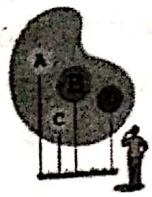
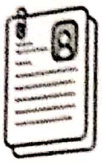
EXERCISE



Multiple Choice Questions

1. In distributed system, each processor has its own.....
   a) Local memory
   b) clock
   c) Both local memory and clock
   d) none of the mentioned
2. The capability of a system to adapt the increased service load is called.....
   a) Scalability
   b) tolerance
   c) Capacity
   d) none of the mentioned
3. Distributed system have.....
   a) High security
   b) Better resource sharing
   c) Better system utilization
   d) Low system overhead
4. Distributed OS works on the ..... principle.
   a) File Foundation
   b) Single system image
   c) Multi system image
   d) Networking image
5. A distributed system contains..... nodes.
   a) Zero node
   b) One node
   c) Two node
   d) multiple node
6. The nodes in the distributed systems can be arranged in the form of?
   a) client/server systems
   b) Peer to peer systems
   c) Both A and B
   d) None of the above
7. In which system, tasks are equally divided between all the nodes?
   a) .client/server systems
   b) .peer to peer systems
   c) .user to client system
   d) .all of the above
8. In distributed system, each processor has its own .....
   a) . Local memory
   b). Clock
   c) . Both a and b
   d). None of the above

9. What are the characteristics of a distributed file system?
   a) Its users, servers and storage devices are dispersed
   b) Service activity is not carried out across the network
   c) They have single centralized data repository
   d) There are multiple dependent storage devices
10. What are the characteristics of Distributed Operating system?
   a) Users are aware of multiplicity of machines
   b) Access is done like local resources
   c) Users are aware of multiplicity of machines
   d) They have multiple zones to access files

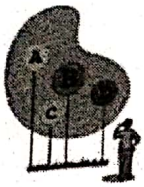


Subjective Questions

1. Explain different characteristics of Distributed System.
2. What is distributed system? Explain different types of distributed system.
3. Explain Challenges in distributed system. What are the limitations of Distributed System?
4. Why would you design a system as a distributed system? List some advantages of distributed systems
5. List three properties of distributed systems.
6. List some disadvantages or problems of distributed systems that local only systems do not show.
7. Define distributed system. Differentiate between an autonomous system and a distributed system with examples.
8. "Distributed system acts as a single coherent system to its end user". Justify the statement with its features and examples.
9. Why do we need a distributed system? Explain the advantages and disadvantages of distributed system in detail.
10. What are the main characteristics (design goals or objectives or requirements) of a distributed system?
11. Why there are challenges in achieving some requirements of a distributed system? Explain the challenges associated with different requirements of distributed system.
12. Define transparency. "Transparency is one the most important feature of a distributed system", justify the statement with example. Explain the challenges in achieving the transparency in distributed system.

13. Use the World Wide Web as an example to illustrate the concept of resource sharing, cli and server. What are the advantages and disadvantages of HTML, URLs and HTTP as c technologies for information browsing? Are any of these technologies suitable as a basis client-server computing in general?
14. We have identified a number of qualities that made the WWW successful: interoperabil portability, remote access, extensibility, and scalability. Which of these do you th contributed most substantially to the Web's success? If any of these qualities had b sacrificed, would the Web still have been successful?
15. The Web did not have performance as one of its early quality goals, which is unusual fo successful system. Why do you think the system was successful anyway? What anything, does this say about the future of computing?
16. Define distributed systems. Give examples of distributed systems.
17. Mention the examples of distributed system.
18. What are the Applications of Distributed system?
19. Write the different trends in distributed systems?
20. What are Advantages of Distributed Systems vs. Centralized?

EXERCISE



Multiple Choice Questions

1. The type of architecture that is considered responsible for the success of application server is said to be.....
   a) Two-tier architecture
   b) Three-tier architecture
   c) n-tier architecture
   d) Peer-to-Peer architecture

2. What is an architectural style?
   a) Defined set of rules only
   b) Undefined rules and guidelines
   c) Predefined set of rules and guidelines
   d) Defined set of guidelines only

3. On what basis is an architectural pattern chosen?
   a) On how much it can compile with the design
   b) Its cost convenience
   c) On how compatible it is with the design and cost convenience
   d) Speed at which it complies with the design

4. Parallel computing is also known as.....
   a) Parallel computation
   b) Parallel processing
   c) Parallel distribution
   d) Parallel development

5. Type of architecture that is considered responsible for success of.....
   a) Two-tier architecture
   b) Three-tier architecture
   c) n-tier architecture
   d) Peer-to-Peer architecture

6. Centralized computing covers many data centers and.....
   a) Minicomputers
   b) Mainframe computers
   c) Supercomputers
   d) Microcomputers

7. Which technique is based on compile-time program transformation for accessing remote data in a distributed-memory parallel system.....
   a) Cache coherence scheme
   b) computation migration
   c) Remote procedure call
   d) message passing

8. If one site fails in distributed system.....
   a) The remaining sites can continue operating
   b) all the sites will stop working
   c) Directly connected sites will stop working
   d) none of the mentioned

9. Uni processor computing is known as.....
   a) Centralized computing
   b) Parallel computing
   c) Distributed computing
   d) Grid computing
10. A computing model of a distributed architecture of large numbers of computers connec to solve a complex problem is called.....
   a) Linear computing
   b) Grid computing
   c) Layout computing
   d) Compound computing

Subjective Questions

1. What are different architecture styles of distributed system? Explain each of them in det
2. Define middleware and explain its importance in distributed system with suita diagram.
3. Describe types of system architecture with suitable example.
4. Define architecture style. Explain their types with example.
5. Define Modifiable middleware. Also define their advantages with suitable example.
6. Define Centralized System with example. Also describe their advantages disadvantages.
7. Define De-centralized System with example. Also describe their advantages disadvantages.
8. Define client-server system with example. Also describe their advantages disadvantages.
9. Define peer-to-peer System with example. Also describe their advantages disadvantages.
10. Define centralized system also describe characteristics of Centralized System example.
11. Describe Architectural styles of distributed system with suitable example.
12. Define Middleware organization of distributed system.
13. Define System Architectures of distributed system with suitable diagram.
14. Compare peer to peer and client server architecture.
15. Define Modifiable middleware with suitable example.
16. Compare centralized and client server architecture with suitable example.
17. Define Layered Architecture. Also compare it with event based architecture
18. Define Object Based Architecture. Also compare it with Data-centered Architecture
19. Define Data-centered Architecture. Also Define Event based architecture with suitab example.
20. Define Hybrid Architecture with suitable example.

ANSWERS KEY

1. (c)	2. (c)	3. (c)	4. (a)	5. (c)	6. (c)	7. (b)	8. (a)	9. (a)	10. (c)

Distributed Systems (3 Cr.)

Course Code: CACS352

Year/Semester: II/VI

Class Load: 4Hrs./Week

(Theory: 3Hrs. Tutorials: 1Hr)

By Prashant Gautam

Course Descriptions

→Give an insight on how modern distributed systems operate.

Objectives

→To make familiar with different aspect of distributed system, middleware, system level support and different issues in designing algorithms and finally systems.

Unit 1: Introduction (4 Hrs.)

Background

1.1 Characteristics

1.2 Design Goals

1.3 Types of Distributed Systems

1.4 Case Study: WWW

Background

- The pace at which computer systems change was, is, and continues to be overwhelming.
- 1945→ when the modern computer era began.
- 1945-1985→computers were large and expensive.
- Moreover, for lack of a way to connect them, these computers operated independently from one another.
- mid-1980s → Development of powerful microprocessors
  → invention of high-speed computer networks

Development of powerful microprocessors

- Initially, these were 8-bit machines, but soon 16-, 32-, and 64-bit CPUs became common.

Invention of high-speed computer networks

- **Local-area networks or LANs** allow thousands of machines within a building or campus to be connected in such a way that small amounts of information can be transferred in a few microseconds or so.
- Larger amounts of data can be moved between machines at rates of billions of bits per second (bps).
- **Wide-area networks or WANs** allow hundreds of millions of machines all over the earth to be connected at speeds varying from tens of thousands to hundreds of millions bps, and sometimes even faster.

Smartphone as the most impressive outcome

- Packed with sensors, lots of memory, and a powerful CPU, these devices are nothing less than full-fledged computers.
- Of course, they also have networking capabilities.

plug computers and nano computers

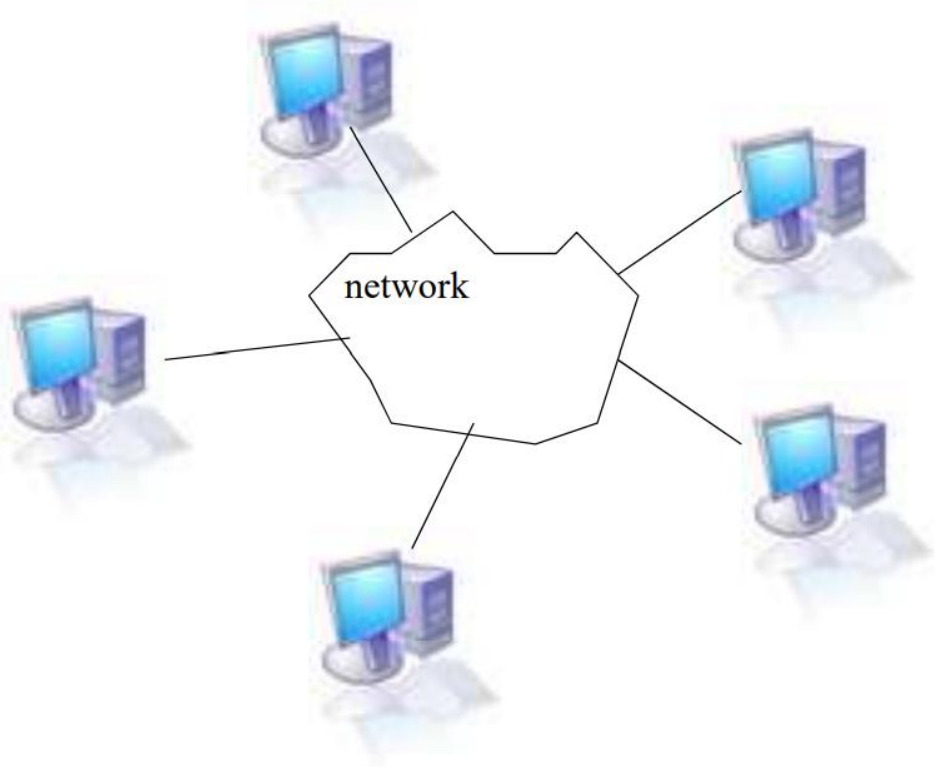
- These small computers, often the size of a power adapter, can often be plugged directly into an outlet and offer near-desktop performance.

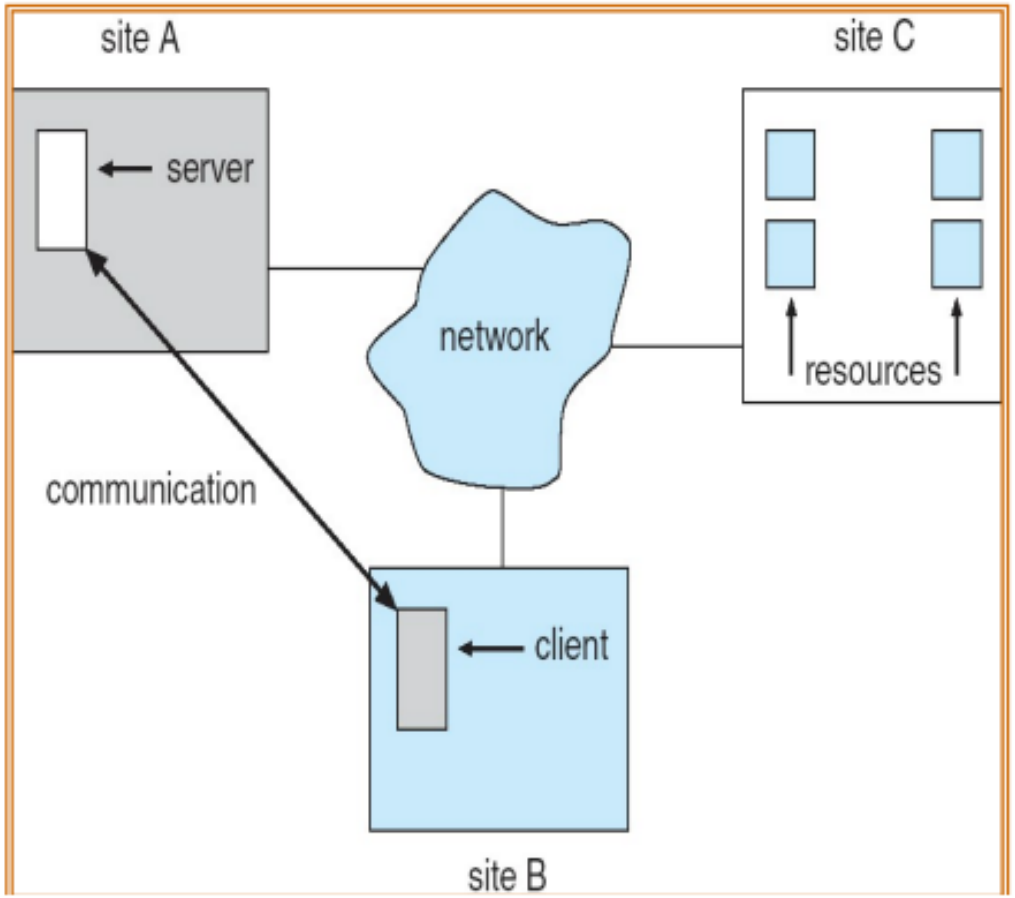
Distributed System

- The size of a distributed system may vary from a handful of devices, to millions of computers.
- The interconnection network may be wired, wireless, or a combination of both.
- Moreover, distributed systems are often highly dynamic, in the sense that computers can join and leave, with the topology and performance of the underlying network almost continuously changing.

What is a distributed system?

- A distributed system is a collection of autonomous computing elements that appears to its users as a single coherent system.
  - a collection of computing elements(Node: H/w device or S/w Process) each being able to behave independently of each other.
  - users (people or applications) believe they are dealing with a single system.





1. Group of autonomous hosts, each host executes components and operates distribution applications.
2. Hosts are Geographically dispersed/separated over (LAN, WAN,……)
3. Hosts Connected via a network
4. The network is used to: transfer messages and mail, and execute applications: airline reservation, stock control, ….)

1.1. Characteristic 1: collection of autonomous computing elements

- In a DS there are multiple components that may be decomposed further.
- These components are autonomous, i.e. they possess full control over their parts at all times.

1.1 Characteristic 2: single coherent system

- In a single coherent system the collection of nodes as a whole operates the same, no matter where, when, and how interaction between a user and the system takes place.

Middleware and distributed systems

- To assist the development of distributed applications, distributed systems are often organized to have a separate layer of software that is logically placed on top of the respective operating systems of the computers that are part of the system.
- This organization is shown

in Fig. 1, leading to what is known as middleware.

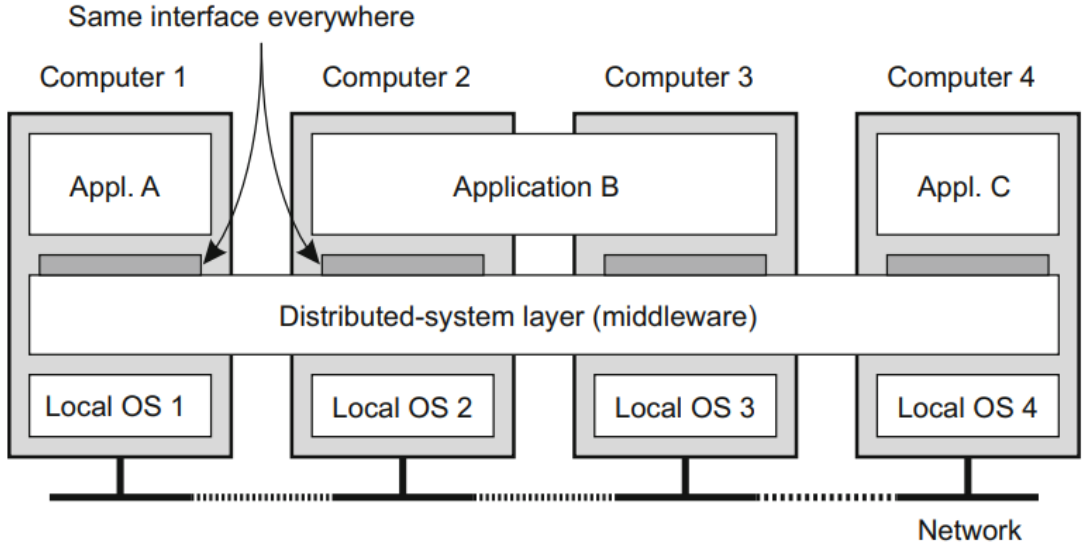


Fig. 1. A distributed system organized as middleware. The middleware layer extends over multiple machines, and offers each application the same interface

1.2 Design Goals

- Making Resources Accessible
- Distribution Transparency
- Openness
- Scalability

1.2.1 Making Resources Accessible

- Goal → to make it easy for the users (and applications) to access remote resources, and to share them in a controlled and efficient way.
- Resources→ printers, computers, storage facilities, data, files, Web pages, and networks, etc.

Why Resources Sharing ?

- Economic →

It is cheaper to let a printer be shared by several users in a small office than having to buy and maintain a separate printer for each user.

Likewise, it makes economic sense to share costly resources such as supercomputers, high-performance storage systems, imagesetters, and other expensive peripherals.

Security Concern

- As connectivity and sharing increase, security is becoming increasingly important.
- There is much room for improvement.
- Technique such as cryptographic encryption can be used while sharing contents.

1.2.2 Distribution Transparency

- Goal → to hide the fact that its processes and resources are physically distributed across multiple computers.
- In other words, it tries to make the distribution of processes and resources transparent, that is, invisible, to end users and applications.

Types of Transparency

- The concept of transparency can be applied to several aspects of a distributed system, the most important ones shown below:

Transparency	Description
Access	Hide differences in data representation and how an object is accessed
Location	Hide where an object is located
Relocation	Hide that an object may be moved to another location while in use
Migration	Hide that an object may move to another location
Replication	Hide that an object is replicated
Concurrency	Hide that an object may be shared by several independent users
Failure	Hide the failure and recovery of an object

Access Transparency

- For example, a distributed system may have computer systems that run different operating systems, each having their own file-naming conventions. Differences in naming conventions, differences in file operations, or differences in how low-level communication with other processes is to take place, are examples of access issues that should preferably be hidden from users and applications.

Location Transparency

- An example of a such a name is the uniform resource locator (URL) http://www.distributed-systems.net/index.php, which gives no clue about the actual location of the site's Web server. The URL also gives no clue as to whether the file index.php has always been at its current location or was recently moved there. F

Relocation Transparency

- For example, the entire site may have been moved from one (part of a) data center to another to make more efficient use of disk space, yet users should not notice.

Migration Transparency

- A typical example is communication between mobile phones: regardless whether two people are actually moving, mobile phones will allow them to continue their conversation.

Replication transparency

- For example, resources may be replicated to increase availability or to improve performance by placing a copy close to the place where it is accessed.

concurrency transparency

- For example, two independent users may each have stored their files on the same file server or may be accessing the same tables in a shared database. In such cases, it is important that each user does not notice that the other is making use of the same resource.

Failure Transparency

- For example, when contacting a busy Web server, a browser will eventually time out and report that the Web page is unavailable. At that point, the user cannot tell whether the server is actually down or that the network is badly congested.

1.2.3 Openness

- Open Distributed System →system that offers services according to standard rules that describe the syntax and semantics of those services.
- For example, in computer networks, standard rules govern the format, contents, and meaning of messages sent and received. Such rules are formalized in protocols. In distributed systems, services are generally specified through interfaces, which are often described in an Interface Definition Language (IDL).

1.2.4 Scalability

- Scalability of a system can be measured along at least three different dimensions:
  - **A system can be scalable with respect to its size**

    add more users and resources to the system
  - **A geographically scalable system**

    add more users and resources to the system
  - **A system can be administratively scalable**

    it can still be easy to manage even if it spans many independent administrative organizations.

1.2.5 Pitfalls when developing DS

- False assumptions that everyone makes when developing a distributed application for the first time:
  1. The network is reliable.
  2. The network is secure.
  3. The network is homogeneous.
  4. The topology does not change.
  5. Latency is zero.
  6. Bandwidth is infinite.
  7. Transport cost is zero.
  8. There is one administrator

1.3 Types of Distributed Systems

1- Distributed Computing Systems : Focus on computation

--Cluster Computing Systems
--Grid Computing Systems

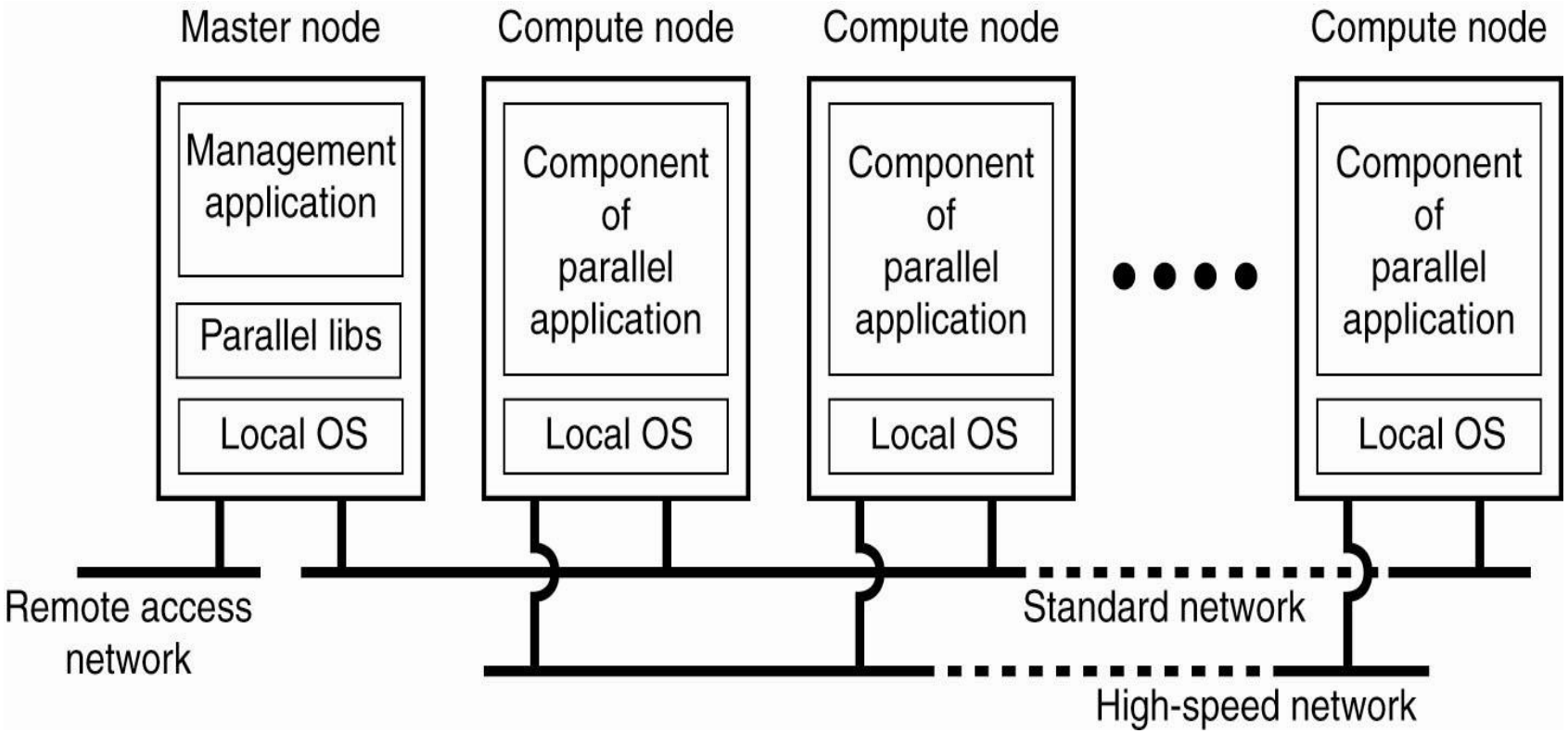
2- Distributed Information Systems: Focus on interoperability

--Transaction Processing Systems
--Enterprise Application Integration
(Exchange info via RPC or RMI)

3- Distributed Pervasive Systems (usually small, battery-powered systems, Mobile & wireless): Focus on mobile, embedded, communicating

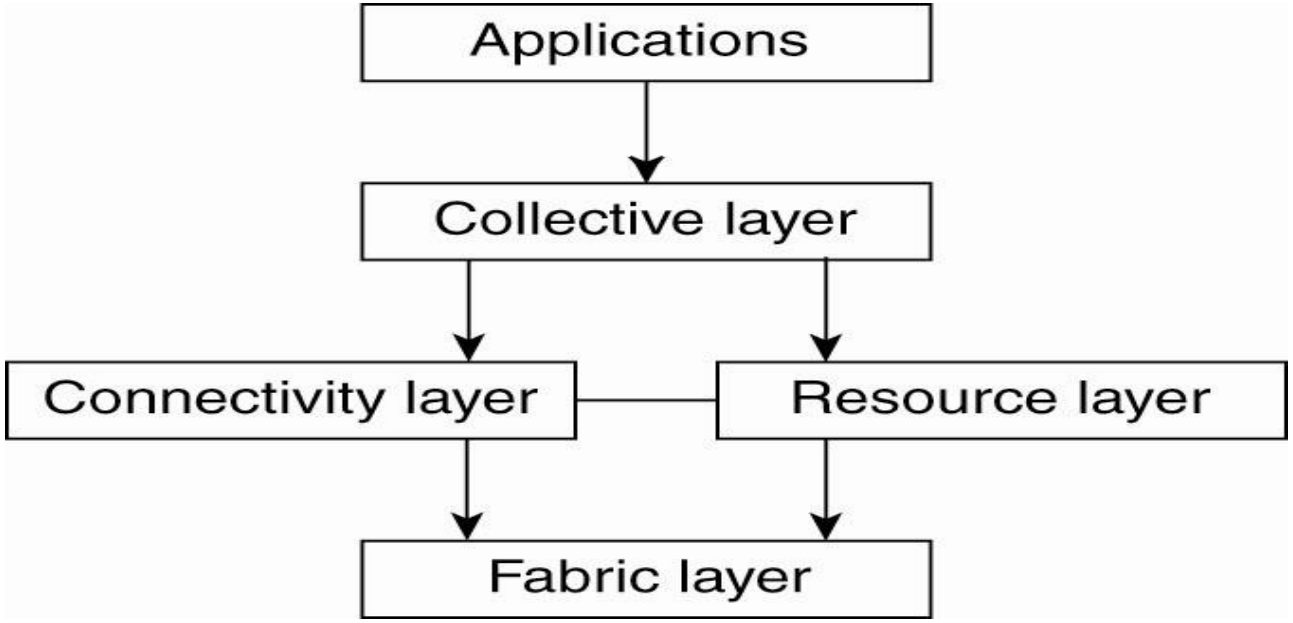
--Home Systems (e.g. Smart phones, PDAs)
--Electronic Health care systems (Heart monitors, BAN: Body Area Networks)
--Sensor Networks (distributed Databases connected wirelessly)

Cluster Computing Systems



- **Hooking up a collection of simple computers via high-speed networks to build a supercomputing environment**
- **Mostly homogenous**
- **Example: server clusters at Banks**

Grid Computing Systems



- have a high degree of heterogeneity
-Users and resources from different organizations are brought together to allow collaboration (i.e. a V.O. = Virtual Organization)
- Members belonging to the same V.O. have access rights to a common set of resources (e.g. Police and some local agencies may form a computing grid)

Transaction Processing Systems (1)

Primitive	Description
BEGIN_TRANSACTION	Mark the start of a transaction
END_TRANSACTION	Terminate the transaction and try to commit
ABORT_TRANSACTION	Kill the transaction and restore the old values
READ	Read data from a file, a table, or otherwise
WRITE	Write data to a file, a table, or otherwise

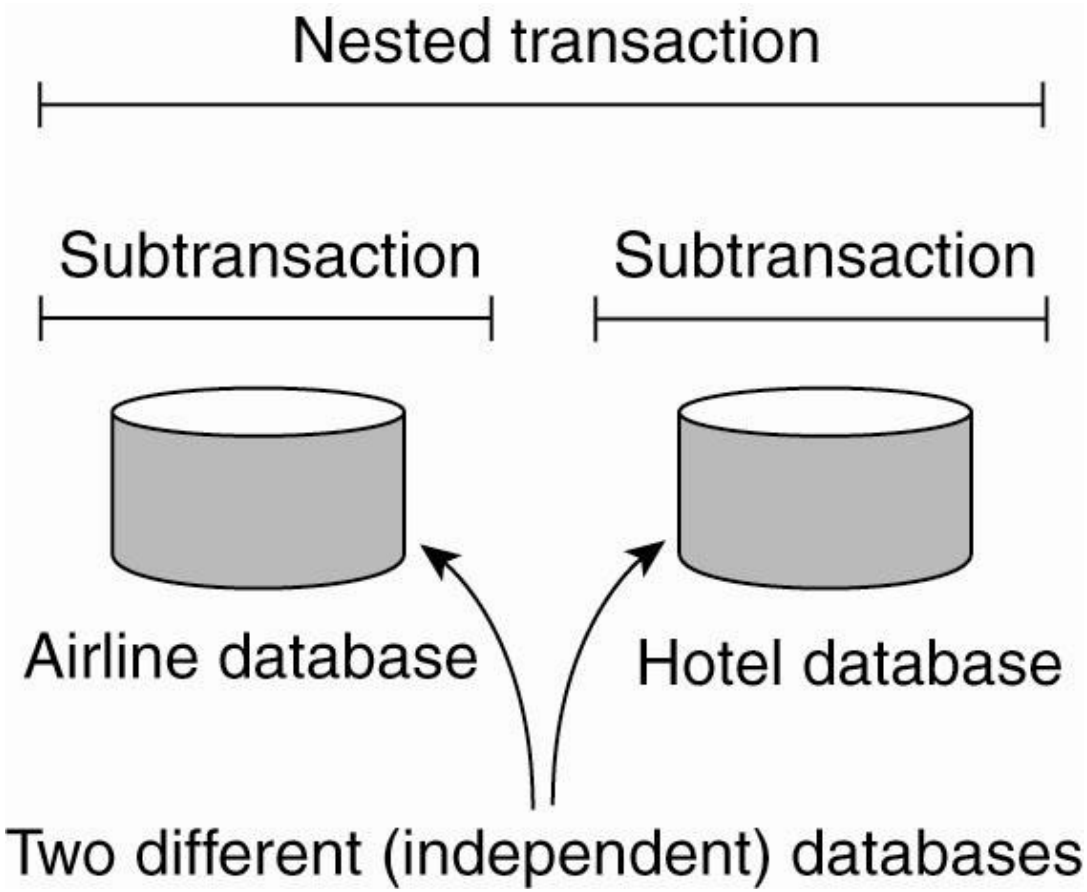
```
BEGIN_TRANSACTION
salary1 = doctor1.getSalary()
doctor1.setSalary(salary1 + bonus)
salary2 = doctor2.getSalary()
doctor2.setSalary(salary2 - bonus)
END_TRANSACTION
```

Transaction Processing Systems (2)

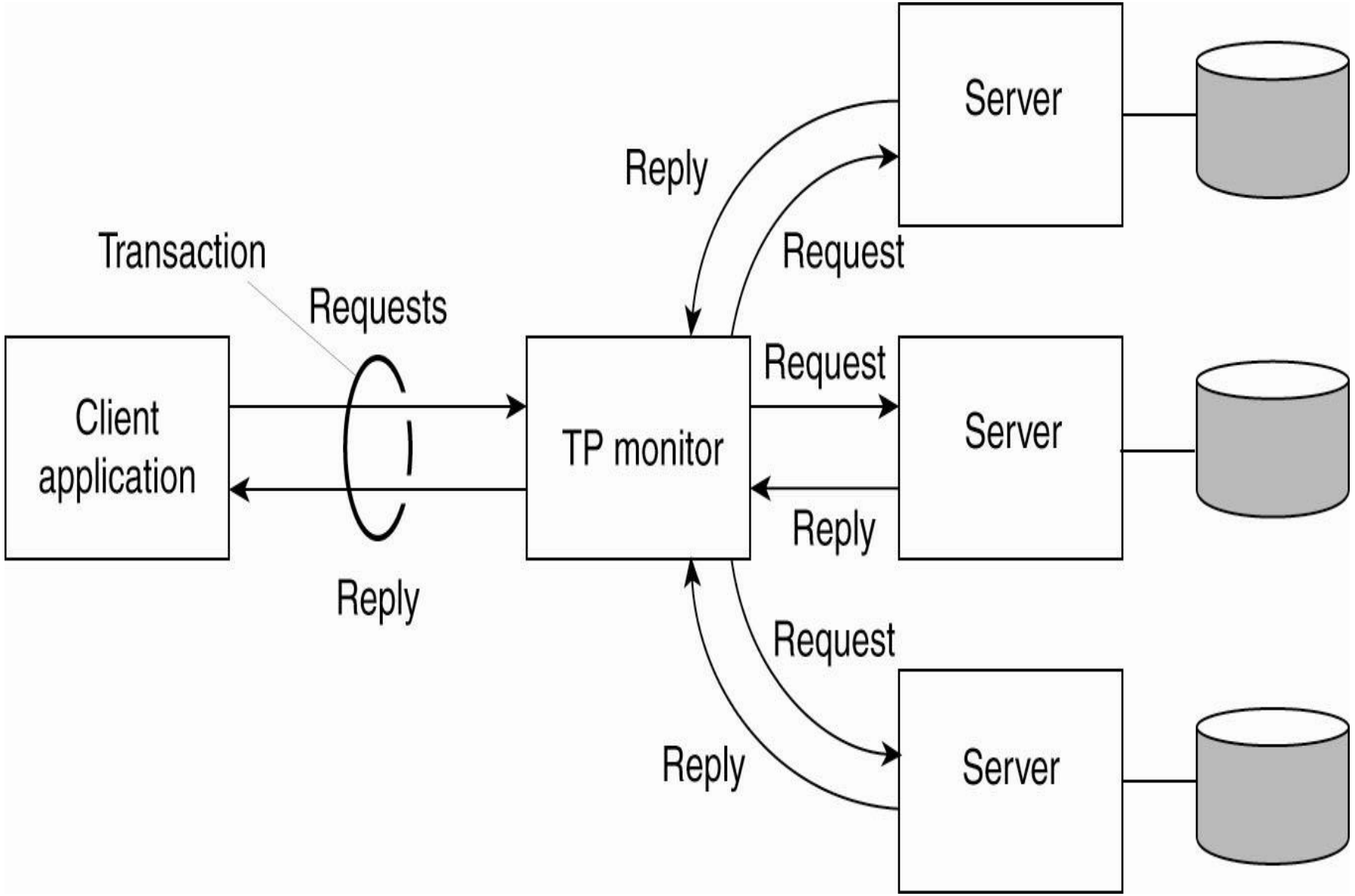
Characteristic:

- Atomic: The transaction happens indivisibly.
- Consistent: The transaction does not violate system invariants.
- Isolated: Concurrent transactions do not interfere with each other.
- Durable: Once a transaction commits, the changes are permanent.

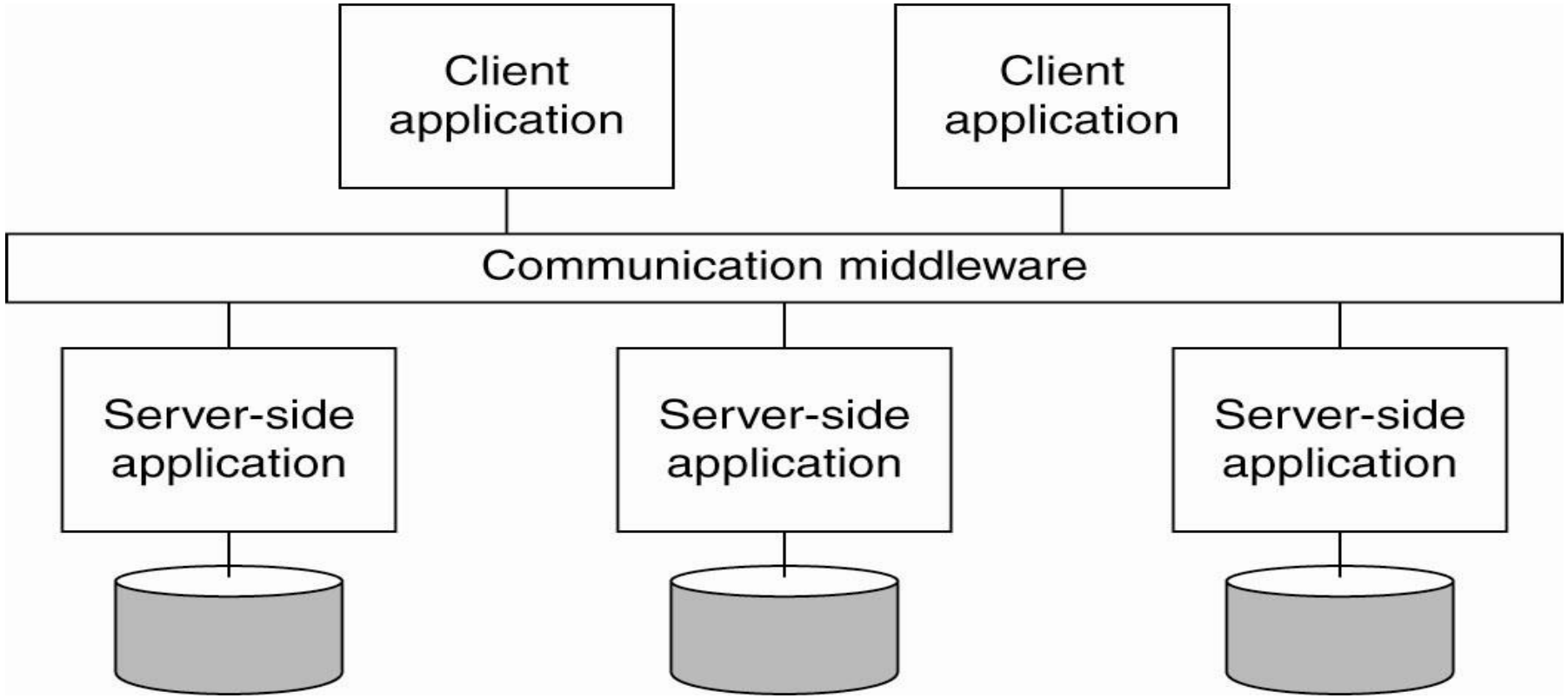
Transaction Processing Systems (3)



Transaction Processing Systems (4)



Enterprise Application Integration



-RPC (Remote Procedure Call)
-RMI (Remote Method Invocation)

Distributed Pervasive Systems

- Embrace contextual changes
- (a phone now is a web access device. A device must continuously be aware of the fact that its environment may change)

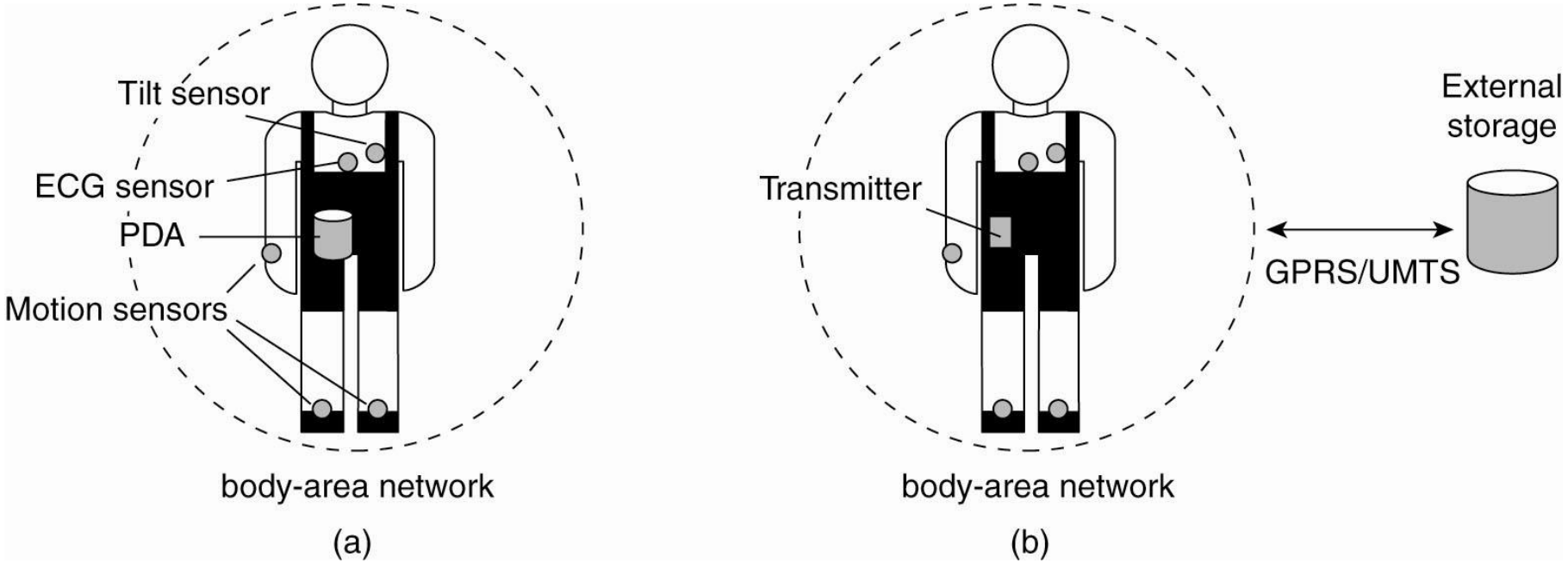
- Encourage ad hoc composition
- (used differently by different users, e.g. PDA)

- Recognize sharing as the default
- (easily read, store, manage, and share info)

Electronic Health Care Systems (1)

- Where and how should monitored data be stored?
- How can we prevent loss of crucial data?
- What infrastructure is needed to generate and propagate alerts?
- How can physicians provide online feedback?
- How can extreme robustness of the monitoring system be realized?
- What are the security issues and how can the proper policies be enforced?

Electronic Health Care Systems (2)

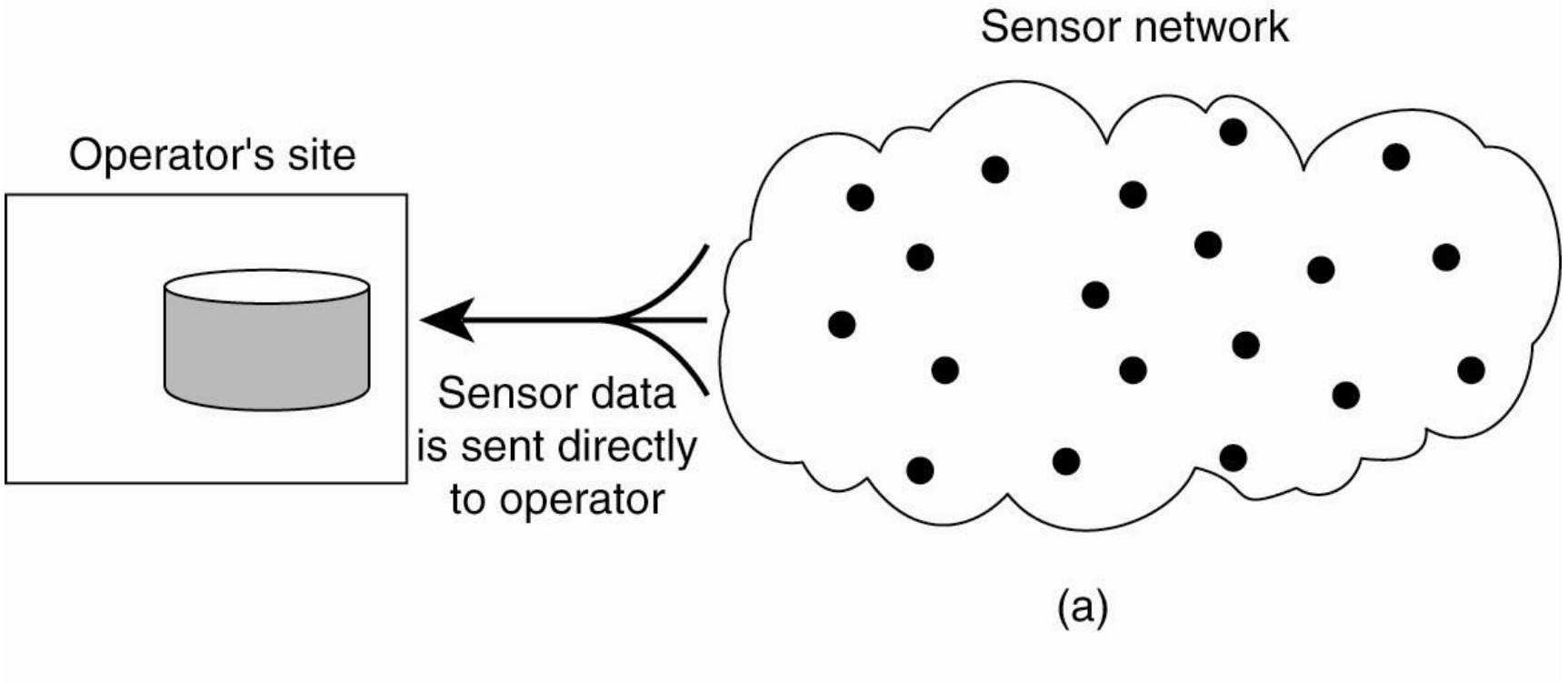


Monitoring a person in a pervasive electronic health care system, using (a) a local hub or - (b) a continuous wireless connection.

Sensor Networks (1)

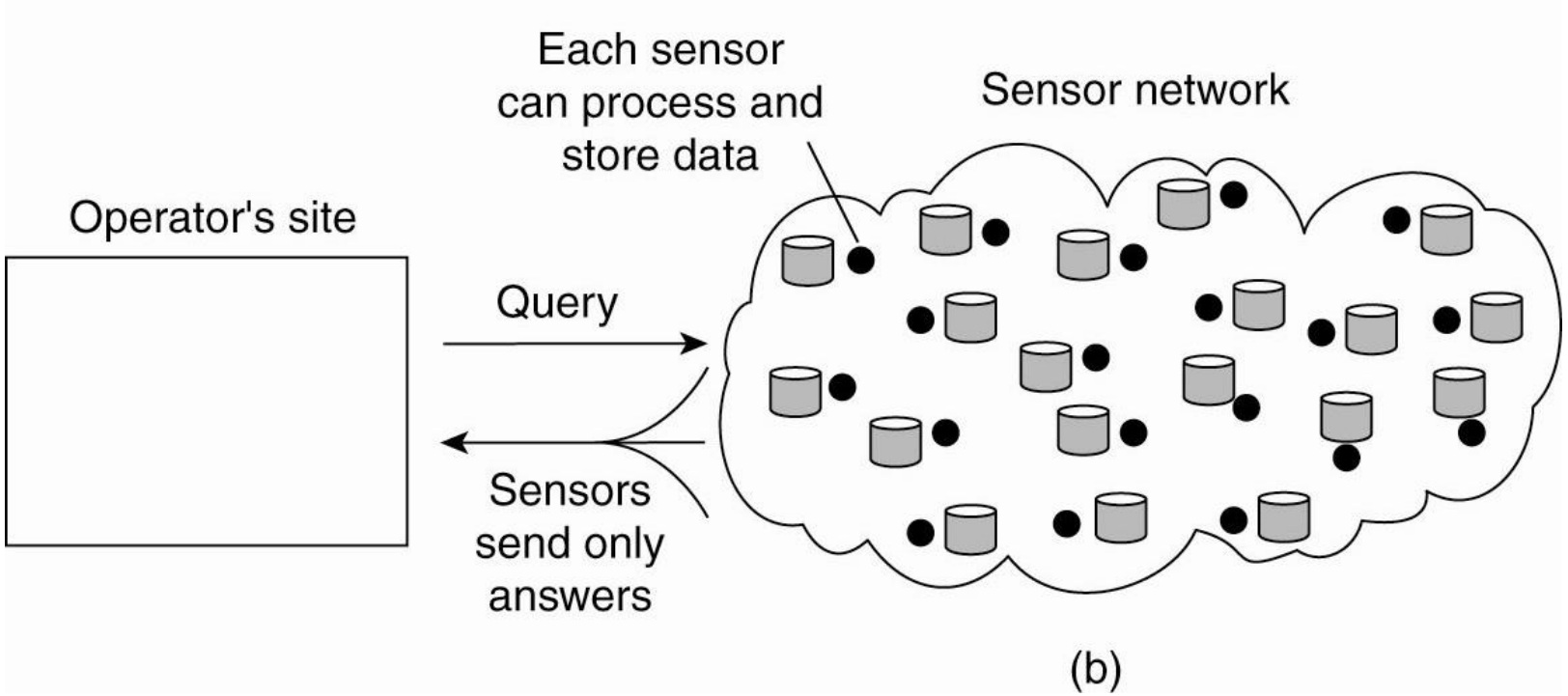
- How do we (dynamically) set up an efficient tree in a sensor network?
- How does aggregation of results take place? Can it be controlled?
- What happens when network links fail?

Sensor Networks (2)



Organizing a sensor network database, while storing and processing data (a) only at the operator's site or …

Sensor Networks (3)



Organizing a sensor network database, while storing and processing data … or (b) only at the sensors.

Unit 2: Architecture (4 Hrs.)

Background

2.1 Architectural Structure

2.2 Middleware organization

2.3 System Architecture

2.4 Example Architecture

Background

- Distributed systems are often complex pieces of software of which the components are dispersed across multiple machines.
- To master their complexity, it is crucial that these systems are properly organized.
- The software architectures tell us how the various software components are to be organized and how they should interact.

2.1 ARCHITECTURAL STYLE

1. Layered architectures
2. Object-based architectures
3. Data-centered architectures
4. Event-based architectures

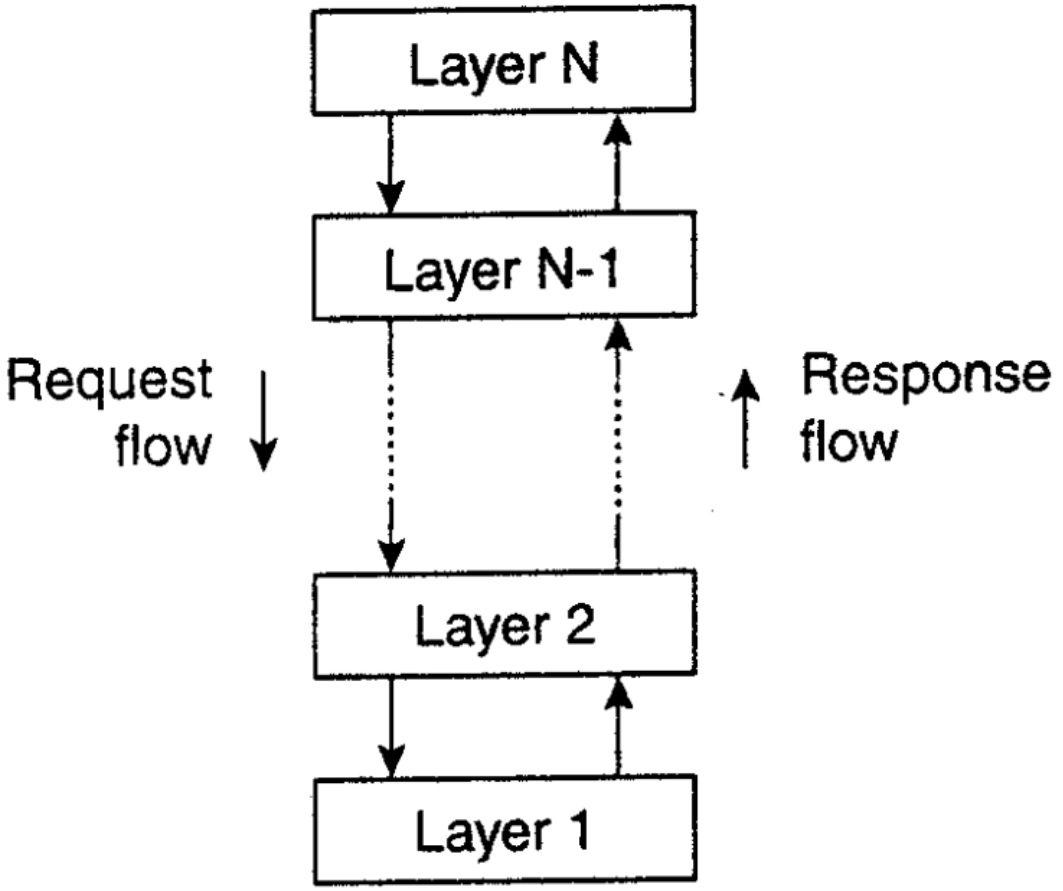
Layered architectures

- **Basic Idea:**

  --components are organized in a layered fashion where a component at layer L; is allowed to call components at Li, but not the other way around, as shown in Fig.

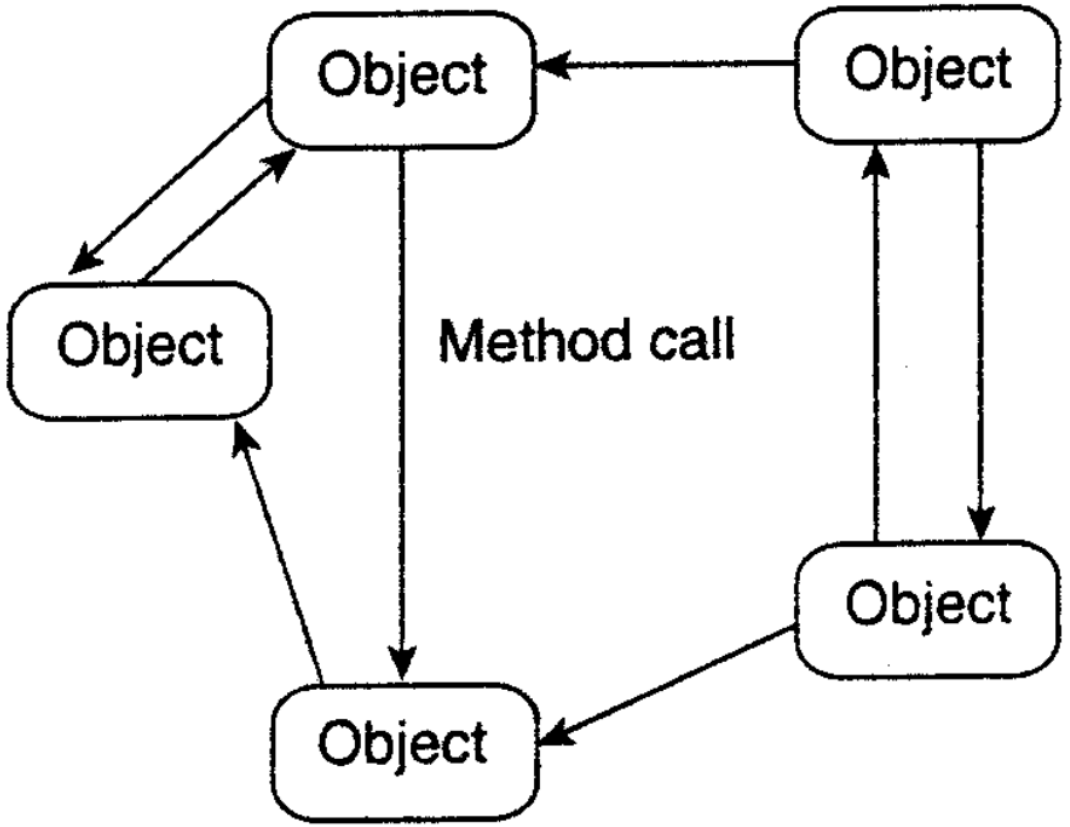
- Widely adopted by the networking community.

- Key observation is that control generally flows from layer to layer: requests go down the hierarchy whereas the results flow upward.



Object-based architectures

- This software architecture matches the client-server system architecture.
- Each object corresponds to what we have defined as a component, and these components are connected through a (remote) procedure call mechanism.



Data-centered architectures

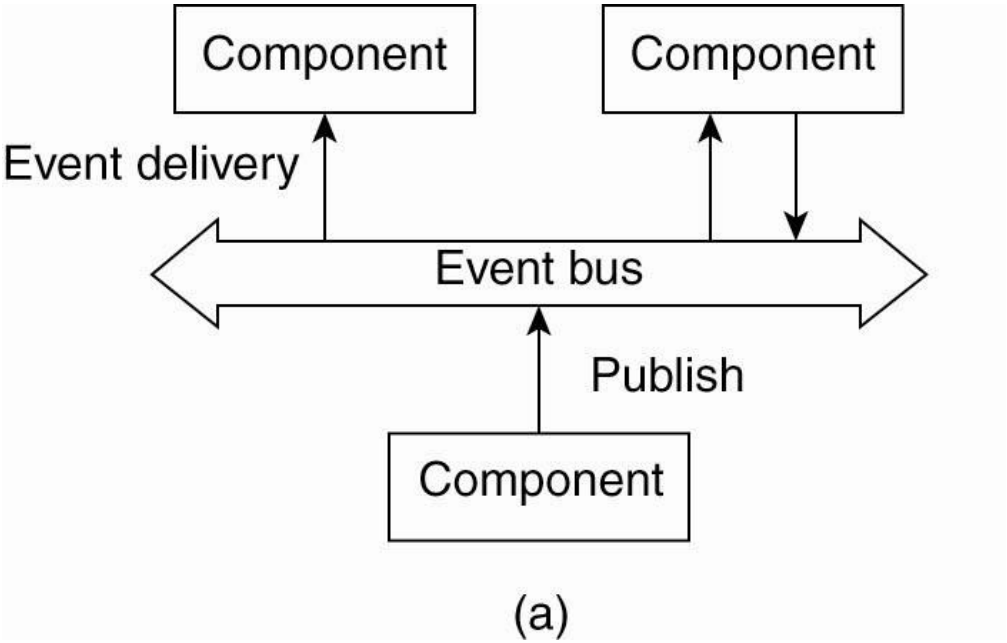
- Basic Idea:

--processes communicate through a common (passive or active) repository.

- As important as the layered and object based architectures. Examples:
  - Network applications: communicate through shared distributed file systems
  - Web applications: processes communicate through the use of shared Web-based data services

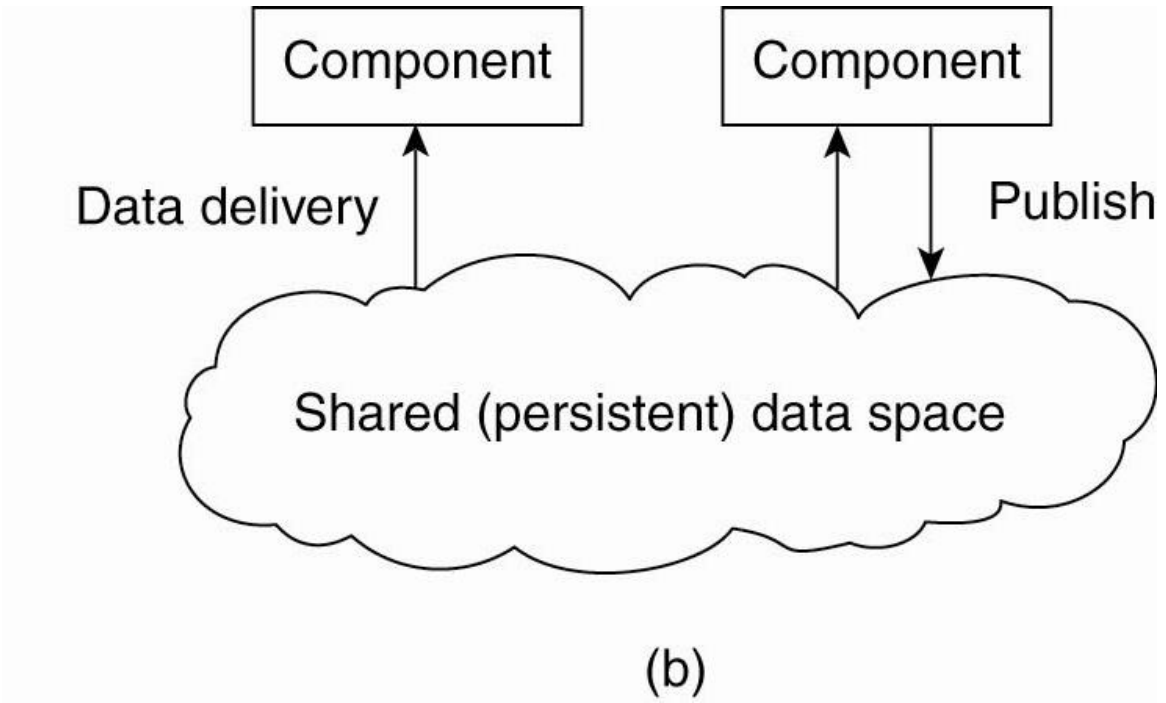
Event-based Architectural Style

- Processes essentially communicate through the propagation of events
  - Optionally also carry data
- Publish/subscribe systems
  - Only subscribed processes will receive the published events
  - **Referentially decoupled**: Processes are loosely coupled



Shared Data-space Architectural Style

- Combine event-based architectures and data-centered architectures
  - Processes are decoupled in time



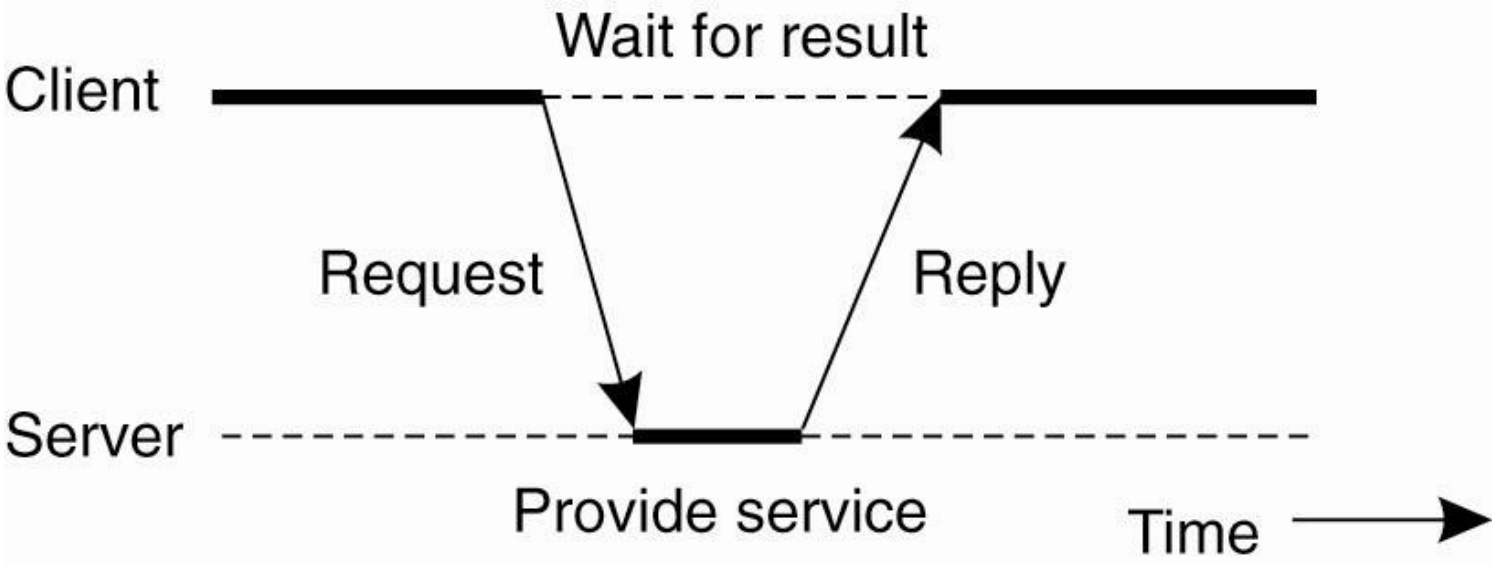
2.2 SYSTEM ARCHITECTURES

How many distributed systems are actually organized by considering where software components are placed.

- Centralized Architectures
- Decentralized Architectures
- Hybrid Architectures

Centralized Architectures

- Client-server model:
  - Processes are divided into two (possibly overlapping) groups
  - **Server**: a process implementing a specific service
    for example, a file system service or a database service.
  - **Client**: a process sending a request to a server and subsequently waiting for the server's reply



Communication between Clients and Servers

- Connectionless protocol
  - Efficient, but unreliable
    - Good for LANs

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.

- Connection-oriented protocol
  - Inefficient, but reliable
    - Good for WANs
- For example, virtually all Internet application protocols are based on reliable TCP/IP connections. In this case, whenever a client requests a service, it first sets up a connection to the server before sending the request. The server generally uses that same connection to send the reply message, after which the connection is torn down. The trouble is that setting up and tearing down a connection is relatively costly, especially when the request and reply messages are small.

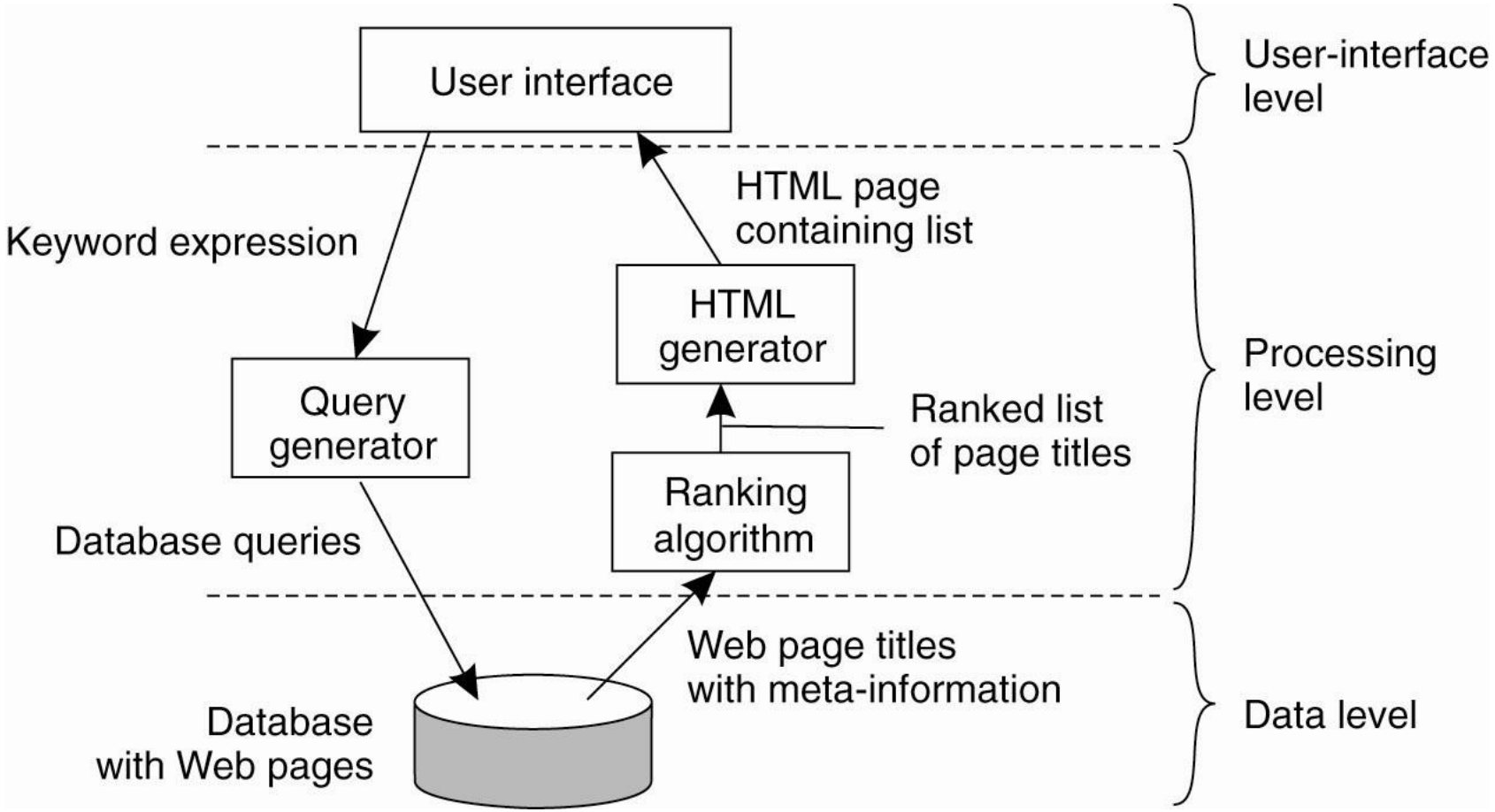
Application Layering

- Traditional three-layered view:
  - User-interface layer
    - Contains units for an application's user interface
  - Processing layer
    - Contains the functions of an application, i.e. without specific data
  - Data layer
    - Contains the data that a client wants to manipulate through the application components

- Observation:
  - This layering is found in many distributed information systems, using traditional database technology and accompanying applications.

Internet Search Engine

- The core : information retrieval part



More Examples

- A Stock Brokerage System
  - User Interface
  - Process Level
    - Analysis of financial data requires sophisticated methods and techniques from statistics and artificial intelligence
  - Data Level
    - Financial database
- Word Processor

Data Level

- Persistency of data
- Keeping data consistent across different applications
- Database
  - Relational database
  - Object-oriented database

Assignment-II (January 23rd)

- Difference between connectionless and connection oriented Protocol. Also focus on their comparative advantages and disadvantages.
- What is the main importance of separating the data level and application layer in Multi-tier Client Server Model. Illustrate.

Multi-Tiered Architecture

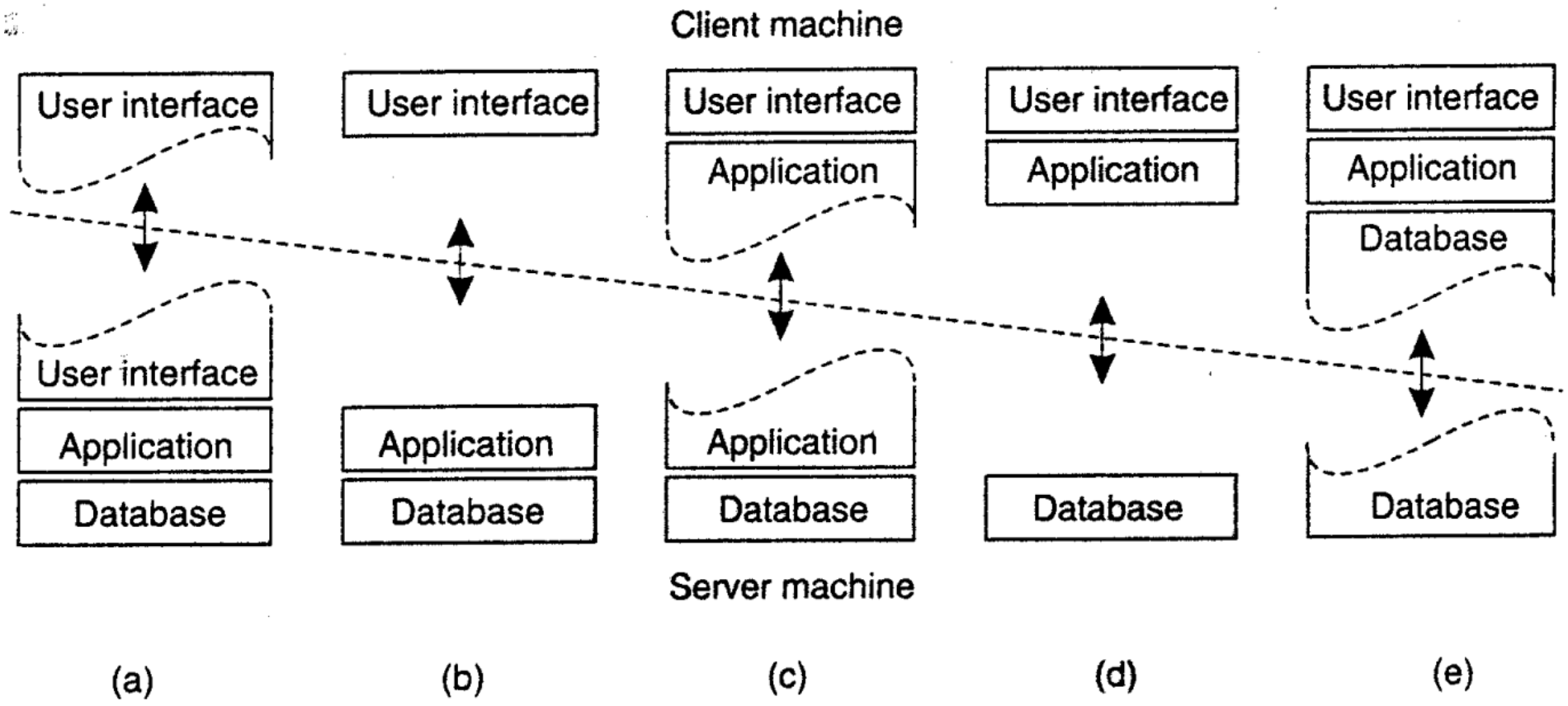
- The distinction into three logical levels as discussed so far, suggests a number of possibilities for physically distributing a client-server application across several machines.
- The simplest organization is to have only two types of machines:

  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

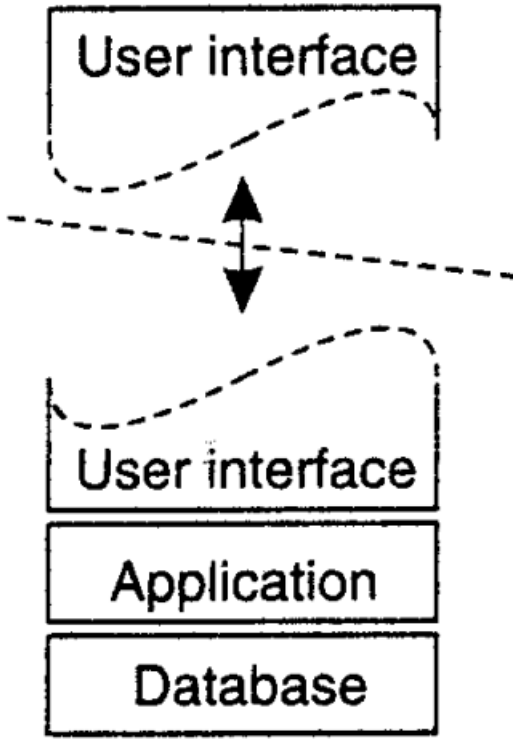
- In this organization everything is handled by the server while the client is essentially no more than a dumb terminal, possibly with a pretty graphical interface.

Alternative Client Server Organization



Client Server organization-I

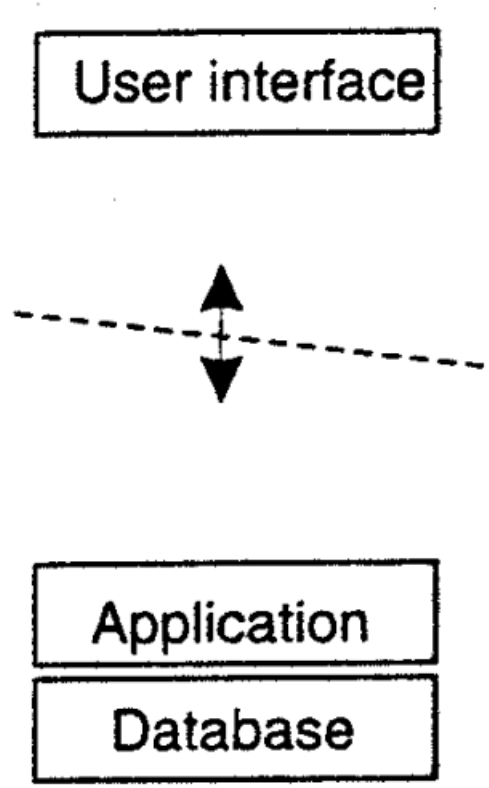
- One possible organization is to have only the terminal-dependent part of the user interface on the client machine and give the applications remote control over the presentation of their data.



(a)

Client Server organization-II

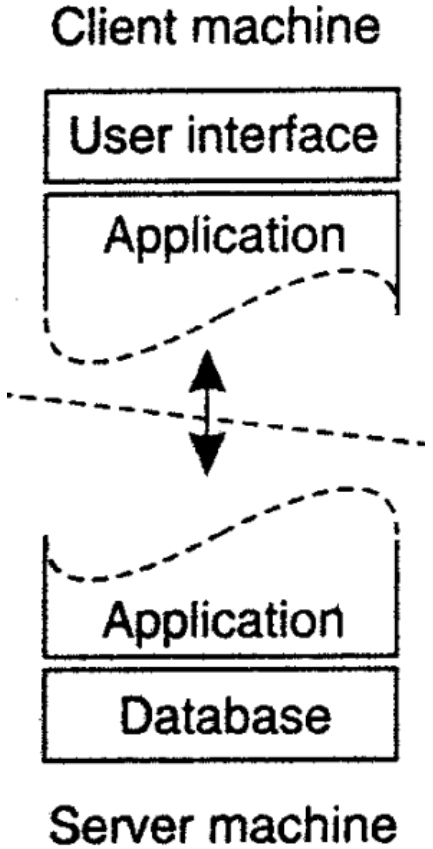
- An alternative is to place the entire user-interface software on the client side.
- In such cases, we essentially divide the application into a graphical front end, which communicates with the rest of the application (residing at the server) through an application-specific protocol.
- In this model, the front end (the client software) does no processing other than necessary for presenting the application's interface.



(b)

Client Server organization-III

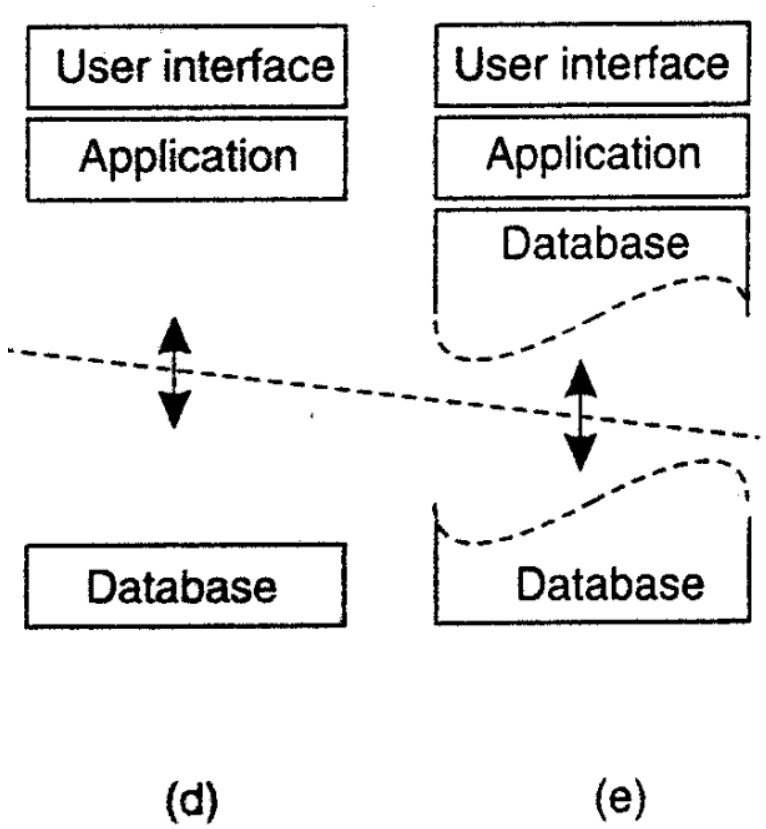
- Move part of the application to the front end.
- EG1. the application makes use of a form that needs to be filled in entirely before it can be processed. The front end can then check the correctness and consistency of the form, and where necessary interact with the user.
- EG2. a word processor in which the basic editing functions execute on the client side where they operate on locally cached, or in-memory data. but where the advanced support tools such as checking the spelling and grammar execute on the server side



(c)

Popular Client Server organization

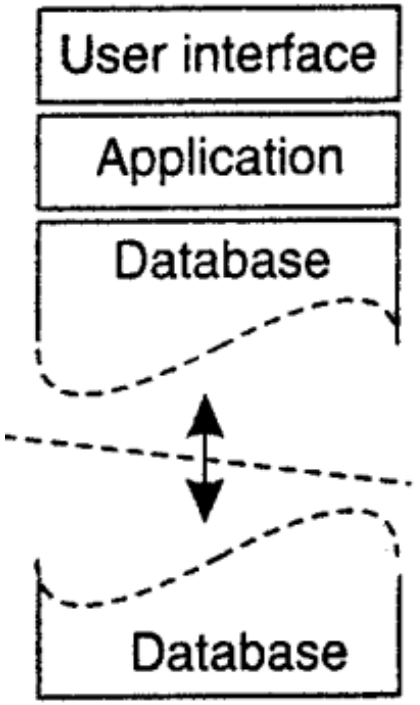
- used where the client machine is a PC or workstation, connected through a network to a distributed file system or database.
- Essentially, most of the application is running on the client machine, but all operations on files or database entries go to the server.



Example

- Many banking applications run on an end-user's machine where the user prepares transactions and such.
- Once finished, the application contacts the database on the bank's server and uploads the transactions for further processing.

- represents the situation where the client's local disk contains part of the data.
- For example, when browsing the Web, a client can gradually build a huge cache on local disk of most recent inspected Web pages.



(e)

Points to be Noted

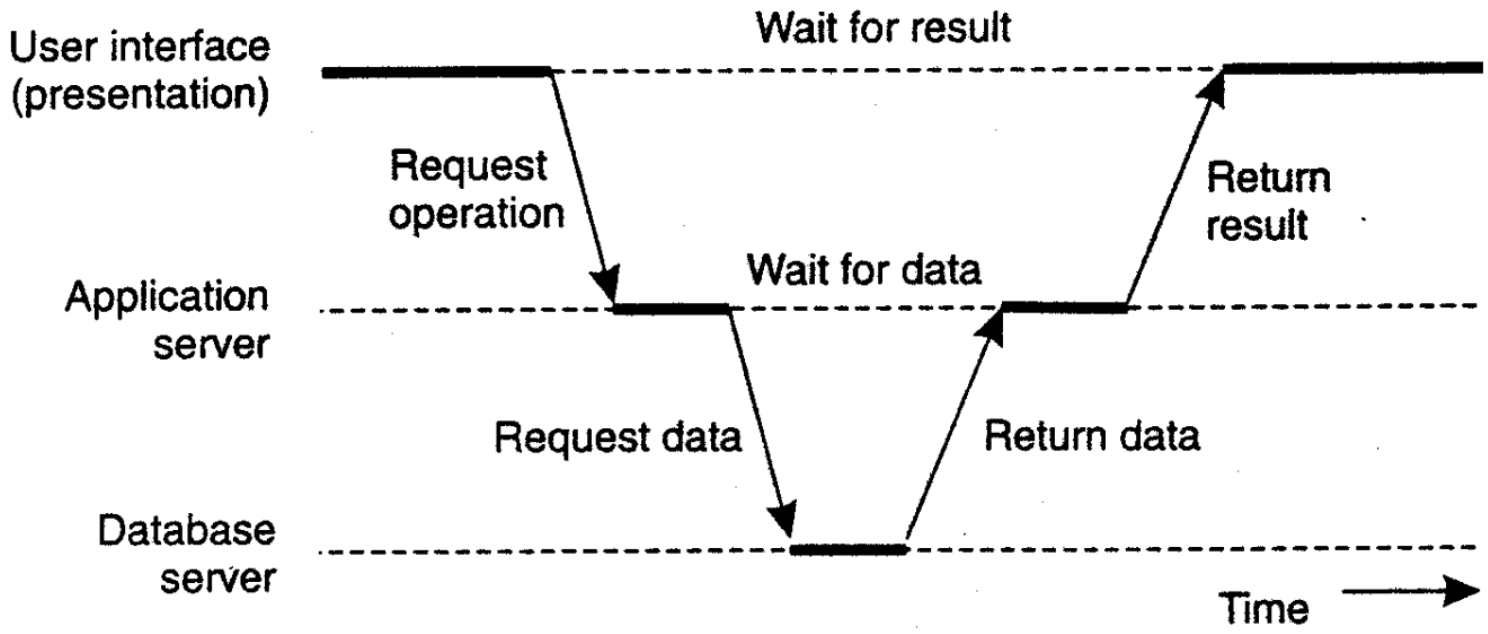
- There been a strong trend to move away from the configurations shown in Fig (d) and Fig (e) in those case that client software is placed at end-user machines.
- In these cases, most of the processing and data storage is handled at the server side.
- The reason for this is simple: although client machines do a lot, they are also more problematic to manage.
- Having more functionality on the client machine makes client-side software more prone to errors and more dependent on the client's underlying platform (i.e., operating system and resources).
- From a system's management perspective, having what are called fat clients is not optimal.
- Instead the thin clients as represented by the organizations shown in Fig (a)-(c) are much easier, perhaps at the cost of less sophisticated user interfaces and client-perceived performance.

Contd…
Point to be noted

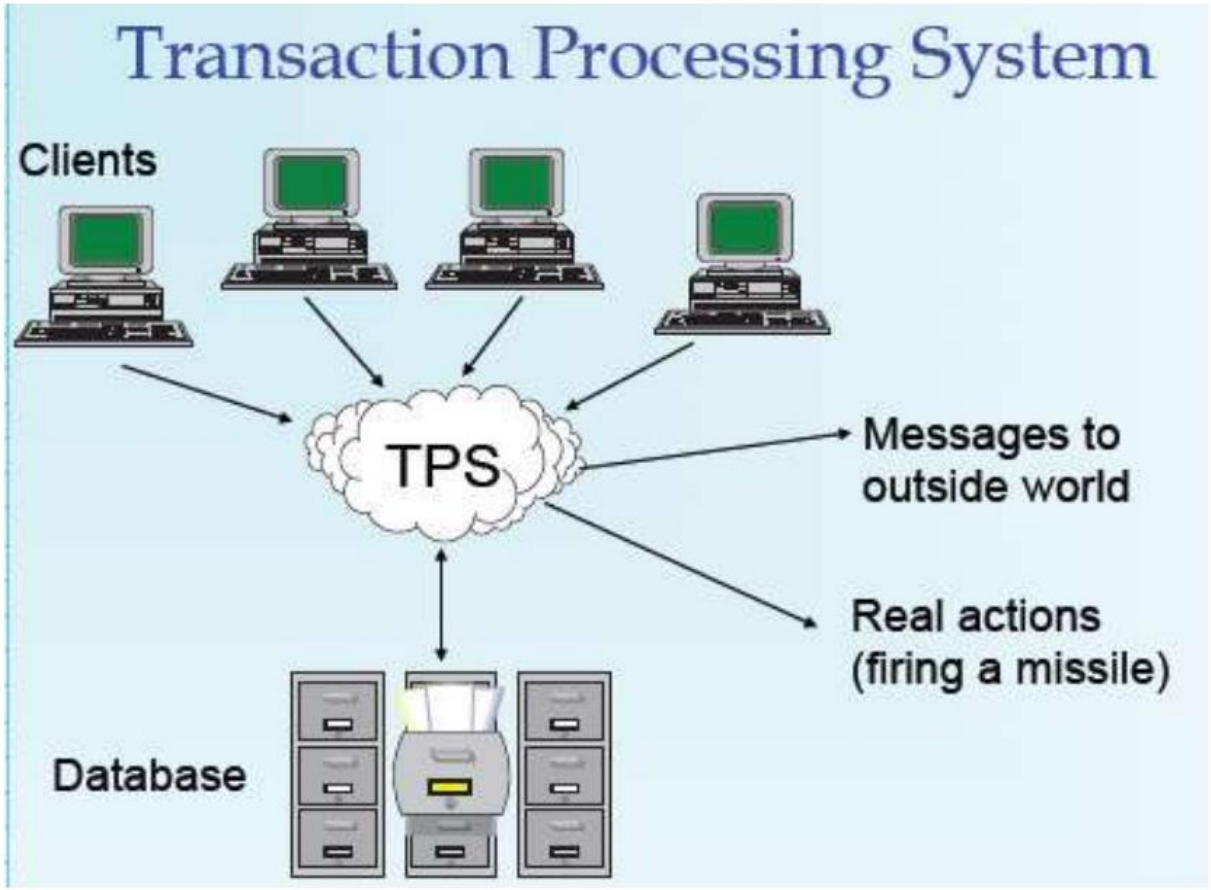
- Note that this trend does not imply that we no longer need distributed systems.
- On the contrary, what we are seeing is that server-side solutions are becoming increasingly more distributed as a single server is being replaced by multiple servers running on different machines.
- In particular, when distinguishing only client and server machines as we have done so far, we miss the point that a server may sometimes need to act as a client, leading to a (physically) three-tiered architecture.

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.
- Eg. Transaction Processing : a separate process, called the transaction processing monitor, coordinates all transactions across possibly different data servers



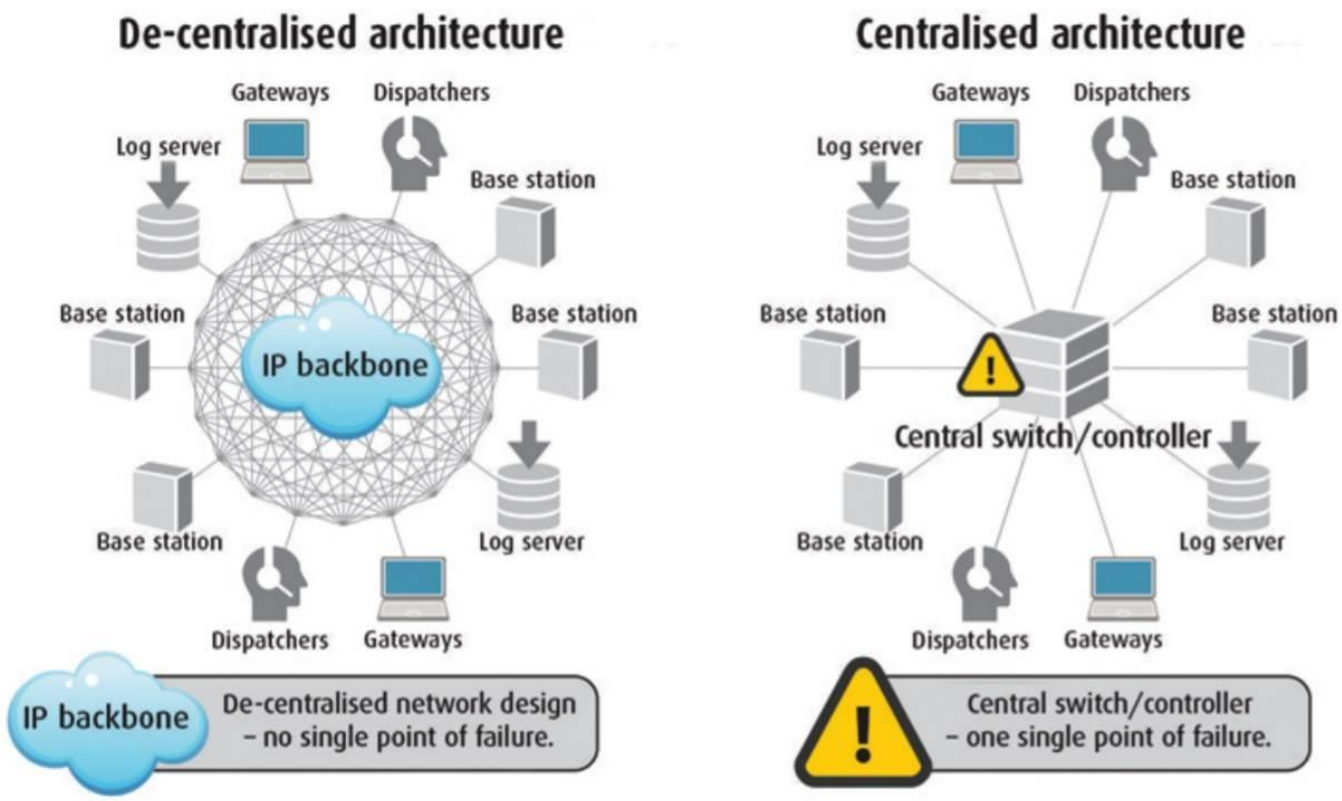
Transaction processing system



Example

- Another, but very different example where we often see a three-tiered architecture is in the organization of Web sites.
- In this case, a Web server acts as an entry point to a site, passing requests to an application server where the actual processing takes place.
- This application server, in turn, interacts with a database server.

Decentralized Architectures



- Multitiered client-server architectures → Vertical Distribution
- Vertical distribution is only one way of organizing client-server applications.
- Modern architectures →Horizontal Distribution → AKA peer-to-peer systems

P2P System

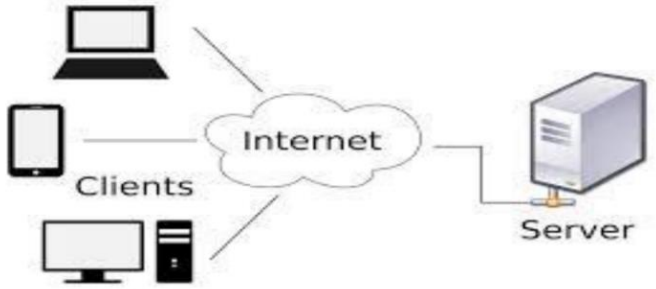
- Peer – to – Peer Computing model is based on how we (human) communicate in real world.
- If we need something then we communicate directly to other corresponding peers (may be friends) who may in turn refer us to their corresponding peers for working towards completion of the request.
- Thus there is a direct access between the peers without any third party intervention.

Examples

- Easy file sharing
- Efficient instant messaging
- Smooth voice communication
- Secure search and communication network
- High performance computing

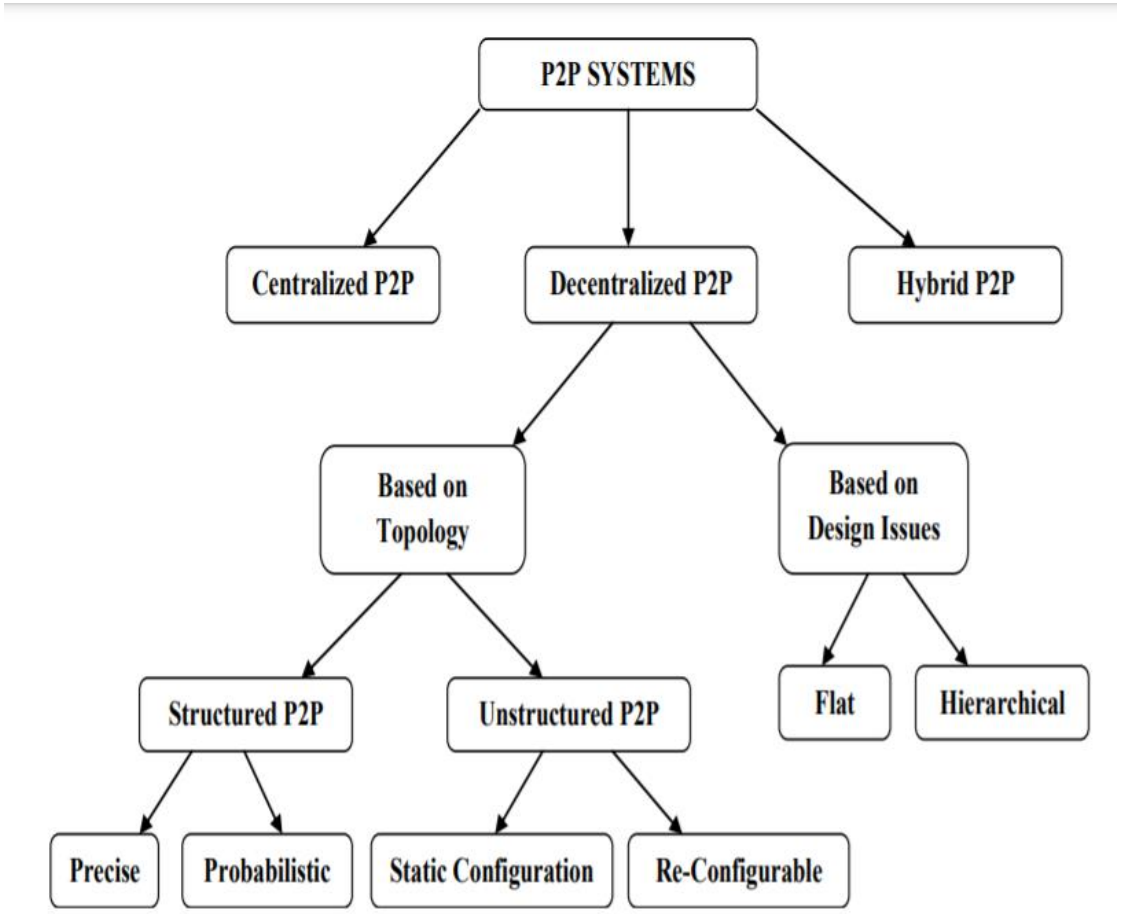
Evolution of P2P from Client-Server Model

- Many of the internet applications are using Client-Server model Example: WWW, email etc.
- In such model there will be a centralized server shared by many clients.
- The clients query the request to server and get services.
- It will be facilitated if the server is available and capable of serving all the requests from distinct clients at a particular moment.



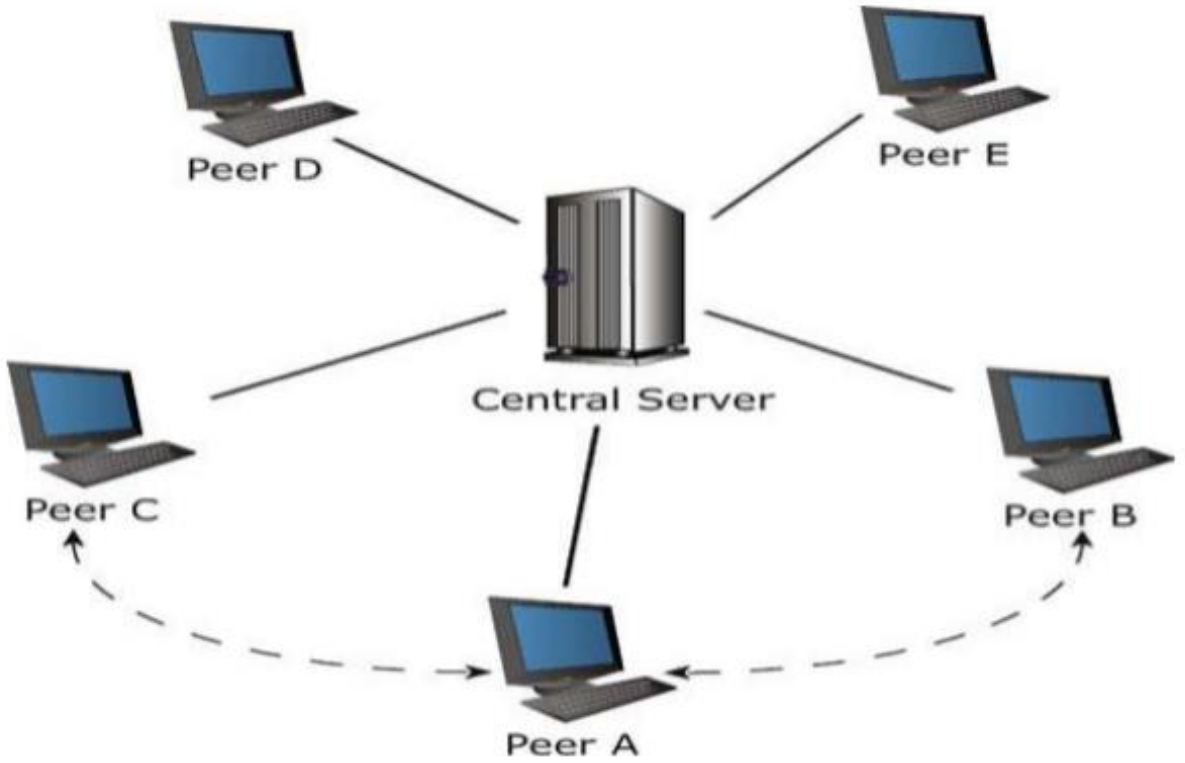
- There may be a chance of getting performance issues because of the unavailability of resources (e.g.: memory, bandwidth, processing speed) at the server system when too many requests arise.

Taxonomy of P2P System

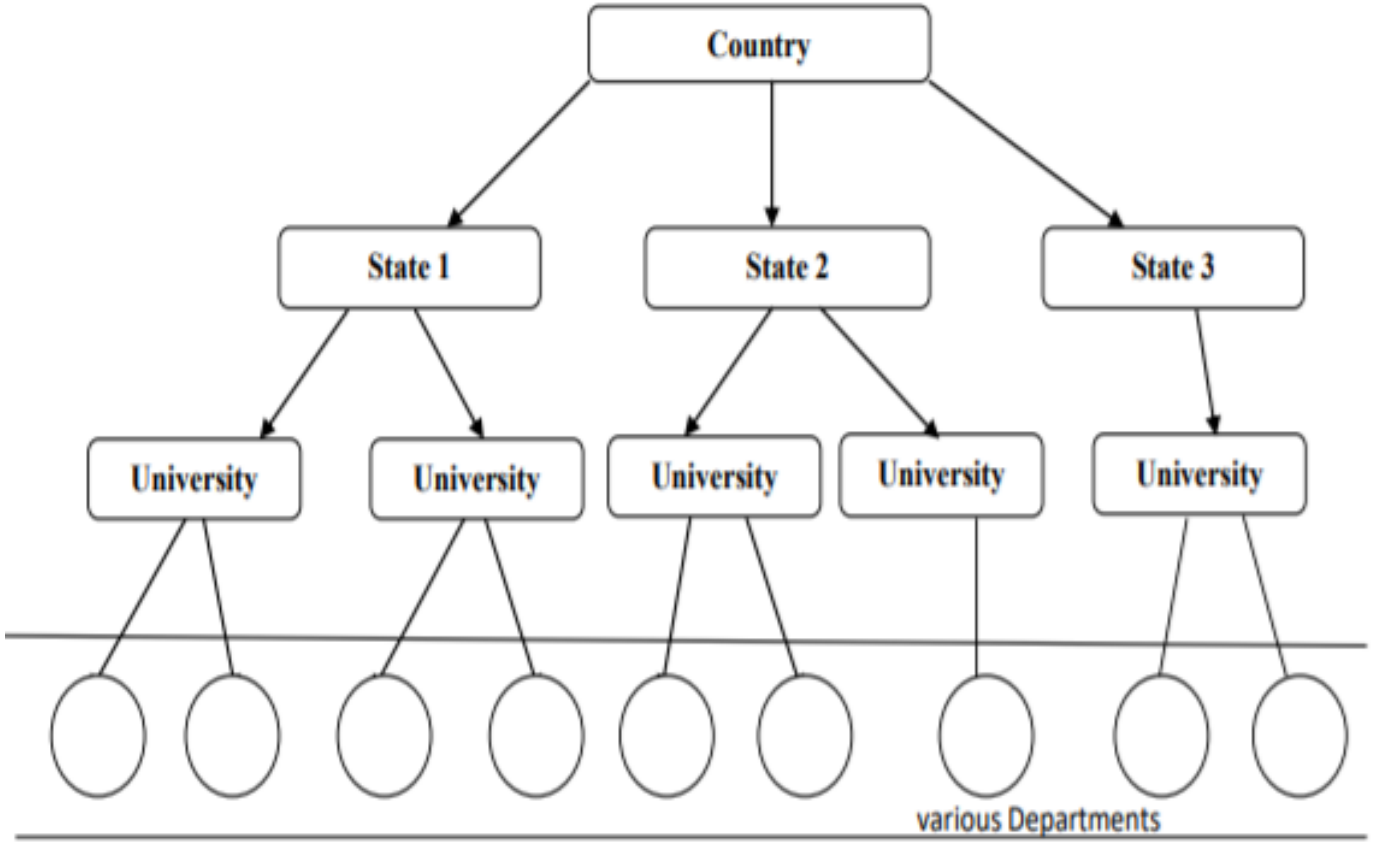


Centralized P2P System

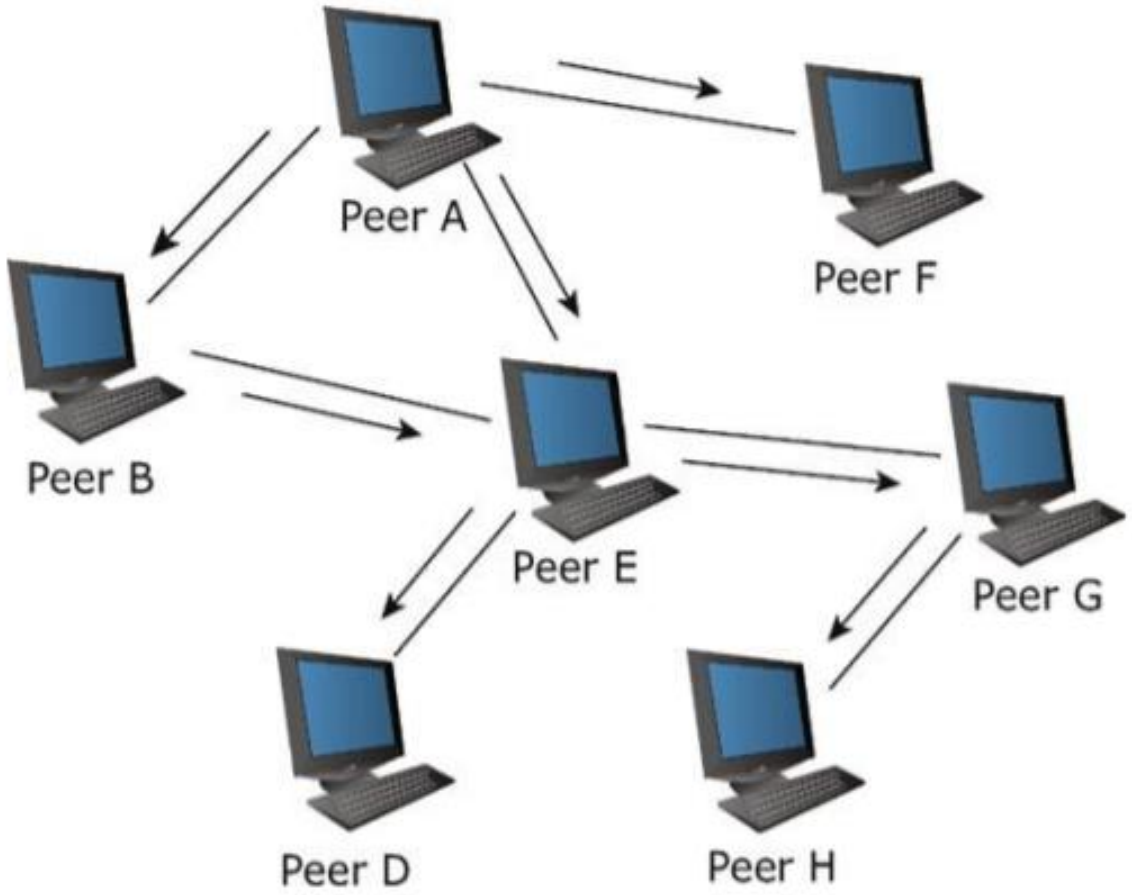
Sharing of MP3 music files



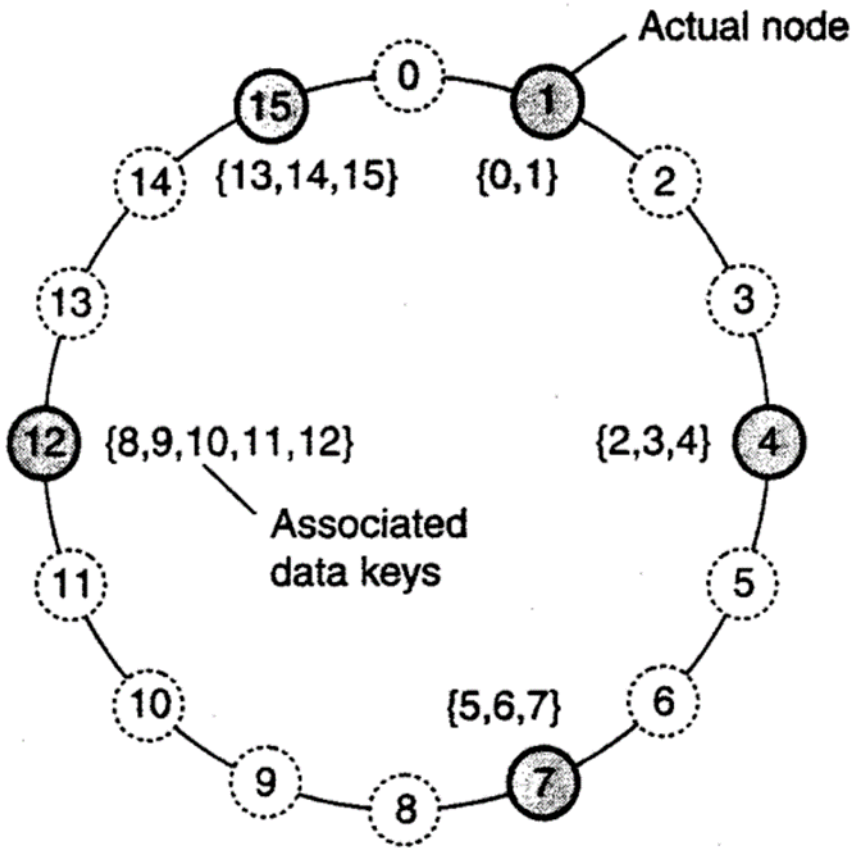
Decentralized P2P Svstem



Unstructured P2P System



Structured P2P System



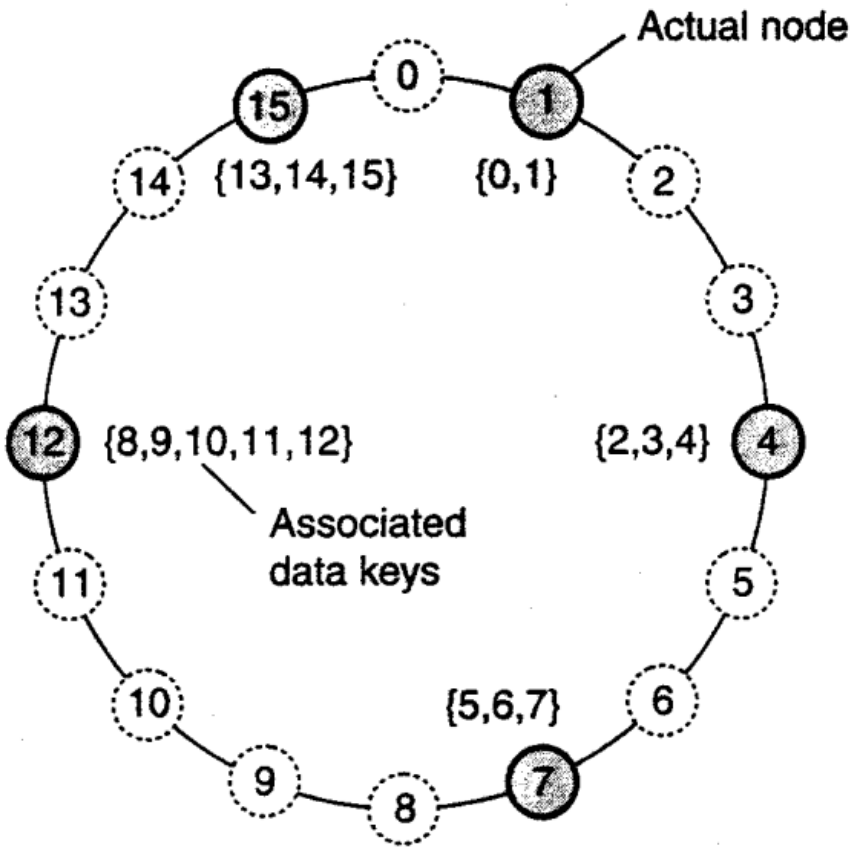
peer-to-peer systems

- From a high-level perspective, the processes that constitute a peer-to-peer system are all equal.
- This means that the functions that need to be carried out are represented by every process that constitutes the distributed system.
- As a consequence, much of the interaction between processes is symmetric: each process will act as a client and a server at the same time (which is also referred to as acting as a servent).
- Types:
  - Structured Peer-to-Peer Architectures
  - Unstructured Peer-to-Peer Architectures

Structured Peer-to-Peer Architectures

- structured → the system already has a predefined structure that other nodes will follow.
- Every structured network inherently suffers from poor scalability, due to the need for structure maintenance.
- In general, the nodes in a structured overlay network are formed in a logical ring, with nodes being connected to the this ring.
- In this ring, certain nodes are responsible for certain services.

- A common approach that can be used to tackle the coordination between nodes, is to use distributed hash tables (DHTs).
- A traditional hash function converts a unique key into a hash value, that will represent an object in the network.
- The hash function value is used to insert an object in the hash table and to retrieve it.



Unstructured P2P Systems

- There is no specific structure in these systems, hence the name "unstructured networks".
- Due to this reason, the scalability of the unstructured p2p systems is very high.
- These systems rely on randomized algorithms for constructing an overlay network.
- As in structured p2p systems, there is no specific path for a certain node. It's generally random, where every unstructured system tried to maintain a random path. Due to this reason, the search of a certain file or node is never guaranteed in unstructured systems.

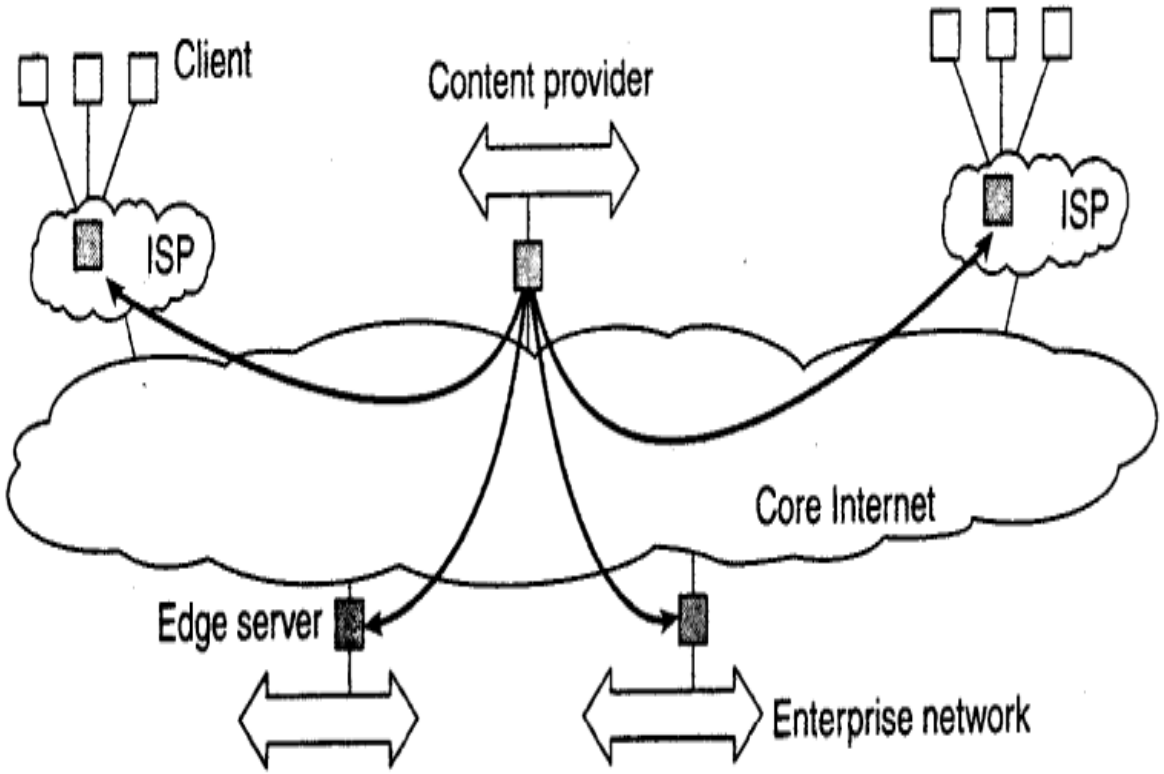
- The basic principle is that each node is required to randomly select another node, and contact it.
    - Let each peer maintain a partial view of the network, consisting of n other nodes
    - Each node P periodically selects a node Q from its partial view
    - P and Q exchange information and exchange members from their respective partial views

Hybrid Architectures

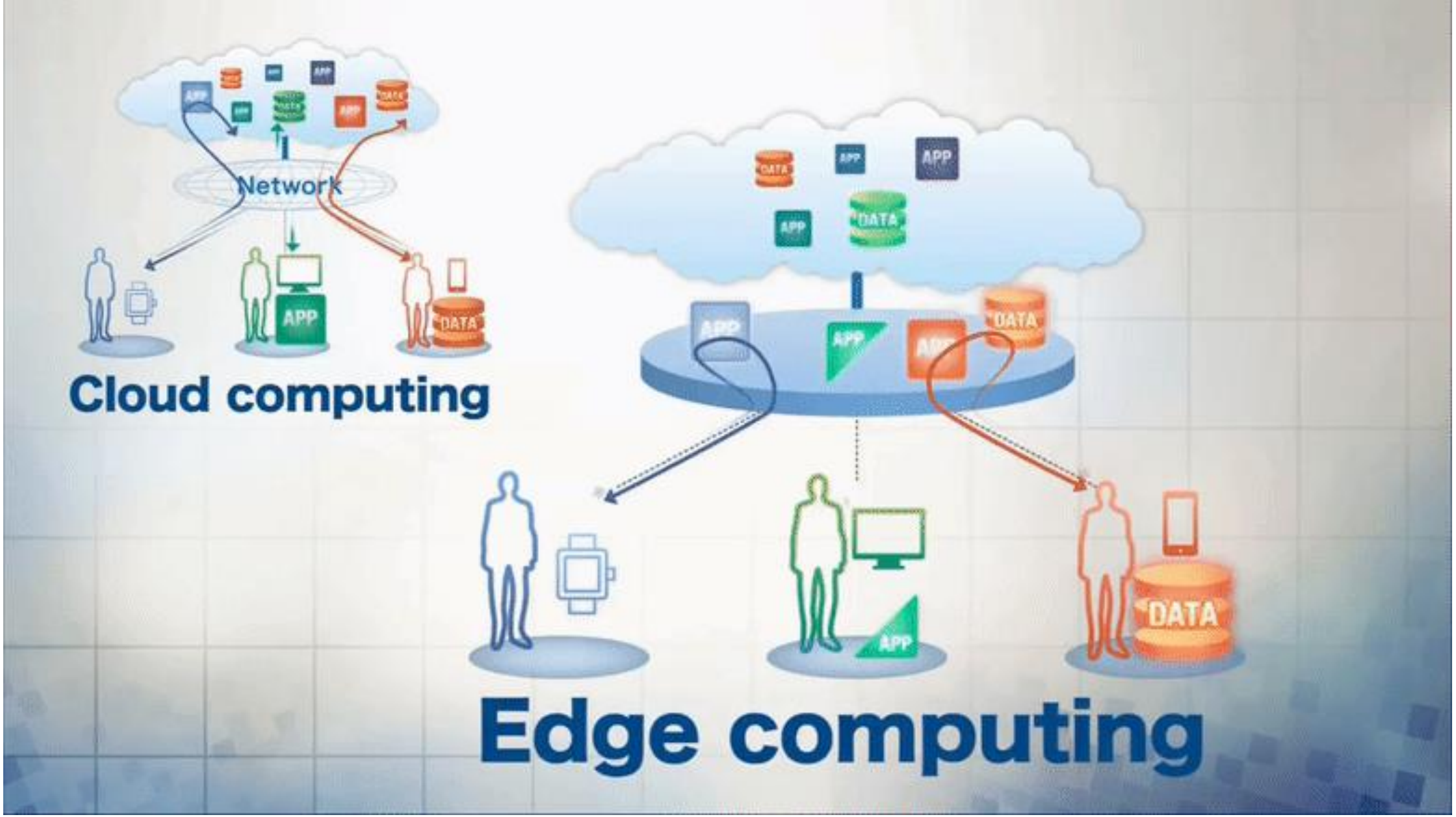
- In this section we take a look at some specific classes of distributed systems in which client-server solutions are combined with decentralized architectures
- Some of these architectures are as follows:
  - Edge-Server Systems
  - Collaborative Distributed Systems

Edge-Server Systems

- An important class of distributed systems that is organized according to a hybrid architecture is formed by edge-server systems.
- These systems are deployed on the Internet where servers are placed "at the edge" of the network. This edge is formed by the boundary between enterprise networks and the actual Internet (ISP).
- EG. Likewise, where end users at home connect to the Internet through their ISP, the ISP can be considered as residing at the edge of the Internet.



Viewing the Internet as consisting of a collection of edge servers.


Network
APP
DATA
Cloud computing
APP
DATA
APP
APP
APP
DATA
APP
DATA
Edge computing

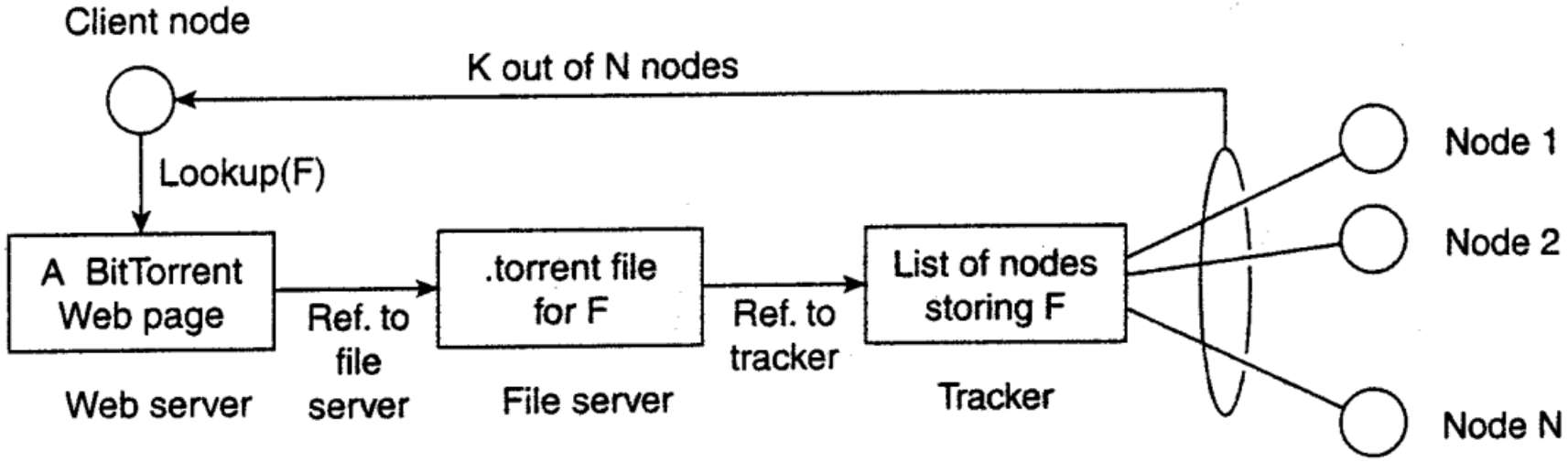
- End users, or clients in general, connect to the Internet by means of an edge server.
- The edge server's main purpose is to serve content, possibly after applying filtering and transcoding functions.
- More interesting is the fact that a collection of edge servers can be used to optimize content and application distribution.
- The basic model is that for a specific organization, one edge server acts as an origin server from which all content originates.
- That server can use other edge servers for replicating Web pages and such.

Collaborative Distributed Systems

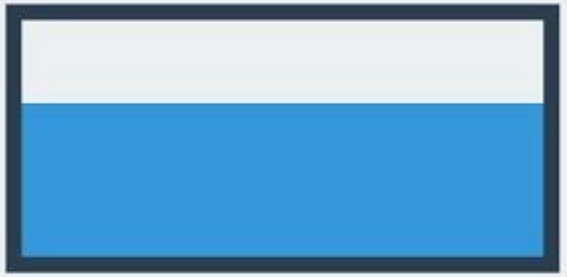
- Hybrid structures are especially deployed in collaborative distributed systems.
- The main issue in many of these systems to first get started, for which often a traditional client-server scheme is deployed.
- Once a node has joined the system, it can use a fully decentralized scheme for collaboration

BitTorrent file-sharing system

- BitTorrent is a peer-to-peer file downloading system.
- The basic idea is that when an end user is looking for a file, he downloads chunks of the file from other users until the downloaded chunks can be assembled together yielding the complete file.



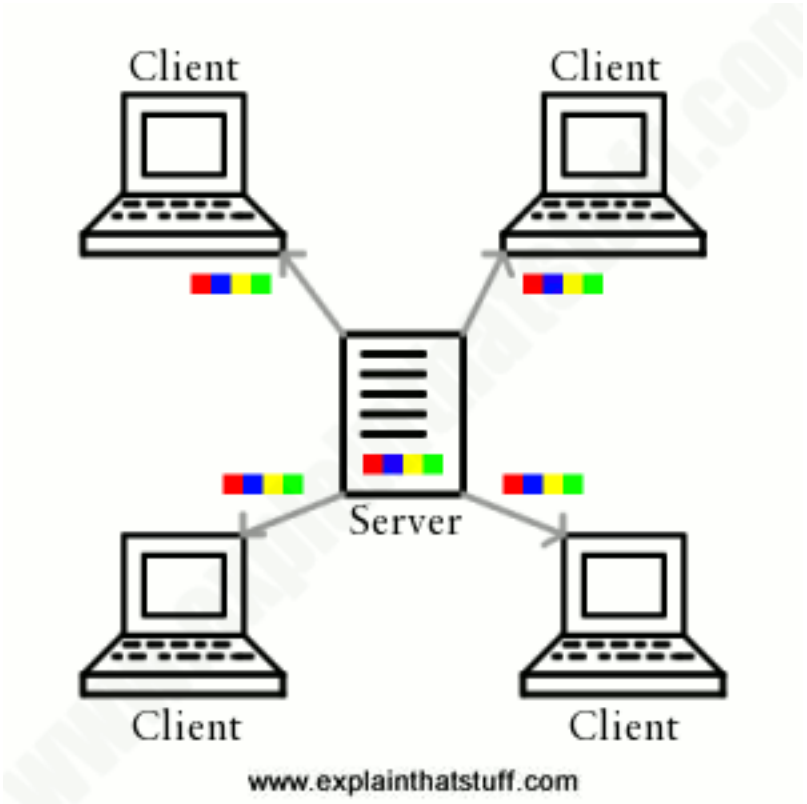
DIRECT DOWNLOAD SERVER

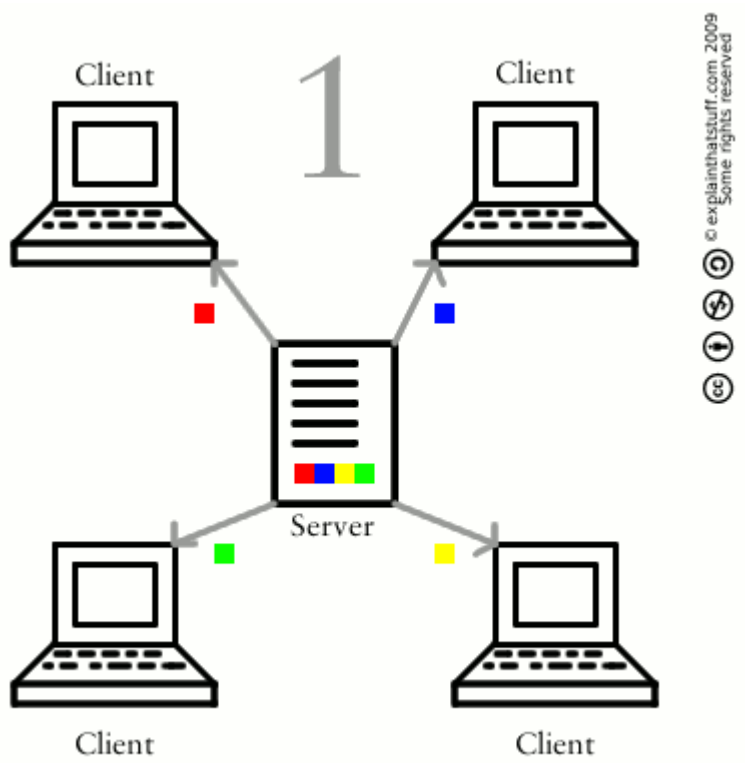


USERS

BITTORRENT PEER-TO-PEER

USERS


Client
Client
Server
Client
Client
www.explainthatstuff.com


Client
Client
Server
Client
Client
© explainthatstuff.com 2009
Some rights reserved

Unit 3: Processes

we take a closer look at how the different types of processes plays crucial role in distributed systems.

Outlines

Background

- Thread
- Virtualization
- Clients
- Servers
- Code Migration

Background

- The concept of a process originates from the field of operating systems.
- Process→ Program in execution
- EG. To efficiently organize client-server systems, it is often convenient to make use of multithreading technique.
- A main contribution of threads in distributed systems is that they results in a high level of performance of multithreading in distributed system.

Multithreading

- The ability of an OS to support multiple, concurrent paths of execution within a single process.

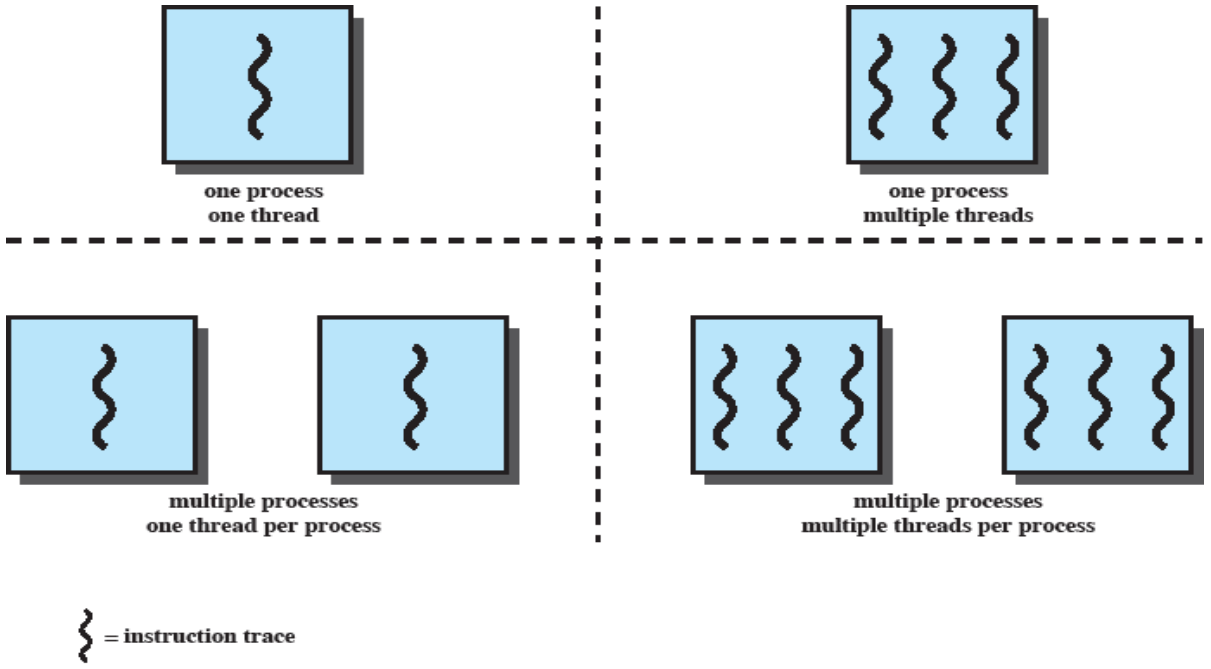


Figure 4.1 Threads and Processes [ANDE97]

Threads vs. processes

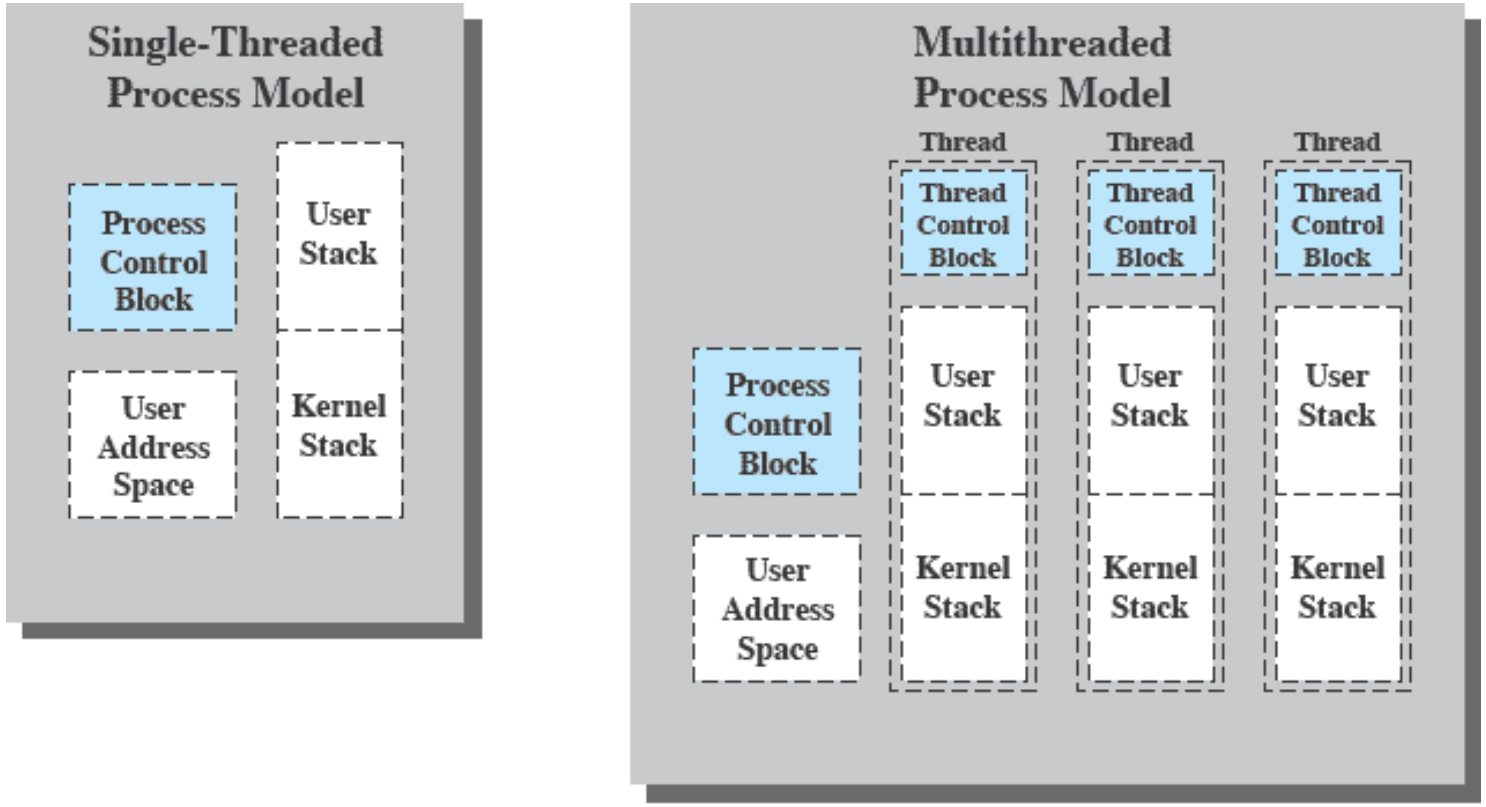


Figure 4.2 Single Threaded and Multithreaded Process Models

Introduction to Processes and Threads

- To understand the role of threads in distributed systems, it is important to understand what a process is, and how processes and threads relate.
- To execute a program, an operating system creates a number of **virtual processors**, each one for running a different program.

- To keep track of these virtual processors, the operating system has a process table, containing entries to store CPU register values, memory maps, open files, accounting information, privileges, etc.

- A process is often defined as a program in execution, that is, a program that is currently being executed on one of the operating system's virtual processors.

- In other words, the fact that multiple processes may be concurrently sharing the same CPU and other hardware resources is made transparent.

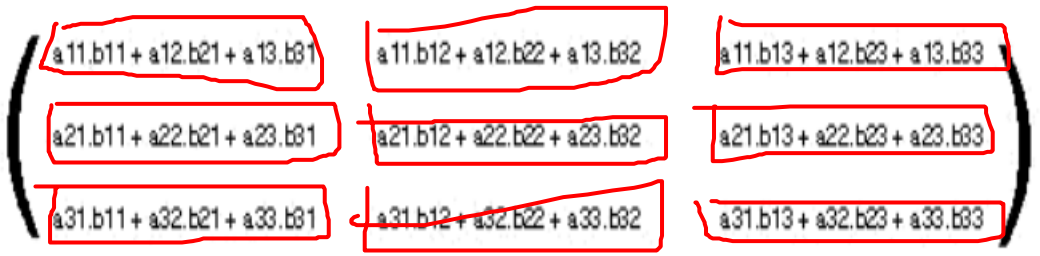
- Each time a process is created, the operating system must create a complete independent address space.
- Allocation can mean initializing memory segments(for example, a data segment) by copying the associated program into a code segment, and setting up a stack segment for temporary data.
- Likewise, **switching the CPU between two processes** may be relatively **expensive** as well.
- Apart from saving the CPU context (which consists of register values, program counter, stack pointer, etc.), the operating system will also have to modify registers of the memory management unit (MMU) and invalidate address translation caches such as in the translation look-aside buffer (TLB).
- In addition, if the operating system supports more processes than it can simultaneously hold in main memory, it may have to swap processes between main memory and disk before the actual switch can take place.
- Like a process, a thread executes its own piece of code, independently from other threads.

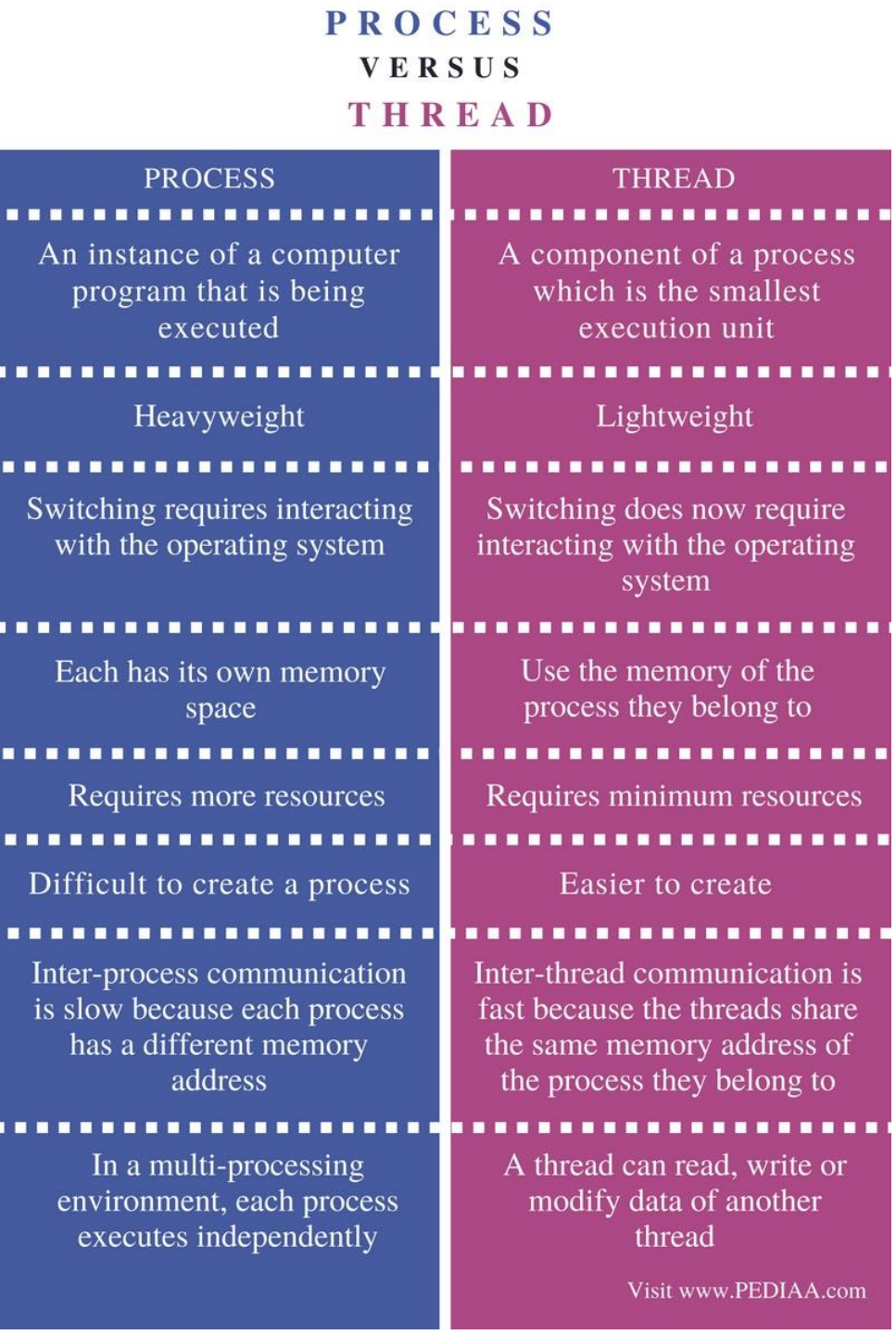
Two implications of multi-thread systems

- There are two important implications of this approach.:
  - First of all, the performance of a multithreaded application is not worse than that of its single-threaded counterpart. In fact, in many cases, multithreading leads to a performance gain.
  - Second, because threads are not automatically protected against each other, development of multithreaded applications requires additional intellectual effort.

Matrix Multiplication

Note that each ***element*** of the resultant matrix can be computed independently, that is to say by a different thread.





Difference	Process	Thread
Resource Allocation	Allocate new resources each time we run a program.	Share resources of process.
Resource Sharing	In general, resources are not shared. The code may be shared for the same program.	Share code, heap, data area except stack.
Address	Have a separate address space	Share address space
Communication	Communicate through IPC.	Communicate freely with modifying shared variables.
Context Switching	Generally slower than thread.	Generally faster than process.

Thread usage in traditional, non-distributed systems

- Spreadsheet: It maintains the inter-dependencies between different cells, often from different spreadsheets

whenever a cell is modified, all dependent cells are automatically updated. When a user changes the value in a single cell, such a modification can trigger a large series of computations. If there is only a single thread of control, computation cannot proceed while the program is waiting for input. Likewise, it is not easy to provide input while dependencies are being calculated. The easy solution is to have at least two threads of control: one for handling interaction with the user and one for updating the spreadsheet. In the mean time, a third thread could be used for backing up the spreadsheet to disk while the other two are doing their work.

2 • possible to exploit parallelism

each thread is assigned to a different CPU while shared data are stored in shared main memory. When properly designed, such parallelism can be transparent: the process will run equally well on a uniprocessor system, abit slower.

Multithreading for parallelism is becoming increasingly important with the availability of relatively cheap multiprocessor workstations. Such computer systems are typically used for running servers in client-server applications.

- useful in the context of large applications

Such applications are often developed as a collection of cooperating programs, each to be executed by a separate process. This approach is typical for a UNIX environment. Cooperation between programs is implemented by means of inter-process communication (IPC) mechanisms.

The major drawback of all IPC mechanisms is that communication often requires extensive context switching, shown at three different points in Figure.

Thread Usage in Non-distributed Systems: IPC and System Call Costs

Because IPC requires kernel intervention, a process will generally:

- First have to switch from user mode to kernel mode. This requires changing the memory map in the MMU, as well as flushing the TLB.
- Within the kernel, a process context switch takes place, after which the other party can be activated by switching from kernel mode to user mode again.
- The latter switch again requires changing the MMU map and flushing the TLB.

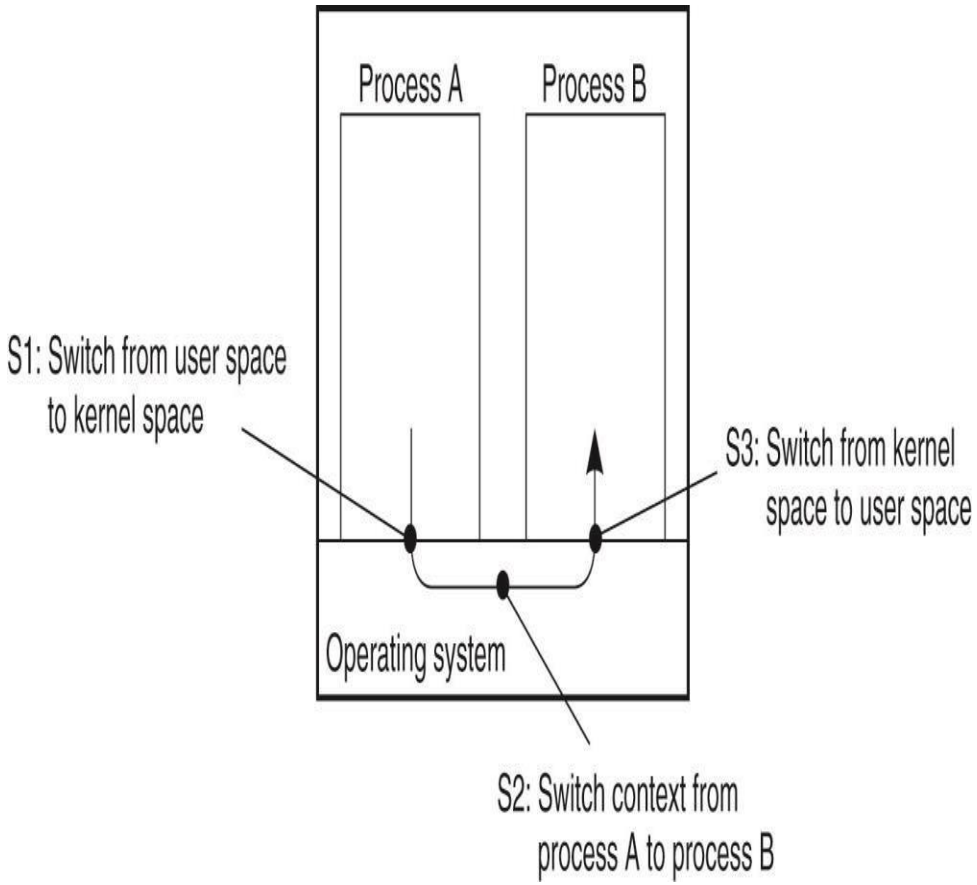


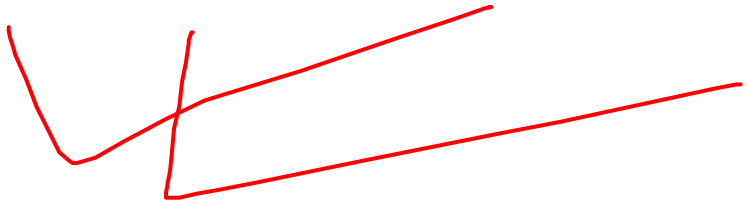
Figure. Context switching as the result of IPC.

Thread Usage in Non-distributed Systems: IPC and System Call Costs

- Instead of using processes, an application can also be constructed such that different parts are executed by separate threads.
- Communication between those parts is entirely dealt with by using shared data.
- Thread switching can sometimes be done entirely in user space, although in other implementations, the kernel is aware of threads and schedules them. The effect can be a dramatic improvement in performance.

- a pure software engineering reason

For example, in the case of a word processor, separate threads can be used for handling user input, spelling and grammar checking, document layout, index generation, etc.

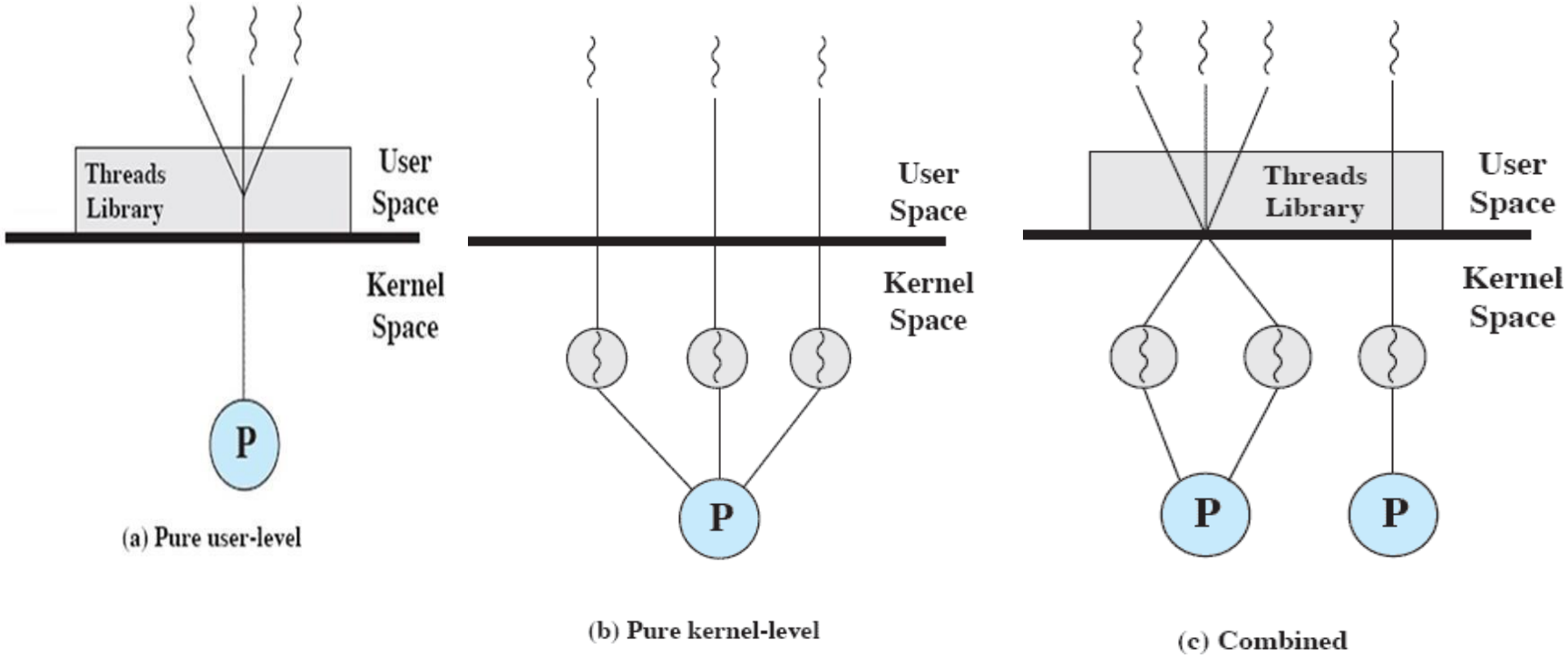


Thread Implementation

• Threads are often provided in the form of a thread package. Such a package contains operations to create and destroy threads as well as operations on synchronization variables such as mutexes and condition variables. There are basically **three** approaches to implement a thread package.

- **The first approach** is to construct a thread library that is executed entirely in user mode, called **User Level Threads (ULT)**.
- **The second approach** is to have the kernel be aware of threads and schedule them, called **Kernel Level Threads (KLT)**.
- **The third approach** is a **Hybrid** Form.

ULT Vs. KLT


Threads Library
User Space
Kernel Space
P
(a) Pure user-level
User Space
Kernel Space
P
(b) Pure kernel-level
Threads Library
User Space
Kernel Space
P
P
(c) Combined

User-Level Threads Advantages

- First, it is cheap to create and destroy threads. Because all thread administration is kept in the user's address space:
  - the price of creating a thread is primarily determined by the cost for allocating memory to set up a thread stack.
  - Analogously, destroying a thread mainly involves freeing memory for the stack.
- A second advantage of user-level threads is that switching thread context can often be done in just a few instructions.
  - Basically, only the values of the CPU registers need to be stored and subsequently reloaded with the previously stored values of the thread to which it is being switched. There is no need to change memory maps, flush the TLB, and do CPU operations.

User-level Threads Disadvantages

- A major drawback of user-level threads is that invocation of a blocking system call will immediately block the entire process to which the thread belongs, and thus also all the other threads in that process.

- Threads are particularly useful to structure large applications into parts that could be logically executed at the same time. In that case, blocking on I/O should not prevent other parts to be executed in the meantime.

Kernel-Level Threads

•These problems can be mostly circumvented by implementing threads in the operating system's kernel. Unfortunately, there is a high price to pay:

•Every thread operation (creation, deletion, synchronization, etc.) will have to be carried out by the kernel which require a system call.

•Switching thread contexts may now become as expensive as switching process contexts. As a result, most of the performance benefits of using threads instead of processes then disappears.

Kernel-Level Threads Advantages

- The kernel can simultaneously schedule multiple threads from the same process on multiple processors.
- If one thread in a process is blocked, the kernel can schedule another thread of the same process.
- Kernel routines themselves can be multithreaded.

Kernel-Level Threads Disadvantages

•Every thread operation (creation, deletion, synchronization, etc.) will have to be carried out by the kernel which requires a system call.

•Switching thread contexts may now become as expensive as switching process contexts.

•As a result, most of the performance benefits of using threads instead of processes then disappears.

Hybrid Approach

- A solution lies in a hybrid form of user-level and kernel-level threads, generally referred to as lightweight processes (LWP). An LWP runs in the context of a single (heavy-weight) process, and there can be several LWPs per process.
- In addition to having LWPs, a system also offers a user-level thread package. Offering applications the usual operations for creating and destroying threads.
- In addition, the package provides facilities for thread synchronization such as mutexes and condition variables. The important issue is that the thread package is implemented entirely in user space. All operations on threads are carried out without intervention of the kernel.

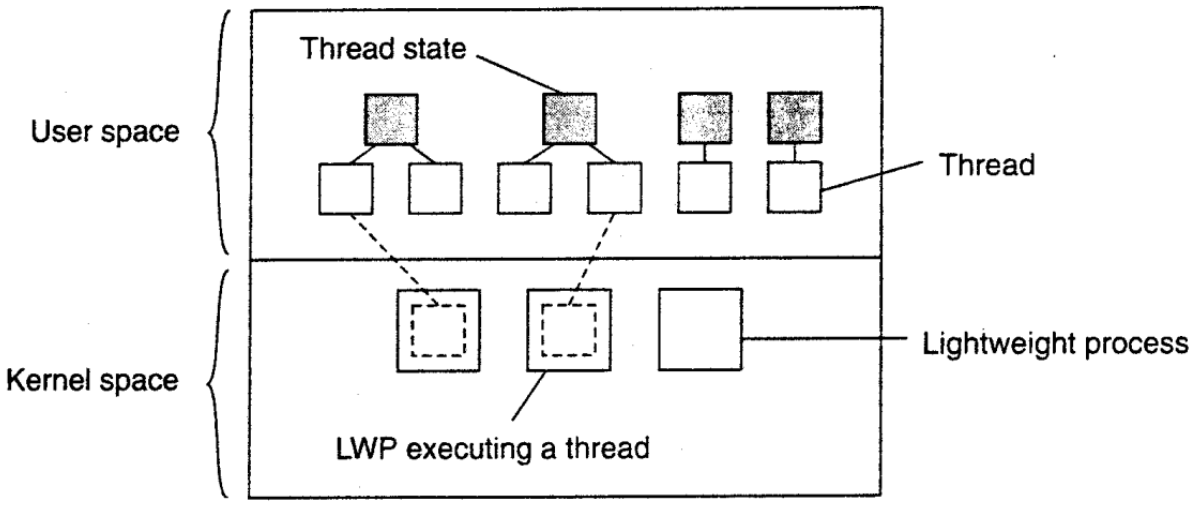


Figure 3-2. Combining kernel-level lightweight processes and user-level threads.

Hybrid Approach Advantages

1. Creating, destroying, and synchronizing threads is relatively cheap and involves no kernel intervention at all.
2. Provided that a process has enough LWPs, a blocking system call will not suspend the entire process.
3. There is no need for an application to know about the LWPs. All it sees are user-level threads.
4. LWPs can be easily used in multiprocessing environments, by executing different LWPs on different CPUs. This multiprocessing can be hidden entirely from the application.

Hybrid Approach disadvantages

1. The only drawback of lightweight processes in combination with user-level threads is that we still need to create and destroy LWPs, which is just as expensive as with kernel-level threads.

However, creating and destroying LWPs needs to be done only occasionally, and is often fully controlled by the operating system.

Threads in Distributed Systems

- Multithreaded Clients → Web Browser to reduce the communication latencies
- Multithreaded Servers → Three types: Multithreaded server, Single Threaded server and Finite-State Machine Server

Multithreaded Clients

- The usual way to hide communication latencies is to initiate communication and immediately proceed with something else. A typical example where this happens is in Web browsers.
- A Web browser often starts with fetching the HTML page and subsequently displays it.
- To hide communication latencies as much as possible, some browsers start displaying data while it is still coming in.
- As soon as the main HTML file has been fetched, separate threads can be activated to take care of fetching the other parts. Each thread sets up a separate connection to the server and pulls in the data. Setting up a connection and reading data from the server can be programmed using the standard (blocking) system calls, assuming that a blocking call does not suspend the entire process.

Multithreaded Servers

- Although there are important benefits to multithreaded clients, the main use of multithreading in distributed systems is found at the server side.
- Practice shows that multithreading not only simplifies server code considerably, but also makes it much easier to develop servers that exploit parallelism to attain high performance, even on uni-processor systems.
- However, now that multiprocessor computers are widely available as general-purpose workstations, multithreading for parallelism is even more useful.
- To understand the benefits of threads for writing server code, consider the organization of a file server that occasionally has to block waiting for the disk.

Three ways to construct a server

Model	Characteristics
Threads	Parallelism, blocking system calls
Single-threaded process	No parallelism, blocking system calls
Finite-state machine	Parallelism, nonblocking system calls

Multithreaded Servers

- The file server normally waits for an incoming request for a file operation, subsequently carries out the request, and then sends back the reply.
- In Fig. , a dispatcher thread, reads incoming requests for a file operation.
- After examining the request, the server chooses an idle (i.e., blocked) worker thread and hands it the request.

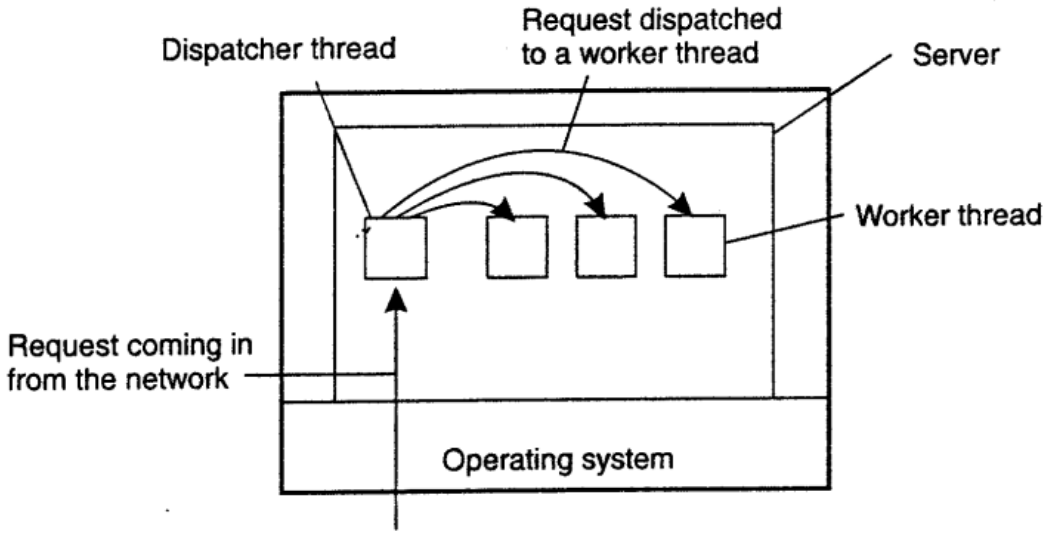


Figure 3-3. A multithreaded server organized in a dispatcher/worker model.

Single-thread Server

One possibility is to have it operate as a single thread:

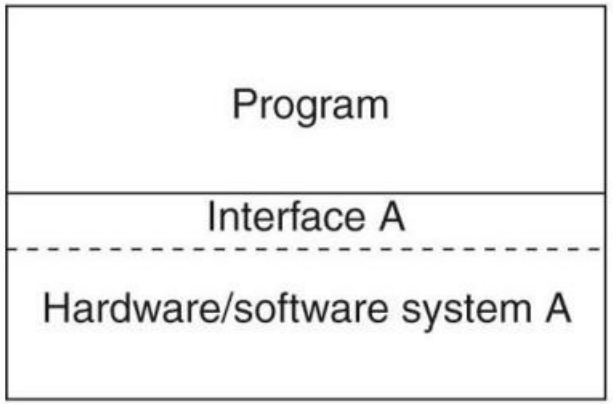
- The main loop of the file server gets a request, examines it, and carries it out to completion before getting the next one.
- While waiting for the disk, the server is idle and does not process any other requests. Consequently, requests from other clients cannot be handled.
- In addition, if the file server is running on a dedicated machine, as is commonly the case, the CPU is simply idle while the file server is waiting for the disk.
- The net result is that many fewer requests/sec can be processed

Finite-state machine Server

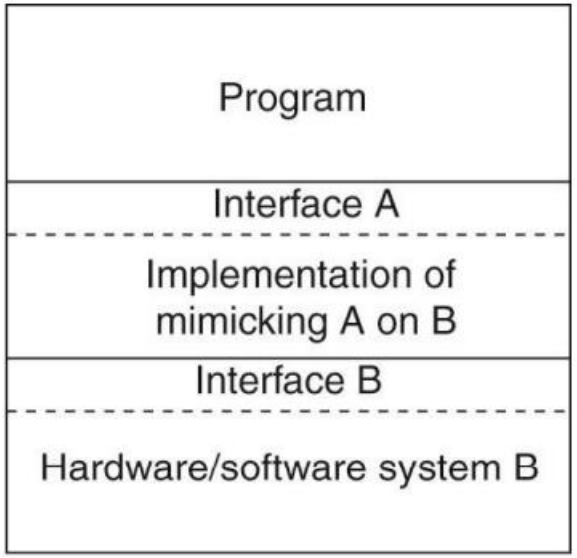
- A third possibility is to run the server as a big finite-state machine. When a request comes in, the one and only thread examines it. If it can be satisfied from the cache, fine, but if not, a message must be sent to the disk.
- However, instead of blocking, it records the state of the current request in a table and then goes and gets the next message.
- The next message may either be a request for new work or a reply from the disk about a previous operation.
- If it is new work, that work is started. If it is a reply from the disk, the relevant information is fetched from the table and the reply processed and subsequently sent to the client.
- In this scheme, the server will have to make use of non-blocking calls to send and receive.

The Role of Virtualization in Distributed Systems

- In practice, every (distributed) computer system offers a programming interface (API) to higher level software, as shown in Fig. (a).
- There are many different types of interfaces, ranging from the basic instruction set as offered by a CPU to the vast collection of application programming interfaces that are shipped with many current middleware systems.
- In its essence, virtualization deals with extending or replacing an existing interface so as to mimic the behavior of another system, as shown in Fig. (b).



(a)



(b)

Architectures of Virtual Machines (1)

Interfaces at different levels

- An interface between the hardware and software consisting of machine instructions
  - that can be invoked by any program.
- An interface between the hardware and software, consisting of machine instructions
  - that can be invoked only by privileged programs, such as an operating system.
- An interface consisting of system calls as offered by an operating system.

Architectures of Virtual Machines (2)

- An interface consisting of library calls

– generally forming what is known as an application programming interface (API).

– In many cases, the aforementioned system calls are hidden by an API.

Architectures of Virtual Machines (3)

- The essence of virtualization is to mimic the behavior of these interfaces.

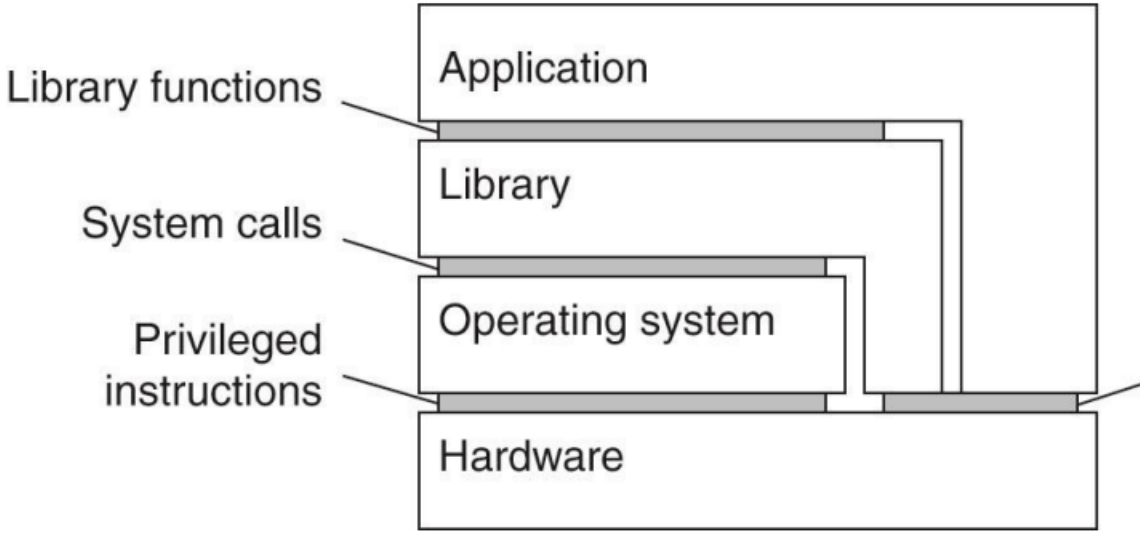


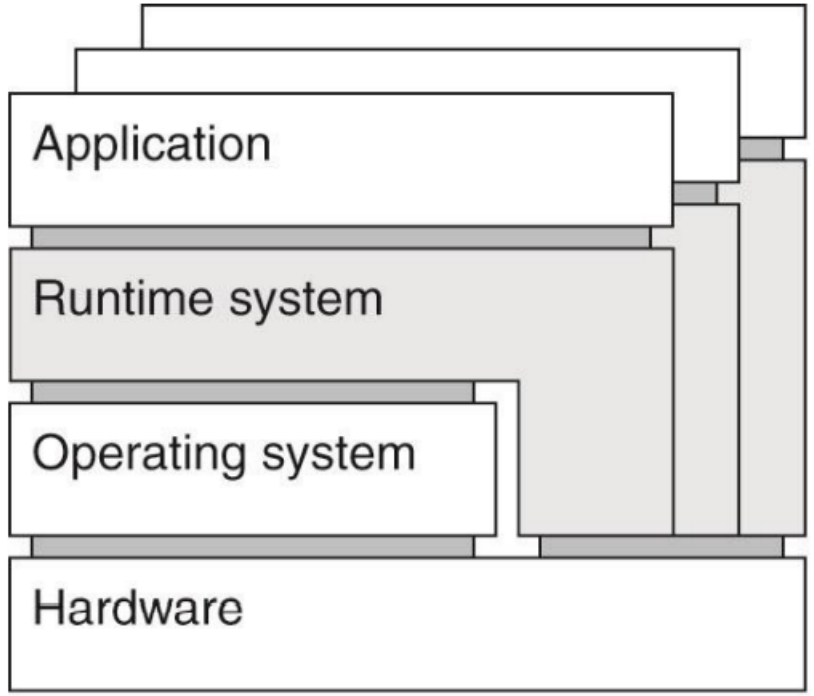
Figure . Various interfaces offered by computer systems.

Two Different Ways of Making Virtual Machines

- Virtualization can take place in two different ways:

1- First, we can build a runtime system that essentially provides an abstract instruction set that is to be used for executing applications.

Instructions can be **interpreted** (i.e. the Java runtime environment), but could also be **emulated** (i.e. running Windows applications on UNIX platforms). This type of virtualization leads to **process virtual machine**, stressing that virtualization is done essentially **only for a single process**.



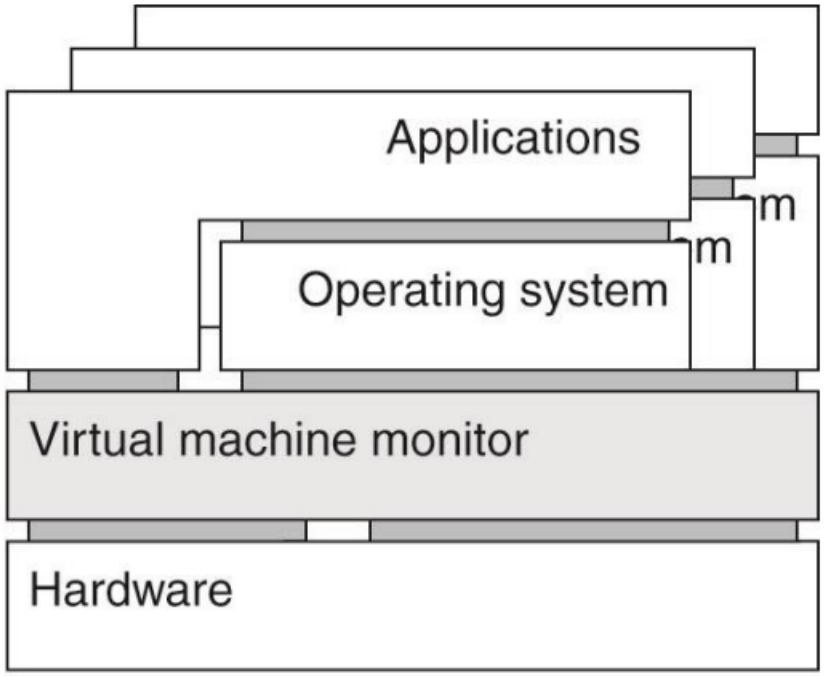
(a)

A process virtual machine, with multiple instances of (application, runtime) combinations.

Two Different Ways of Making Virtual Machines

- Virtualization can take place in two different ways:

2- An alternative approach is to provide a system that is essentially implemented as a layer completely shielding the original hardware, but offering the complete instruction set of hardware as an interface. This interface can be offered simultaneously to different programs. As a result, it is now possible to have multiple, and different operating systems run independently and concurrently on the same platform. The layer is generally referred to as a **Virtual Machine Monitor** (VMM). (i.e. VMware)



(b)

A virtual machine monitor, with multiple instances of (applications, operating system) combinations.

Clients and Servers

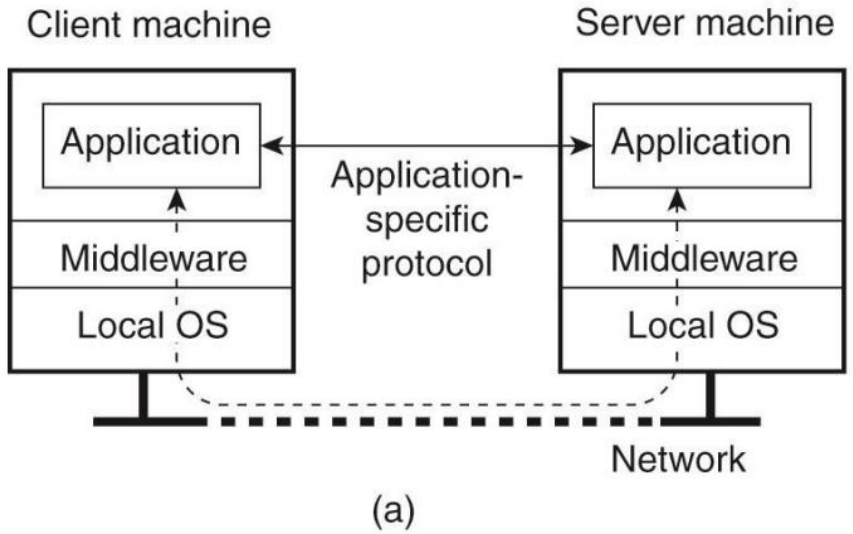
- In the previous chapters we discussed the client-server model the roles of clients and servers, and the ways they interact.
- Let us now take a closer look at the anatomy of clients and servers, respectively.

- Clients → Networked User Interfaces

Clients

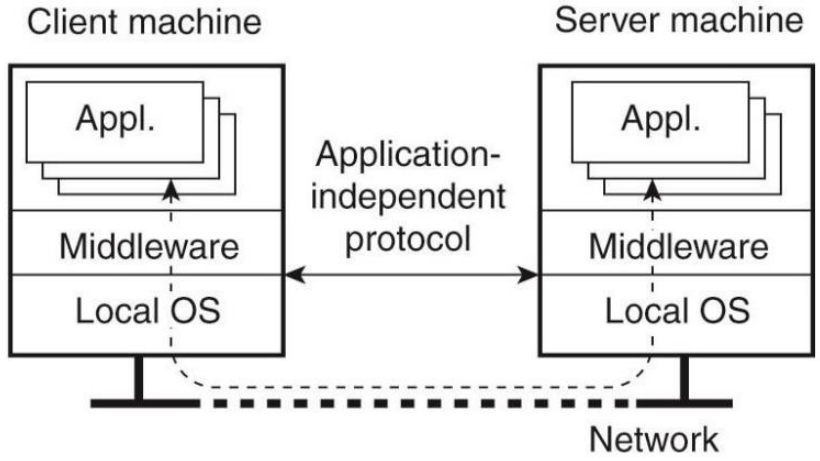
- A major task of client machines is to provide the means for users **to interact with remote servers.** There are roughly two ways in which this interaction can be supported.
- First, for each remote service the client machine will have a separate counterpart that can contact the service over the network (Fig. (a), Fig. (b)).

E.g. A typical example is an agenda running on a user's PDA that needs to synchronize with a remote, possibly shared agenda.



(a)

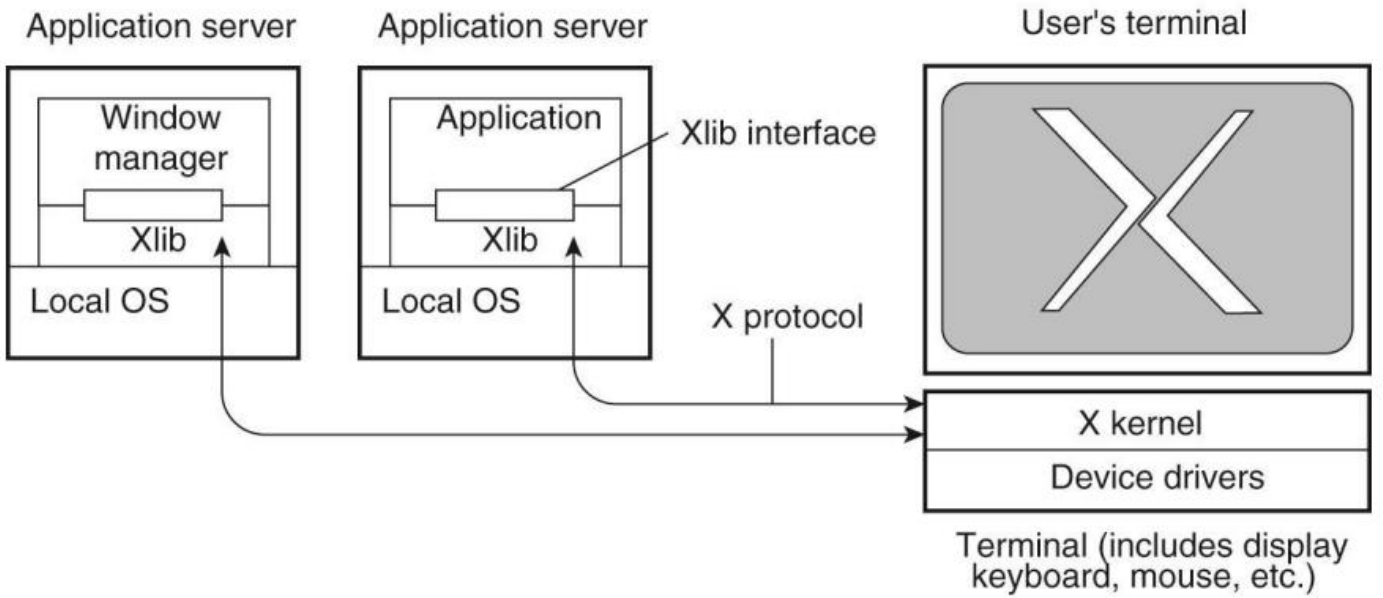
A networked application with its own protocol.



(b)

A general solution to allow access to remote applications.

- A second solution is to provide direct access to remote services by only offering a convenient user interface. Effectively, this means that the client machine is used only as a terminal with no need for local storage, leading to an application neutral solution as shown in (Fig. C)



C) The basic organization of the XWindow System

Example: The X Window System

- one of the oldest and still widely-used networked user interfaces is the X Window system.
- X can be viewed as that part of an operating system that controls the terminal.
- The heart of the system is formed by what we shall call the X kernel. It contains all the terminal-specific device drivers, and as such, is generally highly hardware dependent.
- The interesting aspect of X is that the X kernel and the X applications need not necessarily reside on the same machine.
- In particular, X provides the X protocol, which is an application-level communication protocol by which an instance of Xlib can exchange data and events with the X kernel.
- For example, Xlib can send requests to the X kernel for creating or killing a window, setting colors, and defining the type of cursor to display, among many other requests. In turn, the X kernel will react to local events such as keyboard and mouse input by sending event packets back to Xlib.

Assignment-II: February 4 (Tomorrow)

- Explain the working of Xwindows System as a Networked User Interface.
- Explain Virtualization. Describe the two Different Ways of Making Virtual Machines.

Revisit: Distribution Transparency

- Goal → to hide the fact that its processes and resources are physically distributed across multiple computers.
- In other words, it tries to make the distribution of processes and resources transparent, that is, invisible, to end users and applications.

Transparency	Description
Access	Hide differences in data representation and how an object is accessed
Location	Hide where an object is located
Relocation	Hide that an object may be moved to another location while in use
Migration	Hide that an object may move to another location
Replication	Hide that an object is replicated
Concurrency	Hide that an object may be shared by several independent users
Failure	Hide the failure and recovery of an object

Client-Side Software for Distribution Transparency

- Client software comprises more than just user interfaces. In many cases, parts of the processing and data level in a client-server application are executed on the client side as well. A special class is formed by embedded client software, such as for automatic teller machines (ATMs), cash registers, barcode readers, TV set-top boxes, etc.
- Besides the user interface and other application-related software, client software comprises components for achieving **distribution transparency**. **Access transparency** is generally handled through the generation of a **client stub** from an **interface definition (IDL)** of what the server has to offer. There are different ways to handle **location, migration, and relocation transparency**. Using a convenient **naming system** is crucial. Many distributed systems implement **replication transparency** by means of client-side solutions

- In **failure transparency** a client middleware can be configured to repeatedly attempt to connect to a server, or perhaps try another server after several attempts. **Concurrency transparency** requires less support from client software. **Persistence transparency** is often completely handled at the server.

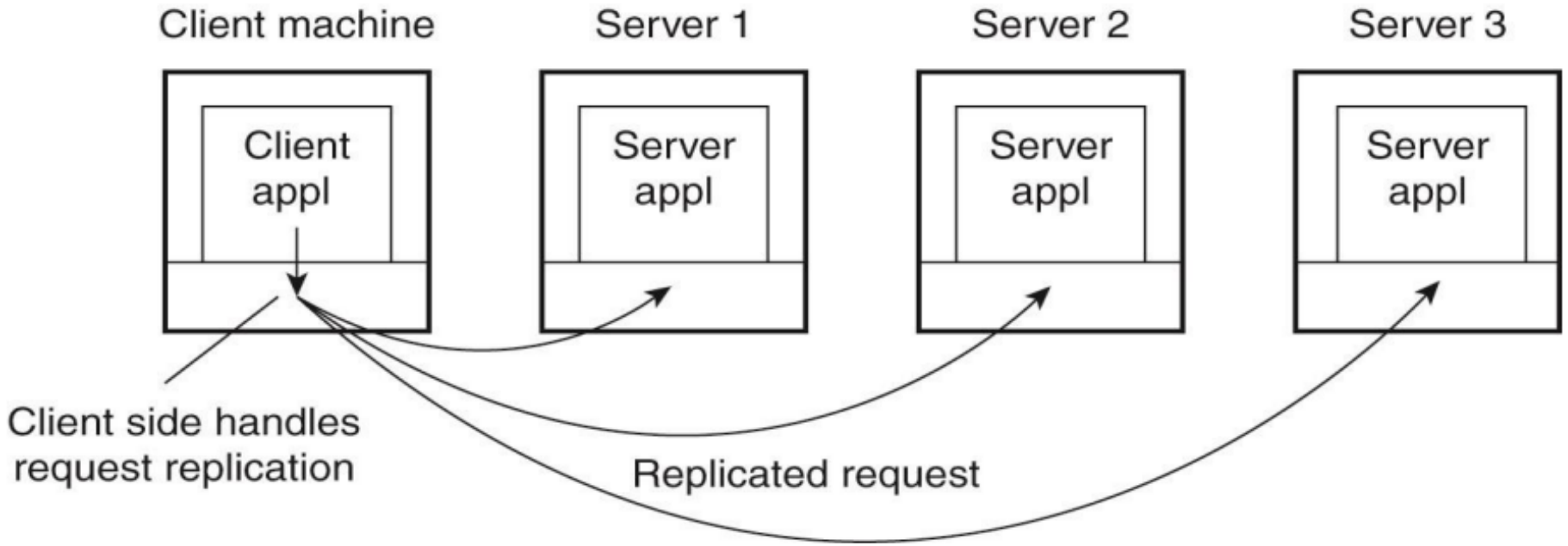


Fig: Transparent replication of a server using a client-side solution.

Servers: General Issues

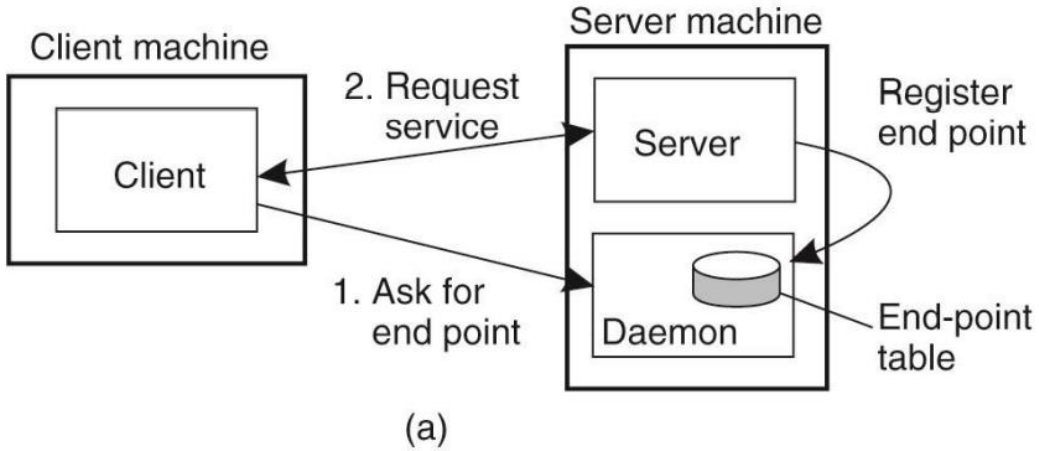
- A server is a process implementing a specific service on behalf of a collection of clients. In essence, each server is organized in the same way: it waits for an incoming request from a client and subsequently ensures that the request is taken care of, after which it waits for the next incoming request.
- In the case of an **iterative server**, the server itself handles the request and, if necessary, returns a response to the requesting client.
- **A concurrent server** does not handle the request itself, but passes it to a separate thread or another process, after which it immediately waits for the next incoming request.
- **A multithreaded server** is an example of a concurrent server.
- An alternative implementation of a concurrent server is **to fork a new process for each new incoming request.**

Servers: General Issues

- An issue is where clients contact a server. In all cases, clients send requests to an **end point**, also called a **port**, at the machine where the server is running. Each server listens to a specific end point. How do clients know the end point of a service?
- The approach is to globally **assign end points for well-known services**.
- For example, servers that handle Internet FTP requests always listen to TCP port 21. Likewise, an HTTP server for the World Wide Web will always listen to TCP port 80.

Servers General Design Issues

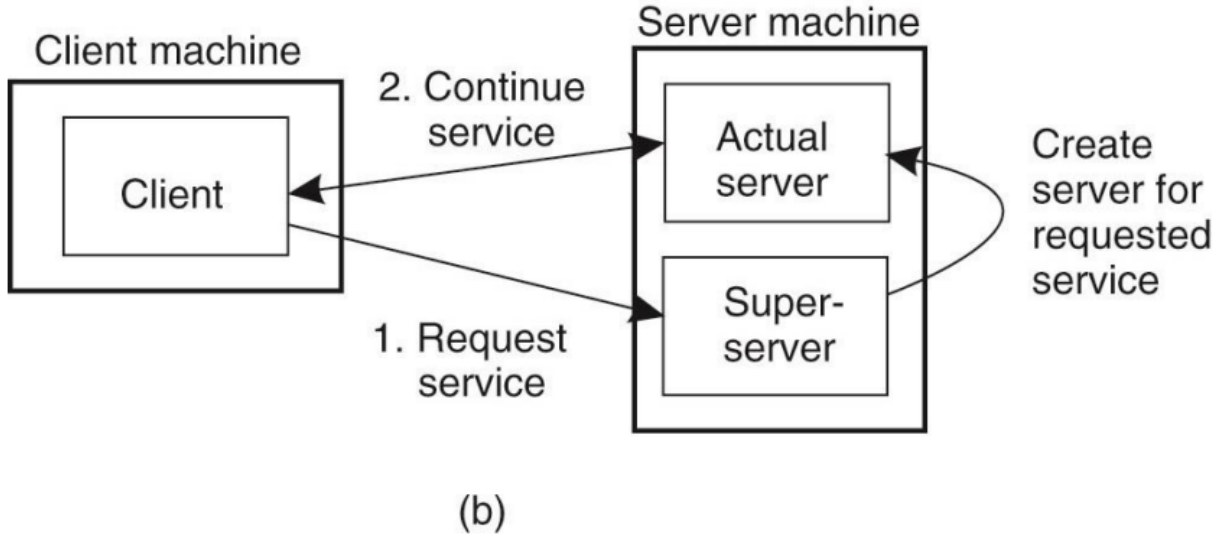
- There are many services that do not require a pre-assigned end point (i.e. a time-of-day server). So, a client will first have to look up the end point.
- One solution is to have a special **daemon** running on each machine that runs servers. The daemon keeps track of the current end point of each service implemented by a co-located server. **The daemon itself listens to a well-known end point**. A client will first contact the daemon, request the end point, and then contact the specific server, as shown in Fig. (a).



Client-to-server binding using a daemon.

Servers General Design Issues

- It is often more efficient to have a single **super-server** listening to each end point associated with a specific service, as shown in Fig. (b).
- *For example, the daemon in UNIX Listens to a number of well-known ports for Internet services. When a request comes in, the daemon forks a process to process the request. That process will exit after it is finished.*



(b)

Client-to-server binding using a super-server.

Servers: General Issues

- A final, important design issue, is whether or not the server is **stateless or stateful**.
- A **stateless server** does not keep information on the state of its clients, and can change its own state without having to inform any client (for example, a Web Server).
- In contrast, **a stateful server** generally maintains persistent information on its clients. This means that the information needs to be explicitly deleted by the server. A typical example is a file server that allows a client to keep a local copy of a file, even for performing update operations. Such a server would maintain a **table (state)** containing (client, file) entries

Terminology

- **Distributed** : refers to splitting a business into different sub-services and distributing them on different machines.
- **Cluster** : It means that multiple servers are grouped together to achieve the same business and can be regarded as one computer.

Server Clusters

- Simply put, a server cluster is nothing else but a collection of machines connected through a network, where each machine runs one or more servers.

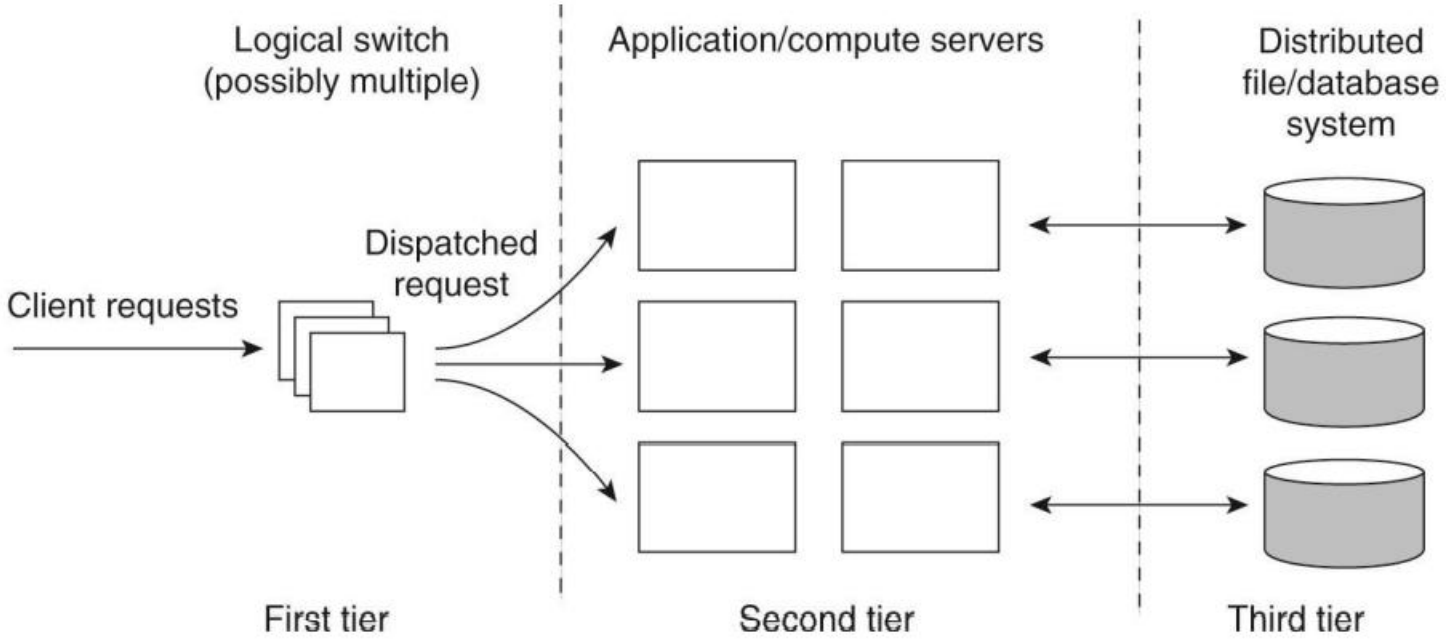


Fig: The general organization of a three-tiered server cluster.

Server Clusters

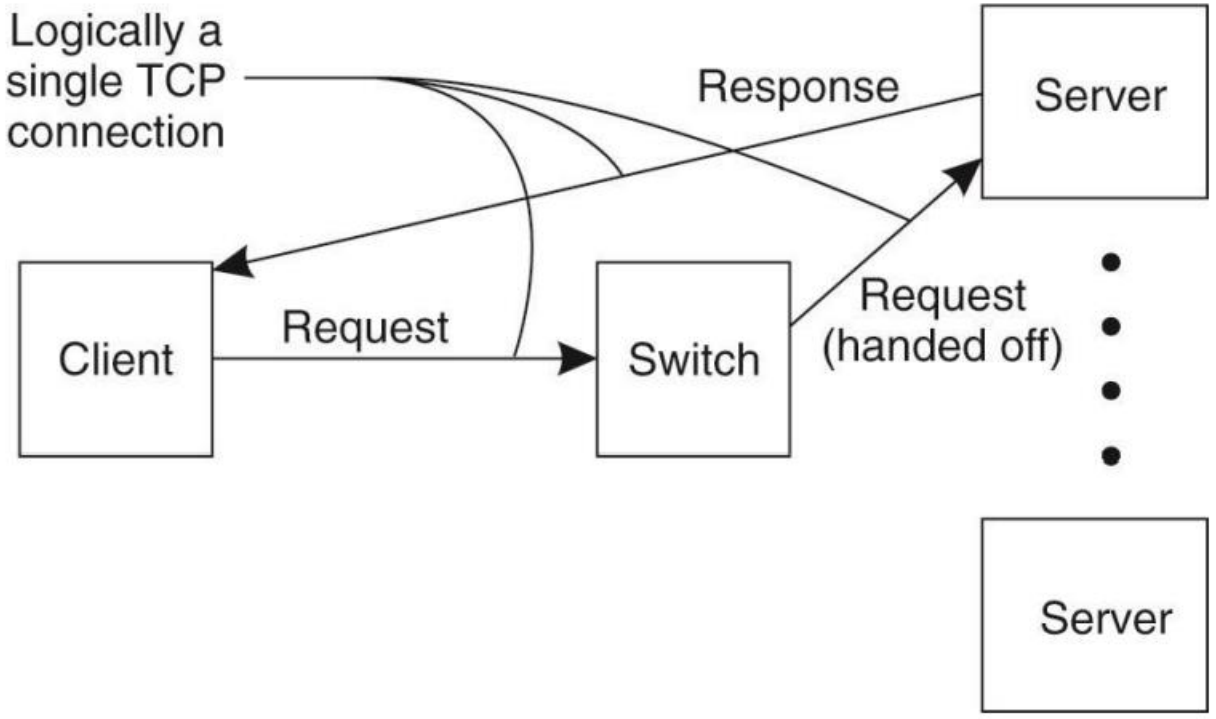


Fig: The principle of TCP handoff.

Revisit

- Server Cluster → Statically configured
  → Single Access point → Single point of Failure
- Access Point → Publicly Available

Eg. DNS

It returns multiple addresses (Same host)

If any one of the address fails, client will have to do several attempts.

-doesn’t solve the problem of requiring static AP.

What we need ?

- Static, high living Access Point
- Flexibility in configuration (No more Static Configurations)

- This Lead to a design of Distributed Server.

Distributed Server

- Stable AP
- Mobile IPV6

MIPv6

- Mobile Computing
  - Communication with mobile host, WLANs
- Why we need MIPv6 ?
  - Ipv4 → local network → It works fine
  - When it left Local Network → Problem arises

  - Therefore we need the concept of MIPV6.

Mobile IPv6

Basic Idea:

- Every Mobile end devices receives two IP Address simultaneously:
  - Primary IP Address
    - Home Address
  - Secondary IP Address
    - Care-of Address (temporary)

Primary IP Address: Home Address

- Stationary IP Address of the mobile host in its local network
- Always remains unchanged, even when location is changed
- Application running on the mobile host only use primary IP address.

Secondary IP Address: Care-of Address (temporary)

- Temporarily Valid
- Change with every location change and is only valid as long as the host remains in certain guest network.
- Mobile host receives second IP address upon registration in new network and communicate it promptly to a specific agent(Router in home network).

Distributed Servers

- The basic idea behind a distributed server is that clients benefit from a robust, high-performing, stable server. **These properties can often be provided by high-end mainframes, of which some have an acclaimed mean time between failure of more than 40 years.**

- However, by grouping simpler machines transparently into a cluster, and not relying on the availability of a single machine, it may be possible to achieve a better degree of stability than by each component individually. For example, such a cluster could be dynamically configured from end user machines as in the case of a collaborative distributed system

Distributed Servers

So far, server clusters are generally:

1- rather statically configured. There is often a separate administration machine that keeps track of available servers, and passes this information to other machines as appropriate, such as the switch.

2- offer a single access point. When that point fails, the cluster becomes unavailable.

Distributed Servers

Two requirements:

1) Having a stable and long-living access point,

2) High level of flexibility in configuring a server cluster.

To eliminate this potential problem, several access points can be provided, of which the addresses are made publicly available. For example, the Domain Name System (DNS) can return several addresses, all belonging to the same host name. This approach still requires clients to make several attempts if one of the addresses fails. Moreover, this does not solve the problem of requiring static access points

Distributed Servers

- Let us concentrate on how a **stable access point** can be achieved in such a system. The main idea is to make use of available networking services, notably **mobility support** for IP version 6 (MIPv6).
- In MIPv6, a mobile node is assumed to have a **home network** where it normally resides and for which it has an associated stable address, known as its **Home Address (HoA).** This home network has a special router attached, known as the **home agent**, which will take care of traffic to the mobile node when it is away.
- To this end, when a mobile node attaches to a foreign network, it will receive a **temporary Care-of Address (CoA)** where it can be reached. This care-of address is reported to the node's home agent who will then see to it that all traffic is forwarded to the mobile node. Note that applications communicating with the mobile node **will only see the address associated with the node's home network.** They will never see the care-of address.

Assignment-IV (deadline Feb-8)

1. Distinguish between server cluster and Distributed server.
2. Explain the concept behind MIPv6 and explain its working with suitable diagram(Architecture).

Code Migration

- So far, communication is limited to passing data in distributed systems.
- However, there are situations in which passing programs, simplifies the design of a distributed system.
  - What code migration actually is.
  - Different approaches to code migration,
  - How to deal with the local resources that a migrating program uses

A particularly hard problem is migrating code in heterogeneous systems

Approaches to Code Migration

- Before taking a look at the different forms of code migration, let us first consider why it may be useful to migrate code.

Reasons for Migrating Code (imp)

1- Overall system performance can be improved if processes are moved from heavily-loaded to lightly-loaded machines.

2- To minimize communication. If a client application needs to perform many database operations involving large quantities of data, it may be better to ship part of the client application to the server and send only the results across the network. This same reason can be used for migrating parts of the server to the client.

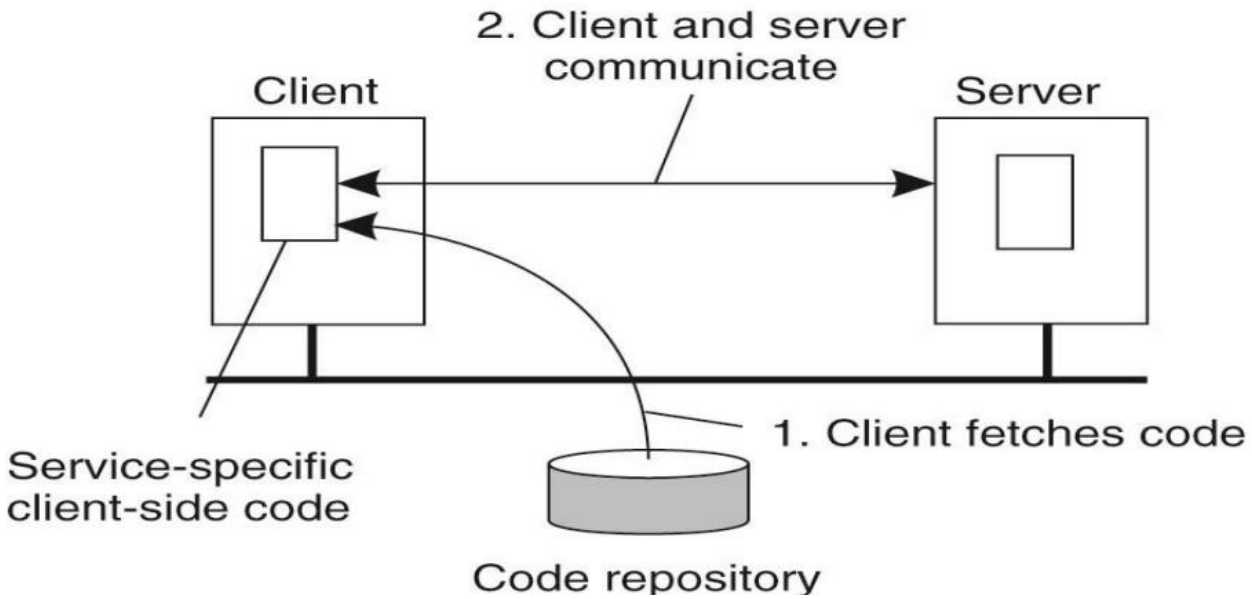
3- To improve performance by exploiting parallelism. A typical example is searching for information in the Web. It is relatively simple to implement a search query in the form of a small mobile program, called a mobile agent, that moves from site to site.

4- Flexibility is another reason. An approach to building distributed applications is to partition the application into different parts, and decide in advance where each part should be executed.

5- Dynamically configure distributed systems.

Dynamically configuring a client to communicate to a server

- Let the server provide the client's implementation no sooner than is strictly necessary, that is, when the client binds to the server.
- At that point, the client dynamically downloads the implementation, goes through the necessary initialization steps, and subsequently invokes the server.
- This model of dynamically moving code from a remote site does require that the protocol for downloading and initializing code is standardized. Also, it is necessary that the downloaded code can be executed on the client's machine.



The principle of dynamically configuring a client to communicate to a server. The client first fetches the necessary software, and then invokes the server.

Dynamically configuring a client to communicate to a server

Advantage

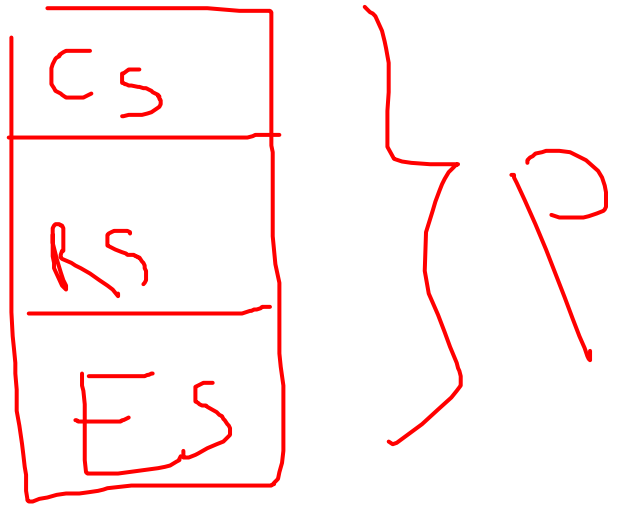
- Clients need not have all the software preinstalled to talk to servers.
- As long as interfaces are standardized, we can change the client-server protocol and its implementation as often as we like. Changes will not affect existing client applications that rely on the server.

Disadvantages

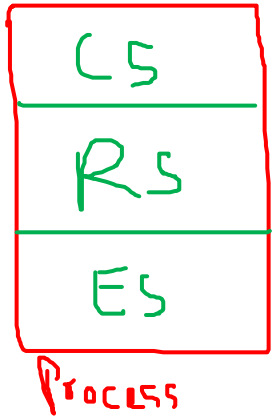
- ***Security:*** Blindly trusting that the downloaded code implements only the advertised interface while accessing your unprotected hard disk and does not send the juiciest parts to heaven-knows-who may not always be such a good idea.

Models for Migrating Code

- Traditionally, communication in distributed systems is concerned with exchanging data between process.
- Code migration in the broadest sense deals with moving programs between machines, with the intention to have those programs be executed at the target.
- A Process consists of three segments:
  - Code Segment
  - Resources Segment
  - Execution Segment



Models for Migrating Code



- **Weak Migration**

only the code segment can be transferred, along with perhaps some initialization data.

Eg. with Java applets, which always start execution from the beginning. The benefit of this approach is its simplicity.

- **Strong Migration**

The code segment and execution segment can be transferred.

Features:

✓ a running process can be stopped, subsequently moved to another machine, and then resume execution.

✓ more general than weak mobility

✓ harder to implement

Models for Migrating Code

- **sender-initiated migration**

uploading programs to a compute server

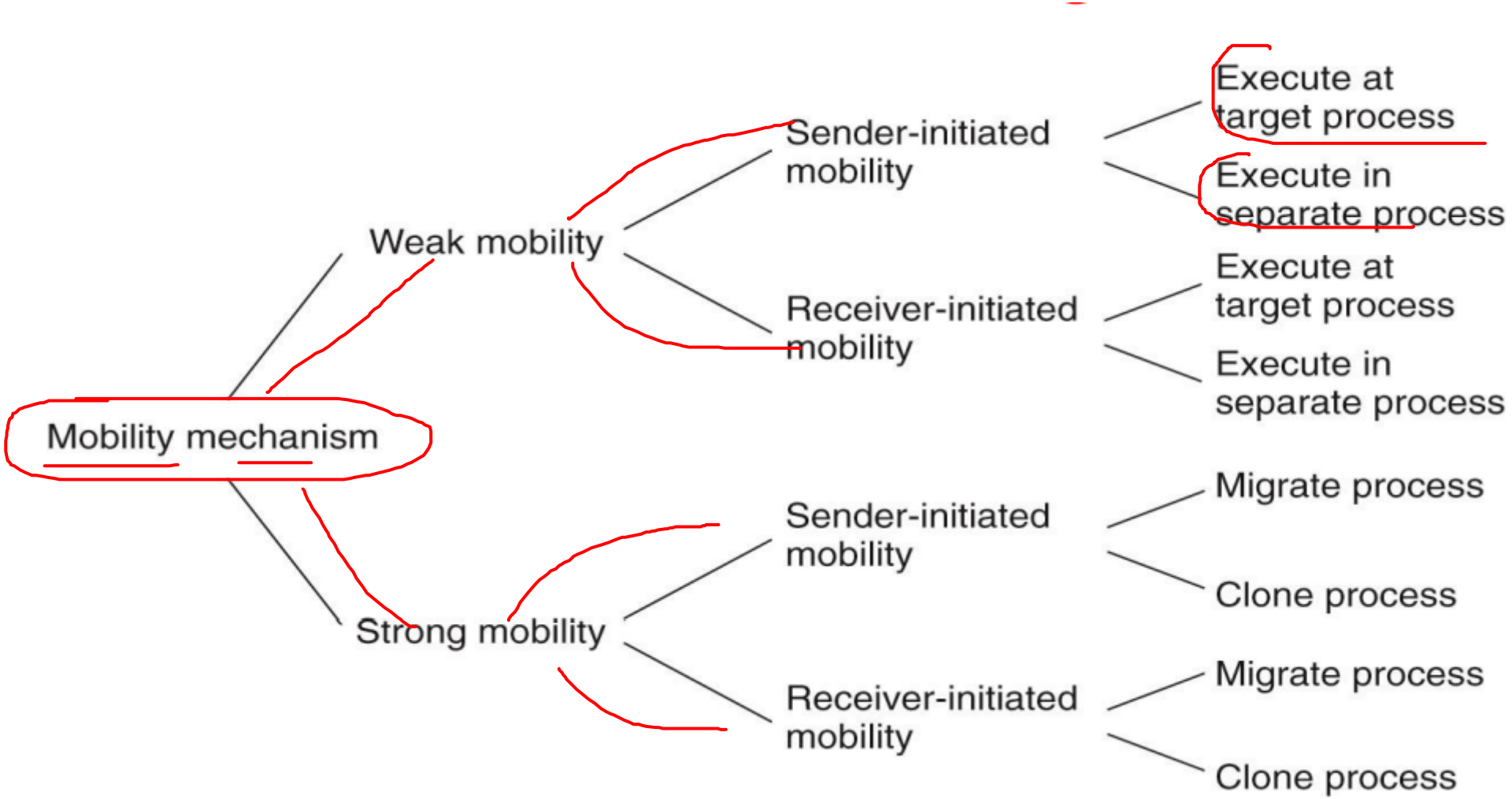
  - sending a search/query program across the Internet to a Web database server to perform the queries at that server.

- **receiver-initiated migration**

Downloading code from server by a client

  - Java applets are an example of this approach. (i.e. Java applets)

Models for Migrating Code



Migration and Local Resources

- So far, only the migration of the code and execution segment has been taken into account. The resource segment requires some special attention. What often makes code migration so difficult is that:
  - the resource segment cannot always be simply transferred along with the other segments without being changed.

Resources Migration

- Depends on type of “resources”
  - By Identifier: Specific website, ftp server
  - By Value: Java Libraries
  - By Type: Printer, Local devices

- Depends on type of “attachments”
  - Un attached to any node: Datafiles
  - Fastened resources(can be moved at high cost)
  - E.G. local databases and complete Web sites.

3 types of process-to-resource bindings

- Binding-by-identifier
  - the strongest that precisely the referenced resource, and nothing else, has to be migrated
  - E.g. when a process uses an URL
- Binding-by-value
  - weaker than BI, but only the value of the resource need be migrated
  - A program relying on a libraries (use the locally available one)
- Binding-by-type
  - nothing is migrated, but a resource of a specific type needs to be available after migration
  - E.g. local devices like monitors, printers

3 types of resource-to-machine bindings

- Unattached resources
  - a resource that can be moved easily from machine to machine (e.g. files)
- Fastened resource
  - migration is possible, but at a high cost (e.g. local databases, complete web sites)
- Fixed resources
  - a resource is bound to a specific machine or environment, and cannot be migrated.
  - (e.g. local devices, ports)

9 Possible Combinations

Process-to-resource binding	Resource-to-machine binding: Unattached	Fastened	Fixed
By identifier	MV (or GR)	GR (or MV)	GR
By value	CP (or MV,GR)	GR (or CP)	GR
By type	RB (or MV,CP)	RB (or GR,CP)	RB (or GR)

GR	Establish a global systemwide reference
MV	Move the resource
CP	Copy the value of the resource
RB	Rebind process to locally-available resource

Actions to be taken with respect to the references to local resources when migrating code to another machine.

Migrating Code in Heterogeneous Systems

- So far, we have assumed that the migrated code can be easily executed at the target machine. This assumption is in order when dealing with homogeneous system.
- In general, however, distributed systems are constructed on a heterogeneous collection of platforms, each having their own operating system and machine architecture. Migration in such systems requires that each platform is supported, that is, that the code segment can be executed on each platform.

Migrating Code in Heterogeneous Systems

- Is more complex.
- Requires code portability.
- A virtual machine approach is used.
- Weak mobility is easier
- In strong mobility it is necessary to handle the execution segment.
- A migration stack is used.

Unit 4: Communication

By Prashant Gautam

Outlines

Background

- Foundation
- RPC
- Message-Oriented Communication
- Multicast Communication
- Case Study: Java RMI and MPI

Background

- Inter-process communication is at the heart of all distributed systems.
- It makes no sense to study distributed systems without carefully examining the ways that processes on different machines can exchange information.
- Communication in distributed systems is always based on low-level message passing as offered by the underlying network.
- Expressing communication through message passing is harder than using primitives based on shared memory, as available for non-distributed platforms.

- Modern distributed systems often consist of thousands or even millions of processes scattered across a network with unreliable communication such as the Internet.
- Unless the primitive communication facilities of computer networks are replaced by something else, development of large-scale distributed applications is extremely difficult.

What we will study ?

- The rules that communicating processes must adhere to, known as protocols, and concentrate on structuring those protocols in the form of layers.
- Widely-used models for communication:
  - Remote Procedure Call (RPC),
  - Message-Oriented Middleware (MOM), and
  - data streaming.
- Multicasting concept: the general problem of sending data to multiple receivers

Before we start our discussion on communication in distributed systems, we first recapitulate some of the fundamental issues related to communication.

Communication Protocol

- Protocol: Set of rules on communication
- To allow a group of computers to communicate over a network, they must all agree on the protocols to be used.
- Protocols can be connectionless and connection oriented

Connection-oriented Protocol

- before exchanging data the sender and receiver first explicitly establish a connection.
- When they are done, they must release (terminate) the connection.
- The telephone is a connection-oriented communication system

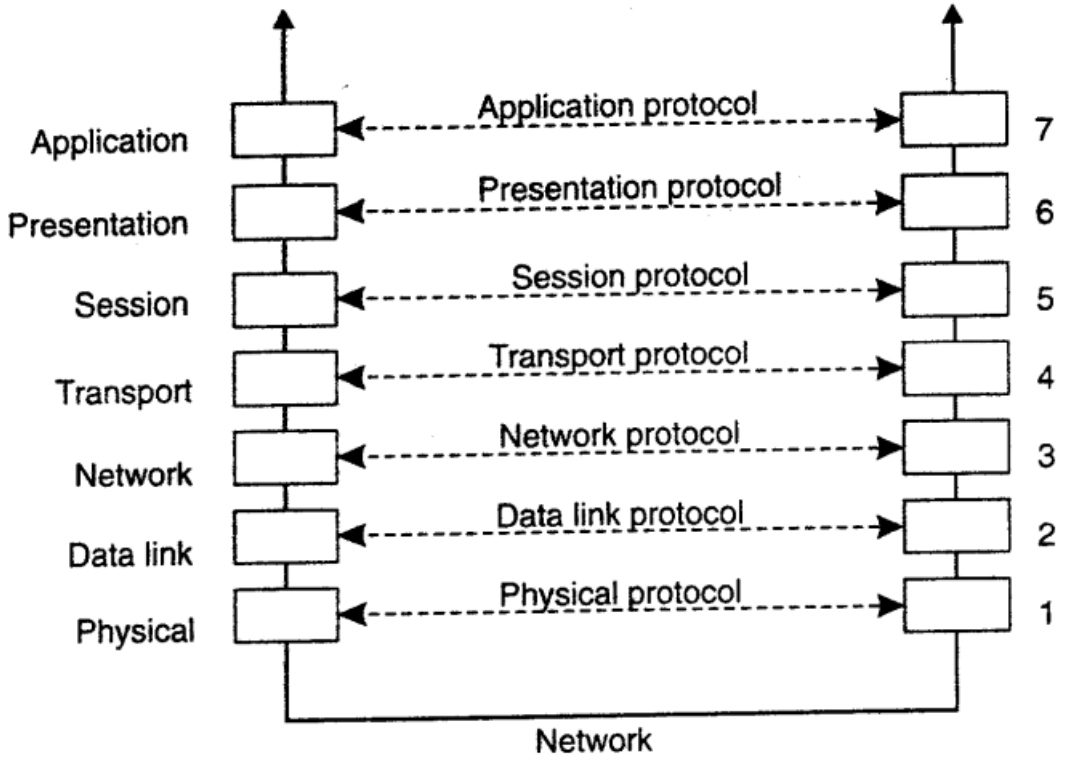
Connectionless Protocol

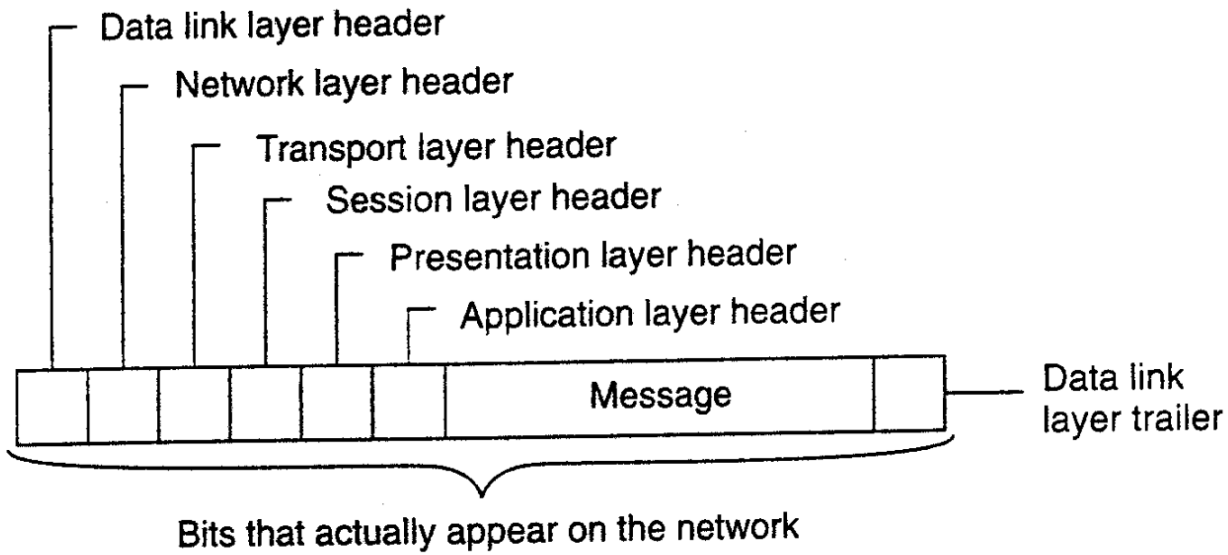
- No setup in advance is needed.
- The sender just transmits the first message when it is ready.
- Dropping a letter in a mailbox is an example of connectionless communication.

With computers, both connection-oriented and connectionless communication are common.

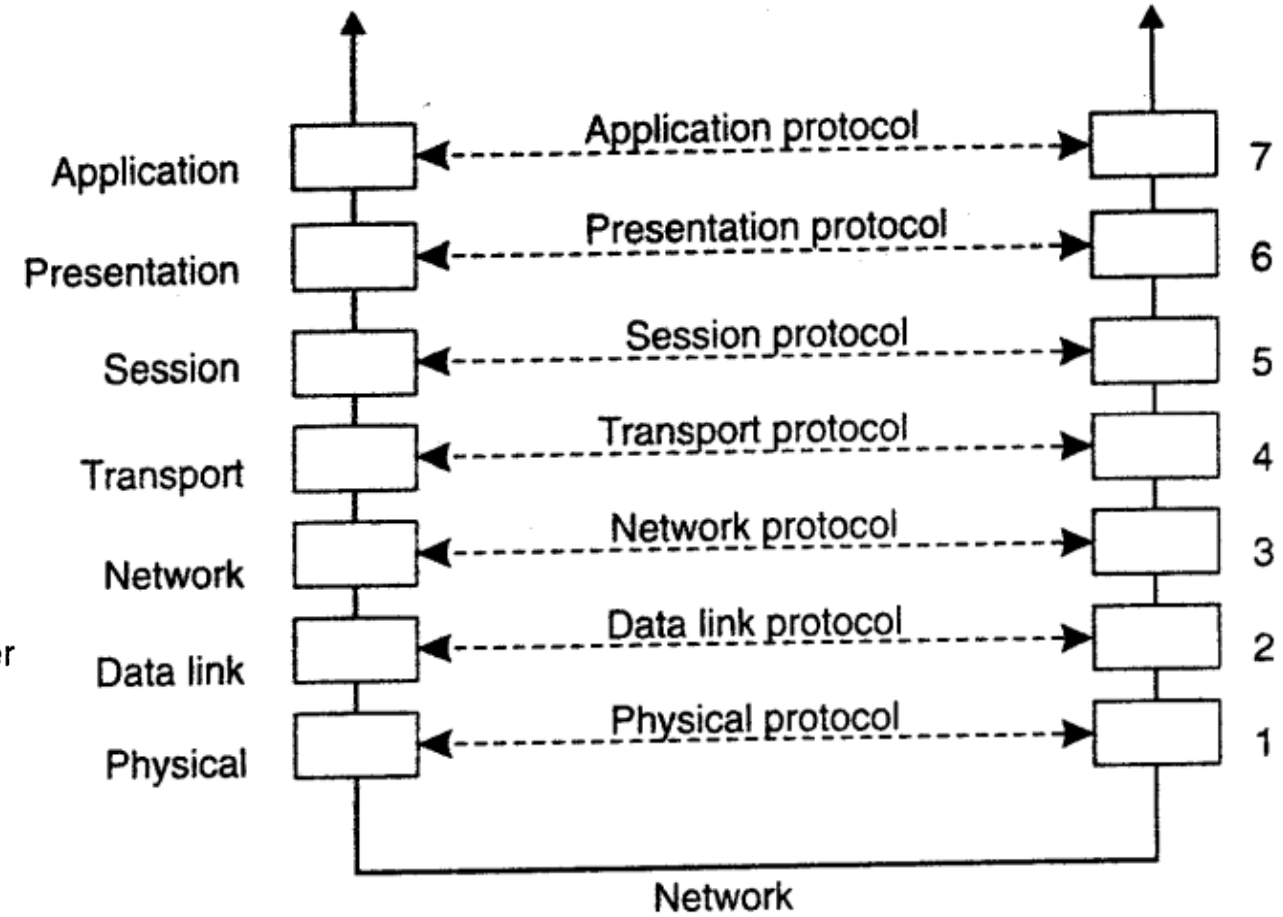
OSI model

- In the OSI model, communication is divided up into seven levels or layers, as shown in Fig.
- Each layer deals with one specific aspect of the communication.
- In this way, the problem can be divided up into manageable pieces, each of which can be solved independent of the others.
- Each layer provides an interface to the one above it.
- The interface consists of a set of operations that together define the service the layer is prepared to offer its users.

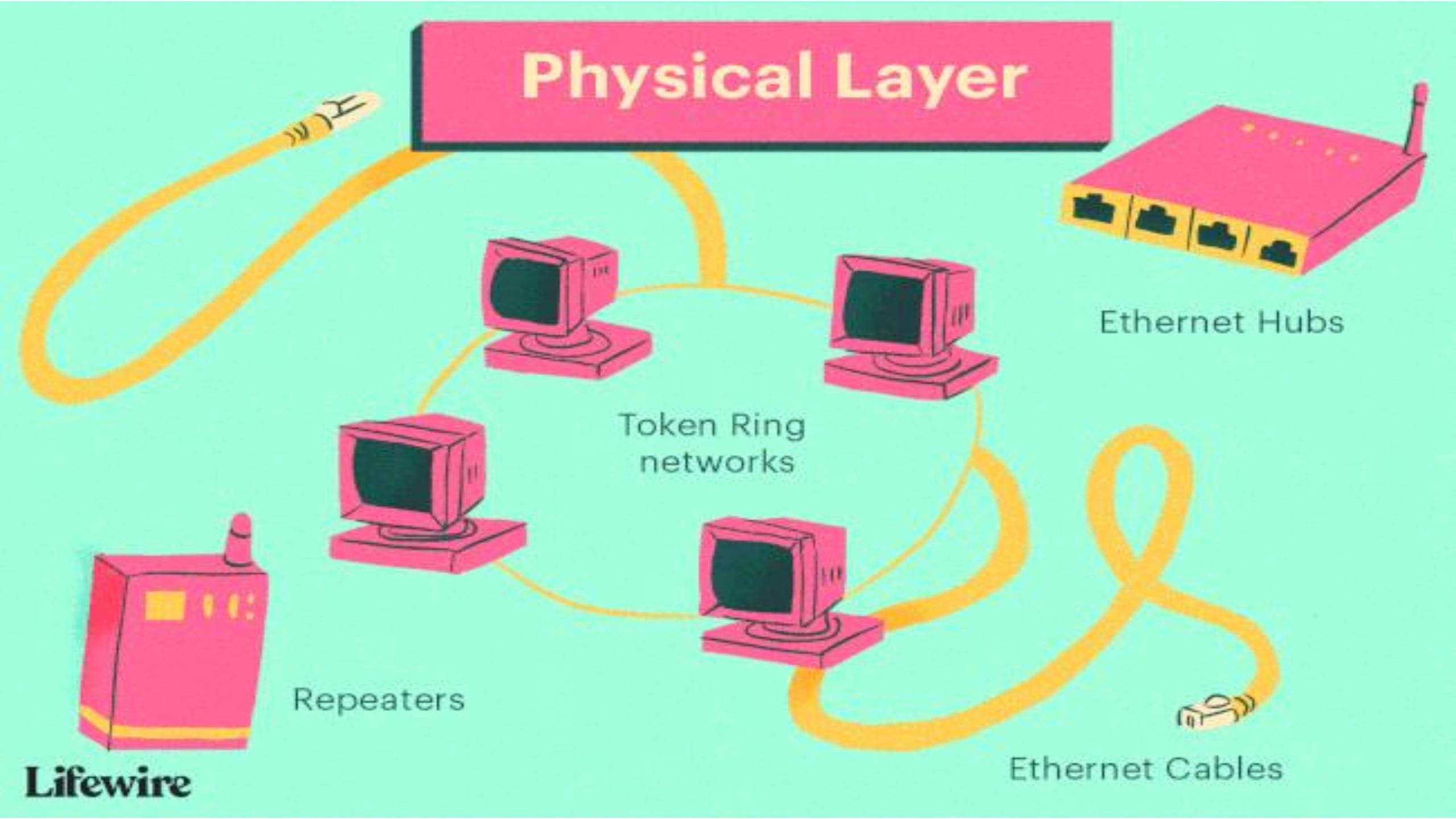




A typical message as it appears on the network



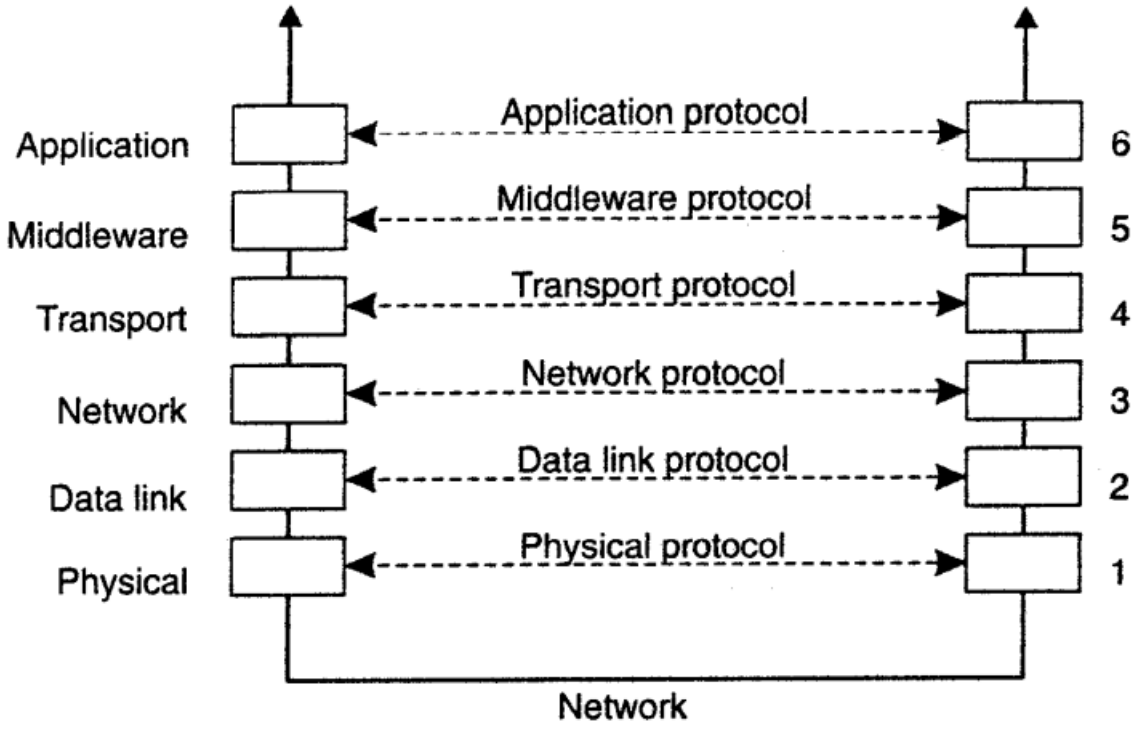
Layers, Interfaces, and Protocols in the OSI Model


Physical Layer
Ethernet Hubs
Token Ring
networks
Repeaters
Ethernet Cables
Lifewire

Video

Protocols

- Lower Level Protocols
- Transport Protocols
- Higher- Level Protocols
- Middleware Protocols



Lower Level Protocols

- We start with discussing the three lowest layers of the OSI protocol suite. Together, these layers implement the basic functions that encompass a computer network.

- The physical layer is concerned with transmitting the 0s and 1s.
- Key Issues in physical layer:
  - How many volts to use for 0 and 1,
  - how many bits per second can be sent, and
  - whether transmission can take place in both directions simultaneously
  - the size and shape of the network connector (plug), as well as the number of pins and meaning of each are of concern here.
- The physical layer protocol deals with standardizing the electrical, mechanical, and signaling interfaces so that when one machine sends a 0 bit it is actually received as a 0 bit and not a 1 bit.
- Many physical layer standards have been developed (for different media), for example, the RS-232-C standard for serial communication lines.

- The physical layer just sends bits.
- As long as no errors occur, all is well.
- However, real communication networks are subject to errors, so some mechanism is needed to detect and correct them.
- This mechanism is the main task of the data link layer.
- What it does is to group the bits into units, sometimes called frames, and see that each frame is correctly received.
- The data link layer does its work by putting a special bit pattern on the start and end of each frame to mark them, as well as computing a checksum by adding up all the bytes in the frame in a certain way.

- The data link layer appends the checksum to the frame.
- When the frame arrives, the receiver recomputes the checksum from the data and compares the result to the checksum following the frame.
- If the two agree, the frame is considered correct and is accepted.
- If they disagree. the receiver asks the sender to retransmit it. Frames are assigned sequence numbers (in the header), so everyone can tell which is which.

- On a LAN, there is usually no need for the sender to locate the receiver.
- It just puts the message out on the network and the receiver takes it off.
- A wide-area network, however, consists of a large number of machines, each with some number of lines to other machines, rather like a large-scale map showing major cities and roads connecting them.
- For a message to get from the sender to the receiver it may have to make a number of hops, at each one choosing an outgoing line to use. The question of how to choose the best path is called routing, and is essentially the primary task of the network layer.

- The problem is complicated by the fact that the shortest route is not always the best route.
- What really matters is the amount of delay on a given route, which, in tum, is related to the amount of traffic and the number of messages queued up for transmission over the various lines.
- The delay can thus change over the course of time. Some routing algorithms try to adapt to changing loads, whereas others are content to make decisions based on long-term averages.

- At present, the most widely used network protocol is the connectionless IP (Internet Protocol), which is part of the Internet protocol suite.
- An IP packet (the technical term for a message in the network layer) can be sent without any setup.
- Each IP packet is routed to its destination independent of all others. No internal path is selected and remembered.

Transport Protocols

- The transport layer forms the last part of what could be called a basic network protocol stack, in the sense that it implements all those services that are not provided at the interface of the network layer, but which are reasonably needed to build network applications.
- In other words, the transport layer turns the underlying network into something that an application developer can use.
- Packets can be lost on the way from the sender to the receiver.
- Although some applications can handle their own error recovery, others prefer a reliable connection.
- The job of the transport layer is to provide this service.
- The idea is that the application layer should be able to deliver a message to the transport layer with the expectation that it will be delivered without loss.

- Upon receiving a message from the application layer, the transport layer breaks it into pieces small enough for transmission, assigns each one a sequence number, and then sends them all.
- The discussion in the transport layer header concerns which packets have been sent, which have been received, how many more the receiver has room to accept, which should be retransmitted, and similar topics.
- Reliable transport connections (which by definition are connection oriented) can be built on top of connection-oriented or connectionless network services.
- In the former case all the packets will arrive in the correct sequence (if they arrive at all), but in the latter case it is possible for one packet to take a different route and arrive earlier than the packet sent before it.
- It is up to the transport layer software to put everything back in order to maintain the illusion that a transport connection is like a big tube-you put messages into it and they come out undamaged and in the same order in which they went in. Providing this end-to-end communication behavior is an important aspect of the transport layer.

- The combination TCP/IP is now used as a de facto standard for network communication.
- The Internet protocol suite also supports a connectionless transport protocol called UDP (Universal Datagram Protocol), which is essentially just IP with some minor additions.
- User programs that do not need a connection-oriented protocol normally use UDP.
- Additional transport protocols are regularly proposed.
- For example, to support real-time data transfer, the Real-time Transport Protocol (RTP) has been defined.
- RTP is a framework protocol in the sense that it specifies packet formats for real-time data without providing the actual mechanisms for guaranteeing data delivery.

Higher- Level Protocols

- Above the transport layer, OSI distinguished three additional layers.
- In practice, only the application layer is ever used.
- In fact, in the Internet protocol suite, everything above the transport layer is grouped together.
- In the face of middleware systems, we shall see in this section that neither the OSI nor the Internet approach is really appropriate.

- The session layer is essentially an enhanced version of the transport layer.
- It provides dialog control, to keep track of which party is currently talking, and it provides synchronization facilities.
- The latter are useful to allow users to insert checkpoints into long transfers, so that in the event of a crash, it is necessary to go back only to the last checkpoint, rather than all the way back to the beginning.
- In practice, few applications are interested in the session layer and it is rarely supported.
- It is not even present in the Internet protocol suite.
- However, in the context of developing middleware solutions, the concept of a session and its related protocols has turned out to be quite relevant, notably when defining higher-level communication protocols.

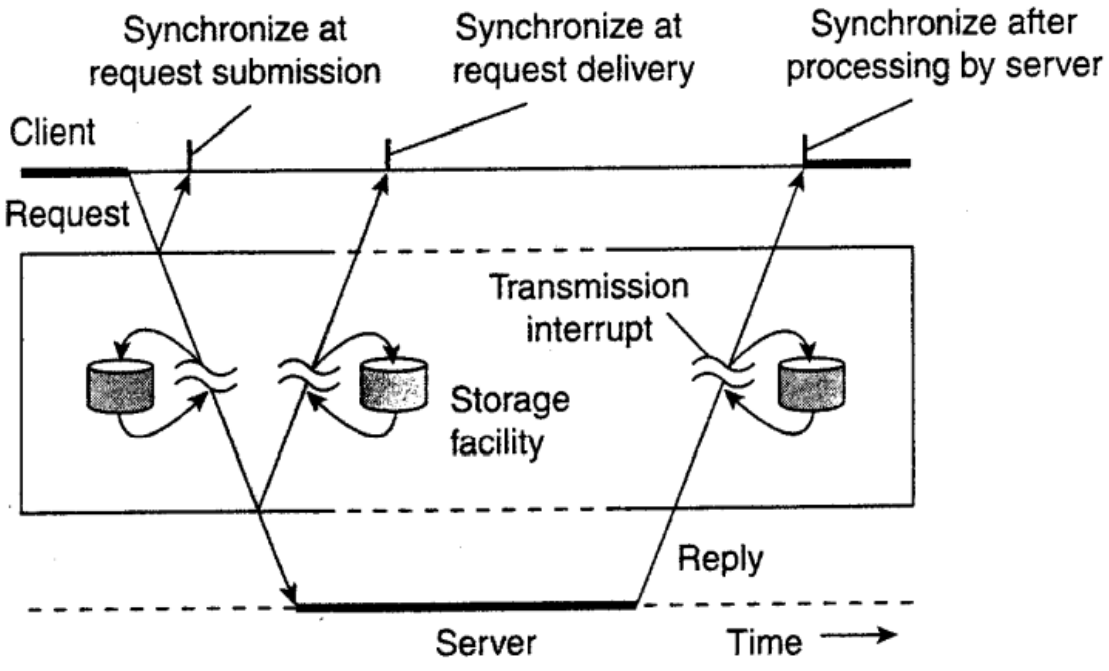
- Unlike the lower layers, which are concerned with getting the bits from the sender to the receiver reliably and efficiently, the presentation layer is concerned with the meaning of the bits.
- Most messages do not consist of random bit strings, but more structured information such as people's names, addresses, amounts of money, and so on.
- In the presentation layer it is possible to define records containing fields like these and then have the sender notify the receiver that a message contains a particular record in a certain format.
- This makes it easier for machines with different internal representations to communicate with each other.

Middleware Protocols

- Middleware is an application that logically lives (mostly) in the application layer, but which contains many general-purpose protocols that warrant their own layers, independent of other, more specific applications.
- A distinction can be made between high-level communication protocols and protocols for establishing various middleware services.
- Authentication protocols
- Authorization protocols

Types of Communication

- To understand the various alternatives in communication that middleware can offer to applications, we view the middleware as an additional service in client-server computing, as shown in Fig.



Example: electronic mail system

- In principle, the core of the mail delivery system can be seen as a middleware communication service.
- Each host runs a user agent allowing users to compose, send, and receive e-mail.
- A sending user agent passes such mail to the mail delivery system, expecting it, in tum, to eventually deliver the mail to the intended recipient.
- Likewise, the user agent at the receiver's side connects to the mail delivery system to see whether any mail has come in.
- If so, the messages are transferred to the user agent so that they can be displayed and read by the user.

Persistence and Synchronicity

- Persistent vs Transient Communication
  - Persistent: submitted messages are stored by the comm. system as long as it takes to deliver it the receiver
  - Transient: messages are stored by the comm system as long as the sending and receiving app. are executing.
- Asynchronous vs synchronous
  - Sync: a client is blocked until its message is stored in a local buffer at the receiving host, or actually delivered to the receiver
  - Async: A sender continues immediately after it has submitted its message for transmission. (The message is stored in a local buffer at the sending host, or otherwise at the first communication server.

Persistent Communication

- An electronic mail system is a typical example in which communication is persistent.
- With persistent communication, a message that has been submitted for transmission is stored by the communication middleware as long as it takes to deliver it to the receiver.
- In this case, the middleware will store the message at one or several of the storage facilities shown in Fig in previous slide.
- As a consequence, it is not necessary for the sending application to continue execution after submitting the message.
- Likewise, the receiving application need not be executing when the message is submitted.

Transient Communication

- In contrast, with transient communication, a message is stored by the communication system only as long as the sending and receiving application are executing.
- More precisely, in terms of Fig in previous slide, the middleware cannot deliver a message due to a transmission interrupt, or because the recipient is currently not active, it will simply be discarded.
- Typically, all transport-level communication services offer only transient communication.
- In this case, the communication system consists traditional store-and-forward routers.
- If a router cannot deliver a message to the next one or the destination host, it will simply drop the message

Asynchronous or Synchronous Communication

- The characteristic feature of asynchronous communication is that a sender continues immediately after it has submitted its message for transmission.
- This means that the message is (temporarily) stored immediately by the middleware upon submission.

- With synchronous communication, the sender is blocked until its request is known to be accepted.
- There are essentially three points where synchronization can take place.
  - First, the sender may be blocked until the middleware notifies that it will take over transmission of the request.
  - Second, the sender may synchronize until its request has been delivered to the intended recipient.
  - Third, synchronization may take place by letting the sender wait until its request has been fully processed, that is, up the time that the recipient returns a response.

Assignment-IV (Deadline: Feb 20)

- Explain OSI Refence Model for data communication in details.
- Explain the different types of communication based on persistence and synchronicity.

REMOTE PROCEDURE CALL

- Background

Many distributed systems have been based on explicit message exchange between processes.

However, the procedures send and receive do not conceal communication at all, which is important to achieve access transparency in distributed system.

RPC: Definition

- Concept: Allowing programs to call procedures located on other machines.
- When a process on machine A calls a procedure on machine B, the calling process on A is suspended, and execution of the called procedure takes place on B.
- Information can be transported from the caller to the callee in the parameters and can come back in the procedure result.
- No message passing at all is visible to the programmer.
- This method is known as Remote Procedure Call, or often just RPC.

- While the basic idea sounds simple and elegant, subtle problems exist.
- To start with, because the calling and called procedures run on different machines, they execute in different address spaces, which causes complications.
- Parameters and results also have to be passed, which can be complicated, especially if the machines are not identical.
- Finally, either or both machines can crash and each of the possible failures causes different problems.
- Still, most of these can be dealt with, and RPC is a widely-used technique that underlies many distributed systems.

Basic RPC Operation

- We first start with discussing conventional procedure calls, and then explain how the call itself can be split into a client and server part that are each executed on different machines.

Conventional Procedure Call

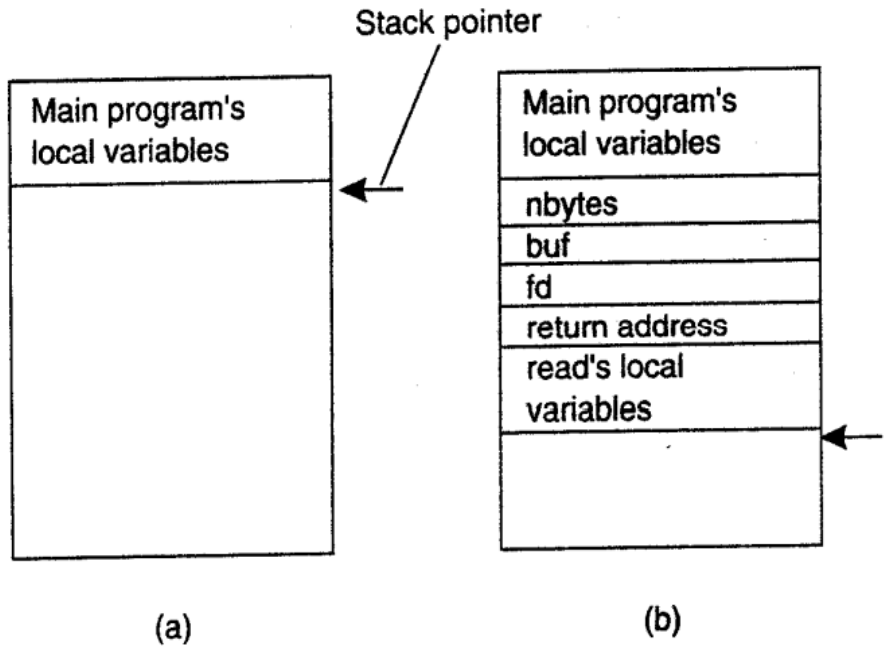
- Conventional (i.e., single machine) Procedure Call
- Consider a call in C like:

Count = tead(fd, but, nbytes)

Where,

fd →integer indicating a file,
buf →an array of characters into which data are read, and
nbytes → another integer telling how many bytes to read

- If the call is made from the main program, the stack will be as shown in Fig. (a) before the call.
- To make the call, the caller pushes the parameters onto the stack in order, last one first, as shown in Fig. (b).
- After the read procedure has finished running, it puts the return value in a register, removes the return address, and transfers control back to the caller.
- The caller then removes the parameters from the stack, returning the stack to the original state it had before the call.



(a) Parameter passing in a local procedure call: the stack before the call to read. (b) The stack while the called procedure is active.

- parameters can be call-by value or call-by-reference.

call-by value

- A value parameter, such as fd or nbytes, is simply copied to the stack as shown in Fig.(b).
- To the called procedure, a value parameter is just an initialized local variable.
- The called procedure may modify it, but such changes do not affect the original value at the calling side.

Call-by-reference

- A reference parameter in C is a pointer to a variable (i.e., the address of the variable), rather than the value of the variable.
- In the call to read. the second parameter is a reference parameter because arrays are always passed by reference in C.
- What is actually pushed onto the stack is the address of the character array.
- If the called procedure uses this parameter to store something into the character
- array, it does modify the array in the calling procedure.

*pass by **reference***

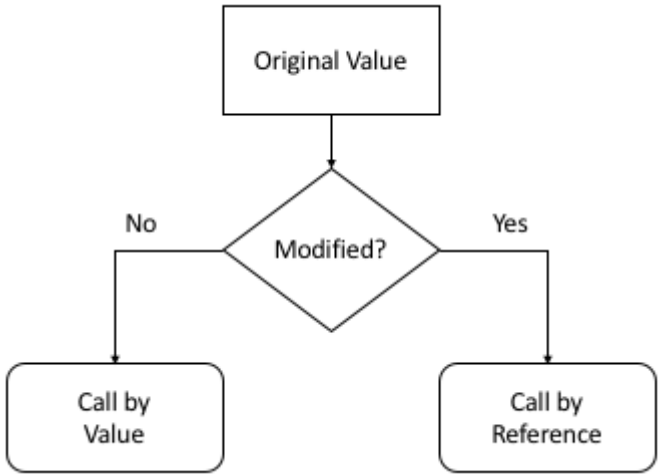
cup =

fillCup( )

*pass by **value***

cup =

fillCup( )



PARAMETERS	CALL BY VALUE	CALL BY REFERENCE
Basic	A copy of the variable is passed.	A variable itself is passed.
effect	Change in a copy of variable doesn't modify the original value of variable.	Change in a copy of variable modify the original value of variable.
Syntax	function_name(variable_name1, variable_name2...)	function_name(&variable_name1, &variable_name2...)
Default calling	Primitive type are passed using "call_by_value".	Objects are implicitly passed using "call_by_reference".

Elements of RPC: Client Server Model

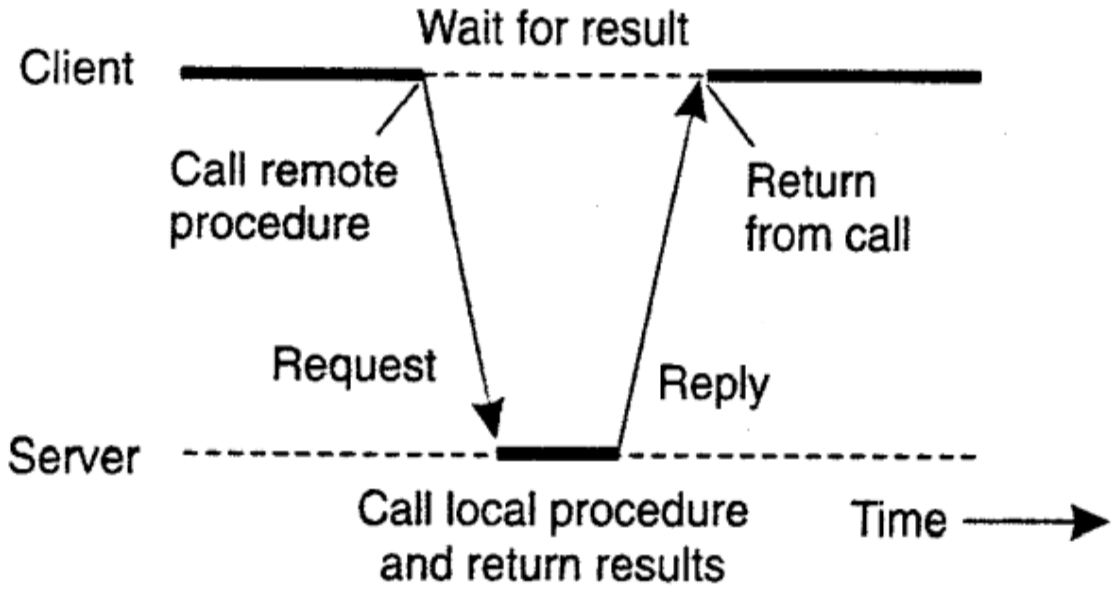
- RPC Uses client server model.

Mainly five components:

- The Client
- The Client Stub

  Stub: Piece of code used for converting parameter
- The RPC Routine

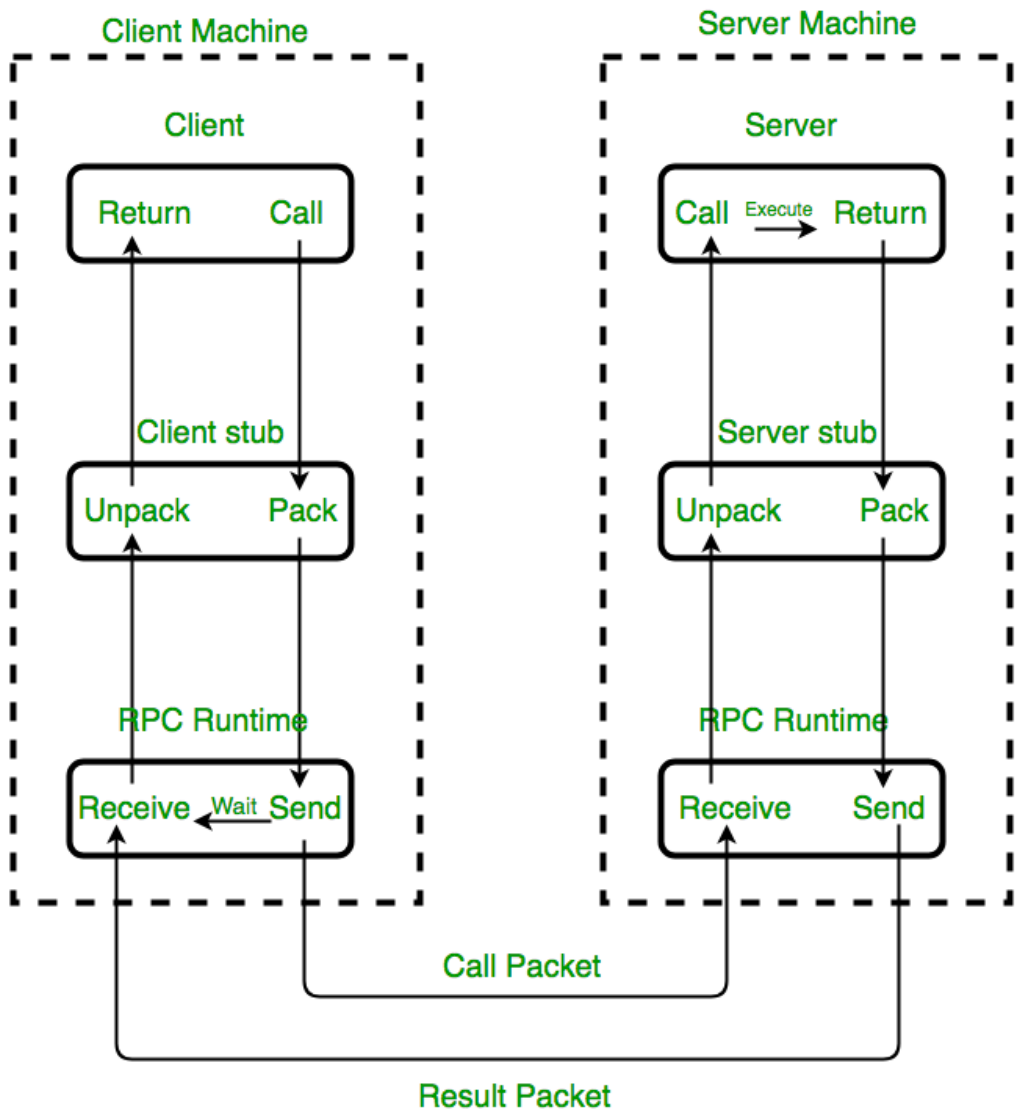
  (Communication Package)
- The Server Stubs
- The Server



Principle of RPC between a client and server program.

- The Client:
  - It is user process which initiates a RPC.
  - The Client makes a perfectly normal call that involves a corresponding procedure in the client stub.
- The Client Stub:
  - On receipt of a request it packs a requirement into a message and asks to RPC runtime to send.
  - On receipt of a result it unpacks the result and passes it to client.

- RPC Runtime:
  - It handles transmission of message between client and Server.
- The Server Stub:
  - It unpacks a call request and make a perfectly normal call to invoke the appropriate procedure in the Server.
- Server:
  - It executes an appropriate procedure and returns the result from a server stub.



Implementation of RPC mechanism

RPC steps

1. The client procedure calls the client stub in the normal way.

2. The client stub builds a message and calls the local operating system.

3. The client's OS sends the message to the remote as.

4. The remote OS gives the message to the server stub.

5. The server stub unpacks the parameters and calls the server.

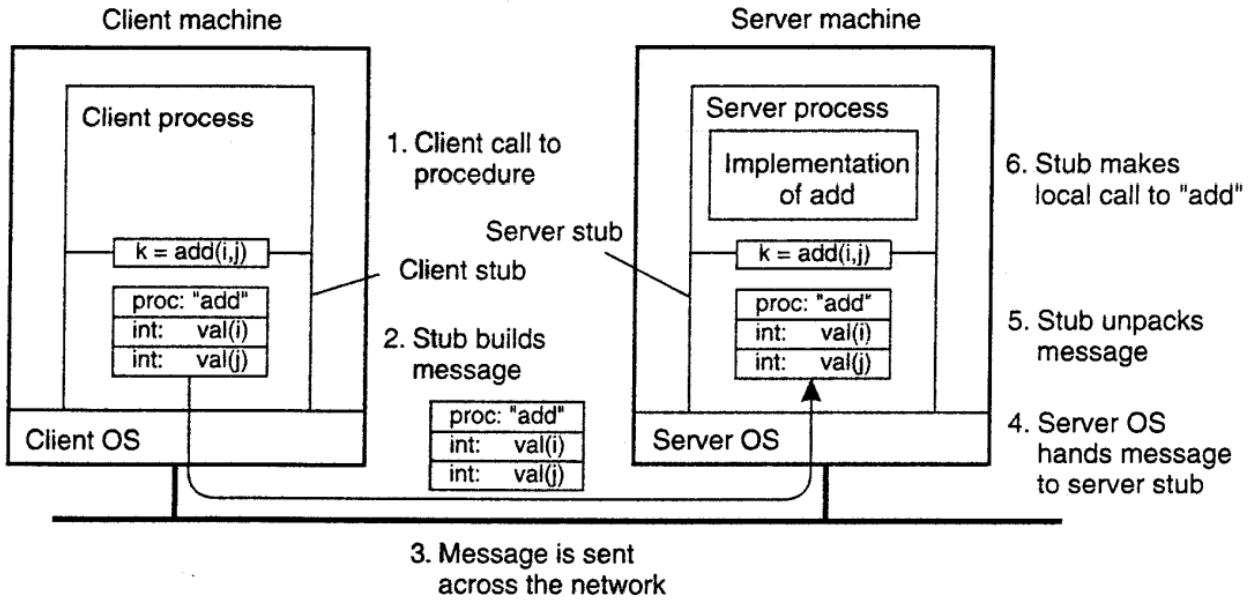
6. The server does the work and returns the result to the stub.

7. The server stub packs it in a message and calls its local OS.

8. The server's OS sends the message to the client's as.

9. The client's OS gives the message to the client stub.

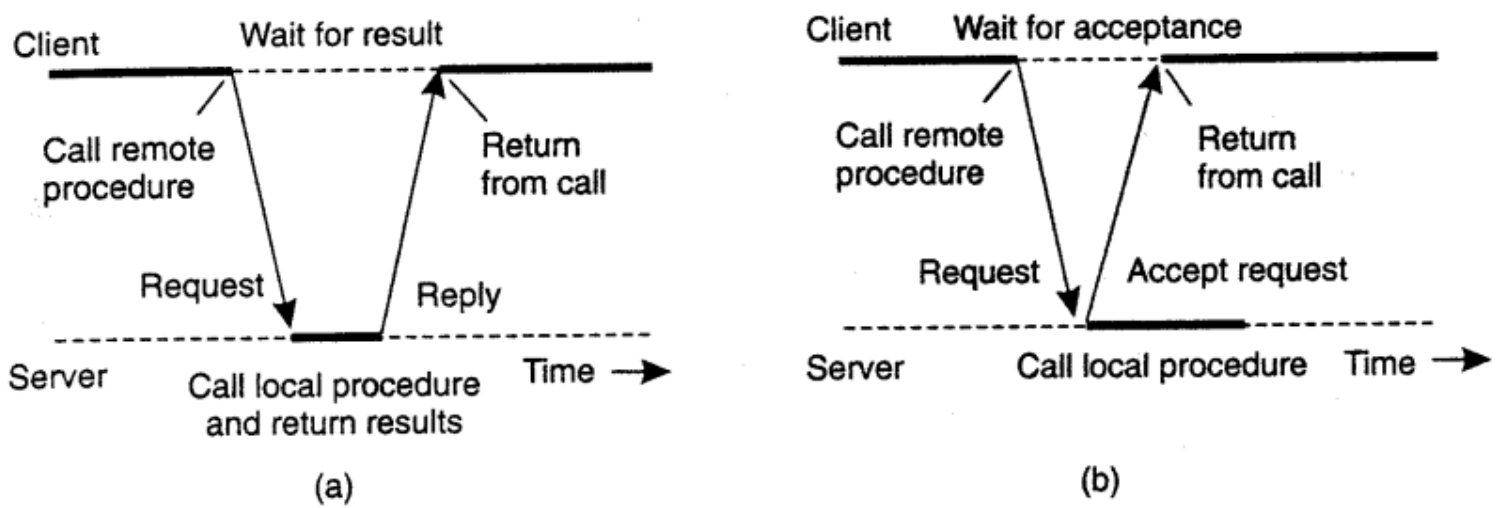
10. The stub unpacks the result and returns to the client.



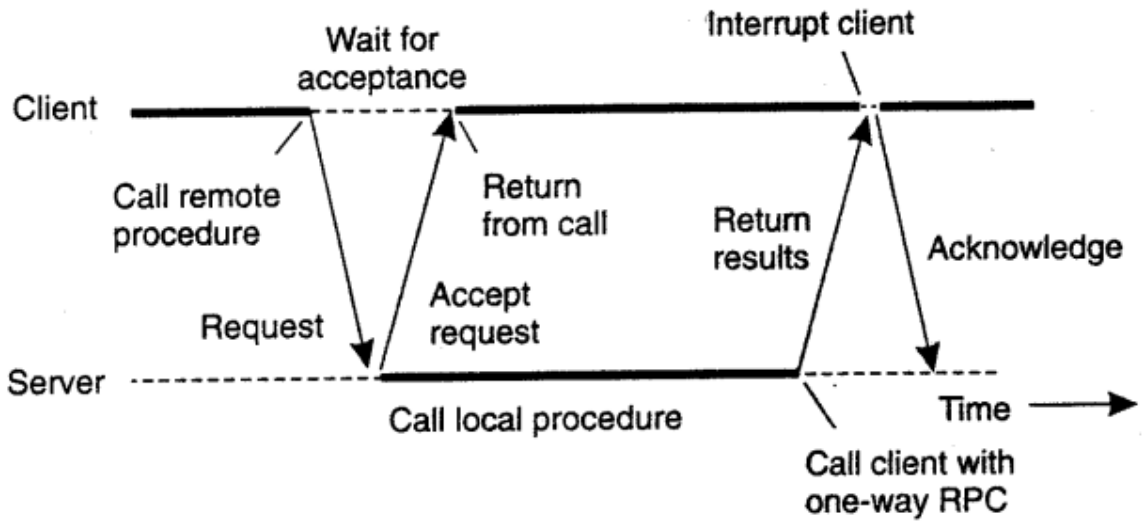
Asynchronous RPC

- As in conventional procedure calls, when a client calls a remote procedure, the client will block until a reply is returned.
- This strict request-reply behavior is unnecessary when there is no result to return, and only leads to blocking the client while it could have proceeded and have done useful work just after requesting the remote procedure to be called.
- Examples of where there is often no need to wait for a reply include:
  - transferring money from one account to another,
  - adding entries into a database,
  - starting remote services,
  - batch processing, and
  - so on.

- To support such situations, RPC systems may provide facilities for what are called asynchronous RPCs, by which a client immediately continues after issuing the RPC request.
- With asynchronous RPCs, the server immediately sends a reply back to the client the moment the RPC request is received, after which it calls the requested procedure.
- The reply acts as an acknowledgment to the client that the server is going to process the RPC.
- The client will continue without further blocking as soon as it has received the server's acknowledgment.



(a) The interaction between client and server in a traditional RPc.
(b) The interaction using asynchronous RPc.



A client and server interacting through two asynchronous RPCs.

MESSAGE-ORIENTED COMMUNICATION

- Remote procedure calls and remote object invocations contribute to hiding communication in distributed systems, that is, they enhance access transparency.
- Unfortunately, neither mechanism is always appropriate.
- In particular, when it cannot be assumed that the receiving side is executing at the time a request is issued, alternative communication services are needed.
- Likewise, the inherent synchronous nature of RPCs, by which a client is blocked until its request has been processed, sometimes needs to be replaced by something else.

- That something else is messaging.
- What we discuss here:
  - what exactly synchronous behavior is and what its implications are
  - messaging systems that assume that parties are executing at the time of communication.
  - examine message-queuing systems that allow processes to exchange information, even if the other party is not executing at the time communication is initiated.

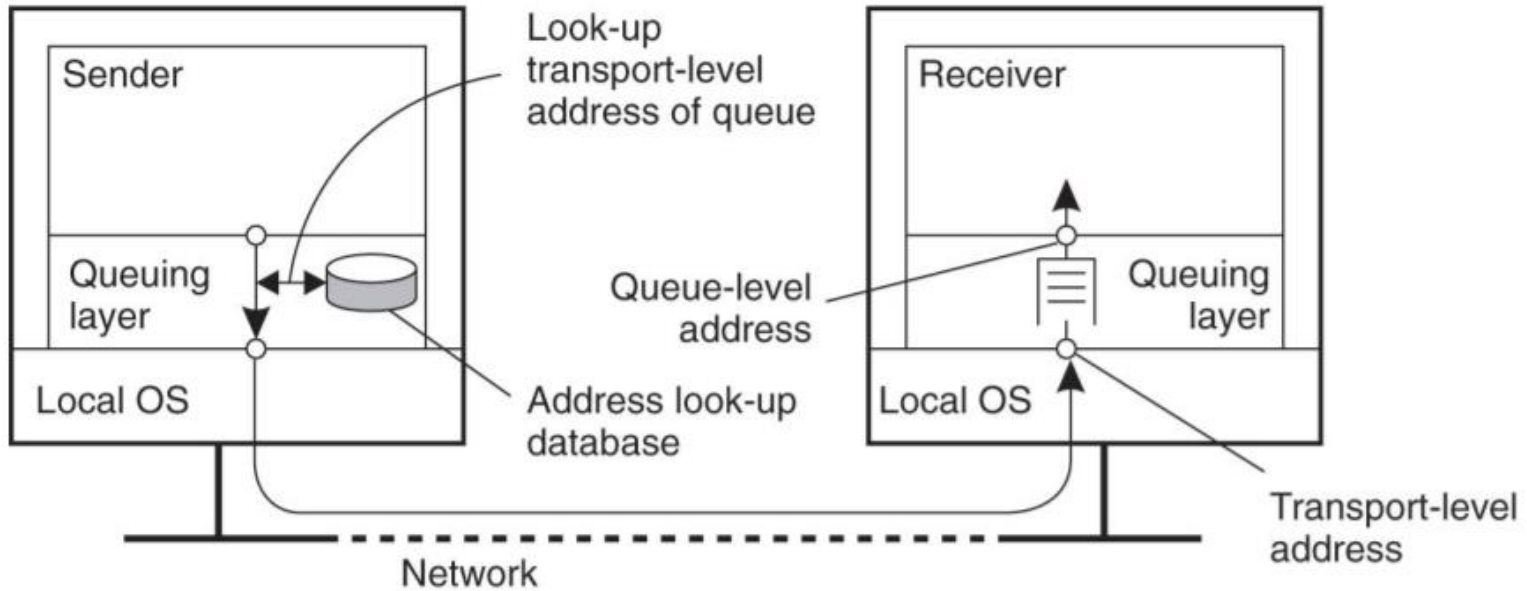
- Message-Oriented Transient Communication
  - Berkeley Socket Primitives, and
  - Message-Passing Interface (MPI)
- Message-Oriented Persistent Communication
  - Message Queuing Systems (MQS)

Message-Oriented Persistent Communication

- Message-queuing systems – a.k.a. Message-Oriented Middleware (MOM)
- Basic idea: MOM provides message storage service.
- A message is put in a queue by the sender, and delivered to a destination queue .
- The target(s) can retrieve their messages from the queue.
- Time uncoupling between sender and receiver
- Example: IBM's WebSphere

Time uncoupling: a sender can send a message even if the receiver is still not available. The message is stored and picked up at a later moment.

Time coupled interactions are observable when communication cannot take place unless both endpoints are operating at the same time.



General architecture of a message-queuing system

Primitive	Meaning
Put	Append a message to a specified queue
Get	Block until the specified queue is nonempty, and remove the first message
Poll	Check a specified queue for messages, and remove the first. Never block
Notify	Install a handler to be called when a message is put into the specified queue

Figure 4-18. Basic interface to a queue in a message-queuing system.

Message-Oriented Transient Communication

- Messages are sent through a channel abstraction.
- The channel connects two running processes.
- Time coupling between sender and receiver.
- Transmission time is measured in terms of milliseconds, typically
- Examples:
  - Berkeley Sockets — typical in TCP/IP-based networks
  - MPI (Message-Passing Interface) — typical in high-speed interconnection networks among parallel processes

Assignment-V (Deadline : feb 23)

- Explain the merits and demerits of RPC and also explain how Message Oriented communication overcome the demerits of RPC.

Berkeley Sockets

- Special attention has been paid to standardizing the interface of the transport layer to allow programmers to make use of its entire suite of (messaging) protocols through a simple set of primitives. Also, standard interfaces make it easier to port an application to a different machine.
- Conceptually, a socket is a communication end point to which an application can write data that are to be sent out over the underlying network, and from which incoming data can be read. A socket forms an abstraction over the actual communication end point that is used by the local operating system for a specific transport protocol.

- Servers generally execute the first four primitives, normally in the order given.
- When calling the socket primitive, the caller creates a new communication end point for a specific transport protocol.
- Internally, creating a communication end point means that the local operating system reserves resources to accommodate sending and receiving messages for the specified protocol.

Primitive	Meaning
Socket	Create a new communication end point
Bind	Attach a local address to a socket
Listen	Announce willingness to accept connections
Accept	Block caller until a connection request arrives
Connect	Actively attempt to establish a connection
Send	Send some data over the connection
Receive	Receive some data over the connection
Close	Release the connection

Figure 4-14. The socket primitives for TCPIIP.

a server should bind the IP address of its machine together with a (possibly well-known) port number to a socket. Binding tells the operating system that the server wants to receive messages only on the specified address and port.

- Let us now take a look at the client side. Here, too, a socket must first be created using the socket primitive, but explicitly binding the socket to a local address is not necessary, since the operating system can dynamically allocate a port when the connection is set up.
- The connect primitive requires that the caller specifies the transport-level address to which a connection request is to be sent.
- The client is blocked until a connection has been set up successfully, after which both sides can start exchanging information through the send and receive primitives.
- Finally, closing a connection is symmetric when using sockets, and is established by having both the client and server call the close primitive.

general pattern followed by a client and server for connection-oriented communication using sockets

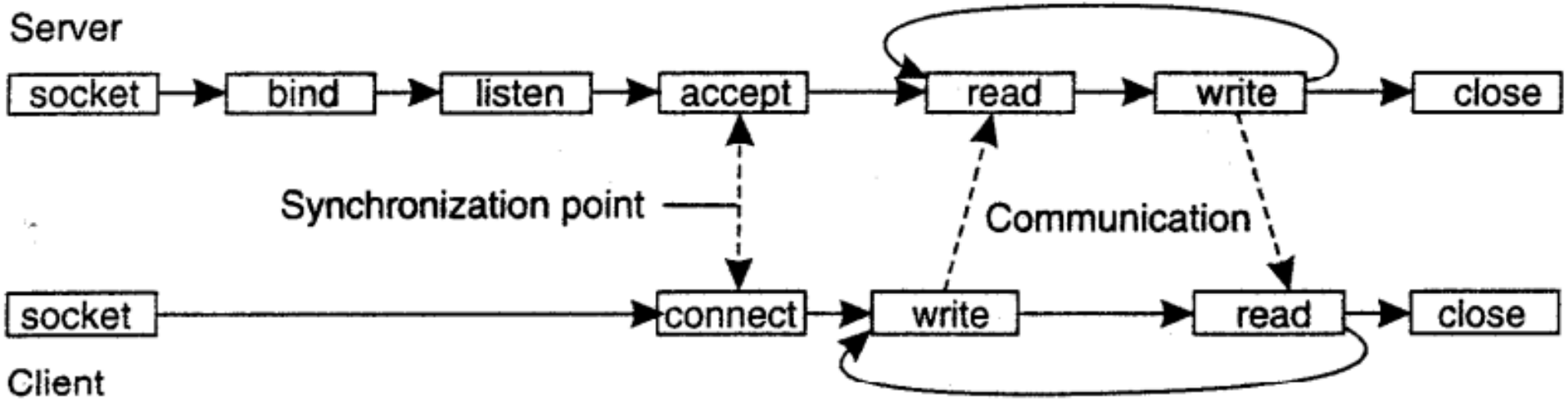


Figure 4-15. Connection-oriented communication pattern using sockets.

The Message-Passing Interface (MPI)

- Group of message-oriented primitives that would allow developers to easily write highly efficient applications.
- Sockets insufficient because:
  - – at the wrong level of abstraction supporting only send and receive primitives,
  - – designed to communicate using general-purpose protocol stacks such as TCP/IP, not suitable in high-speed interconnection networks, such as those used in COWs and MPPs (with different forms of buffering and synchronization).

MPI assumptions:

- – communication within a known group of processes,
- – each group with assigned id,
- – each process withing a group also with assigned id,
- – all serious failures (process crashes, network partitions) assumed as fatal and without any recovery,
- – a (groupID, processID) pair used to identify source and destination of the message,
- – only receipt-based transient synchronous communication (d) not supported, other supported.

The Message-Passing Interface (MPI)

Primitive	Meaning
MPI_bsend	Append outgoing message to a local send buffer
MPI_send	Send a message and wait until copied to local or remote buffer
MPI_ssend	Send a message and wait until receipt starts
MPI_sendrecv	Send a message and wait for reply
MPI_isend	Pass reference to outgoing message, and continue
MPI_issend	Pass reference to outgoing message, and wait until receipt starts
MPI_recv	Receive a message; block if there is none
MPI_irecv	Check if there is an incoming message, but do not block

Figure 4-16. Some of the most intuitive message-passing primitives of MPI.

STREAM-ORIENTED COMMUNICATION

Multicast Communication

- **Rationale**: Often need to a Send-to-Many in Distributed Systems

Examples:

- Financial services: Delivery of news, stock quotes etc.
- E-learning: Streaming content to many students at different levels.

Introduction

- forms of communication in which timing plays a crucial role,
- example:
  - an audio stream built up as a sequence of 16-bit samples each representing the amplitude of the sound wave as it is done through PCM (Pulse Code Modulation),
  - audio stream represents CD quality, i.e. 44100Hz,
  - samples to be played at intervals of exactly 1/44100,

- which facilities a distributed system should offer to exchange time-dependent information such as audio and video streams?
  - support for the exchange of time-dependent information = **support for continuous media,**
  - **continuous** (representation) media vs. **discrete** (representation) media.

Support for Continuous Media

- In continuous media :
  - temporal relationships between data items fundamental to correctly interpreting the data,
  - timing is crucial.

- **Asynchronous transmission mode**

Data items in a stream are transmitted one after the other, but there are no further timing constraints on when transmission of items should take place.

- **Synchronous transmission mode**

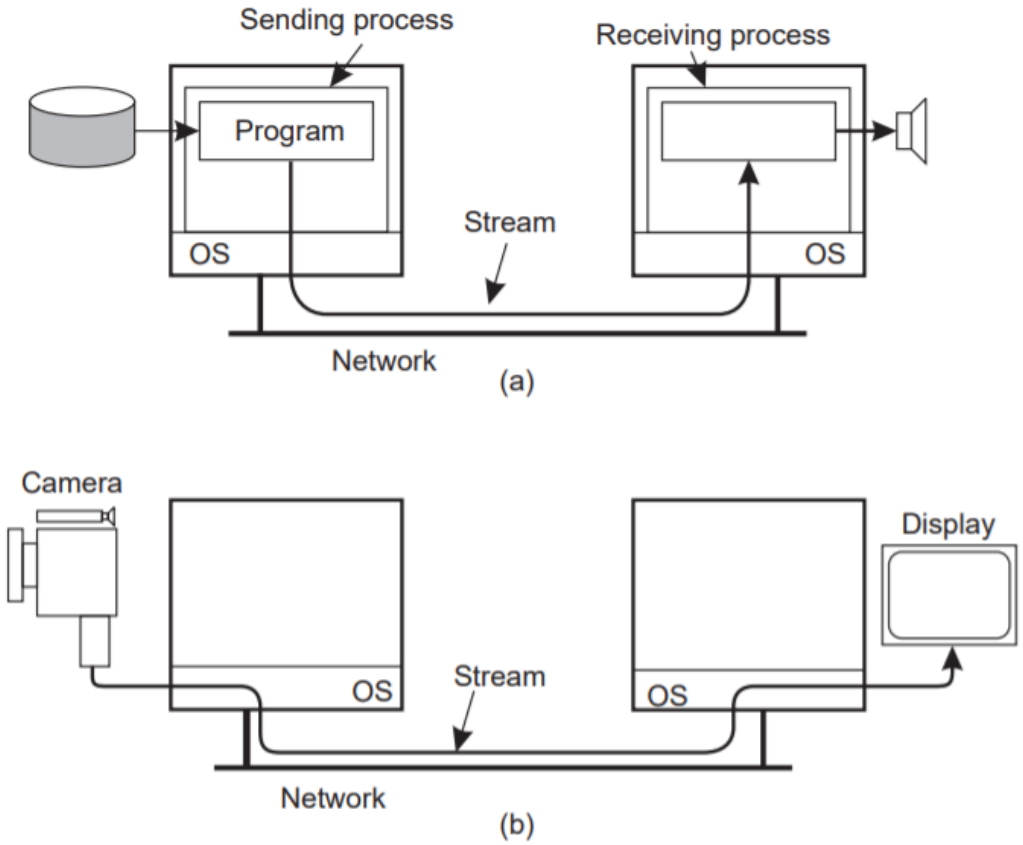
Maximum end-to-end delay defined for each unit in a data stream.

- **Isochronous transmission mode**

It is necessary that data units are transferred on time. Data transfer is subject to bounded (delay) jitter.

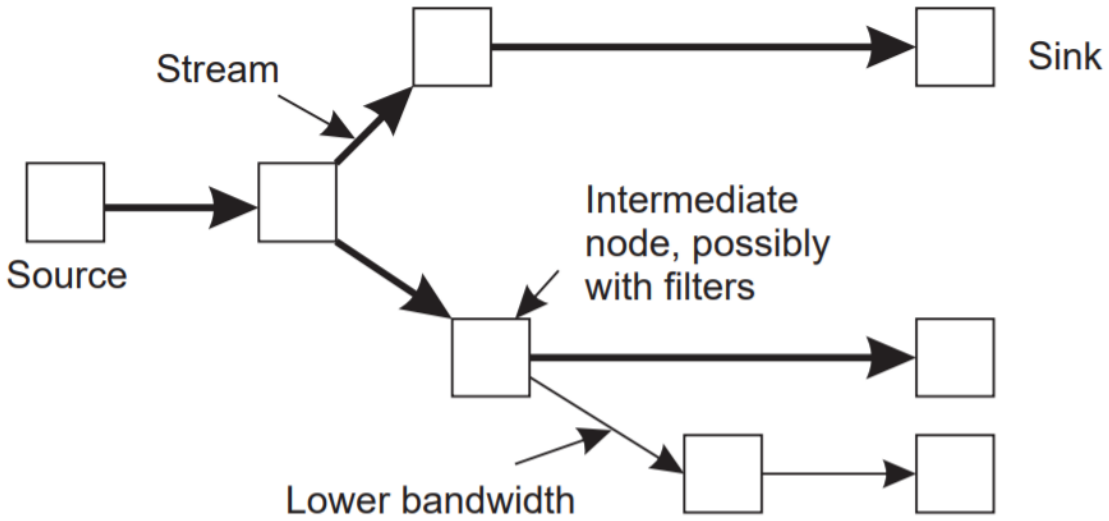
Data Stream

a. Setting up a stream between two processes across a network,
b. Setting up a stream directly between two devices.
   – stream sequence of data units, may be considered as a virtual connection between a source and a sink,
   – simple stream vs. complex stream (consisting of several related sub-streams).



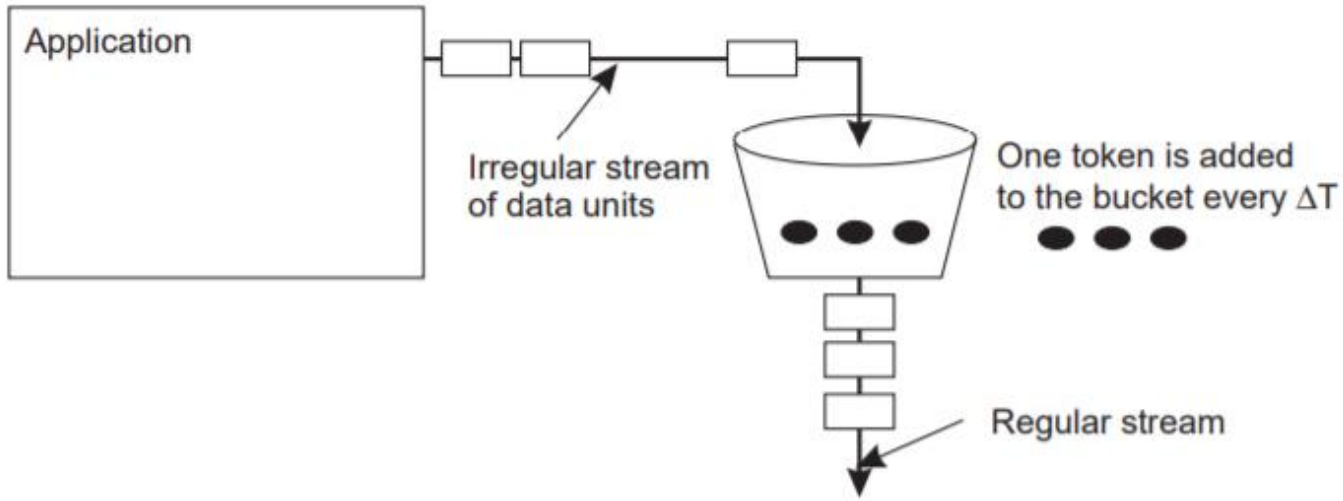
Data Stream

- An example of multicasting a stream to several receivers.
  - – problem with receivers having different requirements with respect to the quality of the stream,
  - – filters to adjust the quality of an incoming stream, differently for outgoing streams.



Specifying QoS: Token Bucket Algorithm

The principle of a token bucket algorithm.
– tokens generated at a constant rate,
– tokens buffered in a bucket which has limited capacity.



Summary

- Middleware enables much functionality in DS
- Especially the many types of interaction/communications necessary
- With rational reasons for every one!
  - Remote Procedure Call (RPC) enables transparency
  - But Message Queuing Systems necessary for persistent communications
    - IBM WebSphere is ok but a bit old, clunky & tired at this stage
    - AMQP open source, more flexible, better Industrial support
  - Multicast Communications are often necessary in DS

Unit-5
Naming

By Prashant Gautam

BCA 6th Semester

Outlines

- Name, Identifiers and Address
- Structured Naming
- Attribute-Based Naming
- Case Study: The Global Name service

Names, Identifiers, and Addresses

- A name in a distributed system is a string of bits or characters that is used to refer to an entity.
- An entity in a distributed system can be practically anything.
  - Typical examples include resources such as hosts, printers, disks, and files.
  - Other well-known examples of entities that are often explicitly named are processes, users, mailboxes, newsgroups, Web pages, graphical windows, messages, network connections, and so on.

For example, a resource such as a printer offers an interface containing operations for printing a document, requesting the status of a print job, and the like. Furthermore, an entity such as a network connection may provide operations for sending and receiving data, setting quality-of-service parameters, requesting the status, and so forth.

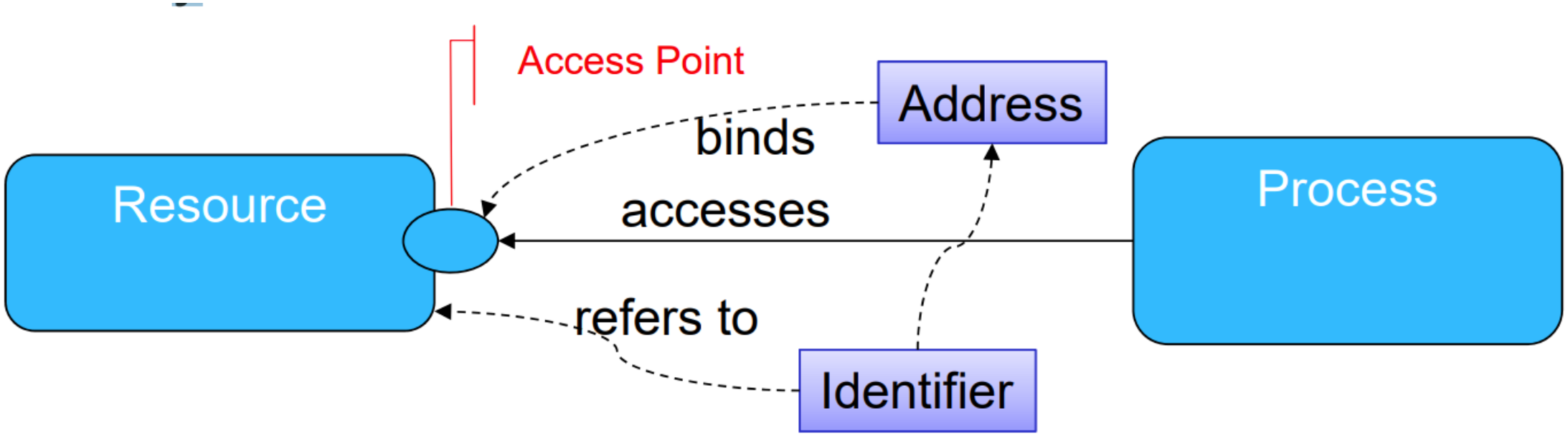
- To operate on an entity, it is necessary to access it, for which we need an access point.
- An access point is yet another, but special, kind of entity in a distributed system.
- The name of an access point is called an address.
- The address of an access point of an entity is also simply called an address of that entity.
- An entity can offer more than one access point.

- As a comparison, a telephone can be viewed as an access point of a person, whereas the telephone number corresponds to an address. Indeed, many people nowadays have several telephone numbers, each number corresponding to a point where they can be reached.

In a distributed system, a typical example of an access point is a host running a specific server, with its address formed by the combination of, for example, an IP address and port number (i.e., the server's transport-level address).

- An entity may change its access points in the course of time.
- For example: when a mobile computer moves to another location, it is often assigned a different IP address than the one it had before. Likewise, when a person moves to another city or country, it is often necessary to change telephone numbers as well.
- In a similar fashion, changing jobs or Internet Service Providers, means changing your e-mail address.

- Identifier: a name that uniquely identifies an entity
  - the identifier is unique and refers to only one entity
- Address: the name of an access point, the location of an entity



3 Classes of naming systems

1. FLAT NAMING / UNSTRUCTURED NAMING
2. STRUCTURED NAMING
3. ATTRIBUTE-BASED NAMING

FLAT NAMING / UNSTRUCTURED NAMING

- Identifiers are convenient to uniquely represent entities.
- In many cases, identifiers are simply random bit strings. which we conveniently refer to as unstructured, or flat names.
- An important property of such a name is that it does not contain any information whatsoever on how to locate the access point of its associated entity.
- We will study these aspects:

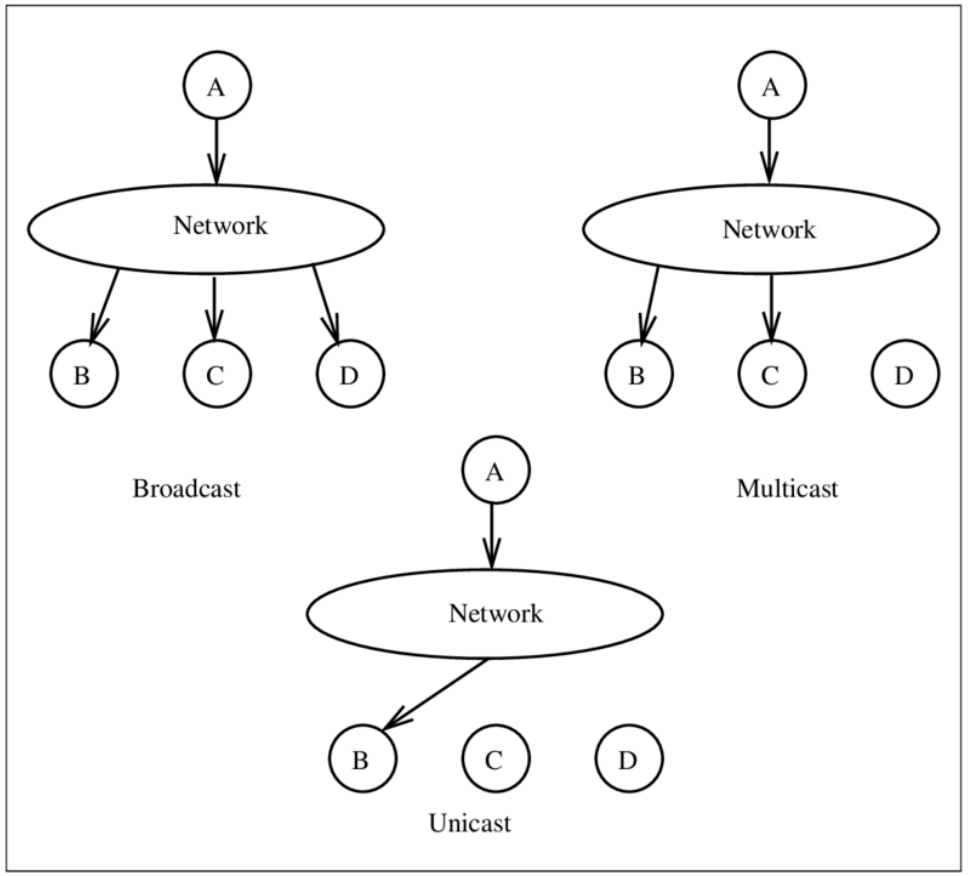
  How flat names can be resolved ?

  How we can locate an entity when given only its identifier ?

FLAT NAMING / UNSTRUCTURED NAMING

- Simple Solution for locating an entity
  - Broadcasting and Multicasting
  - Forwarding Pointers
- Both solutions are applicable only to local-area networks.
- But they often do the job well, making their simplicity particularly attractive.

Broadcasting and Multicasting



BROADCAST VERSUS MULTICAST

BROADCAST	MULTICAST
A method of transferring a message to all recipients simultaneously	A group communication where data transmission is addressed to a group of destination computers simultaneously
Packets are transmitted to all the connected devices in the network	Packets are transmitted to some of the devices in the network
There is no need for group management	Requires group management
Less secure	More secure
More traffic	Less traffic
Slower	Faster

- Consider a distributed system built on a computer network: that offers efficient broadcasting facilities.
- Typically, such facilities are offered by local-area networks and LAN wireless in which all machines are connected to a single cable or equivalent.
- Locating an entity in such an environment is simple: a message containing the identifier of the entity is broadcast to each machine and each machine is requested to check whether it has that entity.
- Only the machines that can offer an access point for the entity send a reply message containing the address of that access point.

Example:

- a machine broadcasts a packet on the local network asking who is the owner of a given IP address.
- When the message arrives at a machine, the receiver checks whether it should listen to the requested IP address.
- If so, it sends a reply packet containing, for example, its Ethernet address.

Problems with Broadcasting:

- Broadcasting becomes inefficient when the network grows.
- Not only is network bandwidth wasted by request messages, but, more seriously, too many hosts maybe interrupted by requests they cannot answer.
- One possible solution is to switch to multicasting, by which only a restricted group of hosts receives the request.
- For example, Ethernet networks support data-link level multicasting directly in hardware.

- Multicasting can also be used to locate entities in point-to-point networks.
- For example, the Internet supports network-level multicasting by allowing hosts to join a specific multicast group.
- Such groups are identified by a multicast address.
- When a host sends a message to a multicast address, the network layer provides a best-effort service to deliver that message to all group members.

- A multicast address can be used as a general location service for multiple entities.
- For example, consider an organization where each employee has his or her own mobile computer.
- When such a computer connects to the locally available network. it is dynamically assigned an IP address.
- In addition, it joins a specific multicast group. When a process wants to locate computer A, it sends a "where is A?" request to the multicast group.
- If A is connected, it responds with its current IP address.

Assignment-VI (Deadline: Feb 28)

- Explain the merits and demerits of broadcasting and multiple casting in locating entities.

Forwarding Pointers

- The principle is simple: when an entity moves from A to B, it leaves behind in A a reference to its new location at B.
- Advantages:
  - simplicity: as soon as an entity has been located, for example by using a traditional naming service, a client can look up the current address by following the chain of forwarding pointer
- Drawbacks
  - a chain for a highly mobile entity can become so long that locating that entity is prohibitively expensive.
  - the vulnerability to broken links

Explore more on Forward pointers

STRUCTURED NAMING

- Flat names:
  - good for machines, but are generally not very convenient for humans to use.

- structured names:
  - composed from simple, human-readable names.

In this section, we concentrate on structured names and the way that these names are resolved to addresses.

Name Spaces

- Names are commonly organized into what is called a name space.
- Name spaces for structured names can be represented as a labeled, directed graph with two types of nodes.

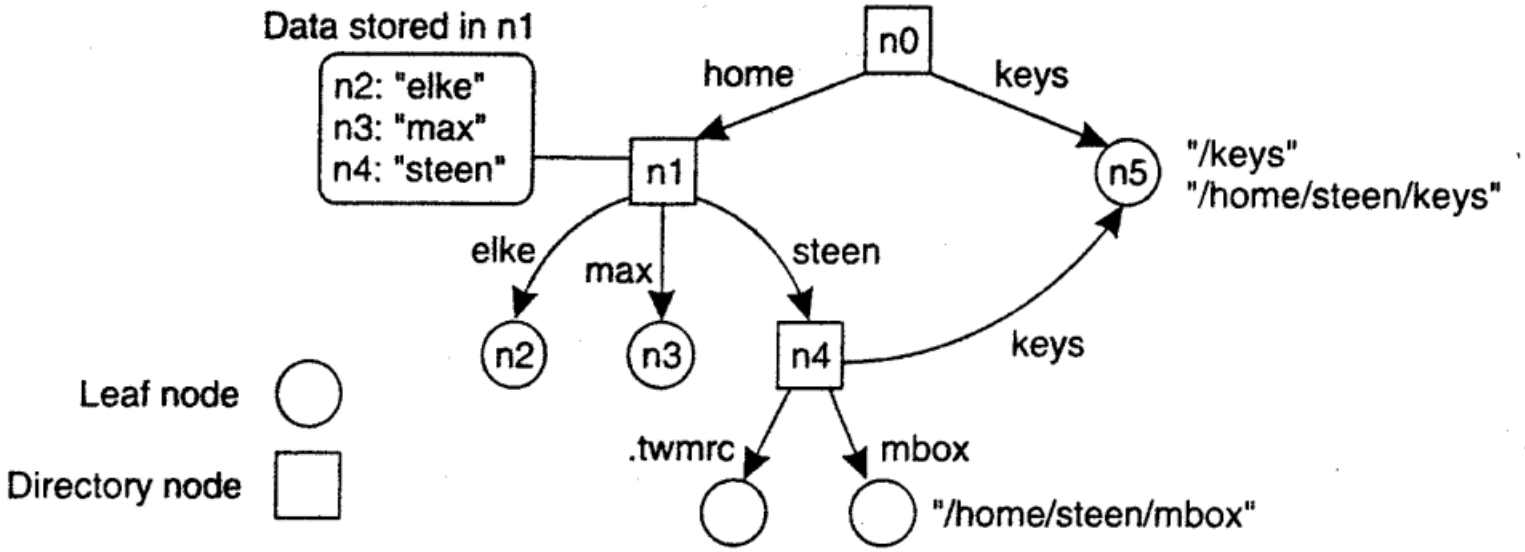


Figure 5-9. A general naming graph with a single root node.

Name Spaces

- **A leaf node** represents a named entity and has the property that it has no outgoing edges.
- **A leaf node** generally stores information on the entity it is representing-for example, its address-so that a client can access it.
- Alternatively, it can store the state of that entity, such as in the case of file systems 'in which a leaf node actually contains the complete file it is representing.
- **A directory node** has a number of outgoing edges, each labeled with a name
- **A directory node** stores a table in which an outgoing edge is represented as a pair (edge label, node identifier). Such a table is called a directory table.

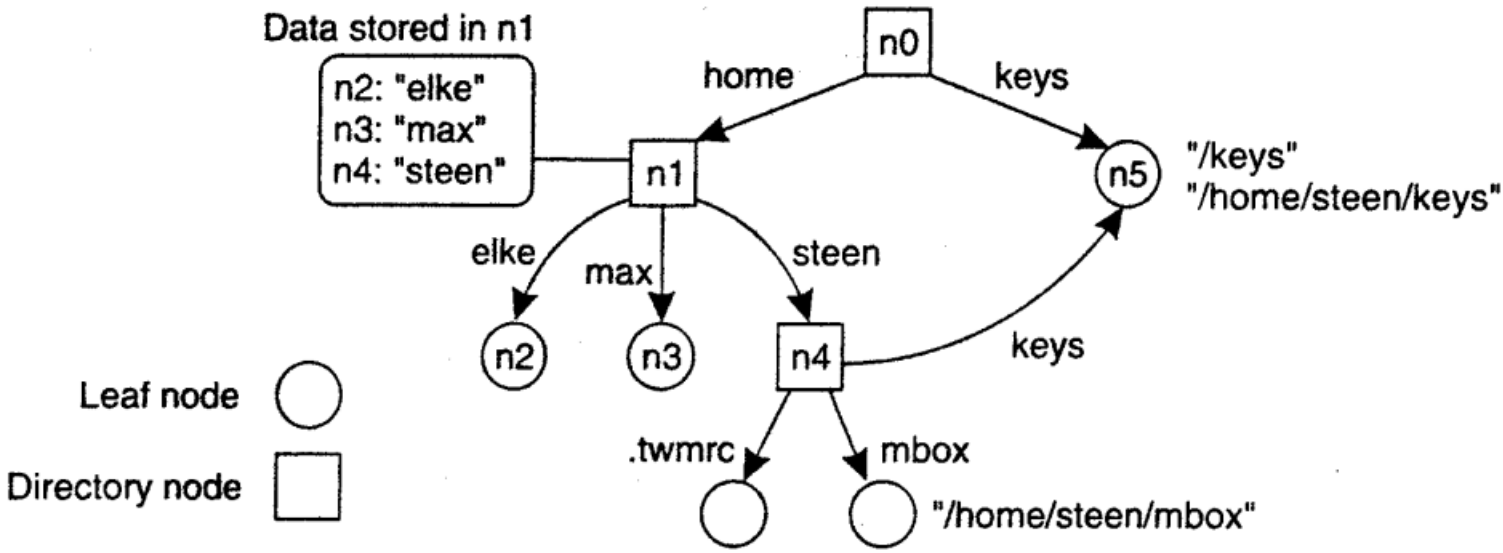


Figure 5-9. A general naming graph with a single root node.

Name Resolution

- Name spaces offer a convenient mechanism for storing and retrieving information about entities by means of names.
- More generally, given a path name, it should be possible to look up any information stored in the node referred to by that name.
- The process of looking up a name is called **name resolution**.

DNS

- We use to remember “human-readable“ machine name
  - we have the name hierarchy
  - E.g., www.facebook.com
- But machines in Internet use IP address
  - E.g., 31.13.84.33
  - Application communication use IP addresses and ports
- DNS
  - Mapping from the domain name hierarchy to IP addresses

Information in records of DNS namespace

Type of record	Associated entity	Description
SOA	Zone	Holds information on the represented zone
A	Host	Contains an IP address of the host this node represents
MX	Domain	Refers to a mail server to handle mail addressed to this node
SRV	Domain	Refers to a server handling a specific service
NS	Zone	Refers to a name server that implements the represented zone
CNAME	Node	Symbolic link with the primary name of the represented node
PTR	Host	Contains the canonical name of a host
HINFO	Host	Holds information on the host this node represents
TXT	Any kind	Contains any entity-specific information considered useful

Source: Andrew S. Tanenbaum and Maarten van Steen, Distributed Systems – Principles and Paradigms, 2nd Edition, 2007, Prentice-Hall

ATTRIBUTE-BASED NAMING

- A tuple (attribute, value) can be used to describe a property
  - E.g., („country“, “Austria“), („language“, „German“)
- A set of tuples (attribute, value) can be used to describe an entity

AustriaInfo

Attribute	Value
CountryName	Austria
Language	German
MemberofEU	Yes
Capital	Vienna

- Employ (attribute, value) tuples for describing entities
  - Why flat and structured naming are not enough?
- Also called directory services.
- Name Resolution
  - Usually based on querying mechanism
  - Querying usually deal with the whole space
- Implementation
  - LDAP (Lightweight Directory Access Protocol)
  - RDF (Resource Description Framework): Framework for semantics

Assignment-VII (Deadline: Mar 3)

- Explore more about LDAP, how it works, considering real world example.
- Explore more about RDF, how it works, considering real world example.