's reply. This client-server interaction, also known as request-reply behavior is shown in Fig. 2-3

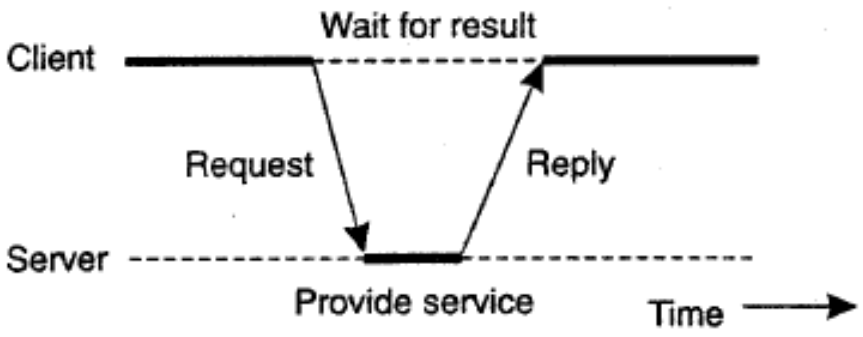


Figure 2-3. General interaction between a client and a server.

Communication between a client and a server can be implemented by means of a simple connectionless protocol when the underlying network is fairly reliable as in many local-area networks. In these cases, when a client requests a service, it simply packages a message for the server, identifying the service it wants, along with the necessary input data. The message is then sent to the server. The latter, in turn, will always wait for an incoming request, subsequently process it, and package the results in a reply message that is then sent to the client.

Application Layering

The client-server model has been subject to many debates and controversies over the years. One of the main issues was how to draw a clear distinction between a client and a server. Not surprisingly, there is often no clear distinction. For example, a server for a distributed database may continuously act as a client because it is forwarding requests to different file servers responsible for implementing the database tables. In such a case, the database server itself essentially does no more than process queries. However, considering that many client-server applications are targeted toward supporting user access to databases, many people have advocated a distinction between the following three levels, essentially following the layered architectural style we discussed previously:

1. The user-interface level
2. The processing level
3. The data level

The user-interface level contains all that is necessary to directly interface with the user, such as display management. The processing level typically contains the applications. The data level manages the actual data that is being acted on.

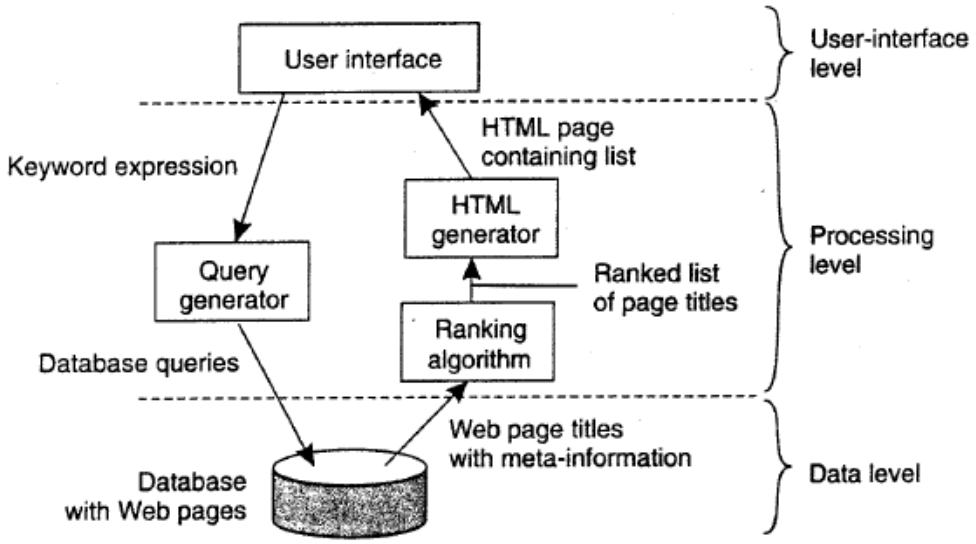


Figure 2-4. The simplified organization of an Internet search engine into three different layers.

Consider an Internet search engine. Ignoring all the animated banners, images, and other fancy window dressing, the user interface of a search engine is very simple: a user types in a string of keywords and is subsequently presented with a list of titles of Web pages. The back end is formed by a huge database of Web pages that have been prefetched and indexed. The core of the search engine is a program that transforms the user's string of keywords into one or more database queries. It subsequently ranks the results into a list, and transforms that list into a series of HTML pages. Within the client-server model, this information retrieval part is typically placed at the processing level. Fig. 2-4 shows this organization.

Multitiered Architecture

The distinction into three logical levels as discussed so far, suggests a number of possibilities for physically distributing a client-server application across several machines. The simplest organization is to have only two types of machines:

1. A client machine containing only the programs implementing (part of) the user-interface level

2. A server machine containing the rest, that is the programs implementing the processing and data level.

In this organization everything is handled by the server while the client is essentially no more than a dumb terminal, possibly with a pretty graphical interface. There are many other possibilities, of which we explore some of the more common ones in this section. One approach for organizing the clients and servers is to distribute the programs in the application layers of the previous section across different machines, as shown in Fig. 2-5

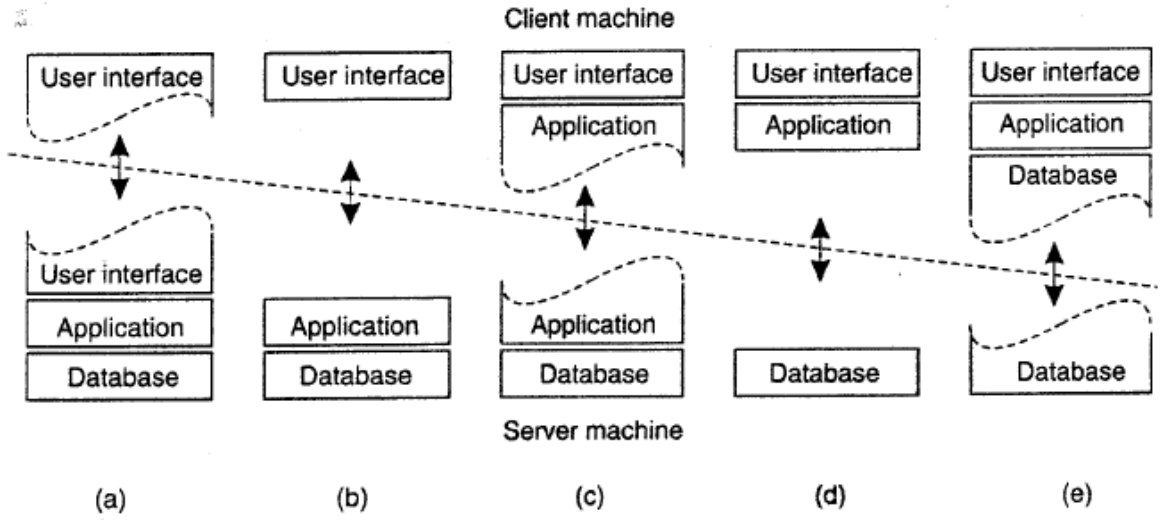


Figure 2-5. Alternative client-server organizations (a)-(e).

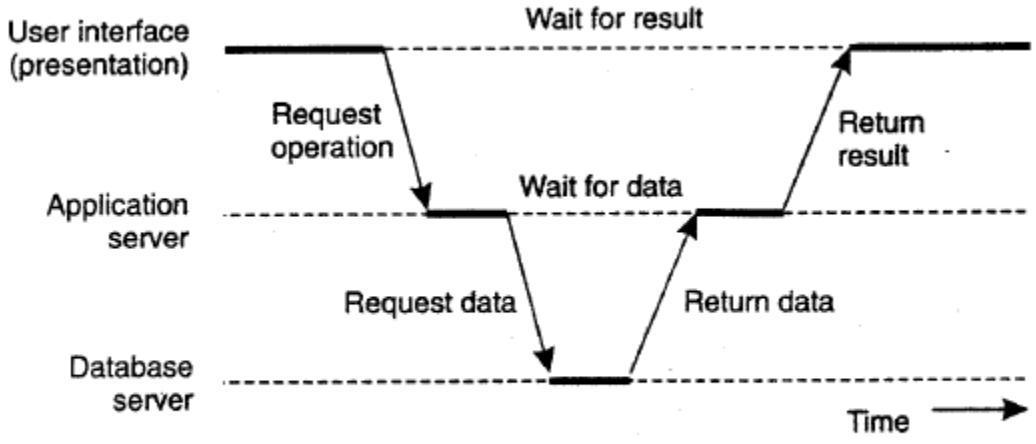


Figure 2-6. An example of a server acting as client.

In this architecture, programs that form part of the processing level reside on a separate server, but may additionally be partly distributed across the client and server machines. A typical example of where a three-tiered architecture is used is in transaction processing. As we discussed in Chap. 1, a separate process, called the transaction processing monitor, coordinates all transactions across possibly different data servers.

Decentralized Architecture

Multitiered client-server architectures are a direct consequence of dividing ap-plications into a user-interface, processing components, and a data level. The dif-ferent tiers correspond directly with the logical organization of applications. In many business environments, distributed processing is equivalent to organizing a client-server application as a multitiered architecture. We refer to this type of dis-tribution as vertical distribution. The characteristic feature of vertical distribu-tion is that it is achieved by placing *logically* different components on different machines. The term is related to the concept of vertical fragmentation as used in distributed relational databases, where it means that tables are split column-wise, and subsequently distributed across multiple machines.

Structured Peer-to-Peer Architectures

In a structured peer-to-peer architecture, the overlay network is constructed using a deterministic procedure. By far the most-used procedure is to organize the processes through a distributed hash table (DHT). In a DHT -based system, data items are assigned a random key from a large identifier space, such as a 128-bit or 160-bit identifier. Likewise, nodes in the system are also assigned a random number from the same identifier space.

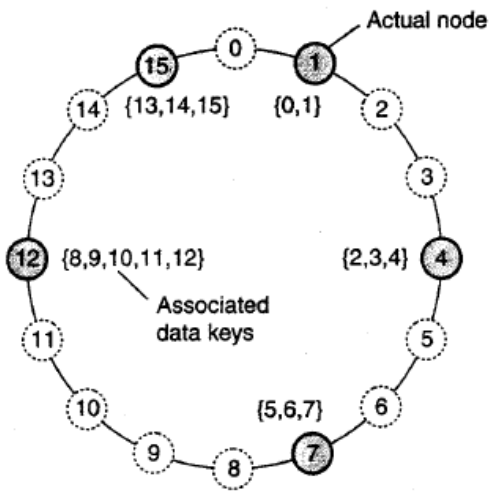


Figure 2-7. The mapping of data items onto nodes in Chord.

Unstructured Peer-to- Peer Architectures

Unstructured peer-to-peer systems largely rely on randomized algorithms for constructing an overlay network. The main idea is that each node maintains a list of neighbors, but that this list is constructed in a more or less random way. Likewise, data items are assumed to be randomly placed on nodes. As a consequence, when a node needs to locate a specific data item, the only thing it can effectively do is flood the network with a search query.

Topology Management

Although it would seem that structured and unstructured peer-to-peer systems form strict independent classes, this 'need actually not be case [see also Castro et al. (2005)]. One key observation is that by carefully exchanging and selecting entries from partial views, it is possible to construct and maintain specific topologies of overlay networks. This topology management is achieved by adopting a twolayered approach, as shown in Fig. 2-10.

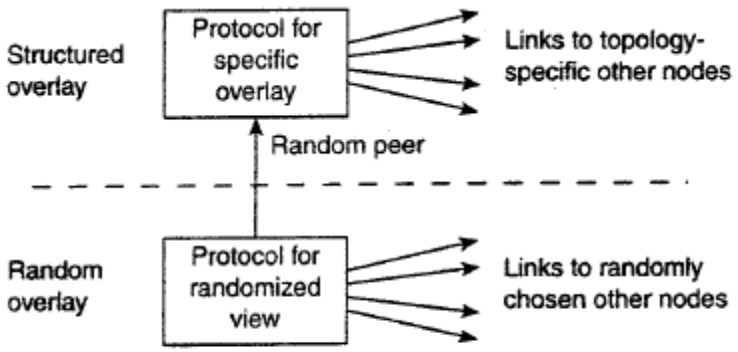


Figure 2·10. A two-layered approach for constructing and maintaining specific overlay topologies using techniques from unstructured peer-to-peer systems.

Interoperability

Interoperability is the ability of a system or a product to work with other systems or products without special effort on the part of the customer. Interoperability becomes a quality of increasing importance for information technology products as the concept that "The network is the computer" becomes a reality. For this reason, the term is widely used in product marketing descriptions.

Products achieve interoperability with other products using either or both of two approaches:

- By adhering to published interface standards
- By making use of a "broker" of services that can convert one product's interface into another product's interface "on the fly"

A good example of the first approach is the set of standards that have been developed for the World Wide Web. These standards include TCP/IP, Hypertext Transfer Protocol, and HTML. The second kind of interoperability approach is exemplified by the Common Object Request Broker Architecture (CORBA) and its Object Request Broker (ORB).

Compatibility is a related term. A product is compatible with a standard but interoperable with other products that meet the same standard (or achieve interoperability through a broker).

Client-Server Interoperability

Reusability of servers is a critical issue for both users and software manufacture due to the high cost of software writing. This issue could be easily resolved in a homogeneous environment because accessing mechanisms of clients may be made compatible with software interfaces, with static compatibility specified by types and dynamic compatibility by protocols.

There are two major mechanisms for interoperation:

1. Interface standardisation: the objective of this mechanism is to map client and server interfaces to a common representation.
The advantages of this mechanism are:
(i) it separates communication models of clients from those of servers, and
(ii) it provides scalability, since it only requires m + n mappings, where m and n are the number of clients and servers, respectively. The disadvantage of this mechanism is that it is closed.

2. Interface bridging: the objective of this mechanism is to provide a two-way mapping between a client and a server. The advantages of this mechanism are:
(i) openness, and
(ii) flexibility — it can be tailored to the requirements of a given client and server pair. However, this mechanism does not scale as well as the interface standardisation mechanism, as it requires m * n mappings.

Openness

Determines whether the system can be extended in various ways without disrupting existing system and services

Hardware extensions (adding peripherals, memory, communication interfaces..)

Software extensions (Operating System features, Communication protocols)

Mainly achieved using published interfaces, standardization
Great example of a distributed, standards-focused effort

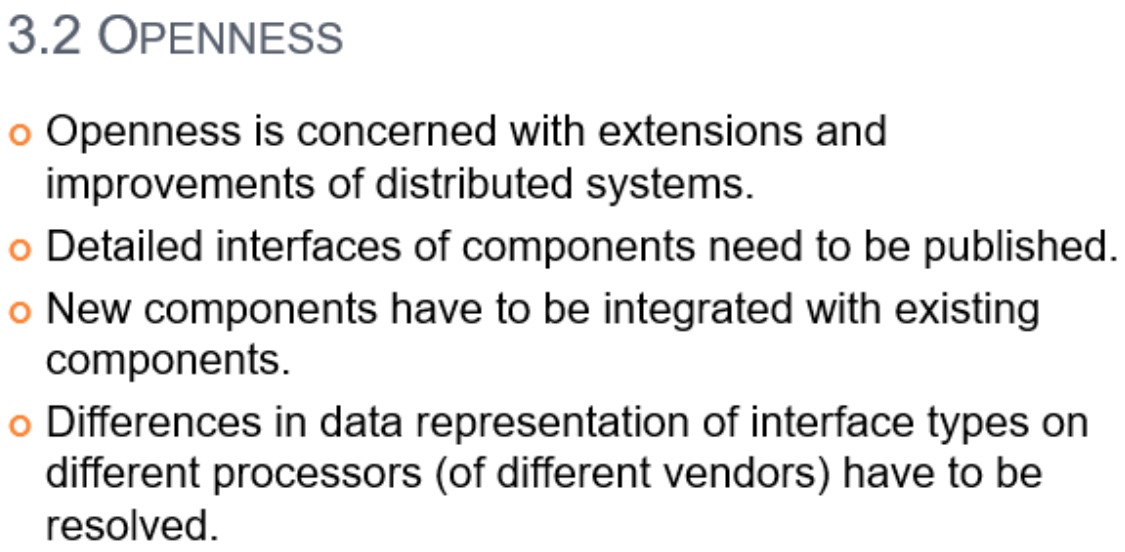
Open Distributed Systems

Are characterized by the fact that their key interfaces are published

 e.g., HTTP Protocol,

Based on the provision of a uniform interprocess communication mechanism and published interfaces for access to shared resources

Can be constructed from heterogeneous hardware and software.



The Client Server Model

A distributed computing system is a set of application and system programs, and data dispersed across a number of independent personal computers connected by a communication network. In order to provide requested services to users the system and relevant application programs must be executed. Because services are provided as a result of executing programs on a number of computers with data stored on one or more locations, the whole computing activity is called distributed computing.

Basic Concepts

The problem is how to formalise the development of distributed computing. The above shows that the main issue of distributed computing is programs in execution, which are called processes. The second issue is that these processes cooperate or compete in order to provide the requested services. This means that these processes are synchronised.

A natural model of distributed computing is the client-server model, which is able to deal with the problems generated by distribution, could be used to describe computation processes and their behaviour when providing services to users, and allows design of system and application software for distributed computing systems.

According to this model there are two processes, the client, which requests a service from another process, and the server, which is the service provider. The server performs the requested service and sends back a response. This response could be a processing result, a confirmation of completion of the requested operation or even a notice about a failure of an operation.

From the user's point of view a distributed computing system can provide the following services: printing, electronic mail, file service, authentication, naming, database service and computing service. These services are provided by appropriate servers. Because of the restricted number of servers (implied by a restricted number of resources on which these servers were implemented), clients compete for these servers.

An association between this abstract model and its physical implementation is shown in Figure 2.1. In particular the basic items of the model: the client and server, and request and response are shown. In this case, the client and server processes execute on two different computers. They communicate at the virtual (logical) level by exchanging requests and responses. In order to achieve this virtual communication, physical messages are sent between these two processes. This implies that operating systems of computers and a communication system of a distributed computing system are actively involved in the service provision.

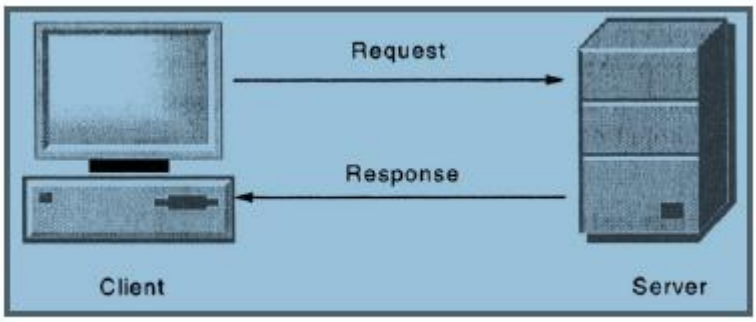


Figure 2.1: The basic client-server model

A more detailed client-server model has three components:

Service: A service is a software entity that runs on one or more machines. It provides an abstraction of a set of well-defined operations in response to applications' requests.

Server: A server is an instance of a particular service running on a single machine.

Client: A client is a software entity that exploits services provided by servers. A client can but does not have to interface directly with a human user.

There are three major problems of the client-server model:

The first is due to the fact that the control of individual resources is centralised in a single server. This means that if the computer supporting a server fails, then that element of control fails. Such a solution is not tolerable if a control function of a server is critical to the operation of the system (e.g., a name server, a file server, an authentication server). Thus, the reliability and availability of an operation depending on multiple servers is a product of reliability of all computers and devices, and communication lines.

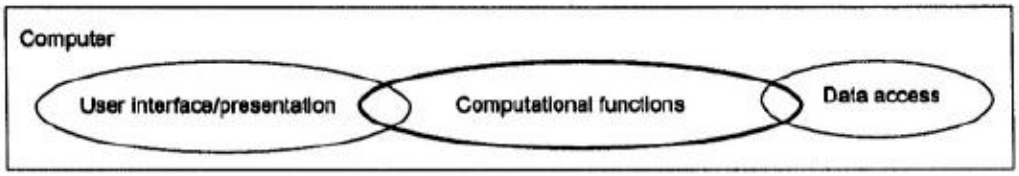
The second problem is that each single server is a potential bottleneck. The problem is exacerbated as more computers with potential clients are added to the system.

The third problem arises when multiple implementations of similar functions are used to improve the performance of a client-server based system because of a need to maintain consistency. Furthermore, this increases the total costs of a distributed computing system.

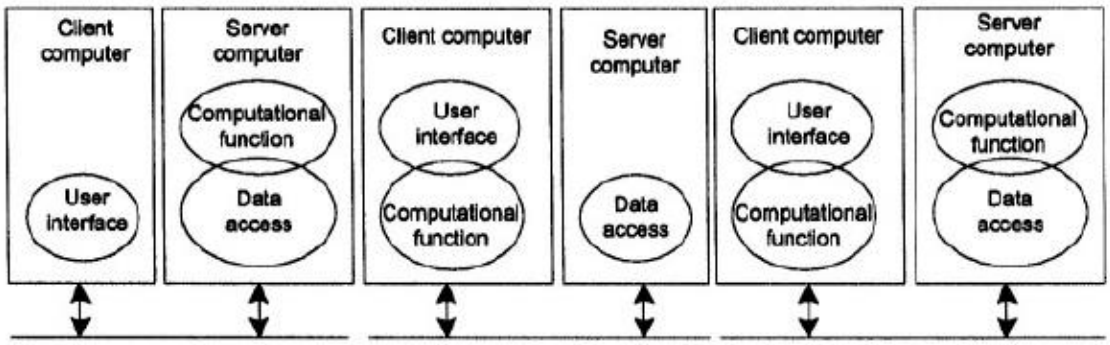
The Three-Tier Client-Server Architecture

Agents and servers acting as clients can generate different architectures of distributed computing systems. The three-tier client-server architecture extends the basic client-server model by adding a middle tier to support the application logic and common services. In this architecture, a distributed application consists of the following three types of components:

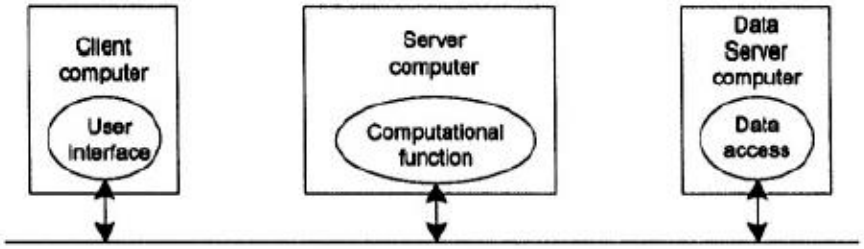
User interface and presentation processing. These components are responsible for accepting inputs and presenting the results. They belong to the client tier;
Computational function processing. These components are responsible for providing transparent, reliable, secure, and efficient distributed computing. They are also responsible for performing necessary processing to solve a particular application problem. We say these components belong to the application tier;
Data access processing. These components are responsible for accessing data stored on external storage devices (such as disk drives). They belong to the back-end tier.



(a) centralised configuration



(b) two-tier configurations



(c) three-tier configuration

Workstations

A workstation is a special computer designed for technical or scientific applications. Intended primarily to be used by one person at a time, they are commonly connected to a local area network and run multi-user operating systems. The term workstation has also been used loosely to refer to everything from a mainframe computer terminal to a PC connected to a network, but the most common form refers to the group of hardware offered by several current and defunct companies such as Sun Microsystems, Silicon Graphics, Apollo Computer, DEC, HP, NeXT and IBM which opened the door for the 3D graphics animation revolution of the late 1990s.

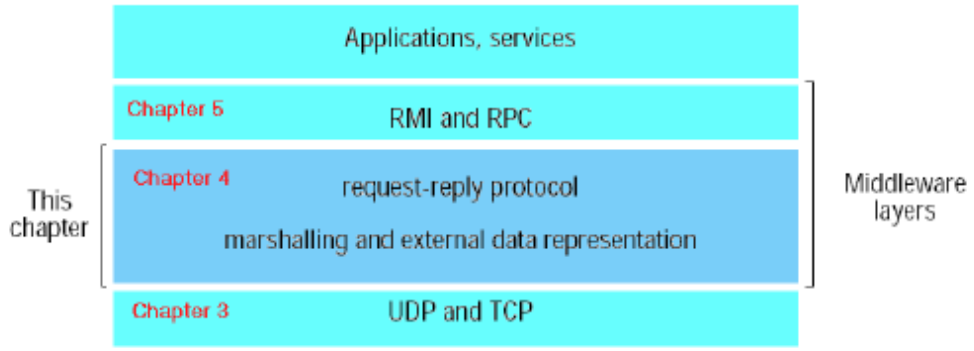
Workstations offered higher performance than mainstream personal computers, especially with respect to CPU and graphics, memory capacity, and multitasking capability. Workstations were optimized for

the visualization and manipulation of different types of complex data such as 3D mechanical design, engineering simulation (e.g., computational fluid dynamics), animation and rendering of images, and mathematical plots. Typically, the form factor is that of a desktop computer, consist of a high resolution display, a keyboard and a mouse at a minimum, but also offer multiple displays, graphics tablets, 3D mice (devices for manipulating 3D objects and navigating scenes), etc. Workstations were the first segment of the computer market to present advanced accessories and collaboration tools.

Unit 3

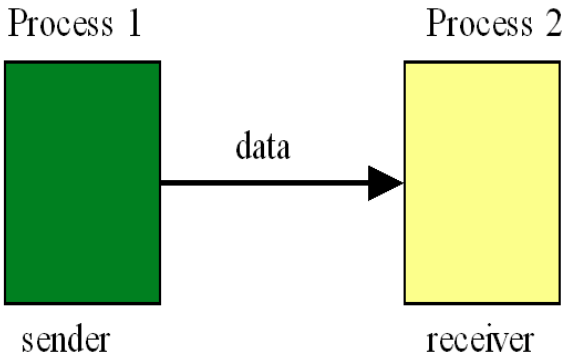
- **Inter-process Communication.**

Figure 4.1 Middleware layers



Communicating entities (what is communicating)		*Communication paradigms (how they communicate)*		
System-oriented entities	*Problem-oriented entities*	*Interprocess communication*	*Remote invocation*	*Indirect communication*
Nodes	Objects	Message passing	Request-reply	Group communication
Processes	Components	Sockets	RPC	Publish-subscribe
	Web services	Multicast	RMI	Message queues
				Tuple spaces
				DSM

- **The API for the Internet protocols**



Characteristics of IPC

Communication operations (defined in terms of destinations & message)
• Send
• Receive
Synchronous

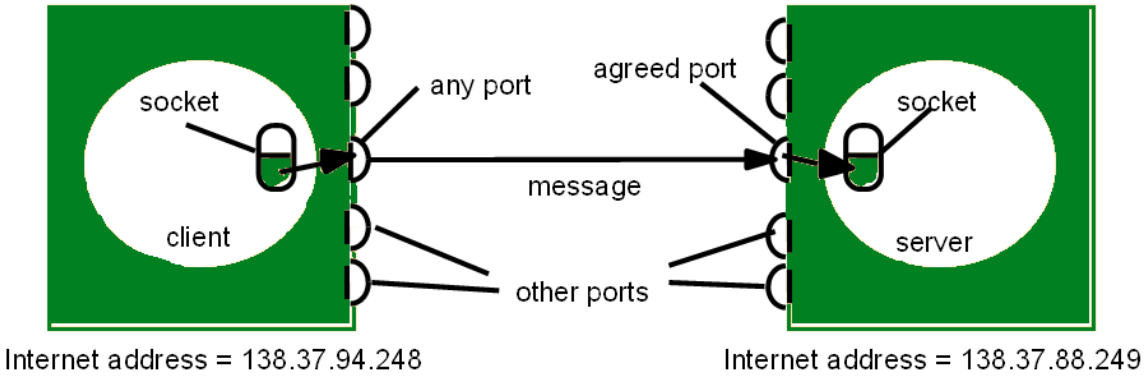
• blocking send – sending process blocked until corresponding receive issued
• blocking receive – receiving process blocked until a message arrives

Asynchronous

• non-blocking send – sending process proceeds with message being copied to local buffer
• blocking/non-blocking receive

Message Destination
• IP address & port
• Location transparency
• Send directly to processes
• Multicast to a group of process

Reliability- validity & integrity
Ordering- message to be delivered in sender order
Sockets – Both UDP and TCP use the socket abstraction, with provides an endpoint for communication between processes.



UDP datagram communication

•Message size (up to 216 bytes)
•Blocking: non-blocking send, blocking receive
•Timeouts- when a receive operation waiting indefinitely is inappropriate
•Receive from any
•Failure Model (Omission, Ordering)
•Use of UDP (eg. DNS, Less overhead- state info., extra message, latency)

Java API for UDP datagram

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

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

```
{System.out.println("Socket:"+e.getMessage());
}catch(IOException e){System.out.println("IO:"+e.getMessage();}
}}
```

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

```
import java.net.*
import java.io.*
public class UDPServer{
public static void main(String args[]){
try{
DatagramSocket aSocket = new DatagramSocket(6789);
byte[] buffer = new byte[1000];
while(true){
DatagramPacket request =
new datagramPacket(buffer, buffer.length);
aSocket.receive(request);
DatagramPacket reply =
new DatagramPacket(request.getData(),
request.getLength(), request.getAddress(),
request.getPort());
aSocket.send(reply);
}
}catch(SocketException e)
{System.out.println("Socket:" + e.getMessage());
}catch(IOException e)
{System.out.println("IO:" + e.getMessage());}
}
}
```

TCP stream communication

Characteristics

•Message size - Unlimited
•Lost Messages - Acknowledgement
•Flow Control- matching speeds of processes reading from & writing to stream
•Message duplication and order- message identifiers
•Message destination- establishing connection before transmission

Outstanding Issues

•Matching of data items
•Blocking- TCP flow control
•Threads
•Failure model – integrity(checksum, sequence nos.), validity (timeouts)
•Use of TCP: http, ftp, telnet, smtp

▪ External data representation

•Info in running programs represented as data structures
•Info in message represented as sequence of bytes
•Flattening of data structures (into sequence of bytes) before transmission
•Different ways to represent int, float, char...
•Byte ordering for integer- big & little - endian
•Enabling exchange of data values
    •Convert values into standard external data representation
    •Send in sender's format and indicates what format, receivers translate if necessary.
•External data representation- an agreed standard for the representation of data structures & primitives values
•Marshalling
    •Taking a collection of data items & assembling them into a form suitable for transmission in a message
    •Translation of structured data items & primitive values into external data representation
•Demarshalling
    •Generation of primitive values from external data representation
    •Rebuilding of data structures

Three Approaches to External Data Representation

1. •CORBA
2. •Java's object serialization
3. •XML

1. CORBA Common Data Representation (CRD) message

•Primitive types- short, long, double, char, boolean…
•Constructed types

Type	*Representation*
sequence	length (unsigned long) followed by elements in order
string	length (unsigned long) followed by characters in order (can also can have wide characters)
array	array elements in order (no length specified because it is fixed)
struct	in the order of declaration of the components
enumerated	unsigned long (the values are specified by the order declared)
union	type tag followed by the selected member

CORBA CDR message

•CORBA IDL complier generates marshalling and unmarshalling routines

•Types of data str. & basic data items described in CORBA Interface Description Language (IDL)

•Structure with string, unsigned long

index in sequence of bytes	← 4 bytes →	*notes on representation*
0–3	5	*length of string*
4–7	"Smit"	*'Smith'*
8–11	"h___"	
12–15	6	*length of string*
16–19	"Lond"	*'London'*
20-23	"on__"	
24–27	1934	*unsigned long*

The flattened form represents a *Person* struct with value: {'Smith', 'London', 1934}

2. Java object serialization

•Serialization- flattening of object or a connected set of objects into a serial form to be transmitted or stored on the disk
•De-seralization – restoring the state of an object or a set of objects from their serialized form
• No knowledge of the types of objects in the serialized form during de-seralization
•Info about the class of each object is in serialized form – class name, version no.

```
public class Person implements Serializable {
        private String name;
         private String place;
         private int year;

         public Person(String aName, String aPlace, int aYear){
            name = aName;
            place = aPlace;
            year = aYear;
         }
}
```

Serialized values				*Explanation*
Person	8-byte version number		h0	*class name, version number*
3	int year	java.lang.String name:	java.lang.String place:	*number, type and name of instance variables*
1934	5 Smith	6 London	h1	*values of instance variables*

The true serialized form contains additional type markers;
h0 and h1 are handles/references to other objects within the serialized form

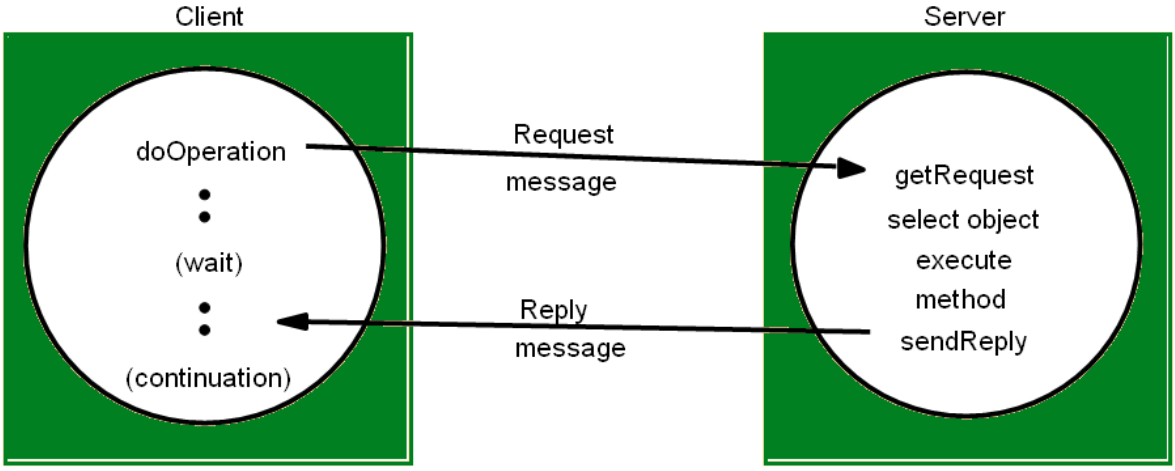
3. Extensible markup language (XML)

•User-defined tags
•Different Apps agree on different set of tags
•e.g. REST, Simple Object Access Protocol (SOAP) for web serves, tags are published
•Tags are in plain text
Illustration of the use of a namespace in the person structure

```
<person pers:id="123456789" xmlns:pers = "http://www.cdk4.net/person">
    <pers:name> Smith </pers:name>
    <pers:place> London </pers:place >
    <pers:year> 1934 </pers:year>
</person>
```

- **Client-server communication**

•Synchronous (client waits for a reply)
•Asynchronous (client doesn't wait)



Request/reply protocol

Request-reply message structure

messageType	*int (0=Request, 1= Reply)*
requestId	*int*
objectReference	*RemoteObjectRef*
methodId	*int or Method*
arguments	*// array of bytes*

UDP – Failure Handling

• Timeout
• Discard of duplicates
• Lost replies - idempotent operations(performed repetitively)
• History- referring a str. containing a transmitted reply record

RPC exchange protocols

• Request (R)
• Request/Reply (RR)
• Request/Reply/Acknowledge Reply(RRA)

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

TCP implementation of Request/Reply Protocols

- HTTP example – allows persistent connection
- HTTP methods – GET, HEAD, POST, PUT, DELETE, OPTIONS,TRACE
- HTTP Message contents

HTTP Request message:

method	*URL or pathname*	*HTTP version*	*headers*	*message body*
GET	//www.dcs.qmw.ac.uk/index.html	HTTP/ 1.1		

24

HTTP Reply message:

HTTP version	*status code*	*reason*	*headers*	*message body*
HTTP/1.1	200	OK		resource data

Group Communication

- Multicast Operation- a single message sent from one Process to all members of a group
- Helpful in attending DS with
    - High fault tolerance based on replicated services
    - Can locate servers
    - Better performance thru replication
    - Propagation of event notifications

IP Multicast

- Multicast Group
- Multicast Routers
- Multicast Address Allocation

IP Multicast – Failure Models

- Same as UDP – no guarantee of delivery
- Effects:
- Replicated services – all or none msg receipt
- Discovery servers – repeat requests
- Replicated Data
- Event Notifications – app determines qualities

Unix Inter-process Communication

- IPC in Unix
    - Layered on TCP and UDP protocol
    - Socket System call – binding to an address
    - Message destinations = socket address
        - Msg queues at sending socket
        - Networking protocol transmits msg
        - Msg queues at receiving socket
        - Receiving process makes system call to receive
        - msg.

Datagram Communication (UDP)

- Sockets identified in each communication
    - Socket call
    - Bind call
    - Send to call
    - Receive from call

Stream Communication (TCP)

- One server is ‘listening’ for requests
    - Socket call for stream socket + bind + listen
    - Accept call, create new socket
    - Client process issues socket, connect
    - Both use write/read
    - Both close when communication is finished

- **Summary**
    - UDP vs. TCP
    - Marshalling data – CORBA, Java, XML
    - Request/Reply Protocols
    - Multicast Messages

Unit 4

Communication between distributed objects

The communication in distributed object is done by various middleware language like RMI (remote invocation method), CORBA (common object request broker). Invoking a method on a remote object is known as RMI or remote invocation, and is the object oriented programming analog of an RPC. Distributed object communication realizes communication between distributed objects. The widely used approach on how to implement the communication channel is realized by using stub and skeletons.

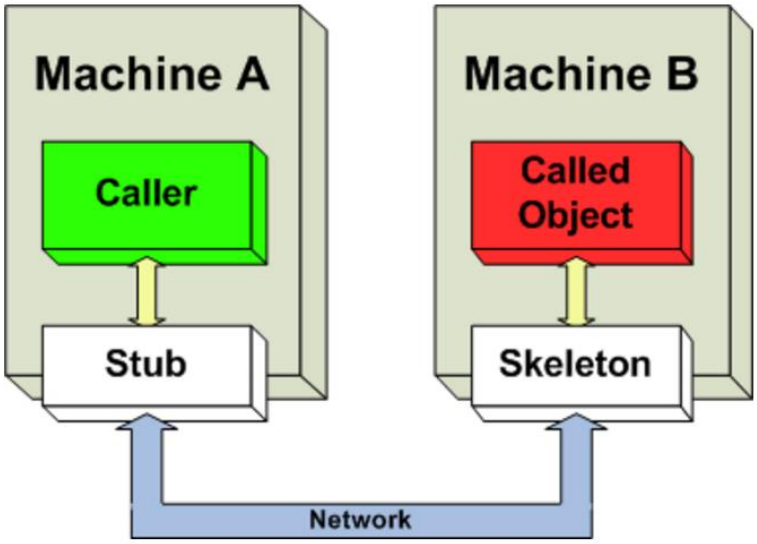


Fig: Communication between distributed objects

In RMI, a stub is defined by the programmer as an interface then the stub passes caller arguments over the network to the server skeleton. The skeleton then passes received data to the called object, waits for a response and returns the result to the client stub.

Stub

The client side object in distributed object communication is known as a stub or proxy. The stub acts as a gateway for client side objects and all outgoing requests to server-side objects that routed through it. The stub wraps client object functionality & by adding the network logic ensures the reliable communication channel between client & server. The stub can be written up manually or generated automatically depending on chosen communication protocol.

Skeleton

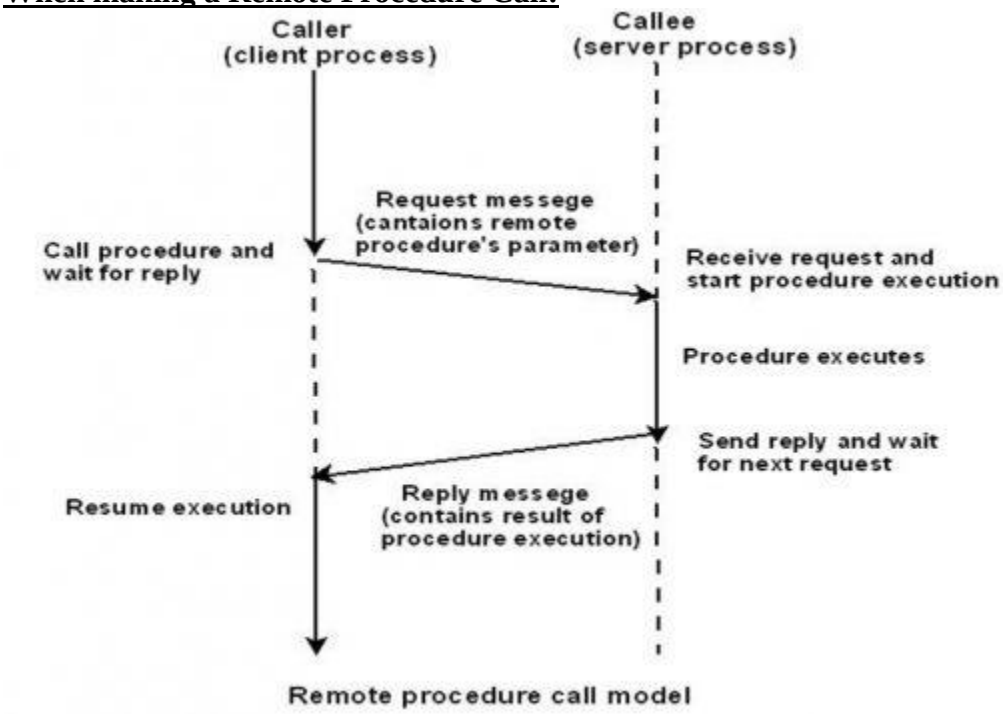
The server side object participating in distributed object communication is known as skeleton. A skeleton act as the gateway for server side objects & all incoming client's requests are routed through it. The skeleton wraps server object functionality & exposes it to the clients, moreover by adding the network logic ensures the reliable communication channel between clients & server.

Skeletons can be written up manually or generated automatically depending on chosen communication protocol.

Remote Procedure Call

Remote Procedure Call (RPC) is a powerful technique for constructing distributed, client-server based applications. It is based on extending the conventional local procedure calling, so that the called procedure need not exist in the same address space as the calling procedure. The two processes may be on the same system, or they may be on different systems with a network connecting them.

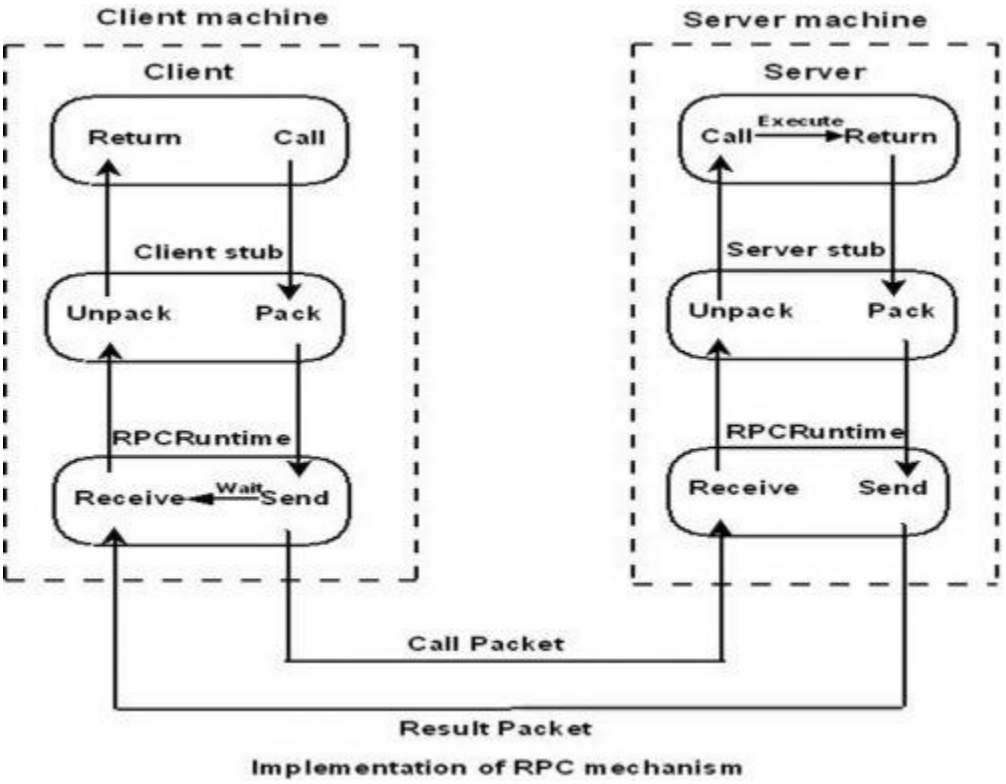
When making a Remote Procedure Call:



Remote procedure call model

1. The calling environment is suspended, procedure parameters are transferred across the network to the environment where the procedure is to execute, and the procedure is executed there.
2. When the procedure finishes and produces its results, its results are transferred back to the calling environment, where execution resumes as if returning from a regular procedure call.
NOTE: RPC is especially well suited for client-server **(e.g. query-response)** interaction in which the flow of control **alternates between the caller and callee**. Conceptually, the client and server do not both execute at the same time. Instead, the thread of execution jumps from the caller to the callee and then back again.

Working of RPC



Implementation of RPC mechanism

The following steps take place during a RPC:

1. A client invokes a **client stub procedure**, passing parameters in the usual way. The client stub resides within the client's own address space.

2. The client stub **marshalls(pack)** the parameters into a message. Marshalling includes converting the representation of the parameters into a standard format, and copying each parameter into the message.

3. The client stub passes the message to the transport layer, which sends it to the remote server machine.

4. On the server, the transport layer passes the message to a server stub, which **demarshalls(unpack)** the parameters and calls the desired server routine using the regular procedure call mechanism.

5. When the server procedure completes, it returns to the server stub **(e.g., via a normal procedure call return)**, which marshalls the return values into a message. The server stub then hands the message to the transport layer.

6. The transport layer sends the result message back to the client transport layer, which hands the message back to the client stub.

7. The client stub demarshalls the return parameters and execution returns to the caller.

RPC ISSUES

- **Issues that must be addressed:**

1. RPC Runtime: RPC run-time system, is a library of routines and a set of services that handle the network communications that underlie the RPC mechanism. In the course of an RPC call, client-side and server-side run-time systems' code handle **binding, establish communications over an appropriate protocol, pass call data between the client and server, and handle communications errors.**

2. Stub: The function of the stub is to **provide transparency to the programmer-written application code**.

On the client side, the stub handles the interface between the client's local procedure call and the run-time system, marshaling and unmarshaling data, invoking the RPC run-time protocol, and if requested, carrying out some of the binding steps.

On the server side, the stub provides a similar interface between the run-time system and the local manager procedures that are executed by the server.

3. Binding: How does the client know who to call, and where the service resides?

The most flexible solution is to use dynamic binding and find the server at run time when the RPC is first made. The first time the client stub is invoked, it contacts a name server to determine the transport address at which the server resides.

Binding consists of two parts:

- Naming:

Remote procedures are named through interfaces. **An interface uniquely identifies a particular service, describing the types and numbers of its arguments**. It is similar in purpose to a type definition in programming languages.

- Locating:

Finding the transport address at which the server actually resides. Once we have the transport address of the service, we can send messages directly to the server.

A Server having a service to offer exports an interface for it. Exporting an interface registers it with the system so that clients can use it.

A Client must import an (exported) interface before communication can begin.

ADVANTAGES

1. RPC provides **ABSTRACTION** i.e message-passing nature of network communication is hidden from the user.

2. RPC often omits many of the protocol layers to improve performance. Even a small performance improvement is important because a program may invoke RPCs often.

3. RPC enables the usage of the applications in the distributed environment, not only in the local environment.

4. With RPC code re-writing / re-developing effort is minimized.

5. Process-oriented and thread oriented models supported by RPC.

Remote Object Invocation

Remote method invocation (RMI) is a distributed object technology developed by Sun for the Java programming language. It is available as part of the core Java application programming interface (API) where the object interfaces are defined as Java interfaces and use object serialization.

RMI permits Java methods to refer to a remote object and invoke methods of the remote object. The remote object may reside on another Java virtual machine, the same host or on completely different hosts across the network. RMI marshals and unmarshals method arguments through object serialization and supports dynamic downloading of class files across networks.

RMI architecture extends the robustness and safety of Java architecture to the distributed computing world. RMI allows the that code defines and implements the behavior to remain on different Java virtual machines. Remote services in RMI are coded using a Java interface where the implementation is coded in a class. In the first class, implementation of the behavior runs on the server. The second class runs on the client and acts as a proxy for the remote service.

RMI is implemented as three layers:

- A stub program in the client side of the client/server relationship, and a corresponding skeleton at the server end. The stub appears to the calling program to be the program being called for a service. (Sun uses the term *proxy* as a synonym for stub.)
- A Remote Reference Layer that can behave differently depending on the parameters passed by the calling program. For example, this layer can determine whether the request is to call a single remote service or multiple remote programs as in a multicast.
- A Transport Connection Layer, which sets up and manages the request.

A single request travels down through the layers on one computer and up through the layers at the other end.

The commonalities between RMI and RPC are as follows:

- They both support programming with interfaces, with the resultant benefits that stem from this approach
- They are both typically constructed on top of request-reply protocols and can offer a range of call semantics such as at-least-once and at-most-once.
- They both offer a similar level of transparency – that is, local and remote calls employ the same syntax but remote interfaces typically expose the distributed nature of the underlying call, for example by supporting remote exceptions.

The following differences lead to added expressiveness when it comes to the programming of complex distributed applications and services.

- The programmer is able to use the full expressive power of object-oriented

programming in the development of distributed systems software, including the use of objects, classes and inheritance, and can also employ related object-oriented design methodologies and associated tools.

• Building on the concept of object identity in object-oriented systems, all objects in an RMI-based system have unique object references (whether they are local or remote), such object references can also be passed as parameters, thus offering significantly richer parameter-passing semantics than in RPC.

Message and Stream Oriented Communication

Message-Oriented Communication

Message-oriented communication is a way of communicating between processes. *Messages*, which correspond to *events*, are the basic units of data delivered. Tanenbaum and Steen classified message-oriented communication according to two factors---*synchronous* or *asynchronous* communication, and *transient* or *persistent* communication. In synchronous communication, the sender blocks waiting for the receiver to engage in the exchange. Asynchronous communication does not require both the sender and the receiver to execute simultaneously. So, the sender and recipient are *loosely-coupled*. The amount of time messages is stored determines whether the communication is transient or persistent. Transient communication stores the message only while both partners in the communication are executing. If the next router or receiver is not available, then the message is discarded. Persistent communication, on the other hand, stores the message until the recipient receives it.

A typical example of asynchronous persistent communication is Message-Oriented Middleware (MOM). Message-oriented middleware is also called a *message-queuing system*, a *message framework*, or just a *messaging system*. MOM can form an important middleware layer for enterprise applications on the Internet. In the *publish and subscribe* model, a client can register as a publisher or a subscriber of messages. Messages are delivered only to the relevant destinations and only once, with various communication methods including one-to-many or many-to-many communication. The data source and destination can be decoupled under such a model.

The Java Message Service (JMS) from Sun Microsystems provides a common interface for Java applications to MOM implementations. Since JMS was integrated with the recent version of the Java 2 Enterprise Edition (J2EE) platform, Enterprise Java Beans (EJB)---the component architecture of J2EE---has a new type of bean, the message-driven bean. The JMS integration simplifies the enterprise development, allowing a decoupling between components.

Stream oriented communication

A (continuous) data stream is a connection-oriented communication facility that supports isochronous data transmission

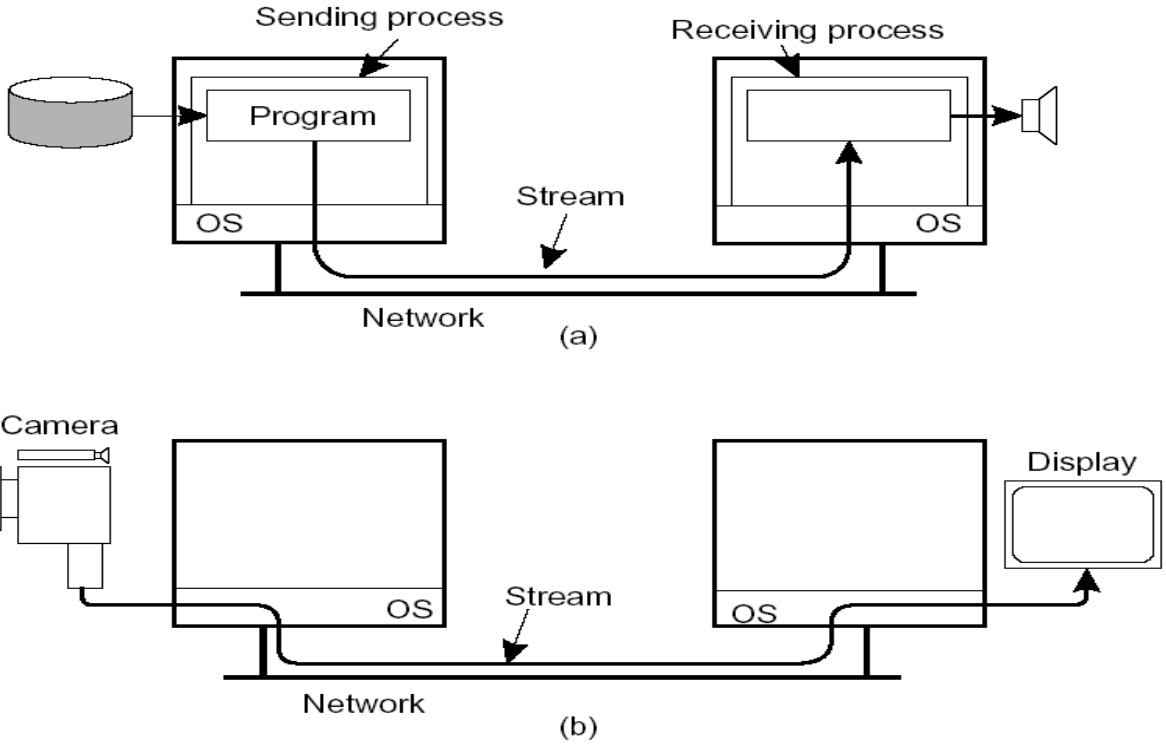
Some common stream characteristics:

- Streams are unidirectional. There is generally a single **source**, and one or more **sinks**
- Often, either the sink and/or source is a wrapper around hardware (e.g., camera, CD device, TV monitor, dedicated storage)

Stream types:

- **Simple**: consists of a single flow of data (e.g., audio or video)
- **Complex**: multiple data flows (e.g., stereo audio or combination audio/video)

- **Issue:** Streams can be set up between two processes at different machines, or directly between two different devices. Combinations are possible as well.



Stream-oriented communication

- Support for continuous media
- Streams in distributed systems
- Stream management

Data Streams

- Data stream = sequence of data items
- Can apply to discrete, as well as continuous media
  - e.g. UNIX pipes or TCP/IP connections which are both byte oriented (discrete) streams
  - Messages are related by send order
- Audio and video require continuous time-based data streams

- **Asynchronous transmission mode:** the order is important, and data is transmitted one after the other, no restriction to when data is to be delivered
- **Synchronous transmission mode** defines a maximum end-to-end delay for individual data packets
- **Isochronous transmission mode** has a maximum and minimum end-to-end delay requirement (jitter is bounded)
  - Not too slow, but not too fast either

12.9 SUMMARY

It can be argued that Web-based distributed systems have made networked applications popular with end users. Using the notion of a Web document as the means for exchanging information comes close to the way people often communicate in office environments and other settings. Everyone understands what a paper document is, so extending this concept to electronic documents is quite logical for most people.

The hypertext support as provided to Web end users has been of paramount importance to the Web's popularity. In addition, end users generally see a simple client-server architecture in which documents are simply fetched from a specific site. However, modem Web sites are organized along multitiered architectures in which a final component is merely responsible for generating HTML or XML pages as responses that can be displayed at the client.

Replacing the end user with an application has brought us Web services. From a technological point of view, Web services by themselves are generally not spectacular, although they are still in their infancy. What is important, however, is that very different services need to be discovered and be accessible to authorized clients. As a result, huge efforts are spent on standardization of service descriptions, communications, directories, and various interactions. Again. each standard by itself does not represent particularly new insights, but being a standard contributes to the expansion of Web services.

Processes in the Web are tailored to handling HTTP requests, of which the Apache Web server is a canonical example. Apache has proven to be a versatile vehicle for handling HTTP-based systems, but can also be easily extended to facilitate specific needs such as replication.

As the Web operates over the Internet, much attention has been paid to improving performance through caching and replication. More or less standard techniques have been developed for client-side caching, but when it comes to replication considerable advances have been made. Notably when replication of Web applications is at stake, it turns out that different solutions will need to co-exist for attaining optimal performance.

Both fault tolerance and security are generally handled using standard techniques that have since long been applied for many other types of distributed systems.

Common Carrier Services

INTRODUCTION

An entity that provides wired and wireless communication services to the general public for a fee. It is contrasted with a contract carrier, also called a private carrier, which provides services to a limited number of customers. It is licensed by a regulatory body(U.S.A- Federal Communications Commission (FCC), under authorization of the Telecommunications Act of 1934.). By classifying ISP's as common carriers the FCC has banned 'paid prioritization'—there will be no fast lanes and slow lanes of the Internet. It supports Net Neutrality (the notion that all internet traffic, regardless of its source, must be treated the same by Internet Service Providers (ISPs))

SERVICES

Satellite Transponder Sales

The FCC found that allowing the sale of satellite transponders would encourage additional satellite entry and facility investment, allow for more efficient use of orbital slots and of the radio spectrum and marketing innovation in the provision of domestic satellite services. The FCC also agreed with the Department of Justice and the Federal Trade Commission that domestic satellite licenses did not possess market power. The FCC has granted numerous applications allowing operators to provide domestic fixed satellite transponders on a non-common carrier basis while the remaining transponders on the satellite are offered under tariff.

Mass Media Services

The FCC followed this transponder sale approach in liberalizing regulation of the Microwave Multipoint Distribution Service (MMDS). Applying the NARUC I test, the FCC concluded that MMDS licenses could, at their election, designate some or all of their channels for non-common carrier service, while providing common carrier service on other channels. The FCC predicted benefits in giving MMDS visions the flexibility to operate on a non-common carriage status. However, the FCC did not require the licenses to offer their services indiscriminately to the public, because their new operations would take place in competitive markets.

Private Land Mobile Services

The FCC considered the regulatory status of private land mobile service and paging services and found both services to be non-common carriage. In 1982, the FCC found that the cooperative sharing of mobile voice telecommunications systems by multiple licenses was not common carriage. The FCC rejected the notion that either the licenses or the entities which supplied equipment to the licensees were common carriers. However, in the course of the proceeding, the FCC did make public interest findings that such private licensing and shared use of facilities were in the public interest. Congress has twice changed the Communications Act's test for determining whether a particular mobile service is common or private carriage.

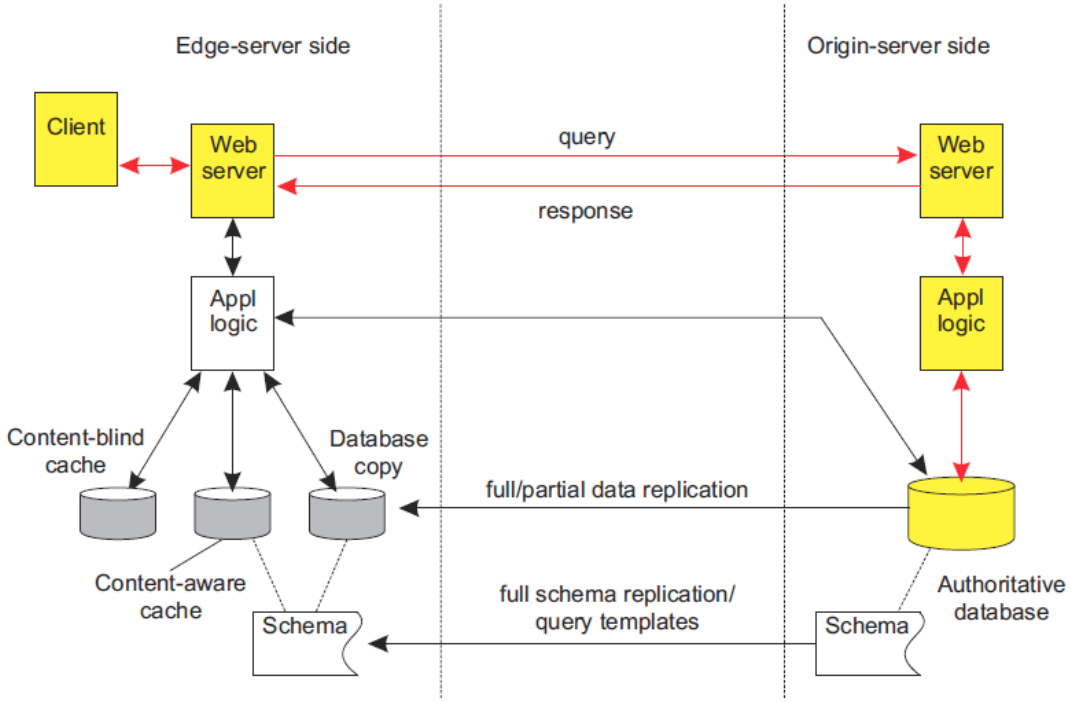
Private Microwave Services:

The FCC broadly defined common carriage when reviewing a number of microwave and fiber-optic cable activities in order to keep those services outside the scope of common carriage and free from the jurisdiction of state regulatory bodies. In 1985, the FCC freed private microwave licensees to offer, on a for-profit basis, telecommunications services to other businesses eligible to use these private frequencies. This new freedom was calculated to foster additional capacity and increased usage of built capacity. Since the services would be offered on a very selective basis, these carriers were distinguishable from common carriers.

LEGAL DIMENSION

In the purely legal dimension, Internet carriers seem presumptively to be common carriers. The principal legal test for whether an entity is a common carrier is whether it has held itself out to serve all indiscriminately, and most Internet carriers seem to do so. Internet carriers seem to exhibit at least some of the public aspects which have accompanied the imposition of common carrier duties, such as the indirect use of eminent domain powers and the manner in which the Internet has become an essential aspect of commerce and communication for many people and industries. The early involvement of ARPA and the NSF provided an important, direct government subsidy to the development of the Internet, a factor which has often pointed in the direction of common carrier regulation. Internet carriers are simply the most recent form of carrier, following the great tradition of steamships, railroads, and telephones; and all of their predecessors have been subject to some form of common carrier regulation.

Replication of Web applications: normal



Replication of Web applications

Alternative solutions

- Full replication: high read/write ratio, often in combination with complex queries.
- Partial replication: high read/write ratio, but in combination with simple queries
- Content-aware caching: Check for queries at local database, and subscribe for invalidations at the server. Works good with range queries and complex queries.
- Content-blind caching: Simply cache the result of previous queries. Works great with simple queries that address unique results (e.g., no range queries).

Unit 5

Distributed file system

11.9 SUMMARY

Distributed file systems form an important paradigm for building distributed systems. They are generally organized according to the client-server model, with client-side caching and support for server replication to meet scalability requirements. In addition, caching and replication are needed to achieve high availability. More recently, symmetric architectures such as those in peer-to-peer file-sharing systems have emerged. In these cases, an important issue is whether whole files or data blocks are distributed.

Instead of building a distributed file system directly on top of the transport layer it is common practice to assume the existence of an RPC layer, so that all operations can be simply expressed as RPCs to a file server instead of having to use primitive message-passing operations. Some variants of RPC have been developed, such as the MultiRPC provided in Coda, which allows a number of servers to be called in parallel.

What makes distributed file systems different from nondistributed file systems is the semantics of sharing files. Ideally, a file system allows a client to always read the data that have most recently been written to a file. These UNIX file-sharing semantics are very hard to implement efficiently in a distributed system. NFS supports a weaker form known as session semantics, by which the final version of a file is determined by the last client that closes a file, which it had previously opened for writing. In Coda, file sharing adheres to transactional semantics in the sense that reading clients will only get to see the most recent updates if they reopen a file. Transactional semantics in Coda do not cover all the ACID properties of regular transactions. In the case that a file server stays in control of all operations, actual UNIX semantics can be provided, although scalability is then an issue. In all cases, it is necessary to allow concurrent updates on files, which brings relatively intricate locking and reservation schemes into play.

To achieve acceptable performance, distributed file systems generally allow clients to cache an entire file. This whole-file caching approach is supported, for example, in NFS, although it is also possible to store only very large chunks of a file. Once a file has been opened and (partly) transferred to the client, all operations are carried out locally. Updates are flushed to the server when the file is closed again.

Replication also plays an important role in peer-to-peer systems, although matters are strongly simplified because files are generally read-only. More important in these systems is trying to reach acceptable load balance, as naive replication schemes can easily lead to hot spots holding many files and thus become potential bottlenecks.

Fault tolerance is usually dealt with using traditional methods. However, it is also possible to build file systems that can deal with Byzantine failures, even when the system as a whole is running on the Internet. In this case, by assuming reasonable timeouts and initiating new server groups (possibly based on false failure detection), practical solutions can be built. Notably for distributed file systems, one should consider to apply erasure coding techniques to reduce the overall replication factor when aiming for only high availability.

Security is of paramount importance for any distributed system, including file systems. NFS hardly provides any security mechanisms itself, but instead implements standardized interfaces that allow different existing security systems to be used, such as, for example Kerberos. SFS is different in the sense it allows file names to include information on the file server's public key. This approach simplifies key management in large-scale systems. In effect, SFS distributes a key by

including it in the name of a file. SFS can be used to implement a decentralized authentication scheme. Achieving security in peer-to-peer file-sharing systems is difficult, partly because of the assumed collaborative nature in which nodes will always tend to act selfish. Also, making lookups secure turns out to be a difficult problem that actually requires a central authority for handing out node identifiers.

10.9 SUMMARY

Most object-based distributed systems use a remote-object model in which an object is hosted by server that allows remote clients to do method invocations. In many cases, these objects will be constructed at runtime, effectively meaning that their state, and possibly also code is loaded into an object server when a client does a remote invocation. Globe is a system in which truly distributed shared objects are supported. In this case, an object's state may be physically distributed and replicated across multiple machines.

To support distributed objects, it is important to separate functionality from extra-functional properties such as fault tolerance or scalability. To this end, advanced object servers have been developed for hosting objects. An object server provides many services to basic objects, including facilities for storing objects, or to ensure serialization of incoming requests. Another important role is providing the illusion to the outside world that a collection of data and procedures operating on that data correspond to the concept of an object. This role is implemented by means of object adapters.

When it comes to communication, the prevalent way to invoke an object is by means of a remote method invocation (RMI), which is very similar to an RPC. An important difference is that distributed objects generally provide a systemwide object reference, allowing a process to access an object from any machine. Global object reference solve many of the parameter-passing problems that hinder access transparency of RPCs.

There are many different ways in which these object references can be implemented, ranging from simple passive data structures describing precisely where a remote object can be contacted, to portable code that need simply be invoked by a client. The latter approach is now commonly adopted for Java RMI.

There are no special measures in most systems to handle object synchronization. An important exception is the way that synchronized Java methods are treated: the synchronization takes place only between clients running on the same machine. Clients running on different machines need to take special synchronization measures. These measures are not part of the Java language.

Entry consistency is an obvious consistency model for distributed objects and is (often implicitly) supported in many systems. It is obvious as we can naturally associate a separate lock for each object. One of the problems resulting from replicating objects are replicated invocations. This problem is more evident because objects tend to be treated as black boxes.

Fault tolerance in distributed object-based systems very much follows the approaches used for other distributed systems. One exception is formed by tryingto make the Java virtual machine fault tolerant by letting it operate as a deterministic finite state machine. Then, by replicating a number of these machines, we obtain a natural way for providing fault tolerance.

Security for distributed objects evolves around the idea of supporting secure method invocation. A comprehensive example that generalizes these invocations to replicated objects is Globe. As it turns out, it is possible to cleanly separate policies from mechanisms. This is true for authentication as well as authorization. Special attention needs to be paid to systems in which the client is required to download a proxy from a directory service, as is commonly the case for Java.

Synchronization

Introduction

In the previous chapters, we have looked at processes and communication between processes. While communication is important, it is not the entire story. Closely related is how processes cooperate and synchronize with one another. Cooperation is partly supported by means of naming, which allows processes to at least share resources, or entities in general..

In this chapter, we mainly concentrate on how processes can synchronize. For example, it is important that multiple processes do not simultaneously access a shared resource, such as printer, but instead cooperate in granting each other temporary exclusive access. Another example is that multiple processes may sometimes need to agree on the ordering of events, such as whether message *ml* from process *P* was sent before or after message *m2* from process Q.

As it turns out, synchronization in distributed systems is often much more difficult compared to synchronization in uniprocessor or multiprocessor systems. The problems and solutions that are discussed in this chapter are, by their nature, rather general, and occur in many different situations in distributed systems.

We start with a discussion of the issue of synchronization based on actual time, followed by synchronization in which only relative ordering matters rather than ordering in absolute time.

In many cases, it is important that a group of processes can appoint one process as a coordinator, which can be done by means of election algorithms. We discuss various election algorithms in a separate section.

6.6 SUMMARY

Strongly related to communication between processes is the issue of how processes in distributed systems synchronize. Synchronization is all about doing the right thing at the right time. A problem in distributed systems, and computer networks in general, is that there is no notion of a globally shared clock. In other words, processes on different machines have their own idea of what time it is.

There are various way to synchronize clocks in a distributed system, but all methods are essentially based on exchanging clock values, while taking into account the time it takes to send and receive messages. Variations in communication delays and the way those variations are dealt with, largely determine the accuracy of clock synchronization algorithms.

Related to these synchronization problems is positioning nodes in a geometric overlay. The basic idea is to assign each node coordinates from an rn-dimensional space such that the geometric distance can be used as an accurate measure for the latency between two nodes. The method of assigning coordinates strongly resembles the one applied in determining the location and time in GPS.

In many cases, knowing the absolute time is not necessary. What counts is that related events at different processes happen in the correct order. Lamport showed that by introducing a notion of logical clocks, it is possible for a collection of processes to reach global agreement on the correct ordering of events. In essence, each event *e,* such as sending or receiving a message, is assigned a globally unique logical timestamp C *(e)* such that when event *a* happened before *b,* C *(a)* < C *(b)*. Lamport timestamps can be extended to vector timestamps: if C *(a)* < C *(b),* we even know that event *a* causally preceded *b*.

An important class of synchronization algorithms is that of distributed mutual exclusion. These algorithms ensure that in a distributed collection of processes, at most one process at a time has access to a shared resource. Distributed mutual exclusion can easily be achieved if we make use of a coordinator that keeps track of whose turn it is. Fully distributed algorithms also exist, but have the drawback that they are generally more susceptible to communication and process failures.

Synchronization between processes often requires that one process acts as a coordinator. In those cases where the coordinator is not fixed, it is necessary that processes in a distributed computation decide on who is going to be that coordinator. Such a decision is taken by means of election algorithms. Election algorithms are primarily used in cases where the coordinator can crash. However, they can also be applied for the selection of superpeers in peer-to-peer systems.

Fault Tolerance

A characteristic feature of distributed systems that distinguishes them from single-machine systems is the notion of partial failure. A partial failure may happen when one component in a distributed system fails. This failure may affect the proper operation of other components, while at the same time leaving yet other components totally unaffected. In contrast, a failure in nondistributed systems is often total in the sense that it affects all components, and may easily bring down the entire system.

An important goal in distributed systems design is to construct the system in such a way that it can automatically recover from partial failures without seriously affecting the overall performance. In particular, whenever a failure occurs, the distributed system should continue to operate in an acceptable way while repairs are being made, that is, it should tolerate faults and continue to operate to some extent even in their presence.

In this chapter, we take a closer look at techniques for making distributed systems fault tolerant. After providing some general background on fault tolerance, we will look at process resilience and reliable multicasting. Process resilience incorporates techniques by which one or more processes can fail without seriously disturbing the rest of the system. Related to this issue is reliable multicasting, by which message transmission to a collection of processes is guaranteed to succeed. Reliable multicasting is often necessary to keep processes synchronized.

Atomicity is a property that is important in many applications. For example, in distributed transactions, it is necessary to guarantee that every operation in a transaction is carried out or none of them are. Fundamental to atomicity in distributed systems is the notion of distributed commit protocols, which are discussed in a separate section in this chapter.

Finally, we will examine how to recover from a failure. In particular, we consider when and how the state of a distributed system should be saved to allow recovery to that state later on.

Type of failure	Description
Crash failure	A server halts, but is working correctly until it halts
Omission failure *Receive omission* *Send omission*	A server fails to respond to incoming requests A server fails to receive incoming messages A server fails to send messages
Timing failure	A server's response lies outside the specified time interval
Response failure *Value failure* *State transition failure*	A server's response is incorrect The value of the response is wrong The server deviates from the correct flow of control
Arbitrary failure	A server may produce arbitrary responses at arbitrary times

Figure 8-1. Different types of failures.

8.2.4 Failure Detection

It may have become clear from our discussions so far that in order to properly mask failures, we generally need to detect them as well. Failure detection is one of the cornerstones of fault tolerance in distributed systems. What it all boils down to is that for a group of processes, nonfaulty members should be able to decide who is still a member, and who is not. In other words, we need to be able to detect when a member has failed.

When it comes to detecting process failures, there are essentially only two mechanisms. Either processes actively send "are you alive?" messages to each other (for which they obviously expect an answer), or passively wait until messages come in from different processes. The latter approach makes sense only when it can be guaranteed that there is enough communication between processes. In practice, actively **pinging** processes is usually followed.

There has been a huge body of theoretical work on failure detectors. What it all boils down to is that a timeout mechanism is used to check whether a process has failed. In real settings, there are two major problems with this approach. First, due to unreliable networks, simply stating that a process has failed because it does not return an answer to a ping message may be wrong. In other words, it is quite easy to generate false positives. If a false positive has the effect that a perfectly healthy process is removed from a membership list, then clearly we are doing something wrong.

Another serious problem is that timeouts are just plain crude. As noticed by Birman (2005), there is hardly any work on building proper failure detection subsystems that take more into account than only the lack of a reply to a single message. This statement is even more evident when looking at industry-deployed distributed systems.

There are various issues that need to be taken into account when designing a failure detection subsystem [see also Zhuang et al. (2005)]. For example, failure detection can take place through gossiping in which each node regularly announces to its neighbors that it is still up and running. As we mentioned, an alternative is to let nodes actively probe each other.

Failure detection can also be done as a side-effect of regularly exchanging information with neighbors, as is the case with gossip-based information dissemination (which we discussed in Chap. 4). This approach is essentially also adopted in Obduro (Vogels, 2003): processes periodically gossip their service availability. This information is gradually disseminated through the network by gossiping. Eventually, every process will know about every other process, but more importantly, will have enough information locally available to decide whether a process has failed or not. A member for which the availability information is old, will presumably have failed.

Another important issue is that a failure detection subsystem should ideally be able to distinguish network failures from node failures. One way of dealing with this problem is not to let a single node decide whether one of its neighbors has crashed. Instead, when noticing a timeout on a ping message, a node requests other neighbors to see whether they can reach the presumed failing node. Of course, positive information can also be shared: if a node is still alive, that information can be forwarded to other interested parties (who may be detecting a link failure to the suspected node).

This brings us to another key issue: when a member failure is detected, how should other nonfaulty processes be informed? One simple, and somewhat radical approach is the one followed in FUSE (Dunagan et al., 2004). In FUSE, processes can be joined in a group that spans a wide-area network. The group members create a spanning tree that is used for monitoring member failures. Members send ping messages to their neighbors. When a neighbor does not respond, the pinging node immediately switches to a state in which it will also no longer respond to pings from other nodes. By recursion, it is seen that a single node failure is rapidly promoted to a group failure notification. FUSE does not suffer a lot from link failures for the simple reason that it relies on point-to-point TCP connections between group members.

8.7 SUMMARY

Fault tolerance is an important subject in distributed systems design. Fault tolerance is defined as the characteristic by which a system can mask the occurrence and recovery from failures. In other words, a system is fault tolerant if it can continue to operate in the presence of failures.

Several types of failures exist. A crash failure occurs when a process simply halts. An omission failure occurs when a process does not respond to incoming requests. When a process responds too soon or too late to a request, it is said to exhibit a timing failure. Responding to an incoming request, but in the wrong way, is an example of a response failure. The most difficult failures to handle are those by which a process exhibits any kind of failure, called arbitrary or Byzantine failures.

Redundancy is the key technique needed to achieve fault tolerance. When applied to processes, the notion of process groups becomes important. A process group consists of a number of processes that closely cooperate to provide a service. In fault-tolerant process groups, one or more processes can fail without affecting the availability of the service the group implements. Often, it is necessary that communication within the group be highly reliable, and adheres to stringent ordering and atomicity properties in order to achieve fault tolerance.

Reliable group communication, also called reliable multicasting, comes in different forms. As long as groups are relatively small, it turns out that implementing reliability is feasible. However, as soon as very large groups need to be supported, scalability of reliable multicasting becomes problematic. The key issue in achieving scalability is to reduce the number of feedback messages by which receivers report the (un)successful receipt of a multicasted message.

Matters become worse when atomicity is to be provided. In atomic multicast protocols, it is essential that each group member have the same view concerning to which members a multicasted message has been delivered. Atomic multicasting can be precisely formulated in terms of a virtual synchronous execution model. In essence, this model introduces boundaries between which group membership does

not change and which messages are reliably transmitted. A message can never cross a boundary.

Group membership changes are an example where each process needs to agree on the same list of members. Such agreement can be reached by means of a commit protocol, of which the two-phase commit protocol is the most widely applied. In a two-phase commit protocol, a coordinator first checks whether all processes agree to perform the same operation (i.e., whether they all agree to commit), and in a second round, multicasts the outcome of that poll. A three-phase commit protocol is used to handle the crash of the coordinator without having to block all processes to reach agreement until the coordinator recovers.

Recovery in fault-tolerant systems is invariably achieved by checkpointing the state of the system on a regular basis. Checkpointing is completely distributed. Unfortunately, taking a checkpoint is an expensive operation. To improve performance, many distributed systems combine checkpointing with message logging. By logging the communication between processes, it becomes possible to replay the execution of the system after a crash has occurred.

Network File System (NFS)

Network File System (NFS) is a distributed file system (DFS) developed by Sun Microsystems. This allows directory structures to be spread over the net- worked computing systems.
A DFS is a file system whose clients, servers and storage devices are dis- persed among the machines of distributed system. A file system provides a set of file operations like read, write, open, close, delete etc. which forms the file services. The clients are provided with these file services. The basic features of DFS are multiplicity and autonomy of clients and servers.
NFS follows the directory structure almost same as that in non-NFS system but there are some differences between them with respect to:

- Naming
- Path Names
- Semantics

Naming

Naming is a mapping between logical and physical objects. For example, users refers to a file by a textual name, but it is mapped to disk blocks. There are two notions regarding name mapping used in DFS.

- **Location Transparency:** The name of a file does not give any hint of file's physical storage location.
- **Location Independence:** The name of a file does not need to be changed when file's physical storage location changes.

A location independent naming scheme is basically a dynamic mapping. NFS does not support location independency.

There are three major naming schemes used in DFS. In the simplest approach, files are named by some combination of machine or host name and the path name. This naming scheme is neither location independent nor location transparent. This may be used in server side. Second approach is to attach or mount the remote directories to the local directories. This gives an appearance of a coherent directory. This scheme is used by NFS. Early NFS allowed only previously mounted remote directories. But with the advent of automount , remote directories are mounted on demand based on the table of mount points and file structure names. This has other advantages like the file-mount table size is much smaller and for each mount point, we can specify many servers. The third approach of naming is to use name space which is identical to all machines. In practice, there are many special files that make this approach difficult to implement.

Mounting

The mount protocol is used to establish the initial logical connection between a server and a client. A mount operation includes the name of the remote directory to be mounted and the name of the server machine storing it. The server maintains an export list which specifies local file system that it exports for mounting along with the permitted machine names. Unix uses /etc/exports for this purpose. Since, the list has a maximum length, NFS is limited in scalabilty. Any directory within an exported file system can be mounted remotely on a machine. When the server receives a mount request, it returns a file handle to the client. File handle is basically a data-structure of length 32 bytes. It serves as the key for further access to files within the mounted system. In Unix term, the file handle consists of a file system identifier that is stored in super block and an inode number to identify the exact mounted directory within the exported file system. In NFS, one new field is added in inode that is called the generic number.

Mount can be is of three types -

1. **Soft mount:** A time bound is there.
2. **Hard mount:** No time bound.
3. **Automount:** Mount operation done on demand.

NFS Protocol and Remote Operations

The NFS protocol provides a set of RPCs for remote operations like **lookup, create, rename, getattr, setattr, read, write, remove, mkdir** etc. The procedures can be invoked only after a file handle for the remotely mounted directory has been esta- blished. NFS servers are stateless servers. A stateless file server avoids to keep state informations by making each request self-contained. That is, each request iden- tifies the file and the position of the file in full. So, the server needs not to store file pointer. Moreover, it needs not to establish or terminate a connection by opening a file or closing a file, repectively. For reading a directory, NFS does not use any file pointer, it uses a **magic cookie**.

Except the opening and closing a file, there is almost one-to-one mapping between Unix system calls for file operations and the NFS protocol RPCs. A remote file operation can be translated directly to the corresponding RPC. Though conceptu- ally, NFS adheres to the remote service paradigm, in practice, it uses buffering and caching. File blocks and attributes are fetched by RPCs and cached locally. Future remote operations use the cached data, subject to consistency constraints.

Since, NFS runs on RPC and RPC runs on UDP/IP which is unreliable, operations should be idempotent.

Cache Update Policy

The policy used to write modified data blocks to the server's master copy has critical effect on the system performance and reliability. The simplest policy is to **write through** the disk as soon as they are placed on any cache. It's advantageous because it ensures the reliability but it gives poor performance. In server site this policy is often followed. Another policy is **delayed write**. It does not ensure reliability. Client sites can use this policy. Another policy is **write-on-close**. It is a variation of delayed write. This is used by Andrews File System (AFS).
In NFS, clients use delayed write. But they don't free delayed written block until the server confirms that the data have been written on disk. So, here, Unix semantics are not preserved. NFS does not handle client crash recovery like Unix. Since, servers in NFS are stateless, there is no need to handle server crash recovery also.

Time Skew
Because of differences of time at server and client, this problem occures. This may lead to problems in performing some operations like " make ".

Performance Issues
To increase the reliability and system performance, the following things are generally done.

1. Cache, file blocks and directory informations are maintained.
2. All attributes of file / directory are cached. These stay 3 sec. for files and 30 sec. for directory.
3. For large caches, bigger block size (8K) is benificial.

This is a brief description of NFS version 2. NFS version 3 has already been come out and this new version is an enhancement of the previous version. It removes many of the difficulties and drawbacks of NFS 2.

Andrews File System (AFS)

AFS is a distributed file system, with scalability as a major goal. Its efficiency can be attributed to the following practical assumptions (as also seen in UNIX file system):

- Files are small (i.e. entire file can be cached)
- Frequency of reads much more than those of writes
- Sequential access common
- Files are not shared (i.e. read and written by only one user)
- Shared files are usually not written
- Disk space is plentiful

AFS distinguishes between client machines (workstations) and dedicated server machines. Caching files in the client side cache reduces computation at the server side, thus enhancing performance. However, the problem of sharing files arises. To solve this, all clients with copies of a file being modified by another client are not informed the moment the client makes changes. That client thus updates its copy, and the changes are reflected in the distributed file system only after the client closes the file. Various terms related to this concept in AFS are:

- **Whole File Serving**: The entire file is transferred in one go, limited only by the maximum size UDP/IP supports
- **Whole File Caching**: The entire file is cached in the local machine cache, reducing file-open latency, and frequent read/write requests to the server
- **Write On Close**: Writes are propagated to the server side copy only when the client closes the local copy of the file

In AFS, the server keeps track of which files are opened by which clients (as was not in the case of NFS). In other words, AFS has **stateful servers**, whereas NFS has **stateless servers**. Another difference between the two file systems is that AFS provides **location independence** (the physical storage location of the file can be changed, without having to change the path of the file, etc.) as well as **location transparency** (the file name does not hint at its physical storage location). But as was seen in the last lecture, NFS provides only location transparency. Stateful servers in AFS allow the server to inform all clients with open files about any updates made to that file by another client, through what is known as a **callback**. Callbacks to all clients with a copy of that file is ensured as a **callback promise** is issued by the server to a client when it requests for a copy of a file.

The key software components in AFS are:

- **Vice**: The server side process that resides on top of the unix kernel, providing shared file services to each client
- **Venus**: The client side cache manager which acts as an interface between the application program and the Vice

All the files in AFS are distributed among the servers. The set of files in one server is referred to as a **volume**. In case a request can not be satisfied from this set of files, the vice server informs the client where it can find the required file.

The basic file operations can be described more completely as:

- Open a file: Venus traps application generated file open system calls, and checks whether it can be serviced locally (i.e. a copy of the file already exists in the cache) before requesting Vice for it. It then returns a file descriptor to the calling application. Vice, along with a copy of the file, transfers a callback promise, when Venus requests for a file.
- Read and Write: Reads/Writes are done from/to the cached copy.
- Close a file: Venus traps file close system calls and closes the cached copy of the file. If the file had been updated, it informs the Vice server which then replaces its copy with the

updated one, as well as issues callbacks to all clients holding callback promises on this file. On receiving a callback, the client discards its copy, and works on this fresh copy.

The server wishes to maintain its states at all times, so that no information is lost due to crashes. This is ensured by the Vice which writes the states to the disk. When the server comes up again, it also informs all the servers about its crash, so that information about updates may be passed to it.

A client may issue an open immediately after it issued a close (this may happen if it has recovered from a crash very quickly). It will wish to work on the same copy. For this reason, Venus waits a while (depending on the cache capacity) before discarding copies of closed files. In case the application had not updated the copy before it closed it, it may continue to work on the same copy. However, if the copy had been updated, and the client issued a file open after a certain time interval (say 30 seconds), it will have to ask the server the last modification time, and accordingly, request for a new copy. For this, the clocks will have to be synchronized.

Unit 6

Grid Computing:

Grid computing is a computer network in which each computer's resources are shared with every other computer in the system. Processing power, memory and data storage are all community resources that authorized users can tap into and leverage for specific tasks. A grid computing system can be as simple as a collection of similar computers running on the same operating system or as complex as inter-networked systems comprised of every computer platform you can think of.

Grid computing is the federation of computer resources from multiple administrative domains to reach a common goal. The grid can be thought of as a distributed system with non-interactive workloads that involve a large number of files. What distinguishes grid computing from conventional high performance computing systems such as cluster computing is that grids tend to be more loosely coupled, heterogeneous, and geographically dispersed. Although a single grid can be dedicated to a particular application, commonly a grid is used for a variety of purposes. Grids are often constructed with general-purpose grid middleware software libraries.

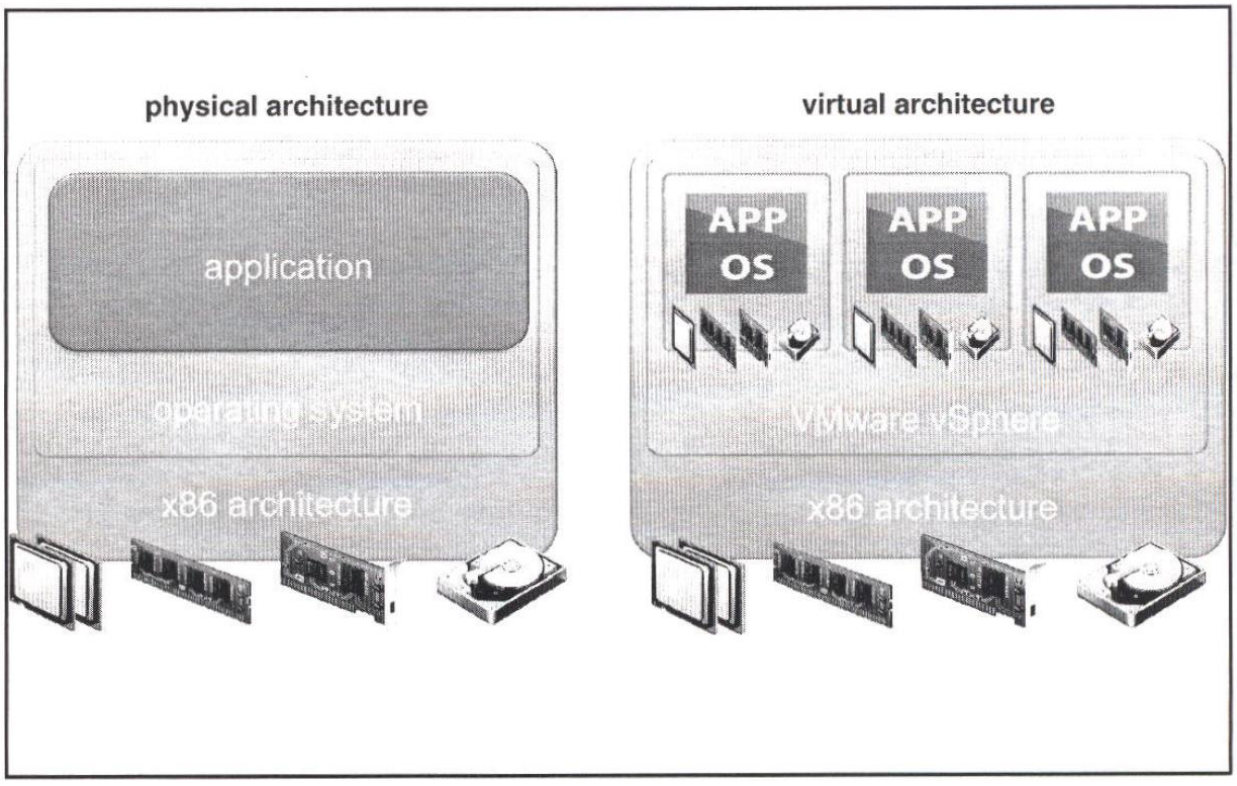
Grid size varies a considerable amount. Grids are a form of distributed computing whereby a “super virtual computer” is composed of many networked loosely coupled computers acting together to perform large tasks. For certain applications, “distributed” or “grid” computing, can be seen as a special type of parallel computing that relies on complete computers (with onboard CPUs, storage, power supplies, network interfaces, etc.) connected to a network (private, public or the Internet) by a conventional network interface, such as Ethernet. This is in contrast to the traditional notion of a supercomputer, which has many processors connected by a local high-speed computer bus.

Grid computing combines computers from multiple administrative domains to reach a common goal, to solve a single task, and may then disappear just as quickly. One of the main strategies of grid computing is to use middleware to divide and apportion pieces of a program among several

computers, sometimes up to many thousands. Grid computing involves computation in a distributed fashion, which may also involve the aggregation of large-scale cluster computing-based systems.

The size of a grid may vary from small—confined to a network of computer workstations within a corporation, for example—to large, public collaborations across many companies and networks. "The notion of a confined grid may also be known as an intra-nodes cooperation whilst the notion of a larger, wider grid may thus refer to an inter-node cooperation".

Virtualization:



Virtualization is the key to cloud computing, since it is the enabling technology allowing the creation of an intelligent abstraction layer which hides the complexity of underlying hardware or software.

Server virtualization enables different operating systems to share the same hardware and make it easy to move operating systems between different hardware, all while the applications are running.

Storage virtualization does the same thing for data. Storage virtualization creates the abstraction layer between the applications running on the servers, and the storage they use to store the data. Virtualizing the storage and incorporating the intelligence for provisioning and protection at the virtualization layer enables companies to use any storage they want, and not be locked into any individual vendor. Storage virtualization makes storage a commodity. All this makes for some interesting ways for companies to reduce their costs.

Any discussion of cloud computing typically begins with virtualization. Virtualization is critical to cloud computing because it simplifies the delivery of services by providing a platform for optimizing complex IT resources in a scalable manner, which is what makes cloud computing so cost effective. Virtualization can be applied very broadly to just about everything you can imagine including memory, networks, storage, hardware, operating systems, and applications.

Virtualization has three characteristics that make it ideal for cloud computing:

- Partitioning: In virtualization, you can use partitioning to support many applications and operating systems in a single physical system.

- Isolation: Because each virtual machine is isolated, each machine is protected from crashes and viruses in the other machines. What makes virtualization so important for the cloud is that it decouples the software from the hardware.

- Encapsulation: Encapsulation can protect each application so that it doesn't interfere with other applications. Using encapsulation, a virtual machine can be represented (and even stored) as a single file, making it easy to identify and present to other applications.

To understand how virtualization helps with cloud computing, you must understand its many forms. In essence, in all cases, a resource actually emulates or imitates another resource. Here are some examples:

- Virtual memory: Disks have a lot more space than memory. PCs can use virtual memory to borrow extra memory from the hard disk. Although virtual disks are slower than real memory, if managed right, the substitution works surprisingly well.

- Software: There is virtualization software available that can emulate an entire computer, which means 1 computer can perform as though it were actually 20 computers. Using this kind of software, you might be able to move from a data center with thousands of servers to one that supports as few as a couple of hundred.

To manage the various aspects of virtualization in cloud computing most companies use hypervisors, an operating system that act as traffic cop managing the various virtualization tasks in the cloud to ensure that they make the things happen in an orderly manner. Because in cloud computing you need to support many different operating environments, the hypervisor becomes

an ideal delivery mechanism by allowing you to show the same application on lots of different systems. Because hypervisors can load multiple operating systems, they are a very practical way of getting things virtualized quickly and efficiently.

Cloud Computing

Cloud computing is the use of computing resources (hardware and software) that are delivered as a service over a network (typically the Internet). The name comes from the use of a cloud-shaped symbol as an abstraction for the complex infrastructure it contains in system diagrams. Cloud computing entrusts remote services with a user's data, software and computation.

There are many types of public cloud computing: Infrastructure as a service (IaaS), Platform as a service (PaaS), Software as a service (SaaS), Storage as a service (STaaS), Security as a service (SECaaS), Data as a service (DaaS), Test environment as a service (TEaaS), Desktop as a service (DaaS), API as a service (APIaaS).

Characteristics:
Cloud computing exhibits the following key characteristics:

- Agility improves with users' ability to re-provision technological infrastructure resources.

- Application programming interface (API) accessibility to software that enables machines to interact with cloud software in the same way the user interface facilitates interaction between humans and computers.

- Cost is claimed to be reduced and in a public cloud delivery model capital expenditure is converted to operational expenditure. This is purported to lower barriers to entry, as infrastructure is typically provided by a third-party and does not need to be purchased for one-time or infrequent intensive computing tasks. Pricing on a utility computing basis is fine-grained with usage-based options and fewer IT skills are required for implementation (in-house).

- Device and location independence enable users to access systems using a web browser regardless of their location or what device they are using (e.g., PC, mobile phone). As infrastructure is off-site (typically provided by a third-party) and accessed via the Internet, users can connect from anywhere.

- Virtualization technology allows servers and storage devices to be shared and utilization be increased. Applications can be easily migrated from one physical server to another.

- Multitenancy enables sharing of resources and costs across a large pool of users thus allowing for:

    o Centralization of infrastructure in locations with lower costs (such as real estate, electricity, etc.)

o Peak-load capacity increases (users need not engineer for highest possible load-levels)

o Utilization and efficiency

- Reliability is improved if multiple redundant sites are used, which makes well designed cloud computing suitable for business continuity and disaster recovery.

- Scalability and elasticity via dynamic ("on-demand") provisioning of resources on a fine-grained, self-service basis near real-time, without users having to engineer for peak loads.

- Performance is monitored, and consistent and loosely coupled architectures are constructed using web services as the system interface.

- Security could improve due to centralization of data, increased security-focused resources, etc., but concerns can persist about loss of control over certain sensitive data, and the lack of security for stored kernels.

- Maintenance of cloud computing applications is easier, because they do not need to be installed on each user's computer and can be accessed from different places.

Cloud Clients:

Users access cloud computing using networked client devices, such as desktop computers, laptops, tablets and smartphones. Some of these devices - *cloud clients* – rely on cloud computing for all or a majority of their applications so as to be essentially useless without it. Examples are thin clients and the browser-based Chromebook. Many cloud applications do not require specific software on the client and instead use a web browser to interact with the cloud application. With Ajax and HTML5 these Web user interfaces can achieve a similar or even better look and feel as native applications. Some cloud applications, however, support specific client software dedicated to these applications (e.g., virtual desktop clients and most email clients). Some legacy applications (line of business applications that until now have been prevalent in thin client Windows computing) are delivered via a screen-sharing technology.

Cloud Service Models:

Cloud computing providers offer their services according to three fundamental models: Infrastructure as a service (IaaS), platform as a service (PaaS), and software as a service (SaaS) where IaaS is the most basic and each higher model abstracts from the details of the lower models.

Iaas:
In this most basic cloud service model, cloud providers offer computers, as physical or more often as virtual machines, and other resources. The virtual machines are run as guests by a hypervisor, such as Xen or KVM. Management of pools of hypervisors by the cloud operational support system leads to the ability to scale to support a large number of virtual machines. Other resources in IaaS clouds include images in a virtual machine image library, raw (block) and file-based storage,

firewalls, load balancers, IP addresses, virtual local area networks (VLANs), and software bundles. IaaS cloud providers supply these resources on demand from their large pools installed in data centers. For wide area connectivity, the Internet can be used or—in carrier clouds -- dedicated virtual private networks can be configured.

To deploy their applications, cloud users then install operating system images on the machines as well as their application software. In this model, it is the cloud user who is responsible for patching and maintaining the operating systems and application software. Cloud providers typically bill IaaS services on a utility computing basis, that is, cost will reflect the amount of resources allocated and consumed.

IaaS refers not to a machine that does all the work, but simply to a facility given to businesses that offers users the leverage of extra storage space in servers and data centers. Examples of IaaS include: Amazon CloudFormation (and underlying services such as Amazon EC2), Rackspace Cloud, Terremark and Google Compute Engine.

Paas:
In the PaaS model, cloud providers deliver a computing platform typically including operating system, programming language execution environment, database, and web server. Application developers can develop and run their software solutions on a cloud platform without the cost and complexity of buying and managing the underlying hardware and software layers. With some PaaS offers, the underlying computer and storage resources scale automatically to match application demand such that cloud user does not have to allocate resources manually. Examples of PaaS include: Amazon Elastic Beanstalk, Heroku, EngineYard, Mendix, Google App Engine, Microsoft Azure and OrangeScape.

Saas:
In this model, cloud providers install and operate application software in the cloud and cloud users access the software from cloud clients. The cloud users do not manage the cloud infrastructure and platform on which the application is running. This eliminates the need to install and run the application on the cloud user's own computers simplifying maintenance and support. What makes a cloud application different from other applications is its elasticity. This can be achieved by cloning tasks onto multiple virtual machines at run-time to meet the changing work demand. Load balancers distribute the work over the set of virtual machines. This process is inconspicuous to the cloud user who sees only a single access point. To accommodate a large number of cloud users, cloud applications can be multitenant, that is, any machine serves more than one cloud user organization. It is common to refer to special types of cloud based application software with a similar naming convention: desktop as a service, business process as a service, test environment as a service, communication as a service.

The pricing model for SaaS applications is typically a monthly or yearly flat fee per user, so price is scalable and adjustable if users are added or removed at any point. Examples of SaaS include: Google Apps, innkeypos, Quickbooks Online, Limelight Video Platform, Salesforce.com and Microsoft Office 365.