

Fault Tolerance

Topics covered in this chapter

- Introduction to faults
- Failure models
- Failure masking
- Process Resilience
- Agreement in Faulty system
- Reliable Client server communication
- Group communication
- Distributed commit
- Recovery

Fault Tolerance

- A system or a component fails due to a fault
- Fault tolerance means that the system continues to provide its services in presence of faults
- A distributed system may experience and should recover also from partial failures
- Includes:
 - preventing faults and failures from affecting other components of the system,
 - automatically recovering from partial failures, and doing so without seriously affecting performance

Basics Concepts

Dependability is the ability to avoid service failures that are more frequent. A key requirement of most systems is to provide some level of dependability. A dependable systems has the following properties.

- **Availability:** system is ready to be used immediately
- **Reliability:** system can run continuously without failure
- **Safety:** when a system (temporarily) fails to operate correctly, nothing catastrophic /dangerous happens
- **Maintainability:** how easily a failed system can be repaired

Basics Concepts

- A system is said to “fail” when it cannot meet its promises.
- A failure is brought about by the existence of “errors” in the system.
- The cause of an error is a “fault”.

Types of Fault

There are three main types of ‘fault’:

1. Transient Fault – appears once, then disappears.
2. Intermittent Fault – occurs, vanishes, reappears; but: follows no real pattern (worst kind).
3. Permanent Fault – once it occurs, only the replacement/repair of a faulty component will allow the DS to function normally.

Failure Models

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

Failure Masking by Redundancy

Strategy: hide the occurrence of failure from other processes using redundancy.

Three main types:

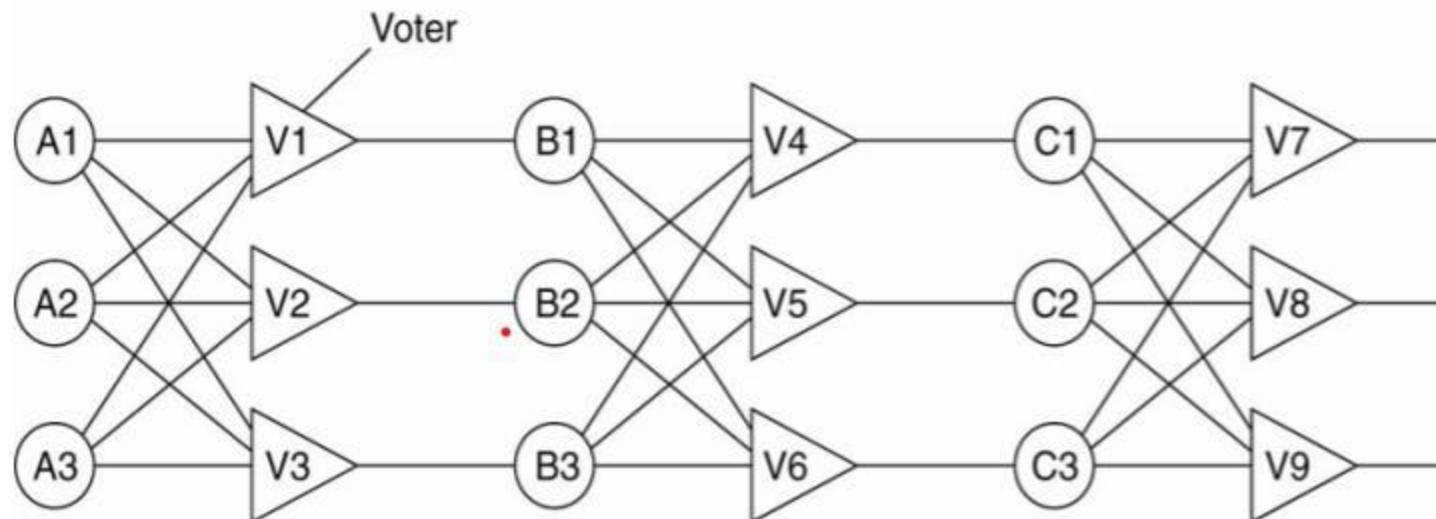
1. Information Redundancy – add extra bits to allow for error detection/recovery (e.g., Hamming codes and the like).
2. Time Redundancy – perform operation and, if needs be, perform it again. Think about how transactions work (BEGIN/END/COMMIT/ABORT).
3. Physical Redundancy – add extra (duplicate) hardware and/or software to the system

Failure Masking by Redundancy

Triple modular redundancy



(a)



(b)

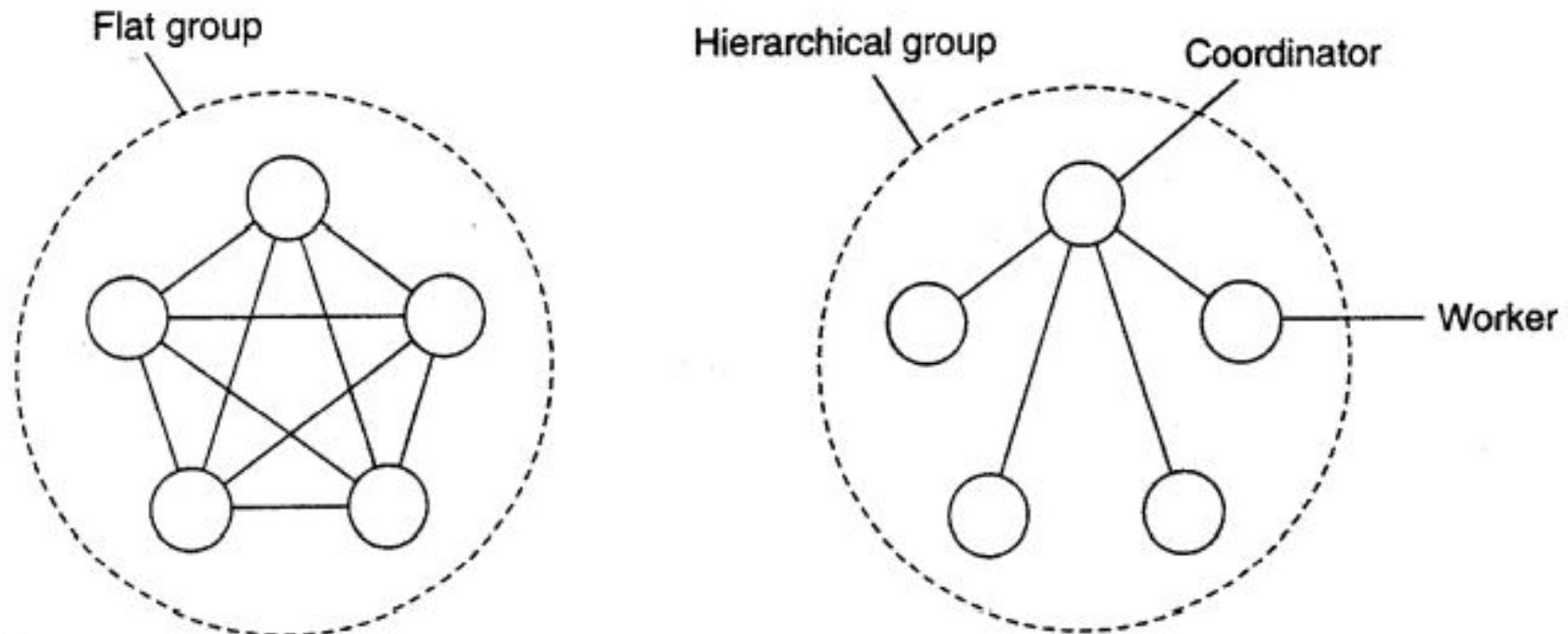
Process Resilience

- Protection against process failures – achieved by replicating processes into groups
- Groups:
 - Organize identical processes into groups
 - Process groups are dynamic
 - Processes can be members of multiple groups
 - Mechanisms are needed for managing groups and group membership

Process Resilience

- Flat vs Hierarchical Groups:
 - Flat group: all decisions made collectively
 - There is no single point of failure
 - Decision making is difficult
 - Hierarchical group: coordinator makes decisions
 - Decision making process is much simpler
 - Single point of failure.

Process Resilience



Flat Groups versus Hierarchical Groups

- Replicate processes and organize them into a group to replace a vulnerable process with a fault tolerant group
- There are two ways to approach such replication:
 1. Primary (backup) Protocols(kind of client-server arch.):
 - primary coordinates all write operations.
 - Its role can be taken over by one of the backup, if need be.
 - When the primary crashes, the backups execute some election algorithm to choose a new primary
 2. Replicated-Write Protocols :
 - organizing identical processes into a flat group.
 - No single point of failure.

Group Membership

- When group communication is present, some method is needed for creating and deleting groups, as well as for allowing processes to join and leave groups.
 - Group server
 - Distributed method

Thank You

Agreement in faulty systems

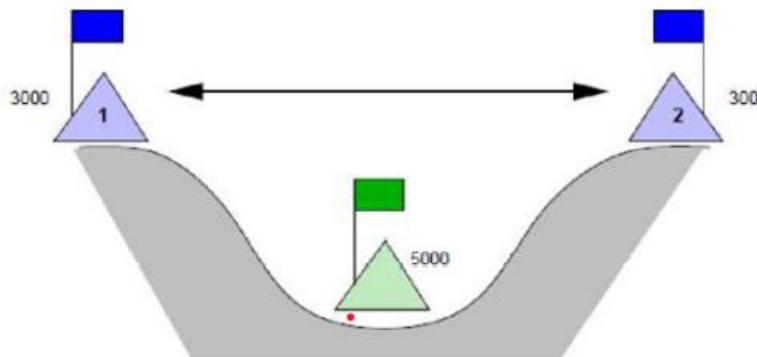
- Examples: Election, transaction commit/abort, dividing tasks among workers, mutual exclusion
 - What happens when processes can fail?
 - What happens when communication can fail?
 - What happens when byzantine failures are possible
- We want all non-faulty processes to reach and establish agreement (within a finite number of steps)

The Goal of Agreement Algorithms

- “To have all non-faulty processes reach consensus on some issue (quickly).”
- The two-army problem -with non-faulty processes, agreement between even two processes is not possible in the face of unreliable communication.
- Byzantine generals problem – communication is perfect but the processes are not

Tow Army Problem

Non-faulty processes but lossy communication.



- 1 → 2 attack!
- 2 → 1 ack
- 2: did 1 get my ack?
- 1 → 2 ack ack
- 1: did 2 get my ack ack?
- etc.

Byzantine Generals Problem

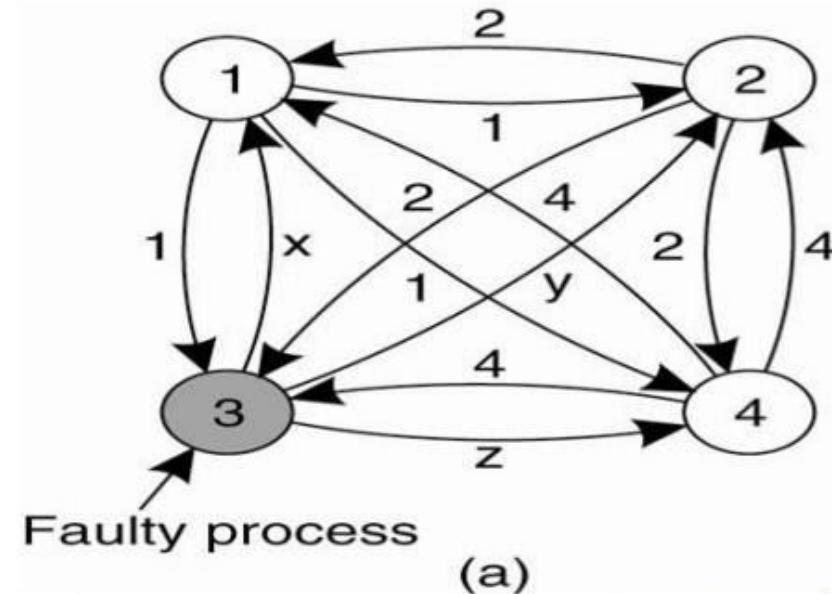
- Time: 330-1453 AD.
- Place: Balkans and Modern Turkey.
- Endless conspiracies, intrigue, and untruthfulness were alleged to be common practice in the ruling circles of the day (sounds strangely familiar ...).
- That is: it was typical for intentionally wrong and malicious activity to occur among the ruling group. A similar occurrence can surface in a DS, and is known as ‘Byzantine failure’.
- Question: how do we deal with such malicious group members within a distributed system?

Agreement in faulty systems

Recursive algorithm was devised by Lamport et. al in 1982

- Step 1: every general sends a (reliable) message to every other general announcing his troop strength
- Step 2: results of announcements of step 1 are collected together in the form of the vectors
- Step 3: every general passing vectors to other generals
- Step 4: each general examines the i th element of each of the newly received vectors. If no majority, corresponding element will be considered to be UNKNOWN

Agreement in faulty systems



The Byzantine agreement problem for three non-faulty and one faulty process. (a) Each process sends their value to the others.

Agreement in faulty systems

1 Got(1, 2, x, 4)
2 Got(1, 2, y, 4)
3 Got(1, 2, 3, 4)
4 Got(1, 2, z, 4)

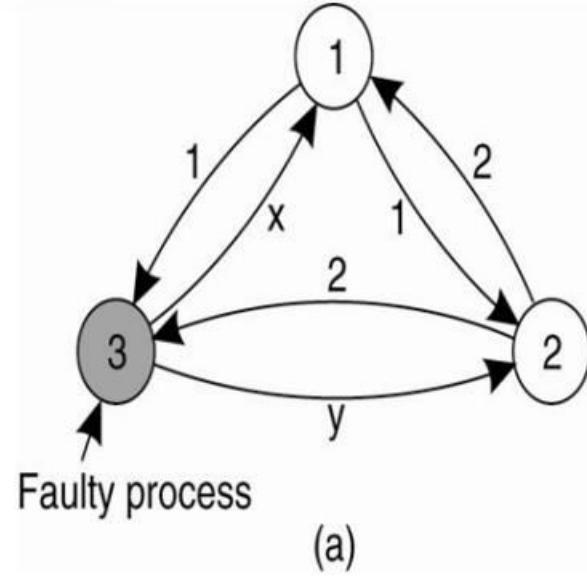
1 Got	2 Got	4 Got
(1, 2, y, 4)	(1, 2, x, 4)	(1, 2, x, 4)
(a, b, c, d)	(e, f, g, h)	(1, 2, y, 4)
(1, 2, z, 4)	(1, 2, z, 4)	(i, j, k, l)

(b)

(c)

- (b) The vectors that each process assembles based on (a).
(c) The vectors that each process receives in step 3

Agreement in faulty systems



```
1 Got(1, 2, x )
2 Got(1, 2, y )
3 Got(1, 2, 3)
```

(b)

<u>1 Got</u> (1, 2, y) (a, b, c)	<u>2 Got</u> (1, 2, x) (d, e, f)
--	--

(c)

The same as before, except now with two correct process and one faulty process

Agreement in faulty systems

- Lamport et.al proved that in a system with m faulty processes, agreement can be achieved only if $2m+1$ correctly functioning processes are present, for a total of $3m+1$ processes.
- Other way to say – agreement is possible only if more than two thirds of the processes are working properly.
- Fischer et.al(1985) proved that agreement becomes worse – slow processes are indistinguishable from crashed ones

Agreement in faulty systems

		Message ordering				Communication delay
		Unordered		Ordered		
Process behavior	Synchronous	X	X	X	X	Bounded
	Asynchronous			X	X	Unbounded
		Unicast	Multicast	Unicast	Multicast	Bounded
					X	Unbounded
					X	
		Message transmission				

Circumstances under which distributed agreement can be reached

Agreement in faulty systems

Agreement in Faulty Systems

Possible cases:

1. Synchronous (lock-step) versus asynchronous systems.
2. Communication delay is bounded (by globally and predetermined maximum time) or not.
3. Message delivery is ordered (in real-time) or not.
4. Message transmission is done through unicasting or multicasting.

Agreement in faulty systems

		Message ordering				Communication delay
		Unordered		Ordered		
Process behavior		X	X	X	X	Bounded
				X	X	Unbounded
Asynchronous					X	Bounded
					X	Unbounded
		Unicast	Multicast	Unicast	Multicast	Message transmission

Circumstances under which distributed agreement can be reached

Reliable Client/Server Communications

- Communication channel may exhibit crash, omission, timing, and arbitrary failures
- Focus is on masking crash and omission failures.
- Example: In DS, reliable point-to-point communication is established using TCP which masks omission failures by guarding against lost messages using ACKs and retransmissions.
- It performs poorly when a crash occurs (DS may try to mask a TCP crash by automatically re-establishing the lost connection).

Reliable Client/Server Communications

The RPC mechanism works well as long as both the client and server function perfectly.

Five classes of RPC failure can be identified:

1. The client cannot locate the server
2. The client's request to the server is lost
3. The server crashes after receiving the request
4. The server's reply is lost on its way to the client
5. The client crashes after sending its request

Reliable Client/Server Communications

(1) Client cannot locate the server

- An appropriate exception handling mechanism can deal with a missing server.
- An appropriate exception handling mechanism can deal with a missing server.

Reliable Client/Server Communications

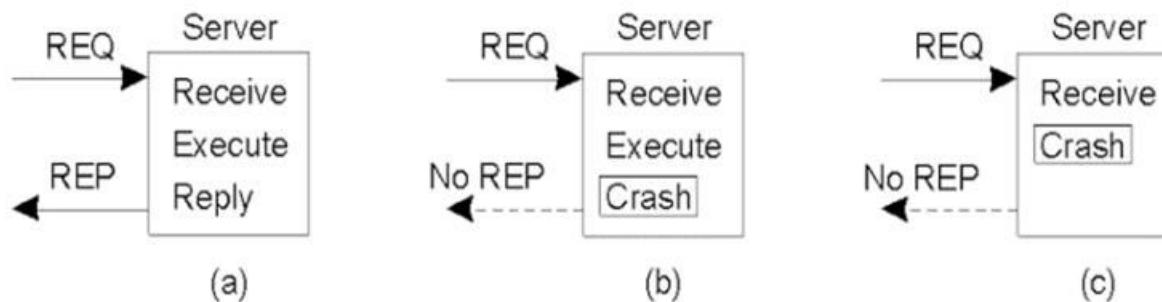
(2) Lost request message

- It can be dealt with easily using timeouts.
- If no ACK arrives in time, the message is resent.
- Server needs to be able to deal with the possibility of duplicate requests

Reliable Client/Server Communications

(3) Server crashes

- a) The normal case.
- b) Crash after service execution.
- c) Crash before service execution



Reliable Client/Server Communications

Server crashes are dealt with by implementing one of three possible implementation philosophies:

1. At least once semantics: keep trying until a reply is received. Guarantee is given that the RPC occurred at least once, but (also) possibly more than once.
2. At most once semantics: gives up immediately and reports back failure. Guarantee is given that the RPC occurred at most once, but possibly not at all.
3. No semantics: When a server crashes, client gets no indication. Nothing is guaranteed, and client and servers take their chances. Easy to implement

It has proved difficult to provide exactly once semantics

Reliable Client/Server Communications

Remote operation: print some text and (when done) send a completion message.

Three events that can happen at the server:

1. Send the completion message (M)
2. Print the text (P)
3. Crash (C)

Reliable Client/Server Communications

These three events can occur in six different orderings:

1. $M \rightarrow P \rightarrow C$: A crash occurs after sending the completion message and printing the text.
2. $M \rightarrow C (\rightarrow P)$: A crash happens after sending the completion message, but before the text could be printed.
3. $P \rightarrow M \rightarrow C$: A crash occurs after sending the completion message and printing the text.
4. $P \rightarrow C (\rightarrow M)$: The text printed, after which a crash occurs before the completion message could be sent.
5. $C (\rightarrow P \rightarrow M)$: A crash happens before the server could do anything.
6. $C (\rightarrow M \rightarrow P)$: A crash happens before the server could do anything. –

Parentheses indicate an event that can no longer happen because the server already crashed

Reliable Client/Server Communications

Client Reissue strategy	Strategy M → P			Strategy P → M		
	MPC	MC(P)	C(MP)	PMC	PC(M)	C(PM)
Always	DUP	OK	OK	DUP	DUP	OK
Never	OK	ZERO	ZERO	OK	OK	ZERO
Only when ACKed	DUP	OK	ZERO	DUP	OK	ZERO
Only when not ACKed	OK	ZERO	OK	OK	DUP	OK

OK = Text is printed once
DUP = Text is printed twice
ZERO = Text is not printed at all

Different combinations of client and server strategies in the presence of server crashes

Reliable Client/Server Communications

(4) Lost reply messages

- Why was there no reply? Is the server dead, slow, or did the reply just go missing?
- A request that can be repeated any number of times without any side-effects is said to be idempotent. (Example: a read of a static web-page is said to be idempotent)
 - Nonidempotent requests (for example, the electronic transfer of funds) are a little harder to deal with.
 - A common solution is to employ unique sequence numbers. More work on server side and reply to clients..
 - Another technique is the inclusion of additional bits in a retransmission to identify it as such to the server

Reliable Client/Server Communications

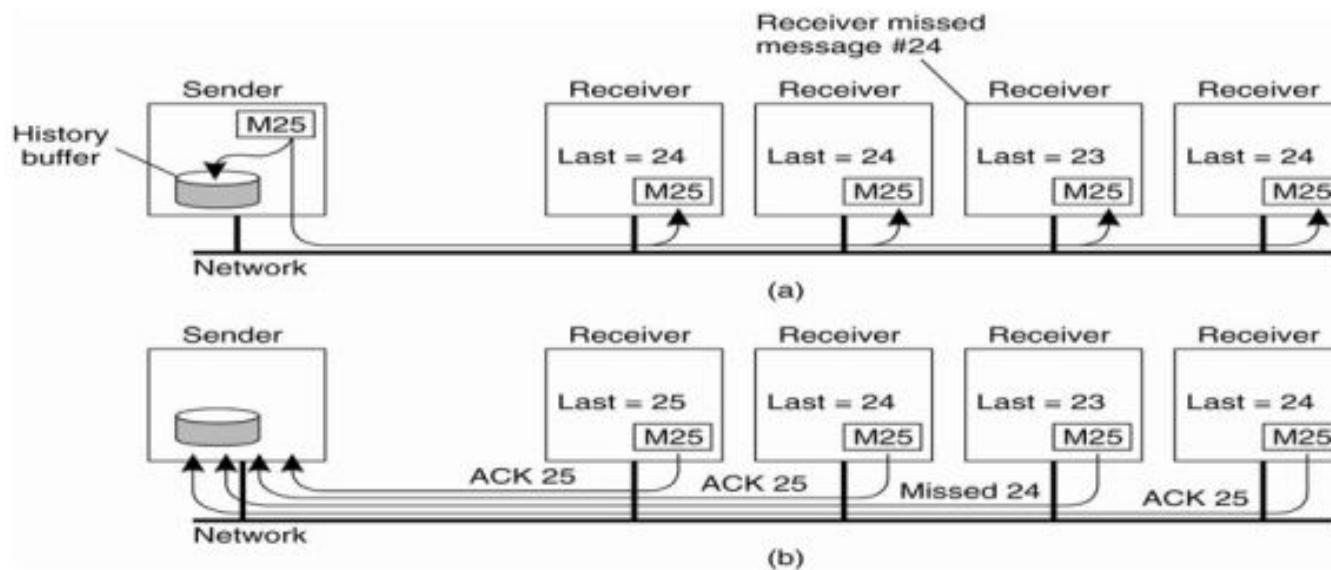
(5) Client crashes after request

- When an ‘old’ reply arrives, such a unwanted computation/reply is known as an orphan.
- Problems – waste CPU cycles, client reboot makes confusion with old replies...
- Four orphan solutions have been proposed by Nelson(1981):
 1. extermination (the orphan is simply killed-off after checking log).
 2. reincarnation (each client session has an epoch associated with it, making orphans easy to spot and obsolete).
 3. gentle reincarnation (when a new epoch is identified, an attempt is made to locate a requests owner, otherwise the orphan is killed).
 4. expiration (if the RPC cannot be completed within a standard amount of time, it is assumed to have expired).
- In practice, however, none of these methods are desirable for dealing with orphans. Orphans may have locks on files/data

Reliable Group Communication

- Reliable multicast services guarantee that all messages are delivered to all members of a process group.
- But it is surprisingly tricky (as multicasting services tend to be inherently unreliable).
- For a small group, multiple, reliable point-to-point channels will do the job, however, such a solution scales poorly as the group membership grows. Also:
 - What happens if a process joins the group during communication?
 - Worse: what happens if the sender of the multiple, reliable point-to-point channels crashes half way through sending the messages?

Basic Reliable-Multicasting Schemes



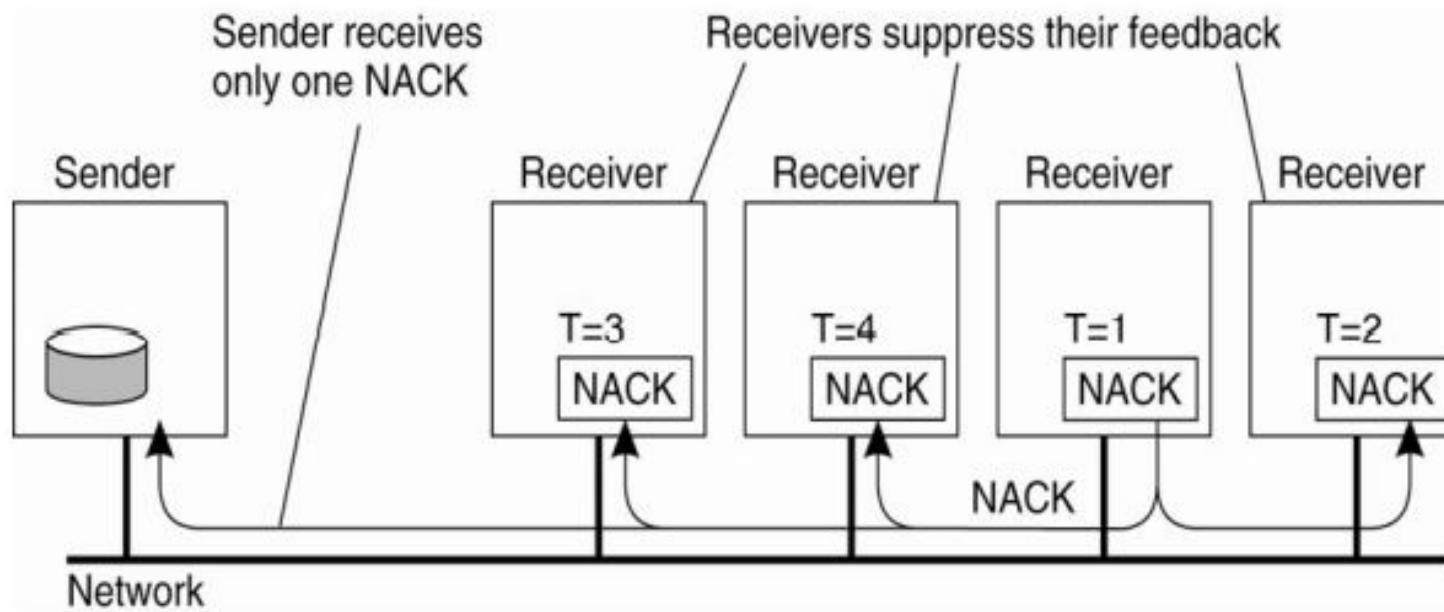
A simple solution to reliable multicasting when all receivers are known and are assumed not to fail.

(a) Message transmission. (b) Reporting feedback.

SRM: Scalable Reliable Multicasting

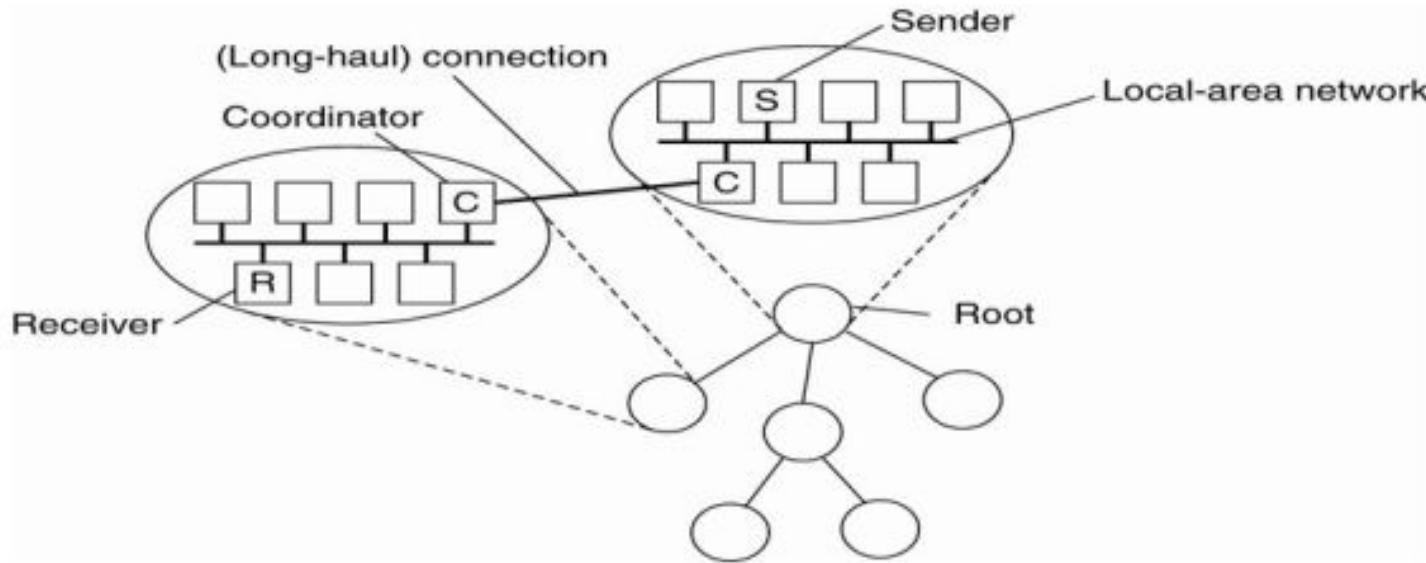
- Based on feedback suppression
- Receivers never acknowledge successful delivery.
- Only missing messages are reported.
- NACKs(negative ack) are multicast to all group members.
- This allows other members to suppress their feedback, if necessary.
- To avoid “retransmission clashes”, each member is required to wait a random delay prior to NACKing

Non-hierarchical Feedback Control



Several receivers have scheduled a request for retransmission, but the first retransmission request leads to the suppression of others

Hierarchical Feedback Control



The essence of hierarchical reliable multicasting.
Each local coordinator forwards the message to
its children and later handles retransmission
requests.

Atomic Multicast

- Reliable group communication in the face of
 - possibly faulty processes, it is useful to look at the atomic multicast problem.
- A message is delivered to either all processes, or none
- Requires agreement about group membership
- Process Group:
 - Group view: view of the group (list of processes) sender had when message sent
 - Each message uniquely associated with a group
 - All processes in group have the same view

Distributed Commit

- The atomic multicasting problem discussed in the previous section is an example of a more general problem, known as distributed commit
- Coordinator tells all other processes that are also involved, called participants, whether or not to (locally) perform the operation in question

Two-Phase Commit

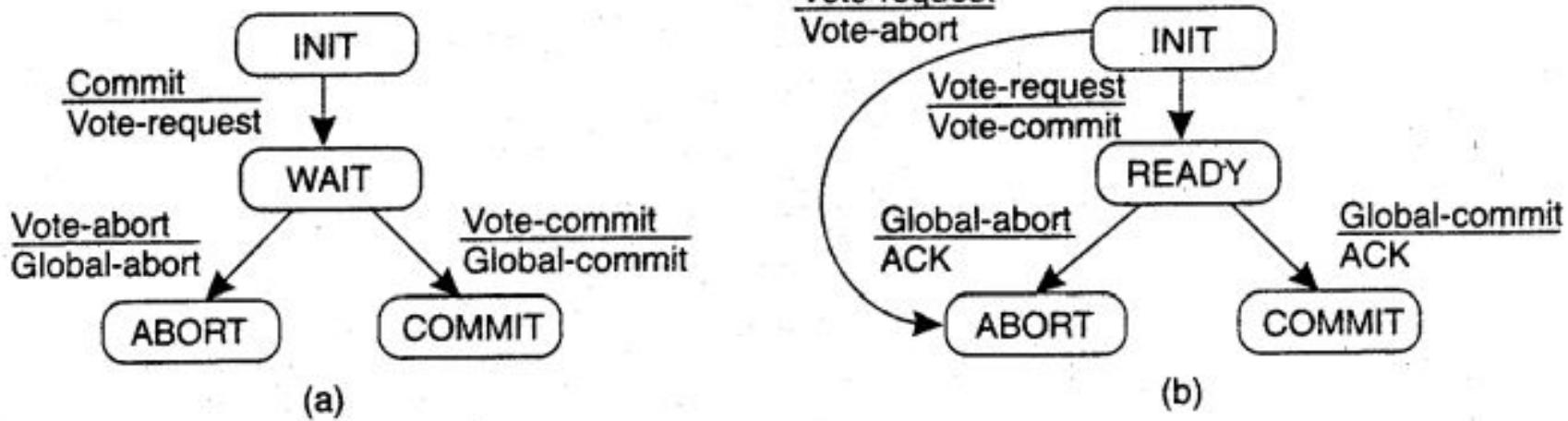


Figure 8~18. (a) The finite state machine for the coordinator in 2PC. (b) The finite state machine for a participant..

Two-Phase Commit

State of Q	Action by P
COMMIT	Make transition to COMMIT
ABORT	Make transition to ABORT
INIT	Make transition to ABORT
READY	Contact another participant

Three-Phase Commit

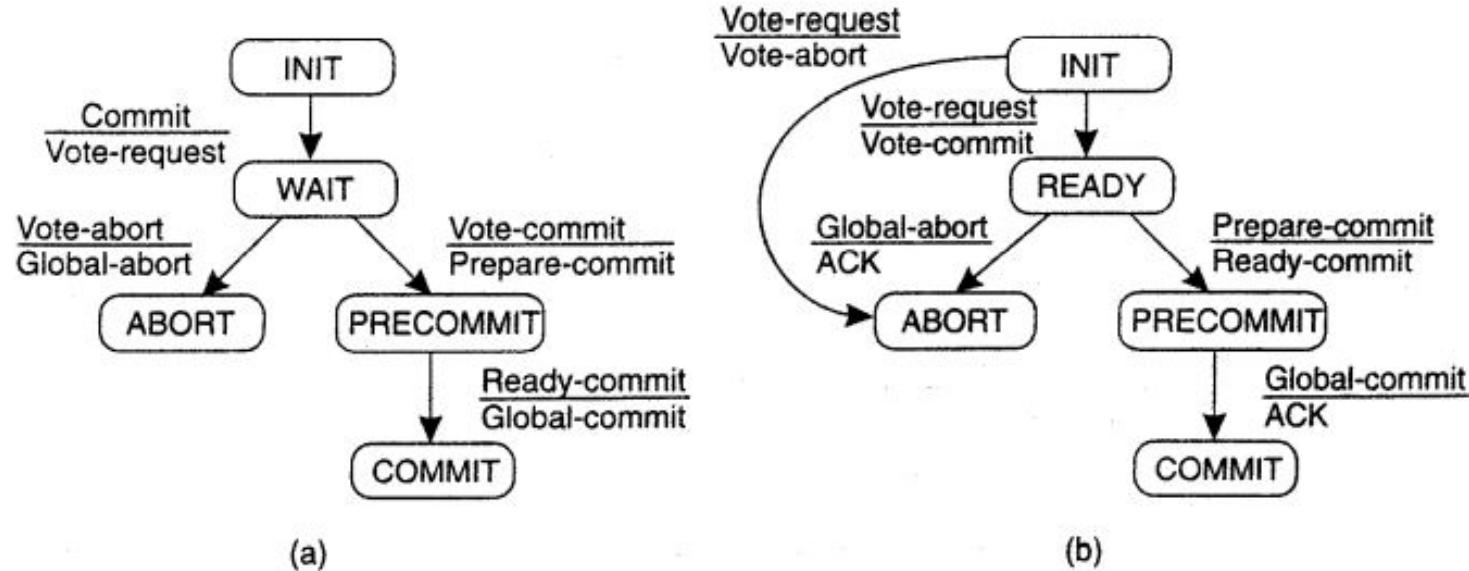


Figure 8-22. (a) The finite state machine for the coordinator in 3PC. (b) The finite state machine for a participant.

Failure Recovery

- Recovery refers- to the process of restoring a (failed) system/process to a normal state of operation.
- Recovery can apply – to the complete system (involving rebooting a failed computer) or – to a particular application (involving restarting of failed process(es)).
- While restarting processes or computers is
 - a relatively straightforward exercise in a centralized system,
 - but it is significantly more complicated in a distributed system.

Restoring an erroneous state to an error free state

Failure Recovery

Issues:

- 1) Reclamation of resources: a process may hold resources, such as locks or buffers, on a remote node. Naively restarting the process or its host will lead to resource leaks and possibly deadlocks.
- 2) Consistency: Naively restarting one part of a distributed computation will lead to a local state that is inconsistent with the rest of the computation. In order to achieve consistency it is, in general, necessary to undo partially completed operations on other nodes prior to restarting.
- 3) Efficiency: One way to avoid the above problems would be to restart the complete computation whenever one part fails. However, this is obviously very inefficient, as a significant amount of work may be discarded unnecessarily

Forward vs. backward recovery

Forward error recovery :

- Correct erroneous state without moving back to a previous state.
- Example: erasure correction - missing packet reconstructed from successfully delivered packets.
 - Here it is known that all communication has been lost,
 - and if appropriate protocols are used (which, for example, buffer all outgoing messages),
 - a forward recovery may be possible (e.g. by resending all buffered messages).
- Possible errors must be known in advance

Forward vs. backward recovery

- Backward error recovery :
- Correct erroneous state by moving to a previously correct state- restores the process or system state to a previous state known to be free from errors
- Example: packet retransmission when packet is lost
 - High overhead- due to the lost computation and the work required to restore the state.
 - Error can reoccur
 - Sometimes impossible to roll back (e.g. ATM has already delivered the money)

State-Based Recovery - Checkpointing

- State-based recovery requires checkpoints to be performed during execution.
- Shadow paging: make modifications to shadow page, after commit, make it to original one

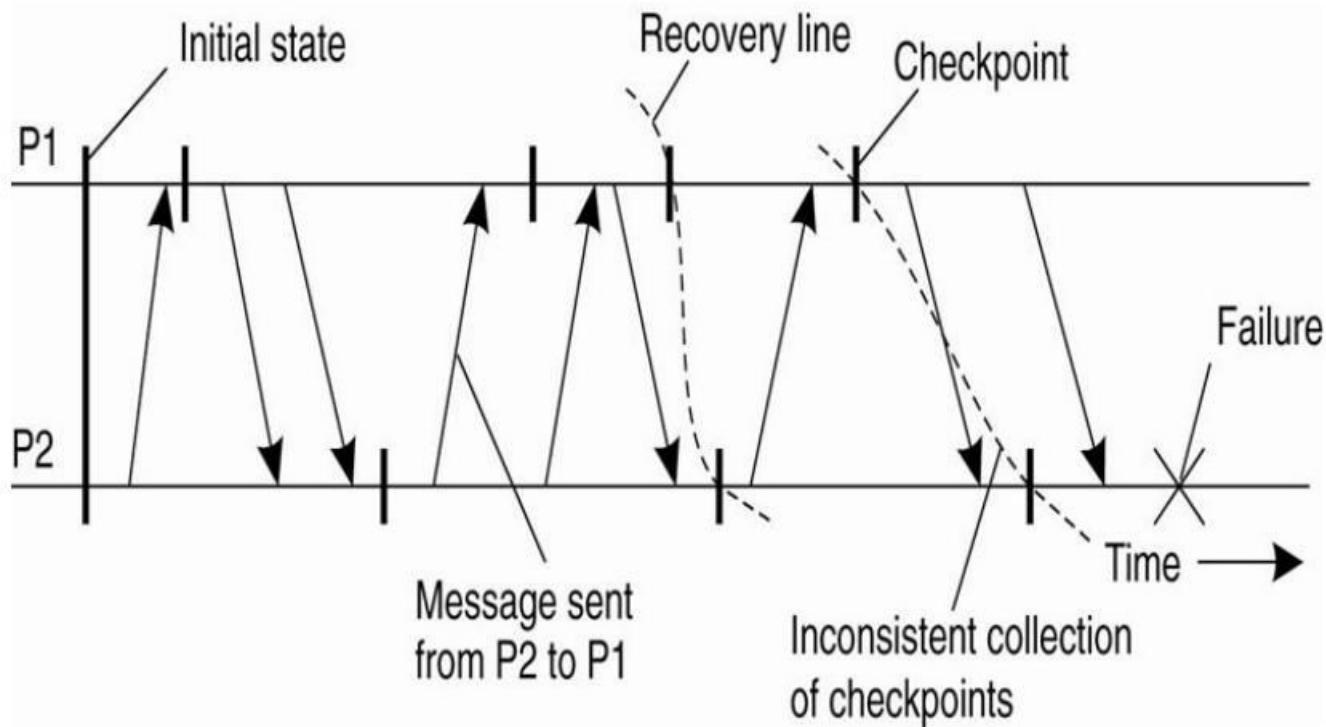
Checkpointing :

1. Pessimistic and optimistic
2. Independent and coordinated
3. Synchronous and asynchronous

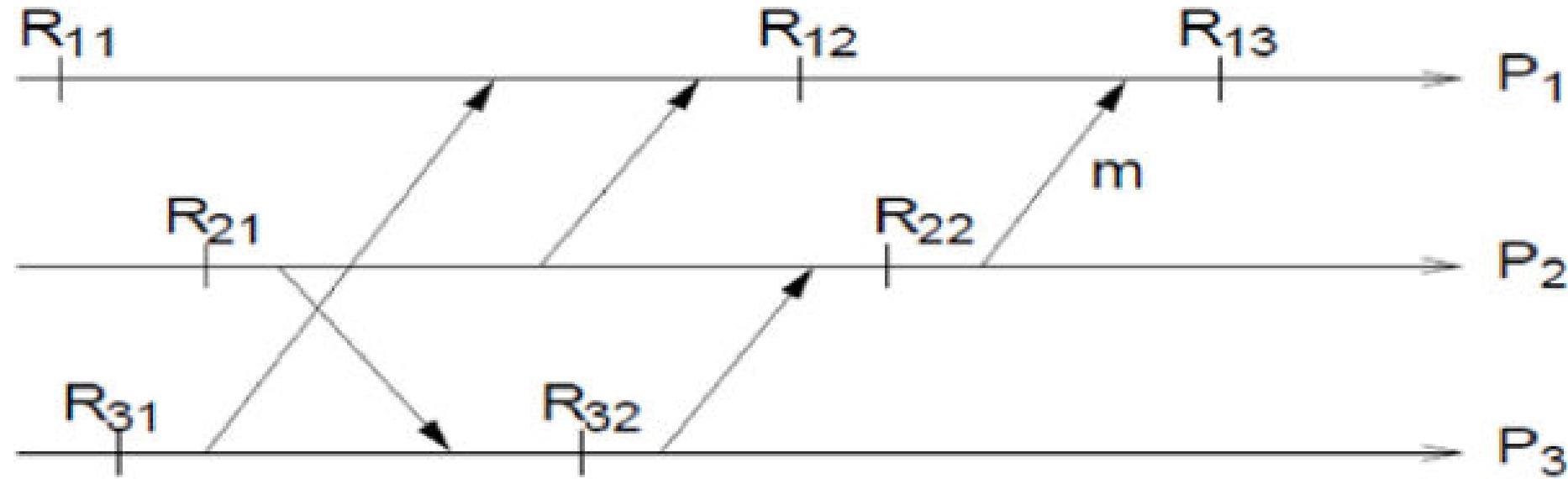
Recovery In Concurrent Systems

- Failed process may have causally affected other processes
- Upon recovery of failed process, must undo effects on other processes
- Must roll back all affected processes
- All processes must establish recovery points
- Must roll back to a consistent global state

Recovery Line

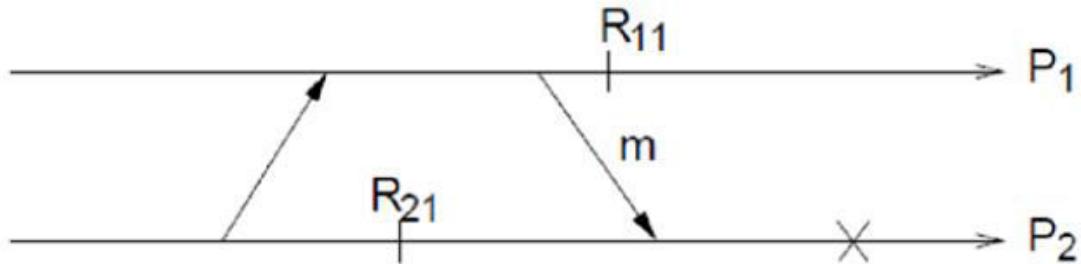


Independent Checkpointing- Domino effect



Message Loss

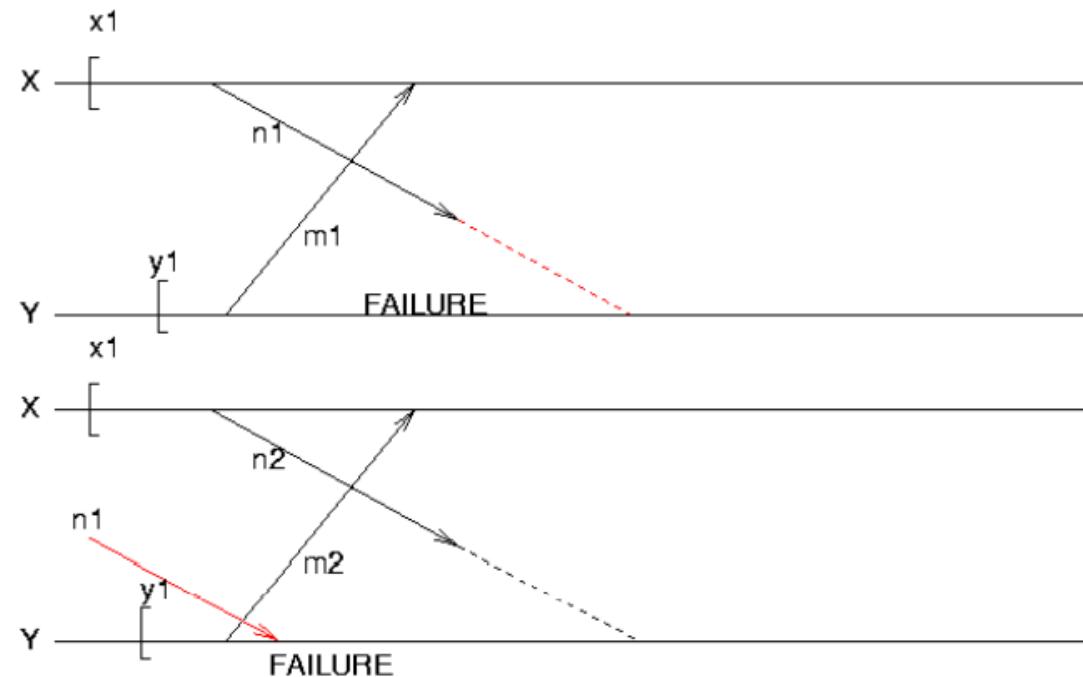
Rollback leading to message loss after failure of P_2



- Failure of $P_2 \rightarrow P_2 \sim R_{21}$
- Message m is now recorded as sent (by P_1) but not received (by P_2), and m will never be received after rollback
- Message m is *lost*
- Whether m is lost due to rollback or due to imperfect communication channels is indistinguishable!
- Require protocols resilient to message loss

Live Lock

Single failure cause an infinite number of rollbacks



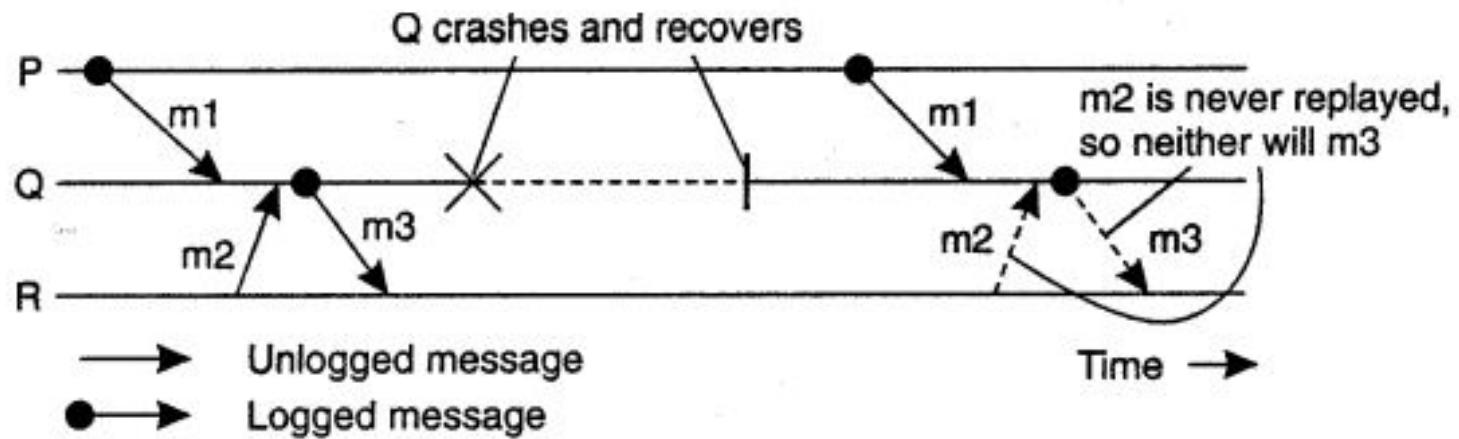
Coordinated Checkpointing

- Coordinated checkpointing all processes synchronize to jointly write their state to local stable storage
- A coordinator first multicasts a CHECKPOINT .-REQUEST message to all processes
- The main advantage of coordinated checkpointing is that the saved state is automatically globally consistent, so that cascaded rollbacks leading to the domino effect are avoided

Message logging

- the checkpointing is an expensive operation, especially concerning the operations involved in writing state to stable storage, techniques have been sought to reduce the number of checkpoints, but still enable recovery.
- An important technique in distributed systems is logging messages
- The basic idea underlying message logging is that if the transmission of messages can be replayed, we can still reach a globally consistent state but without having to restore that state from stable storage

Characterizing Message-logging Schemes



Message Logging

- Pessimistic Logging protocols
- Optimistic logging protocol

Thank you

Distributed Object-Based Systems

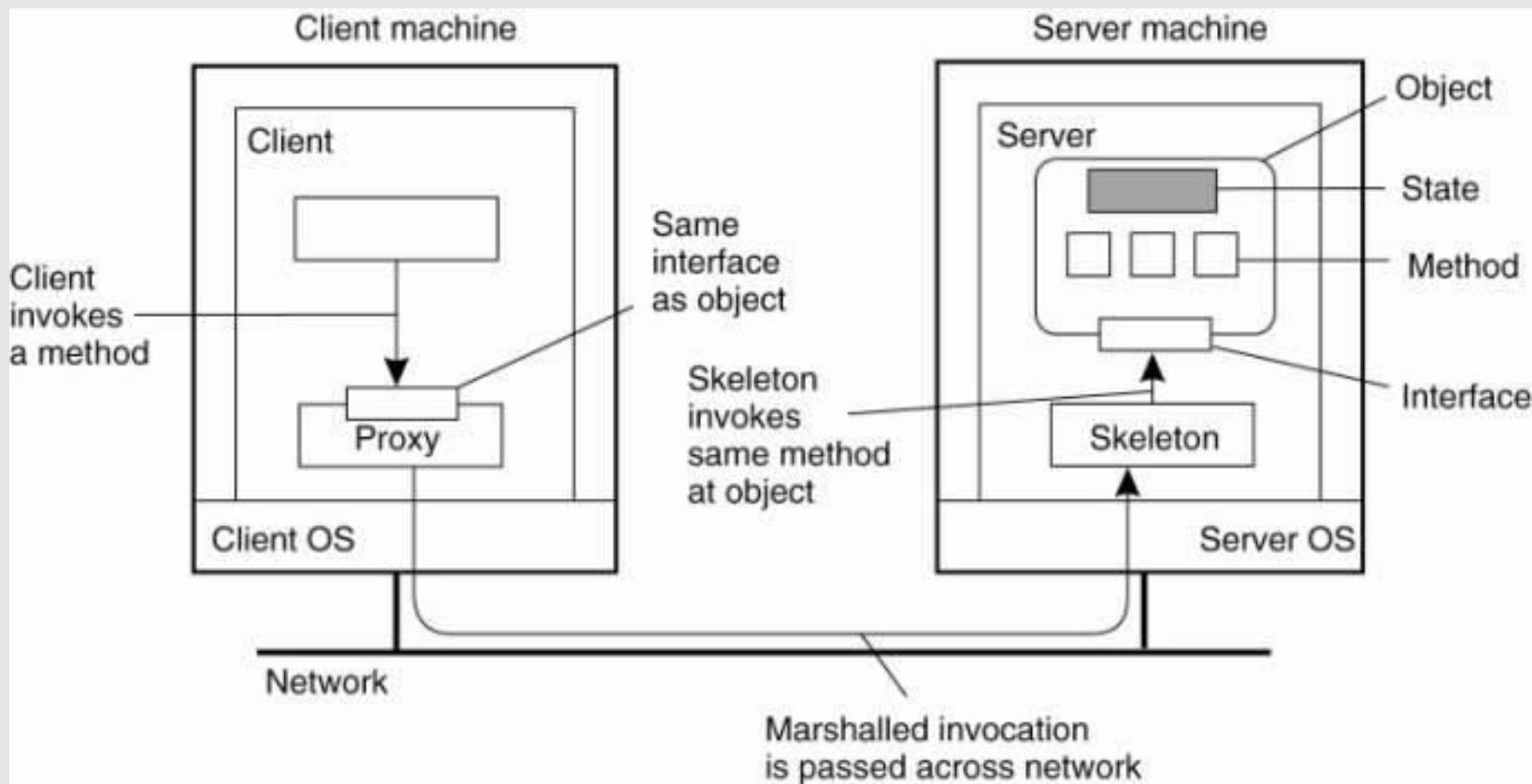
Introduction

- The first paradigm consists of distributed objects. In distributed object based systems, the notion of an object plays a key role in establishing distribution transparency.
- In principle, everything is treated as an object and clients are offered services and resources in the form of objects that they can invoke.
- Distributed objects form an important paradigm because it is relatively easy to hide distribution aspects behind an object's interface.
- Furthermore, because an object can be virtually anything, it is also a powerful paradigm for building systems
- In this chapter, we will take a look at how the principles of distributed systems are applied to a number of well-known object-based systems.

Distributed Objects

- The key feature of an object is that it encapsulates data called the state
- The operations on those data, called the methods. Methods are made available through an interface
- An object may implement multiple interfaces

Distributed Objects



Compile Time versus Runtime Objects

- The most obvious form is the one that is directly related to language-level objects such as those supported by Java, C++, or other object-oriented languages, which are referred to as compile-time objects
- Using compile-time objects in distributed systems often makes it much easier to build distributed applications
- For example , in Java:
 1. An object can be fully defined by means of its class and the interfaces that the class implements.
 2. Compiling the class definition results in code that allows it to instantiate Java objects.
 3. The interfaces can be compiled into client-side and server-side stubs, allowing the Java objects to be invoked from a remote machine.
 4. A Java developer can be largely unaware of the distribution of objects: he sees only Java programming code.

Compile Time versus Runtime Objects

- An alternative way of constructing distributed objects is to do this explicitly during runtime.
- A common approach is to use an object adapter, which acts as a wrapper around the implementation with the sole purpose to give it the appearance of an object

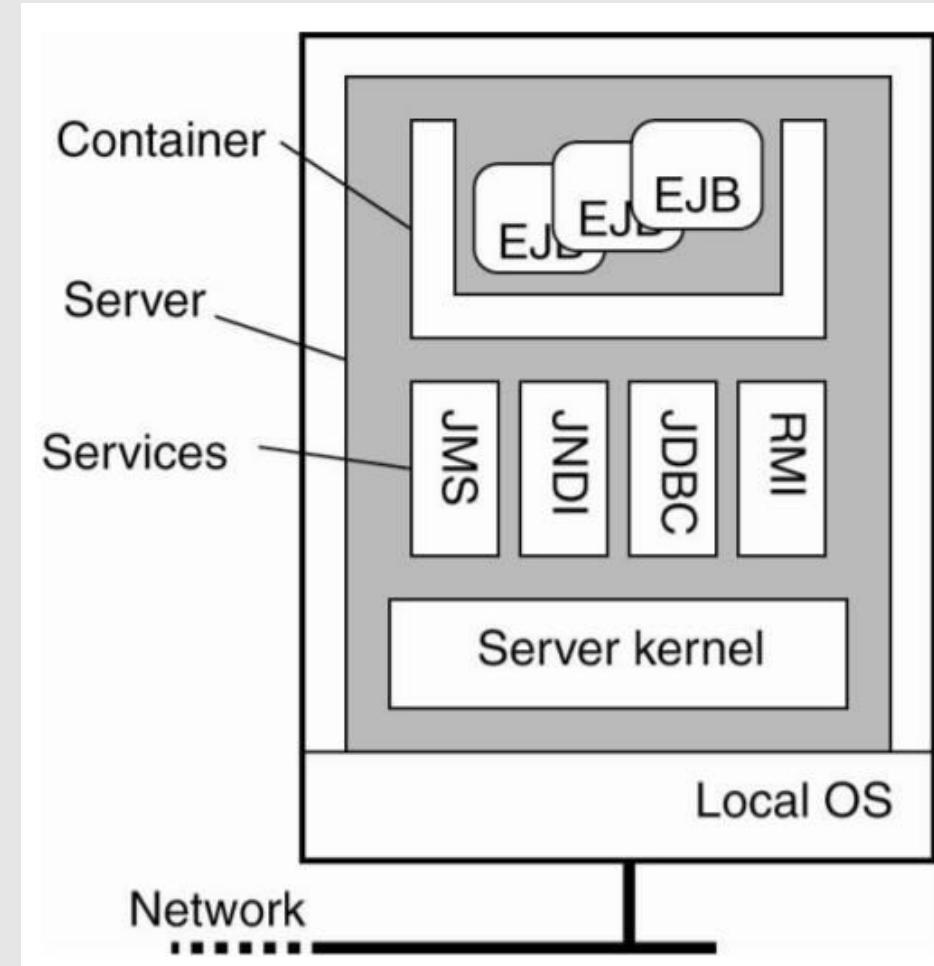
Persistent and Transient Objects

- A persistent object continues to exist even if it is currently not contained in the address space of any server process
- In contrast, a transient object is an object that exists only as long as the server that is hosting the object is.

Enterprise Java Beans (EJB)

- An EJB is essentially a Java object that is hosted by a special server offering different ways for remote clients to invoke that object
- Crucial is that this server provides the support to separate application functionality from systems-oriented functionality
- The latter includes functions for looking up objects, storing objects, letting objects be part of a transaction, and so on
- Typical services include those for remote method invocation (RMI), database access (JDBC), naming (JNDI), and messaging (JMS). Making use of these services is more or less automated

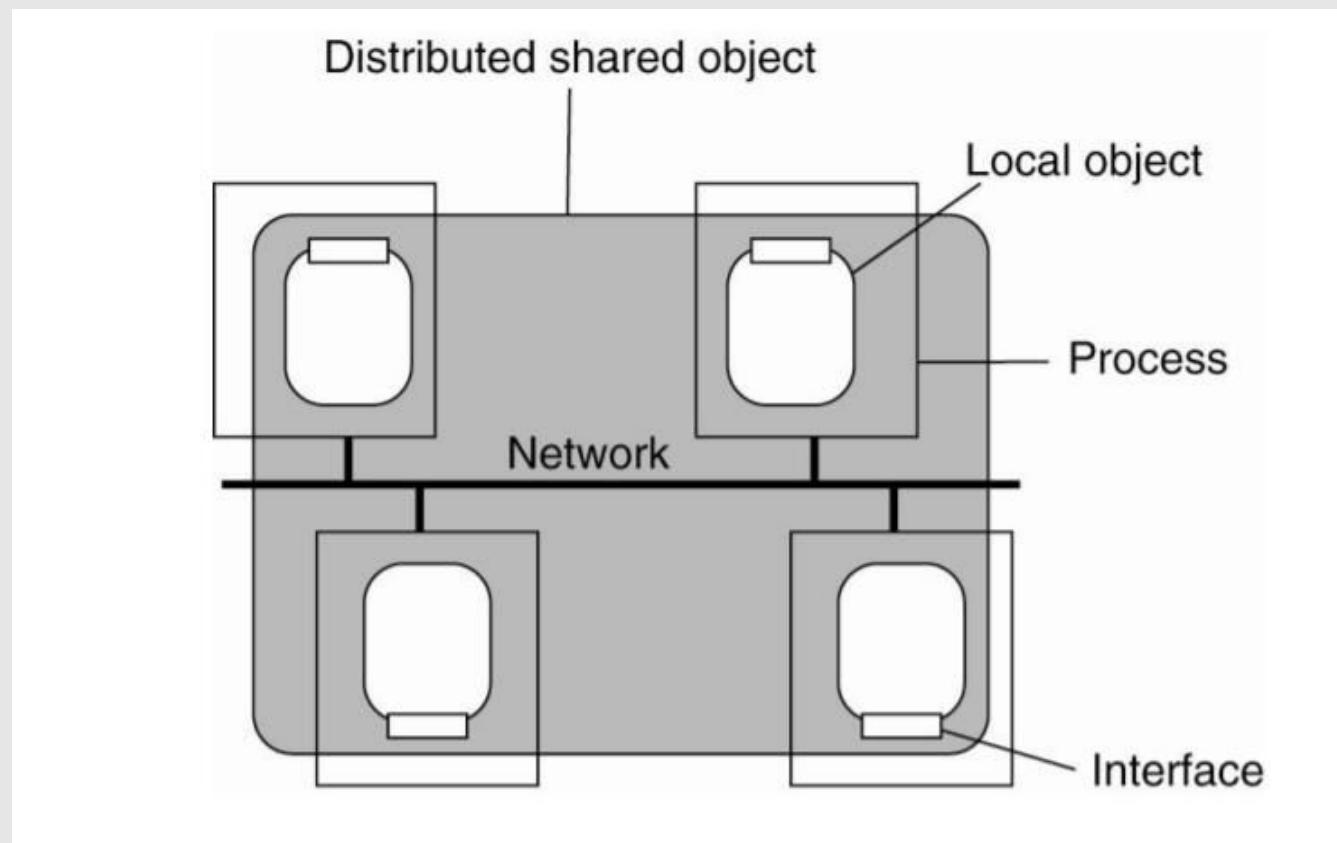
Enterprise Java Beans (EJB)



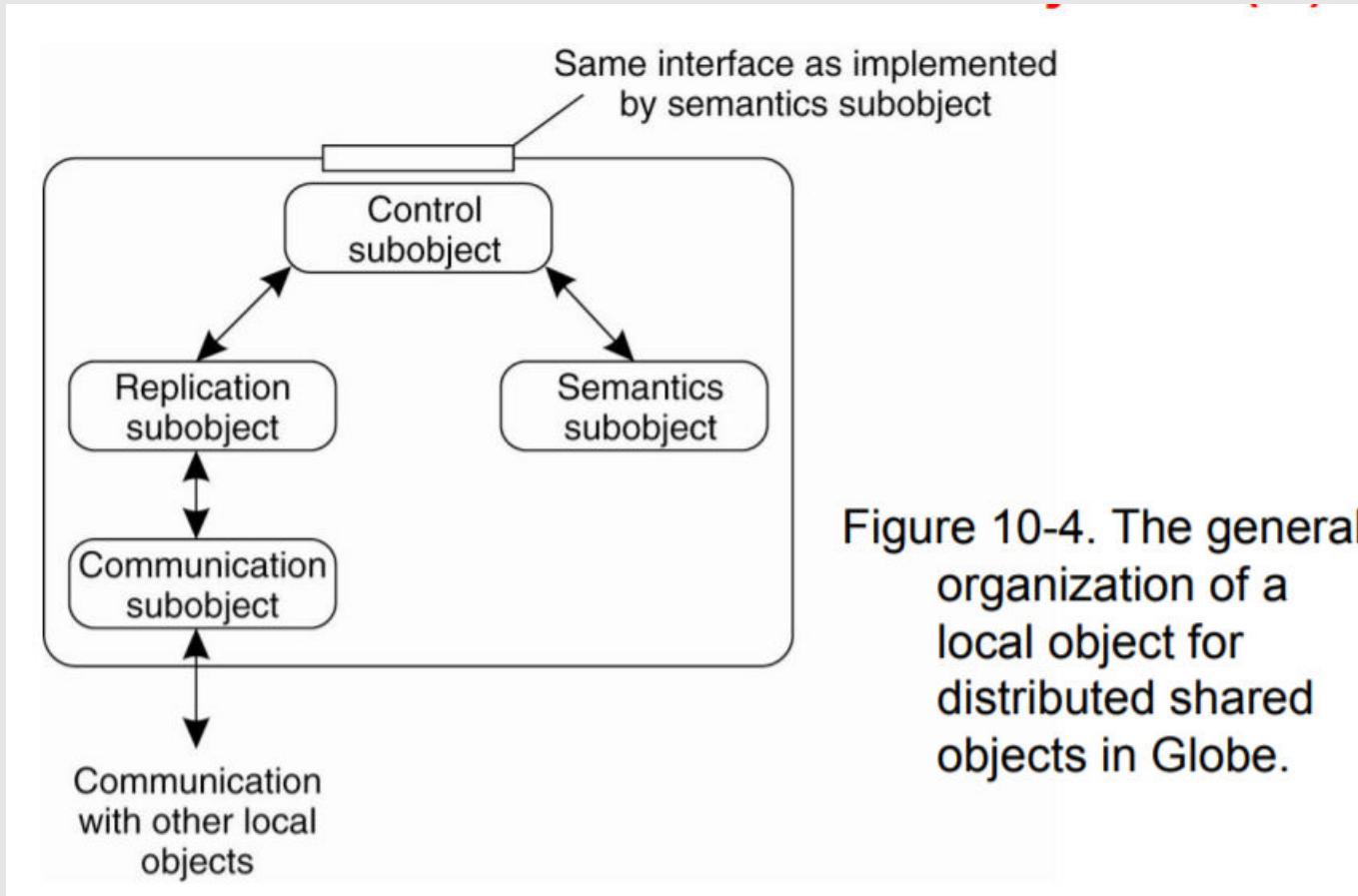
Enterprise Java Beans (EJB)

- Stateless session beans are transient objects that are invoked once, does its work, after which it discards any information it maintains to perform the service it offered to a client.
- Stateful session beans maintain client-related state.
- Entity beans can be considered to be a long-lived persistent object. Such an entity bean will generally be stored in a database, and likewise, will also be part of distributed transactions.
- Message-driven beans are used to program objects that should react to incoming messages (and likewise, be able to send messages). They cannot be invoked directly by a client, but rather fit into a publish-subscribe way of communication

Global Distributed shared objects



Global Distributed shared objects



Object servers

- A key role in object-based distributed systems is played by object servers, that is, the server designed to host distributed objects.
- The important difference between a general object server and other (more traditional) servers is that an object server by itself does not provide a specific service.
- Specific services are implemented by the objects that reside in the server
- An object server thus acts as a place where objects live

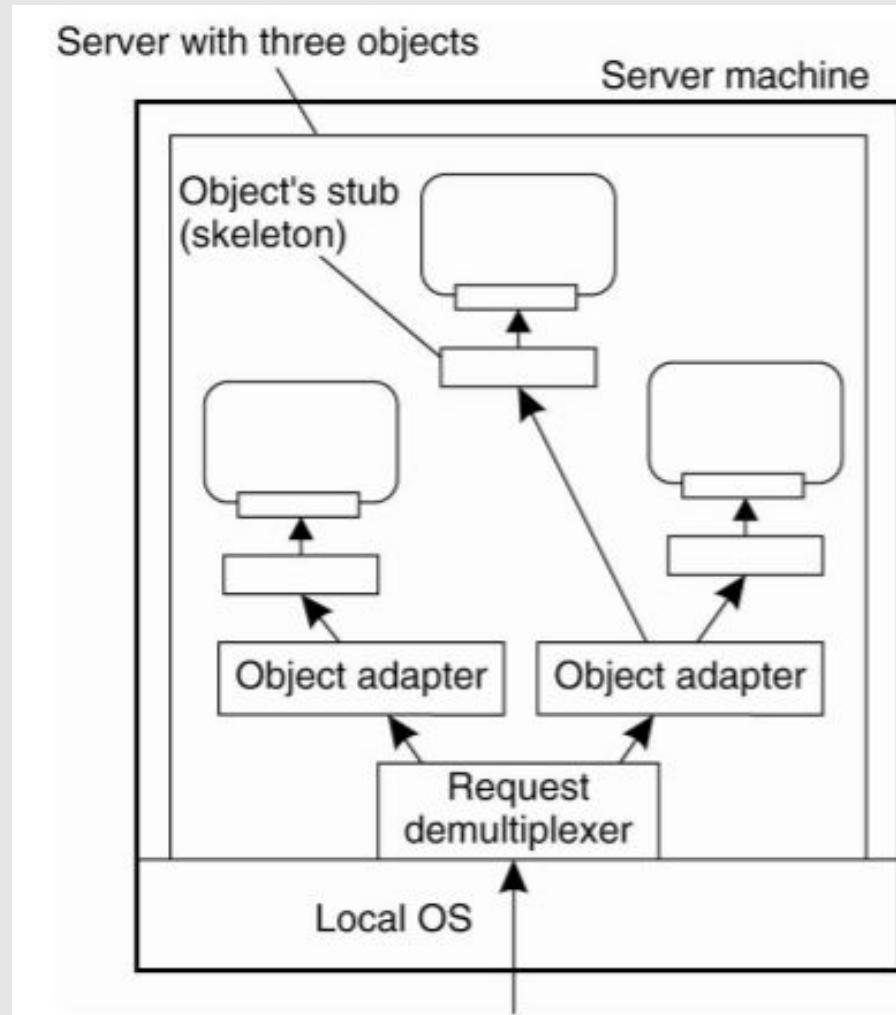
Object servers

- An object consists of two parts: data representing its state and the code for executing its methods
- Whether or not these parts are separated, or whether method implementations are shared by multiple objects, depends on the object server

Object Adapters

- Decisions on how to invoke an object are commonly referred to as activation policies, to emphasize that in many cases the object itself must first be brought into the server's address space (i.e., activated) before it can actually be invoked.
- What is needed then is a mechanism to group objects per policy. Such a mechanism is sometimes called an object adapter, or alternatively an object wrapper. An object adapter can best be thought of as software implementing a specific activation policy

Object Adapters



Communication – Binding a Client to an Object

- These systems generally offer the means for a remote client to invoke an object.
- This mechanism is largely based on remote procedure calls (RPCs).
- A difference between traditional RPC systems and distributed objects is that the latter generally provides system-wide object references.
- Object references can be freely passed between processes on different machines, for example as parameters to method invocations

Communication – Binding a Client to an Object

- By hiding the actual implementation of an object reference, distribution transparency is enhanced compared to traditional RPCs.
- When a process holds an object reference, it must first bind to the referenced object before invoking any of its methods.
- Binding results in a proxy being placed in the process's address space, implementing an interface containing the methods the process can invoke. In many cases, binding is done automatically.
- When an object reference is given, it needs a way to locate the server that manages the actual object, and place a proxy in the client's address space.

Communication – Binding a Client to an Object

- With implicit binding, the client is offered a simple mechanism that allows it to directly invoke methods using only a reference to an object.
- In contrast, with explicit binding, the client should first call a special function to bind to the object before it can actually invoke its methods. Explicit binding generally returns a pointer to a proxy that is then become locally available

Implementation of Object References

- An object reference must contain enough information to allow a client to bind to an object.

Implementation of Object References

Drawbacks

First: if the server's machine crashes and the server is assigned a different end point after recovery, all object references have become invalid.

Implementation of Object References

So far the client and server:

- Have somehow already been configured to use the same protocol stack.
- Not only does this mean that they use the same transport protocol, for example, TCP;
- Furthermore, it means that they use the same protocol for marshaling and unmarshaling parameters.
- They must also use the same protocol for setting up an initial connection, handle errors and flow control the same way, and so on.

Static versus Dynamic Remote Method Invocations

Static RMI: Static invocations require that the interfaces of an object are known when the client application is being developed. It also implies that if interfaces change, then the client application must be recompiled before it can make use of the new interfaces.

Dynamic RMI: The essential difference with static invocation is that an application selects at runtime which method it will invoke at a remote object. Dynamic invocation generally takes a form such as:

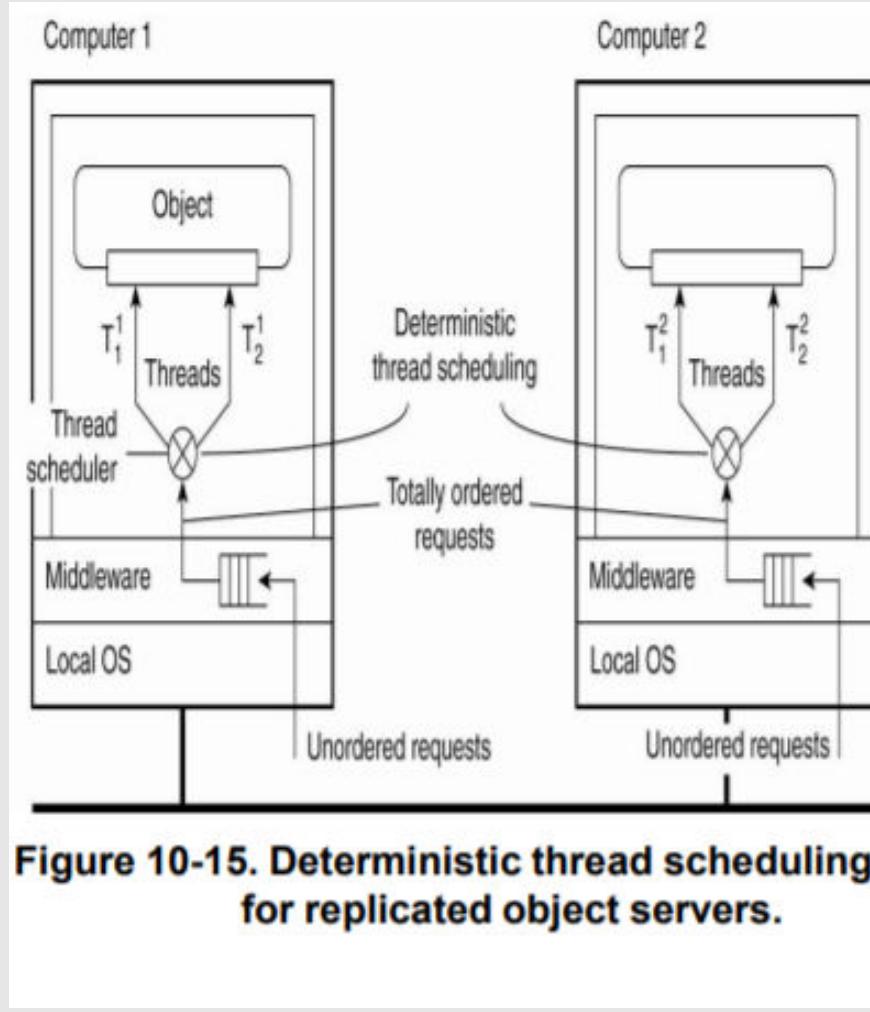
```
invoke(object, method, inputParameters, outputParameters);
```

Consistency and Replication

Entry Consistency

- Data-centric consistency for distributed objects comes naturally in the form of entry consistency. Recall that in this case, the goal is to group operations on shared data using synchronization variables (e.g., in the form of locks).
- Objects naturally combine data and the operations on that data, locking objects during an invocation serializes access and keeps them consistent

Entry Consistency



Replica Frameworks

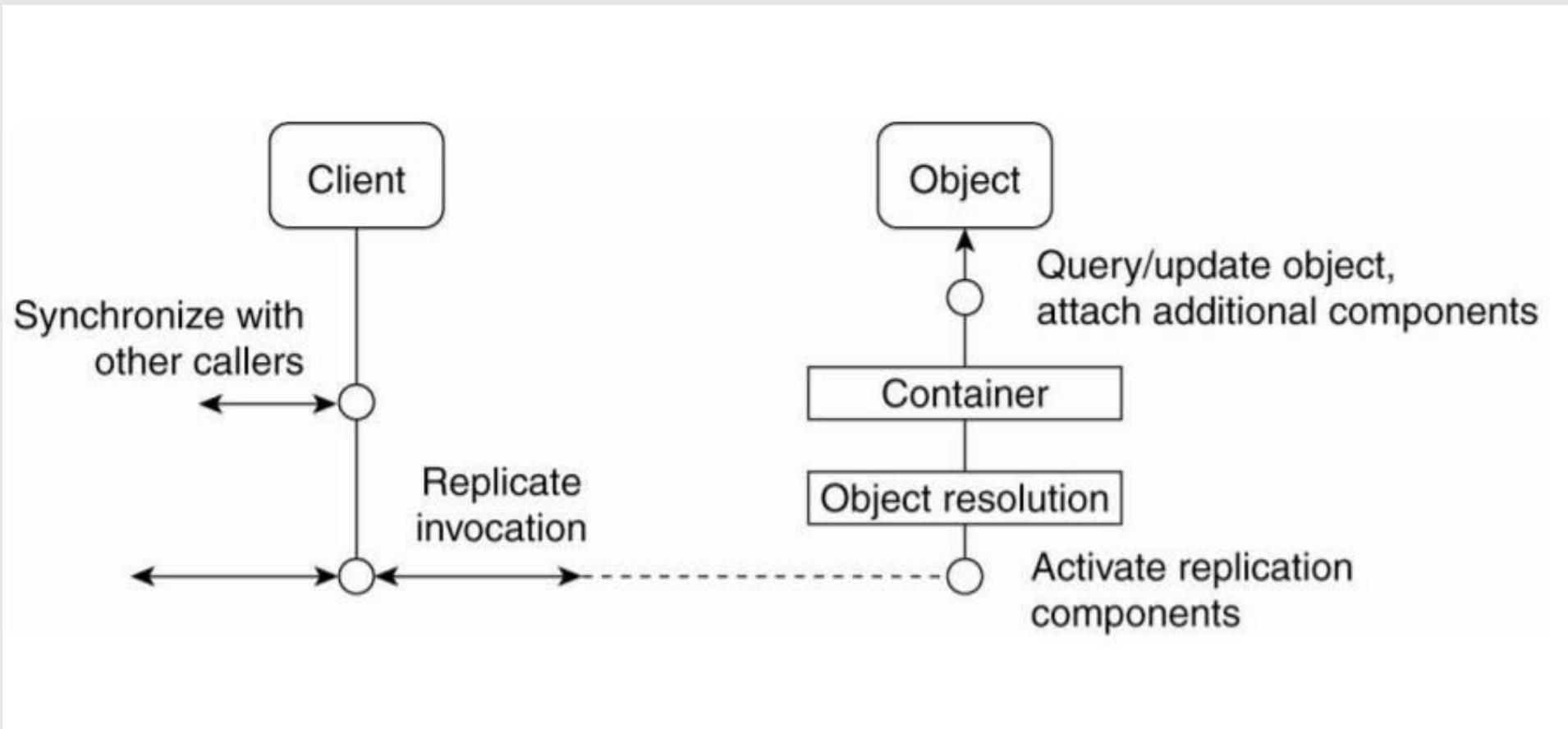
An interesting aspect of most distributed object-based systems is that by nature of the object technology it is often possible to make a clean separation between devising functionality and handling extra functional issues such as replication. A powerful mechanism to accomplish this separation is formed by interceptors

Replica Frameworks

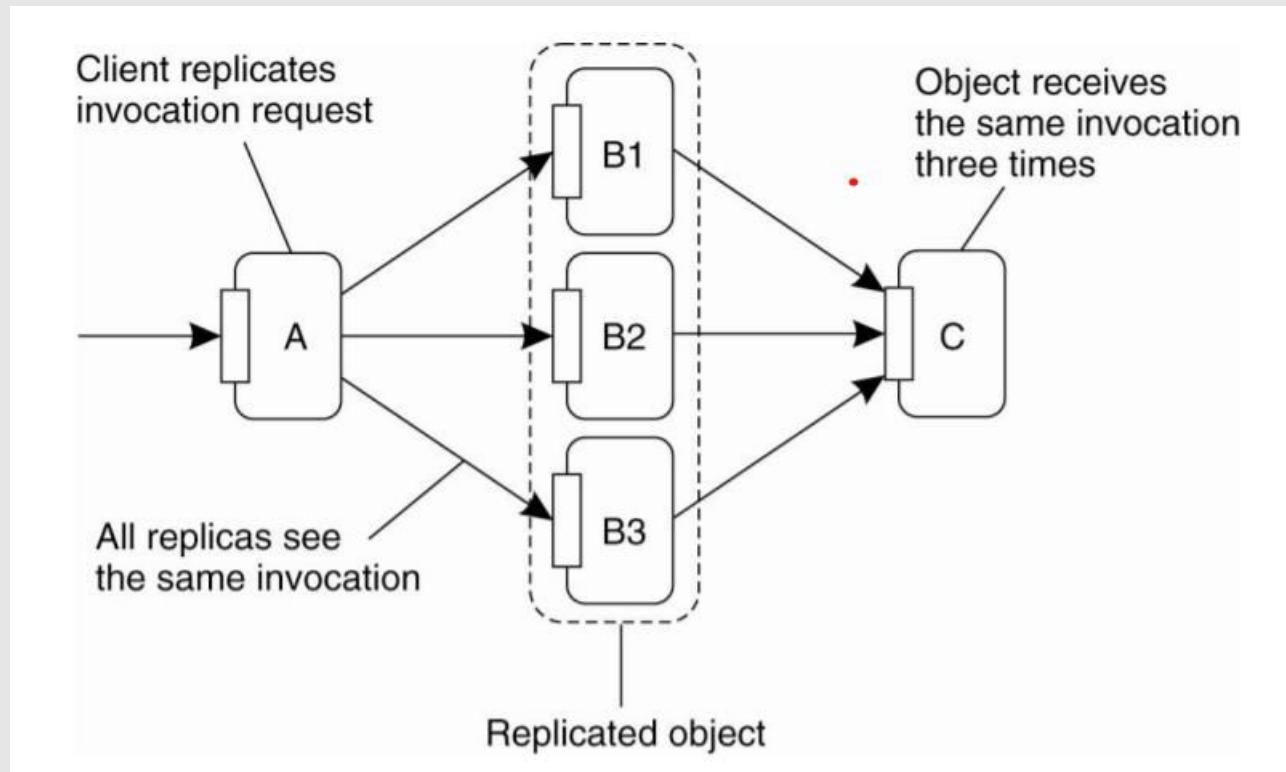
Babaoglu et al. (2004) describe a framework in which they use interceptors to replicate Java beans for J2EE servers. The idea is relatively simple: invocations to objects are intercepted at three different points

- At the client side just before the invocation is passed to the stub.
- Inside the client's stub, where the interception forms part of the replication algorithm.
- At the server side, just before the object is about to be invoked

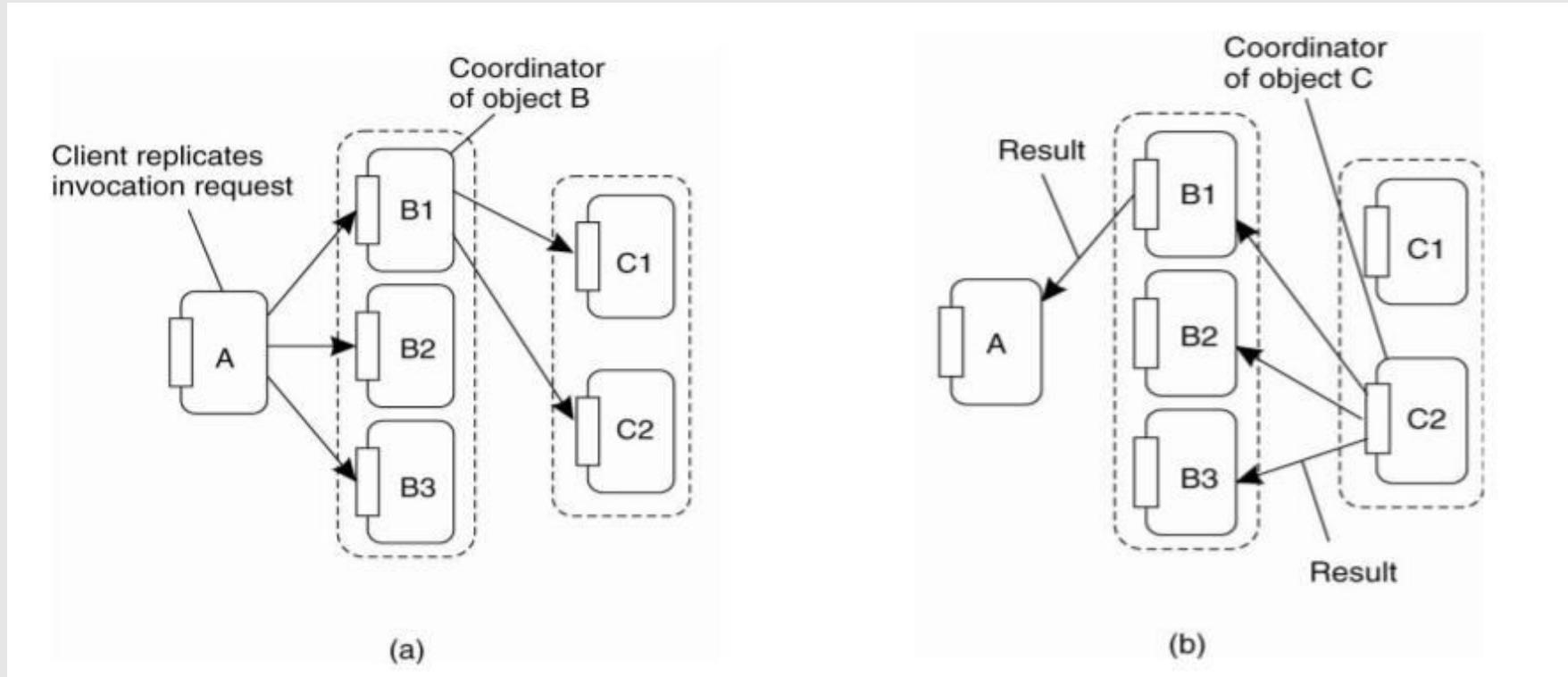
Replica Frameworks



Replicated Invocation



Replicated Invocation



Thank You

Syllabus

- **INTRODUCTION TO DISTRIBUTED SYSTEMS** **(04 Hours)**

Review of Networking Protocols, Point to Point Communication, Operating Systems, Concurrent Programming, Characteristics and Properties of Distributed Systems, Goals of Distributed Systems, Multiprocessor and Multicomputer Systems, Distributed Operating Systems, Network Operating Systems, Middleware Concept, The Client-Server Model, Design Approaches-Kernel Based-Virtual Machine Based, Application Layering.
- **COMMUNICATIONIN DISTRIBUTED SYSTEMS** **(04 Hours)**

Layered Protocols, Message Passing-Remote Procedure Calls-Remote Object Invocation, Message Oriented Communication, Stream Oriented Communication, Case Studies.
- **PROCESS MANAGEMENT** **(04 Hours)**

Concept of Threads, Process, Processor Allocation, Process Migration and Related Issues, Software Agents, Scheduling in Distributed System, Load Balancing and Sharing Approaches, Fault Tolerance, Real Time Distributed System.
- **SYNCHRONIZATION** **(06 Hours)**

Clock Synchronization, Logical Clocks, Global State, Election Algorithms-The Bully algorithm-A Ring algorithm, Mutual Exclusion-A Centralized Algorithm-A Distributed Algorithm-A token ring Algorithm, Distributed Transactions.

Syllabus

- **CONSISTENCY AND REPLICATION** (06 Hours)

Introduction to Replication, Object Replication, Replication as Scaling Technique, Data Centric Consistency Models-Strict-Linearizability and Sequential-Causal-FIFO-Weak-release-Entry, Client Centric Consistency Models-Eventual Consistency-Monotonic Reads and Writes-Read your Writes-Writes Follow Reads, Implementation Issues, Distribution Protocols-Replica Placement-Update Propagation-Epidemic Protocols, Consistency Protocols.
- **FAULT TOLERANCE** (04 Hours)

Introduction, Failure Models, Failure Masking, Process Resilience, Agreement in Faulty Systems, Reliable Client Server communication, Group communication, Distributed Commit, Recovery.
- **DISTRIBUTED OBJECT BASED SYSTEMS** (06 Hours)

Introduction to Distributed Objects, Compile Time Vs Run Time Objects, Persistent and Transient Objects, Enterprise JAVA Beans, Stateful and Stateless Sessions, Global Distributed Shared Objects, Object Servers, Object Adaptors, Implementation of Object References, Static And Dynamic Remote Method Invocations, Replica Framework.
- **DISTRIBUTED FILE SYSTEMS** (04 Hours)

Introduction, Architecture, Mechanisms for Building Distributed File Systems-Mounting-Caching-Hints-Bulk Data Transfer-Encryption, Design Issues-Naming and Name Resolution-Caches on Disk or Main Memory-Writing Policy-Cache consistency-Availability-Scalability-Semantics, Case Studies, Log Structured File Systems.
- **DISTRIBUTED WEB BASED SYSTEMS** (04 Hours)

Architecture, Processes, Communication, Naming, Synchronization, Web Proxy Caching, Replication of Web Hosting Systems, Replication of Web Applications.

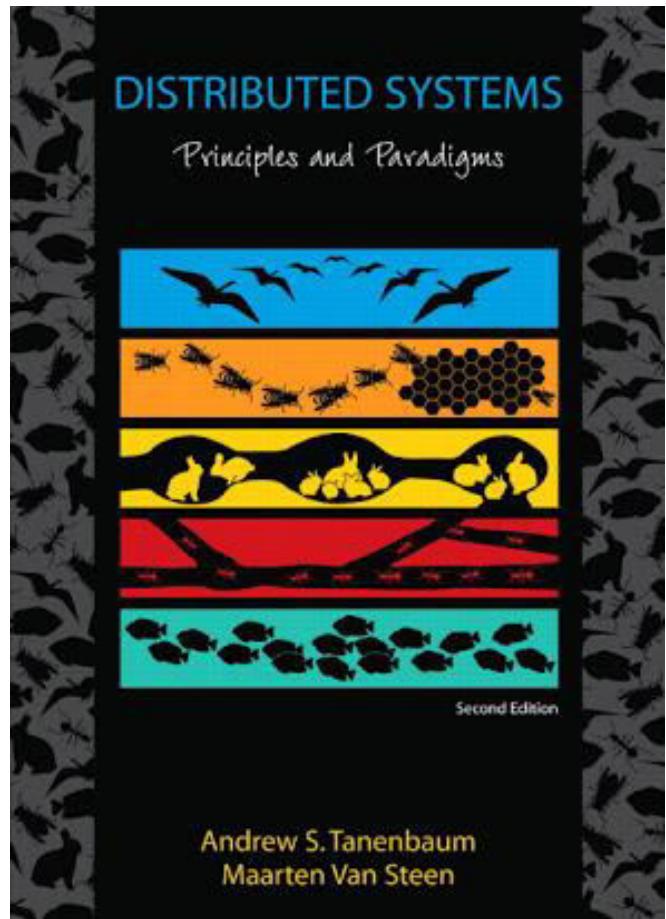
Textbook

- 1.** Andrew S. Tanenbaum & Van Steen, “Distributed Systems Principles and Paradigms”, Publisher: PHI, Year: 2007.
- 2.** P.K.Sinha, “Distributed Operating Systems”, Publisher: PHI, Year: 2007.
- 3.** Mukesh Singhal, Niranjan G. Shivaratri (Contributor) “Advanced Concepts in Operating Systems: Distributed, Database, and Multiprocessor Operating Systems”, MGH, 1994.

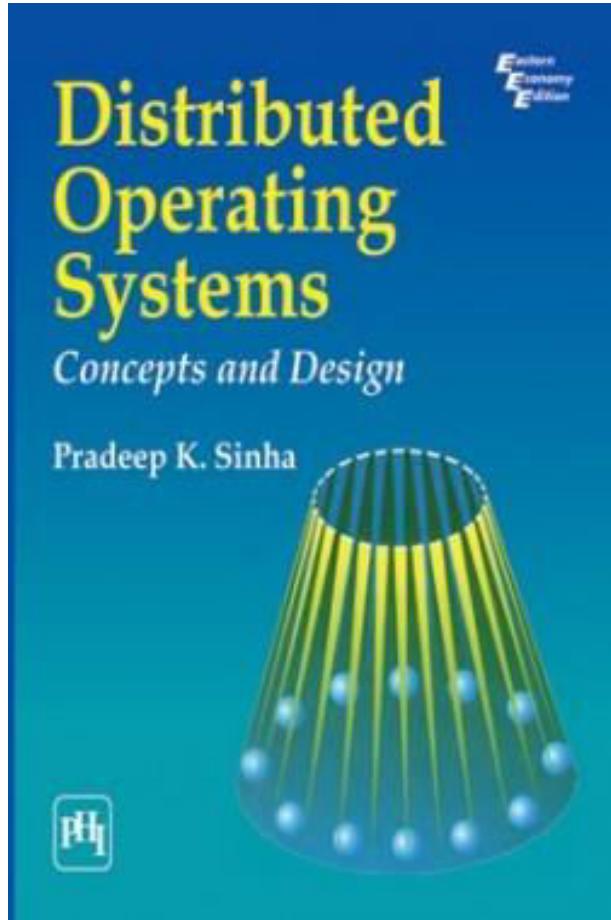
Reference Book

- 1.** Hagit Attiya, Jennifer Welch “Distributed Computing: Fundamentals, Simulations, and Advanced Topics”, 2/E, Jon Wiley & sons, March 2004.
- 2.** M.L.Liu, “Distributed Computing -- Concepts and Application”, Publisher Addison Wesley.

Reference Books



Distributed systems:
principles and paradigms I
Andrew S.Tanenbaum,
Maarten Van Steen



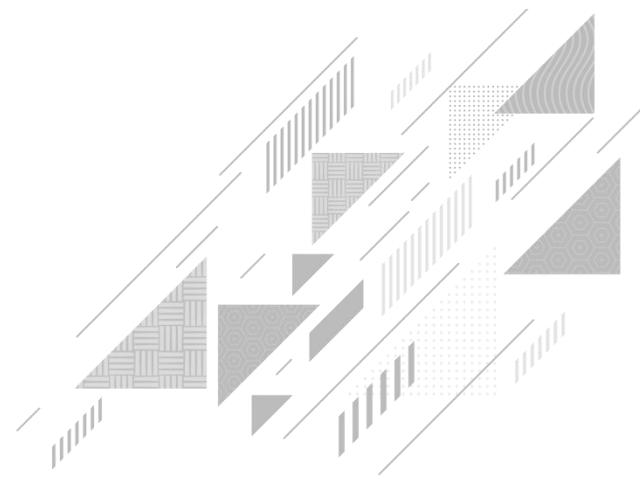
Distributed Operating
Systems Concepts and
Design
By Pradeep K. Sinha, PHI



Unit-1

Introduction to

Distributed System





Topics to be covered

- Definition of a Distributed System
- Goals of a Distributed System
- Types of Distributed Systems
- Challenges

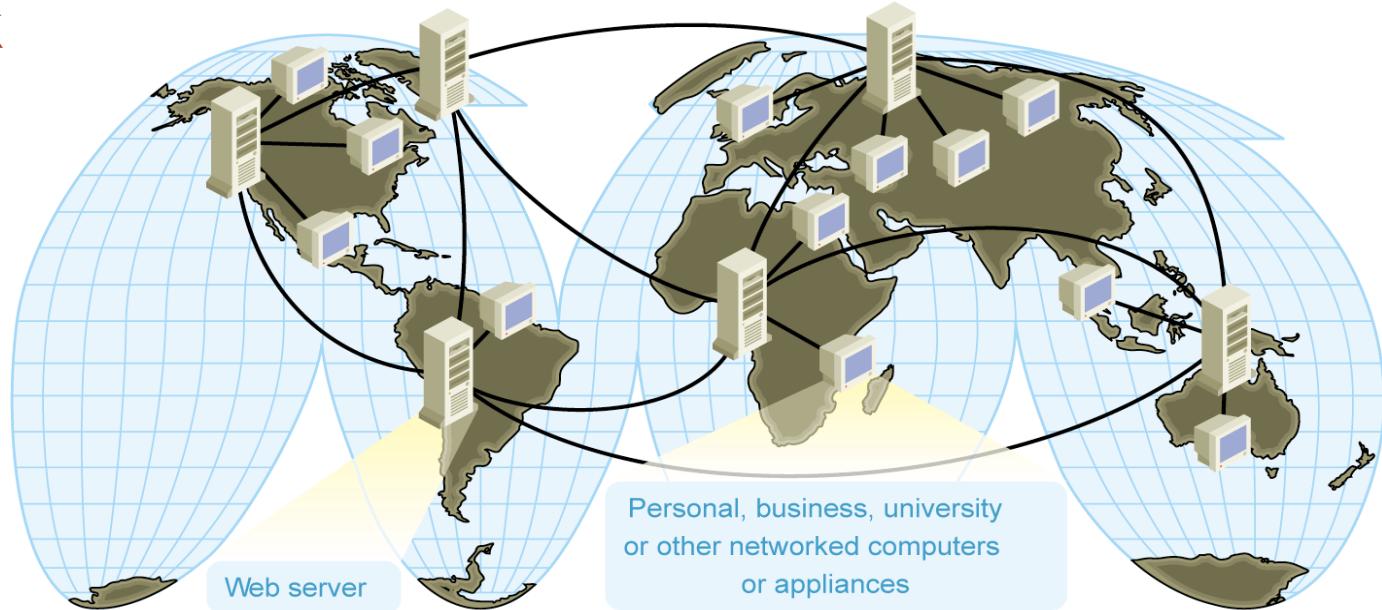
Distributed system, distributed computing

- Early computing was performed on a single processor. Uni-processor computing can be called *centralized computing*.
- A *distributed system* is a collection of independent computers, interconnected via a network, capable of collaborating on a task.
- *Distributed computing* is computing performed in a distributed system.

Distributed Operating System

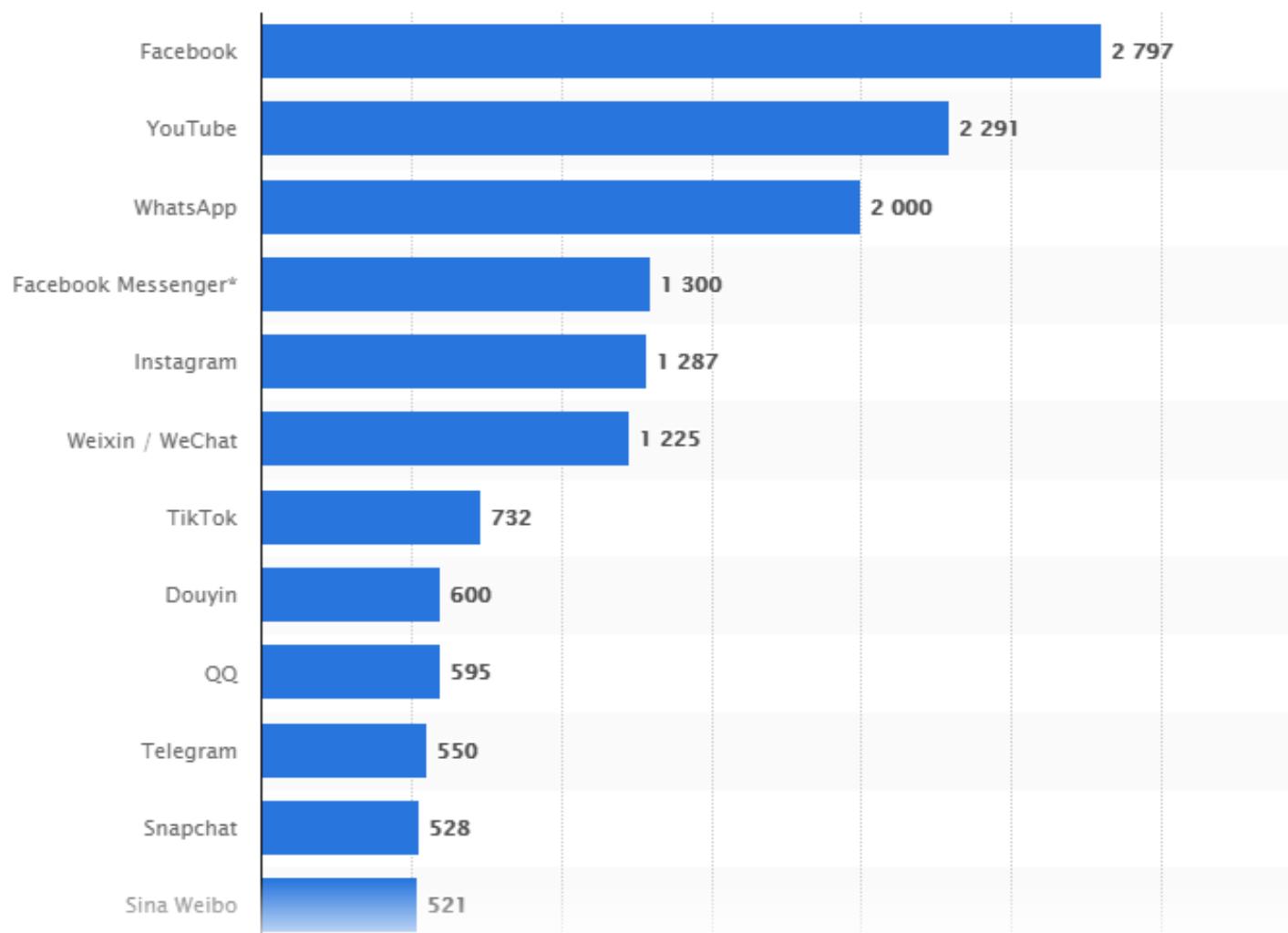
Definition by Coulouris, Dollimore, Kindberg and Blair

- ▶ “A distributed system is defined as one in which components at networked computers **communicate** and **coordinate** their actions only by **passing messages**.”
- ▶ “A Distributed system is **collection of independent computers** which are **connected through network**”



- ▶ This system looks to its users like an ordinary centralized operating system but runs on multiple, independent central processing units (CPUs).

Most popular social networks worldwide as of April 2021



Number of active users monthly (in millions)

Why Distributed Operating System?

- ▶ Facebook, currently, has 2.7 billion active monthly users.
- ▶ Google performs at least 2 trillion searches per year.
- ▶ About 500 hours of video is uploaded in Youtube every minute.
- ▶ A single system would be unable to handle the processing. Thus, comes the need for Distributed Systems.
- ▶ The main answer is to cope with the **extremely higher demand of users in both processing power and data storage.**
- ▶ With this extremely demand, single system could not achieve it.
- ▶ There are many reasons that make distributed systems is viable such as high availability, scalability, resistant to failure, etc.

Why Distributed?

► Resource and Data Sharing

- Printers, databases, multimedia servers etc.

► Availability, Reliability

- The loss of some instances can be hidden

► Scalability, Extensibility

- System grows with demands (e.g. extra servers)

► Performance

- Huge power (CPU, memory etc.) available
- *Horizontal distribution (same logical level is distr.)*

► Inherent distribution, communication

- Organizational distribution, e-mail, video conference
- *Vertical distribution (corresponding to org. struct.)*

Problems of Distribution

▶ Concurrency, Security

- Clients must not disturb each other

▶ Partial failure

- We often do not know, where is the error (e.g. RPC)

▶ Location, Migration, Replication

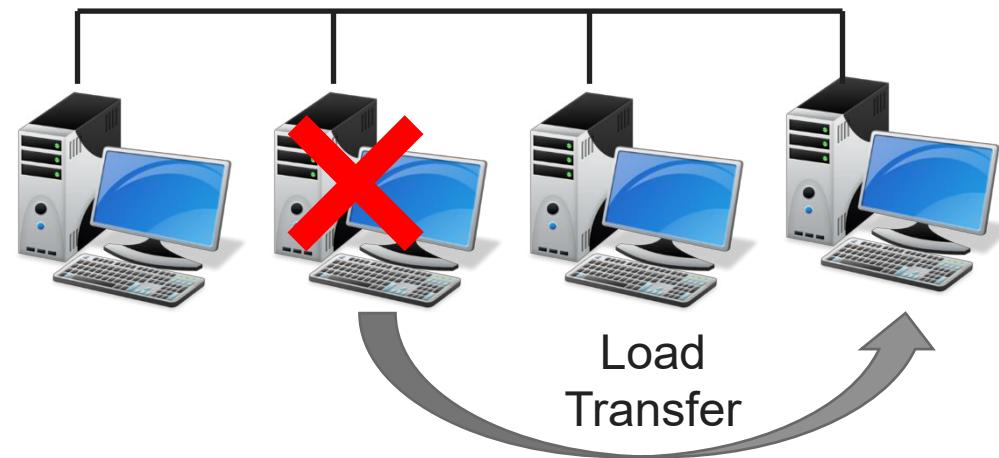
- Clients must be able to find their servers

▶ Heterogeneity

- Hardware, platforms, languages, management

Advantages of Distributed Systems over Centralized Systems

- ▶ **Reliability:** If one machine crashes, the system as a whole can still survive. Higher availability and improved reliability.
- ▶ By having redundant components, the impact of hardware and software faults on users can be reduced.



- ▶ **Data sharing:** Allow many users to access to a common database.

Advantages of Distributed Systems

- ▶ **Resource Sharing:** Expensive peripherals such as color laser printers, photo-type setters and massive storage devices are also among the few things that should be sharable.



- ▶ **Communication:** Enhance human-to-human communication, e.g., email, chat.
- ▶ **Flexibility:** Spread the workload over the available machines

Advantages of Distributed Systems over Centralized Systems

- ▶ **Performance:** By using the combined processing and storage capacity of many nodes, performance levels can be reached that is out of the scope of centralized machines.
- ▶ **Scalability:** Resources such as processing and storage capacity can be increased incrementally.

Disadvantages Of Distributed Systems

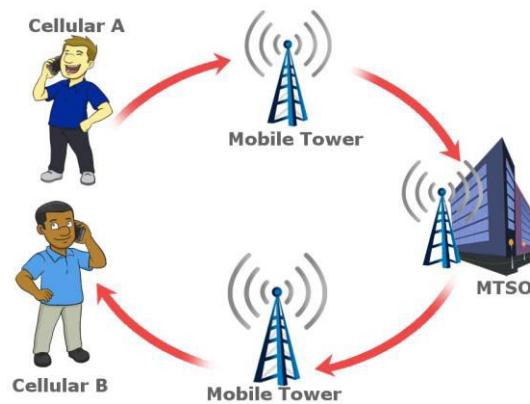
- **Multiple Points of Failures:** the failure of one or more participating computers, or one or more network links, can bring trouble.
- **Security Concerns:** In a distributed system, there are more opportunities for unauthorized attack.
- Possibility of security violation since the private data are visible to others over the network.
- **Software complexity:** Distributed software is more complex and harder to develop than conventional software.

Distributed systems are hard to build and understand.

Examples of Distributed Systems

- ▶ Distributed systems are all around us.
- ▶ From the definition, Distributed Systems also looks the same as **single system**.
- ▶ Let us say about **Google Web Server**, from users perspective while they submit the searched query, they assume google web server as a single system.
- ▶ Just visit google.com, then search.
- ▶ However Google builds a lot of servers even distributes in different geographical area to give you a search result within few seconds.
- ▶ Amazon Platforms
- ▶ Blockchain

Examples of Distributed Systems



Telephone networks and cellular networks



ATM machines



Computer network such internet



Mobile
Computing

Examples of Distributed Systems

▶ Web Search Engines:

- Major growth industry in the last decade.
- 10 billion per month for global number of searches.
- e.g. Google distributed infrastructure



▶ Massively multiplayer online games:

- Large number of people interact through the Internet with a virtual world.
- Challenges include fast response time, real-time propagation of events.



Goals of Distributed Systems

- ▶ Connect Users and Resources
- ▶ Transparency
- ▶ Openness
- ▶ Be Scalable:
 - in size
 - geographically
 - administratively

Connect Users and Resources

- ▶ The main goal of a distributed system is to make it easy for the users (and applications) to access remote resources, and to share them in a controlled and efficient way.
- ▶ Resources can be just about anything, but typical examples include things like printers, computers, storage facilities, data, files, Web pages, and networks
- ▶ There are many reasons for wanting to share resources.
- ▶ One obvious reason is that of economics. For example, it is cheaper to let a printer be shared by several users in a 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.

Transparency

- ▶ To hide the fact that its processes and resources are physically distributed across multiple computers.
- ▶ A distributed system that is able to present itself to users and applications as if it were only a single computer system is said to be transparent.
- ▶ **Access transparency** deals with hiding differences in data representation and the way that resources can be accessed by users.
- ▶ 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, as well as how files can be manipulated, should all be hidden from users and applications.

- ▶ An important group of transparency types has to do with the location of a resource. **Location transparency** refers to the fact that users cannot tell where a resource is physically located in the system.
- ▶ Distributed systems in which resources can be moved without affecting how those resources can be accessed are said to provide **migration transparency**.
- ▶ The situation in which resources can be relocated while they are being accessed without the user or application noticing anything. In such cases, the system is said to support **relocation transparency**.
- ▶ Resources may be replicated to increase availability or to improve performance by placing a copy close to the place where it is accessed. **Replication transparency** deals with hiding the fact that several copies of a resource exist.

Transparency in a Distributed System

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

Different forms of transparency in a distributed system

Openness

- ▶ An open distributed system is a system that **offers services according to standard rules** that describe syntax and semantics of the services.
- ▶ In distributed system, services are generally specified through interfaces, which are often described in an Interface Definition Language (IDL).
- ▶ Interface definitions written in an IDL nearly always capture only the syntax of services. In other words, they specify precisely the names of the functions that are available together with types of the parameters, return values, possible exceptions that can be raised.
- ▶ It should be easy to configure the system out of different components.

Distributed System Goals



Scalability

A system is said to be scalable if it can handle the addition of users and resources without suffering a noticeable loss of performance or increase in administrative complexity

- ▶ **Scale has three dimensions:**

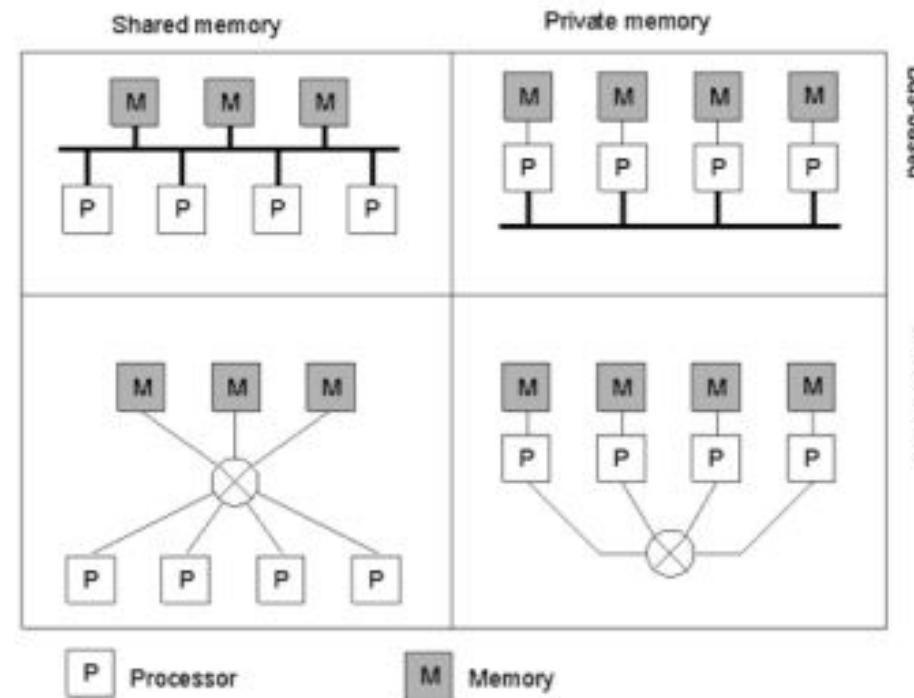
- **Size:** number of users and resources . we can easily add more users and resources to the system.(problem: overloading)
- **Geography:** In which the users and resources may lie far apart. (problem: communication)
- **Administration:**it can still be easy to manage even if it spans many independent administrative organizations. (problem: administrative mess)

DISTRIBUTED SYSTEM Concepts

Hardware Concepts	Software Concepts
Multicomputer	Uniprocessor OS
Multiprocessor	Multiprocessor OS
	Network OS (NOS)
	Distributed OS (DOS)
	Middleware

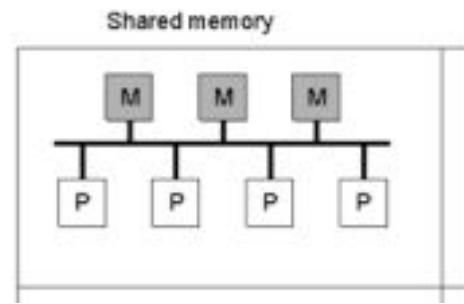
Multiprocessor and multicomputers

- ▶ Computers that have shared memory is called **Multiprocessors**.
- ▶ Those that do not share is called **Multicomputers**.
- ▶ In Multiprocessor , if any CPU writes value 44 to address 100,any other CPU subsequently reading from its address 100 will get the value 44.
- ▶ In Multicomputer.....

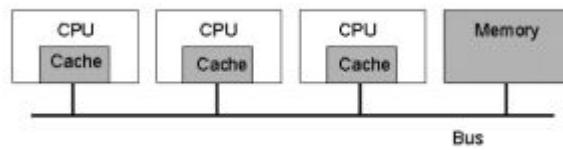


Multiprocessor

► Coherent Memory

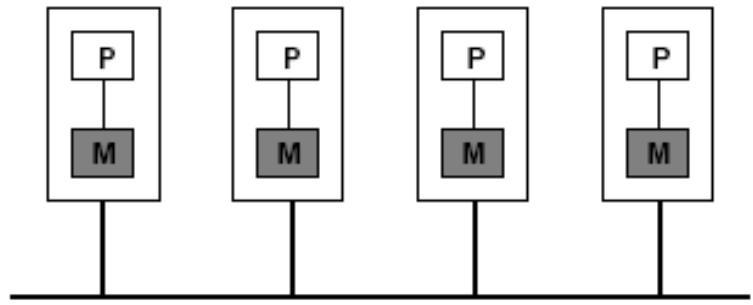


- 4-5 CPU will saturate the BUS
- Add cache memory between the CPU and bus



► Incoherent Memory

Multicomputer



- ▶ Homogeneous (all nodes support same physical architecture)
- ▶ Heterogeneous (does not support same physical architecture)
 - Computers that are part of the system may vary with respect to processor type, memory sizes and I/O bandwidth.

Software Concept

- ▶ Mostly the software determines what a distributed system actually looks like
- ▶ Distributed systems are very much like traditional operating systems.
- ▶ Distributed systems act as a resource managers for the underlying hardware
 - Allowing multiple users and applications to share resources
 - Hide the heterogeneous nature of the underlying hardware.
 - By providing a virtual machine on which applications can be easily executed

Software Concept

- ▶ Operating systems for distributed computers can be roughly divided into two categories:
 - Tightly coupled systems
 - The OS maintain a single, global view of the resources
 - Generally referred to as a distributed operating system (DOS)
 - Used for managing **multiprocessors and homogeneous multicollectors.**
 - Loosely-coupled systems
 - A collection of computers each running their own operating system.
 - However, these operating systems work together to make their own services and resources available to the others.
 - The **loosely-coupled network operating system (NOS) is used for heterogeneous multicollector systems.**

Software Concept

► Middleware

- Enhancement to the services of the network operating system are needed so that better support for distribution transparency can be provided.
- These enhancements leads to **Middleware** which lie at the heart of modern distributed system.

Software Concept

System	Description	Main Goal
DOS	Tightly-coupled operating system for multi-processors and homogeneous multicomputers	Hide and manage hardware resources
NOS	Loosely-coupled operating system for heterogeneous multicomputers (LAN and WAN)	Offer local services to remote clients
Middleware	Additional layer at top of NOS implementing general-purpose services	Provide distribution transparency

Distributed Operating System

- ▶ Two types of distributed operating systems.
 - A multiprocessor operating system
 - Manages the resources of a multiprocessor.
 - A multicenter operating system
 - An operating system that is developed for homogeneous multicenters

Uniprocessor OS

- ▶ To allow users and applications an easy way of sharing resources such as CPU, main memory, disks and peripheral devices.
- ▶ Sharing resources means that different applications make use of same hardware in an isolated fashion.
- ▶ To an application, it appears as if it has its own resources and that there may be several applications executing on same system at the same time each with their own set of resources.
- ▶ Communication Primitives
- ▶ Kernel Mode
- ▶ User Mode

Multiprocessor Operating System

- ▶ Multiprocessor- uses different system services to manage resources connected in a system and use system calls to communicate with the processor.
- ▶ Multiple processors with shared memory
 - Support for multiple processors having access to a shared memory.
- ▶ Problem: consistency
- ▶ Solutions: synchronization
 - Semaphore
 - Monitor: programming-language concept since semaphore is error-prone

Distributed Operating System(DOS)

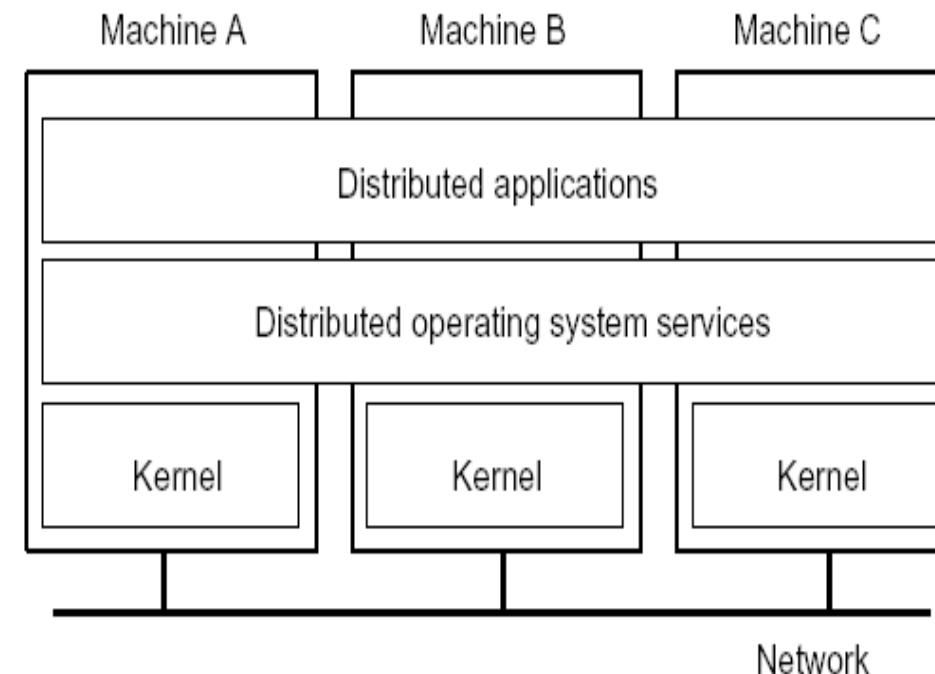
- ▶ A tightly coupled operating system is generally referred to as a distributed operating system (DOS).
- ▶ In distributed OS, a common set of services is shared among multiple processors in such a way that they are meant to execute a distributed application effectively and also provide services to separate independent computers connected in a network as shown in fig below
- ▶ It communicates with all the computer using message passing interface(MPI).
- ▶ It uses Data structure like queue to manages the messages and avoid message loss between sender and receiver computer.
- ▶ Eg Automated banking system, railway reservation system etc.

Distributed Operating System(DOS)

- ▶ Users not aware of multiplicity of machines
 - Access to remote resources similar to access to local resources
 - ▶ High degree of transparency (single system image)
 - ▶ **Homogeneous hardware**
-
- ▶ **Disadvantages:**
 - ▶ It has a problem of scalability as it supports only limited number of independent computers with shared resources.
 - ▶ There is need to define message passing semantics prior to the execution of messages.

Distributed Operating System(DOS)

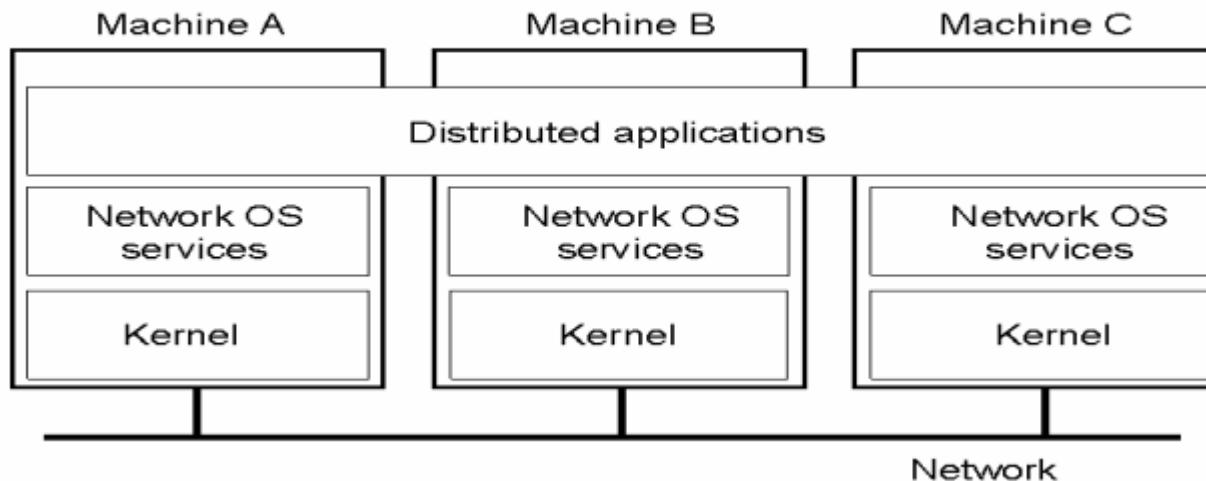
- ▶ Each node has its own kernel containing modules for managing local resources.
- ▶ Above each layer of kernel is a common layer of software that implements the operating system as virtual machine supporting parallel and concurrent execution of various tasks.
- ▶ Apart from that it supports distributed services (services may include, assignment of tasks to processors, masking hardware failure, transparent storage, interprocess communication, etc.)



Network Operating Systems

- ▶ NOS does not assume underlying hardware is homogeneous and be managed as a single system
 - Different from distributed OS
- ▶ NOS is constructed from a collection of uniprocessor systems, each with its own operating systems as shown in figure.
- ▶ The machine and their os may be different but they are connected in a network.
- ▶ Network operating system provides facilities to allow users to make use of the services available on a specific machine.
- ▶ Example,
 - To allow a user to log into another machine remotely by using command rlogin machine.
 - Command to copy files from one machine to another machine.

Network Operating Systems

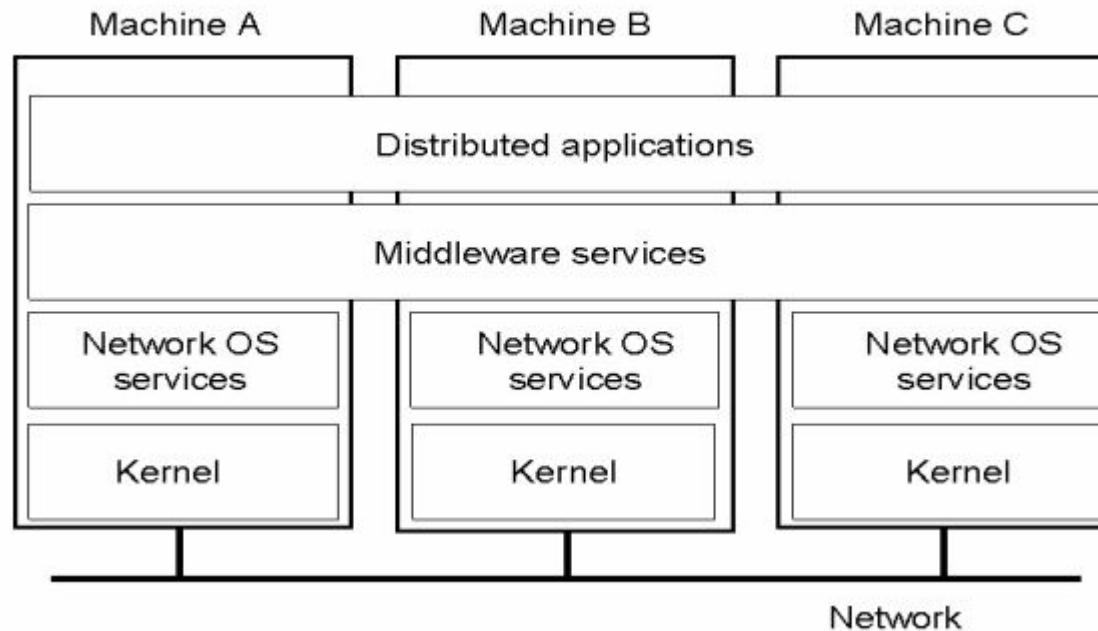


- ▶ Disadvantages
 - Management problem because all machine in nos are independent
- ▶ Difference between NOS and DOS
 - DOS attempts to realize full transparency and provide a single-system view
 - Transparency is missing in NOS

Middleware

- ▶ Neither a distributed operating system or a network operating system really qualifies as a distributed system according to the definition.
- ▶ **Distributed operating system is not intended to handle a collection of *independent* computers, while a network operating system does not provide a view of a *single coherent system*.**
- ▶ *The question comes to mind whether it is possible to develop a distributed system that has the best of both words: the scalability of network operating systems and the transparency and related ease of use of distributed operating systems.*
- ▶ The **solution** is to be found in an additional layer of software that is used in network operating systems to more or less hide the heterogeneity of the collection of underlying platforms but also to improve distribution transparency.
- ▶ Many modern distributed systems are constructed by means of such an additional layer of what is called **middleware**.

Middleware



- ▶ It has a common set of services is provided for the local applications and independent set of services for the remote applications.
- ▶ It provide the services such as locating the objects or interfaces by their names, finding the location of objects, maintaining the quality of services, handling the protocol information, synchronization, concurrency and security of the objects etc.

Comparison between systems

Item	Distributed OS		Network OS	Middleware-based OS
	Multiproc.	Multicomp.		
Degree of transparency	Very High	High	Low	High
Same OS on all nodes	Yes	Yes	No	No
Number of copies of OS	1	N	N	N
Basis for communication	Shared memory	Messages	Files	Model specific
Resource management	Global, central	Global, distributed	Per node	Per node
Scalability	No	Moderately	Yes	Varies
Openness	Closed	Closed	Open	Open

Architectural Styles

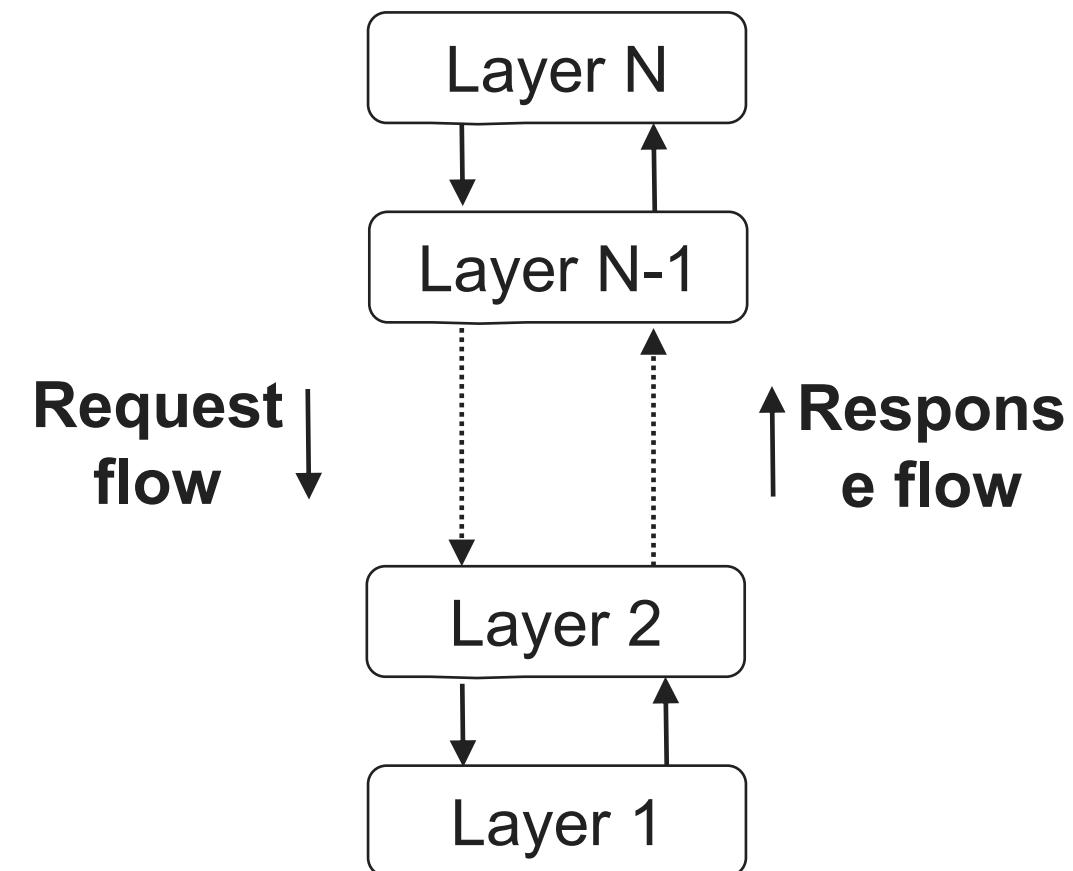
- ▶ Distributed systems are often complex pieces of software of which the components are by definition dispersed across multiple machines.
 - ▶ To master their complexity, it is crucial that these systems are properly organized.
 - ▶ There are different ways on how to view the organization of a distributed system, but an obvious one is to **make a distinction between the logical organization of the collection of software components and on the other hand the actual physical realization.**
-
- ▶ **Software Architecture**
 - ▶ **System Architecture**

Architectural Styles

- An **architectural style** describes a particular way to configure a collection of components and connectors.
 - **Component** - a module with well-defined interfaces; reusable, replaceable
 - **Connector** – communication link between modules
 - RPC, Message Passing or streaming data
- ▶ Important styles of software architecture for distributed systems:
 - Layered architectures
 - Object-based architectures
 - Data-centered architectures
 - Event-based architectures

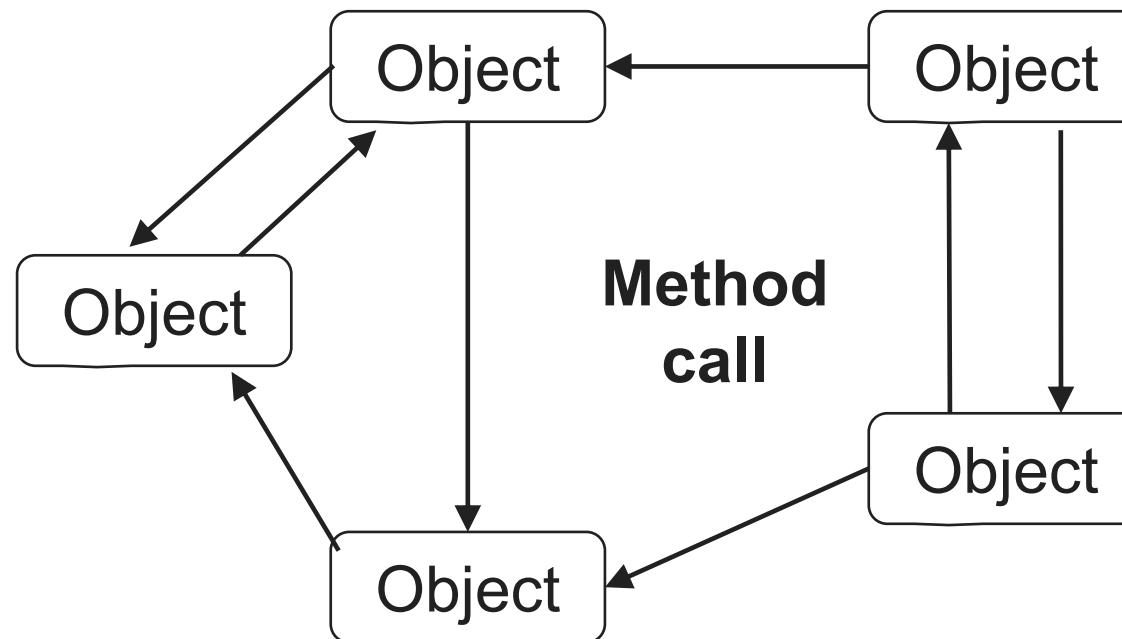
Layered architectures

- ▶ The Components are organized in a layered fashion where a component at layer L_i is allowed to call components at the underlying layer L_{i-1} , but not the other way around,
- ▶ This model has been widely adopted by the networking community
- ▶ A key observation is that control generally flows from layer to layer; requests go down the hierarchy whereas the results flow upward.



Object-based architectures

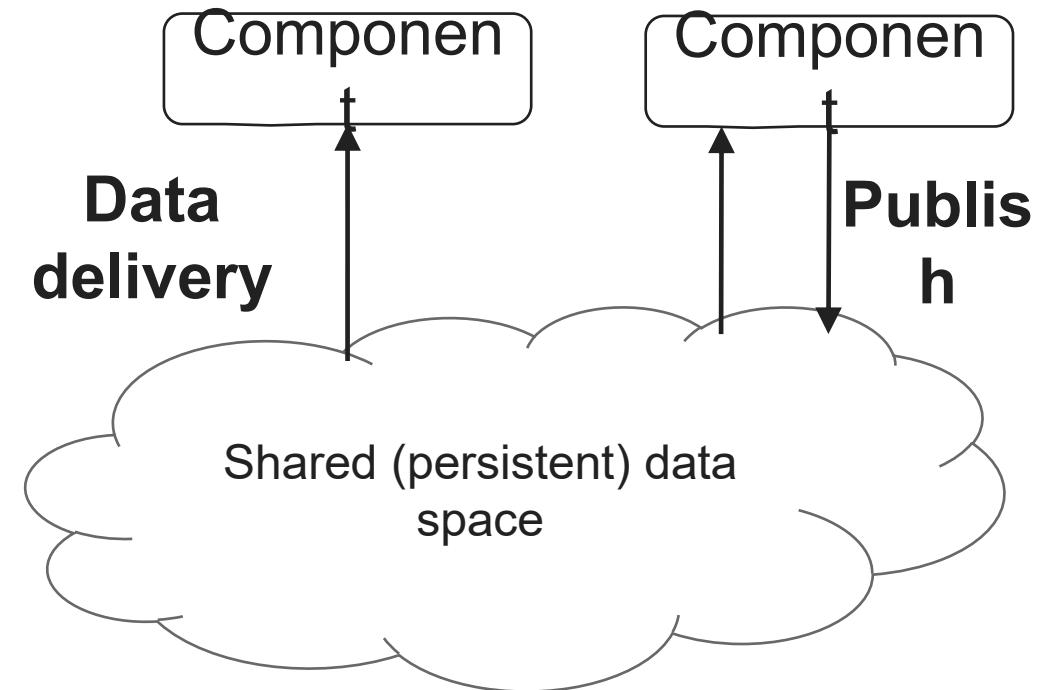
- ▶ Each object corresponds a component
- ▶ Components are connected through a (remote) procedure call mechanism.
- ▶ The layered and object-based architectures still form the most important styles for large software systems



component = object
connector = RPC or RMI

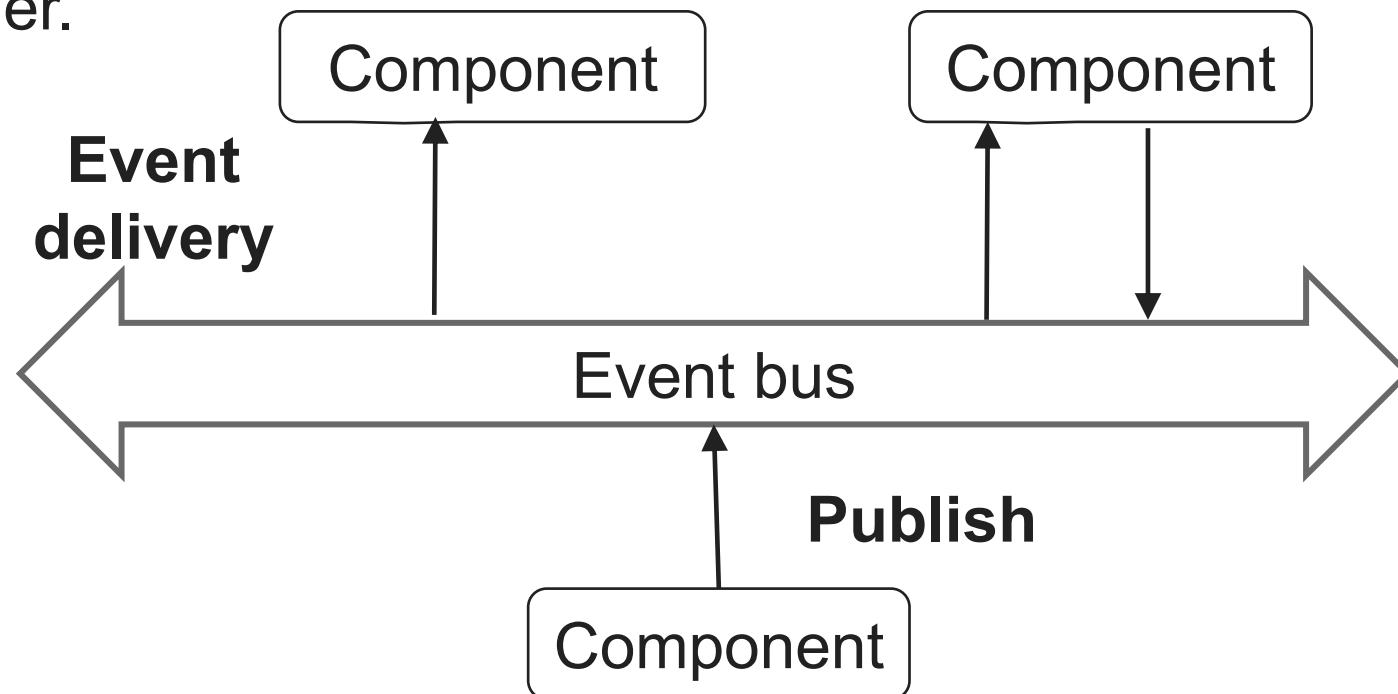
Data-centered architectures

- ▶ It evolves around the idea that processes communicate through a common (passive or active) repository.
- ▶ It can be argued that for distributed systems these architectures are as important as the layered and object-based architectures
- ▶ For example,
 - A wealth of networked applications have been developed that rely on a shared distributed file system in which virtually all communication takes place through files.
 - Web-based distributed systems are largely data-centric: processes communicate through the use of shared Web-based data services.



Event-based architectures

- ▶ Processes communicate through the propagation of events.
- ▶ For distributed systems, event propagation has generally been associated with what are known as publish/subscribe systems.
- ▶ Middleware
- ▶ **Advantage** - processes are loosely coupled. In principle, they need not explicitly refer to each other.



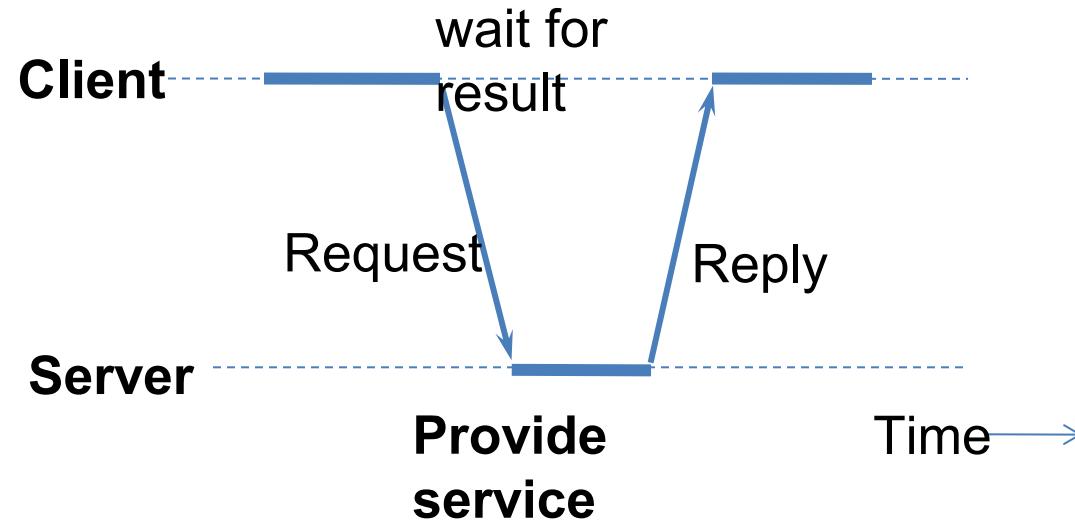
System Architecture

There are three views toward system architectures:

- ▶ Centralized Architectures
- ▶ Decentralized architectures
- ▶ Hybrid Architectures

Centralized Architectures

- ▶ Manage distributed system complexity – think in terms of clients that request services from servers.
- ▶ Processes are divided into two groups:
 1. A server is a process implementing a specific service, for example, a file system service or a database service.
 2. A client is a process that requests a service from a server by sending it a request and subsequently waiting for the server's reply.



Centralized Architectures

- ▶ **Communication** - implemented using a connectionless protocol when the network is reliable e.g. local-area networks.
 1. **Client requests a service** –sends a message for the server, identifying the service it wants, along with the necessary input data.
 2. **The Server will always wait for an incoming request**, process it, and package the results in a reply message that is then sent to the client.

Client Server

```
/* Definitions needed by clients and servers. */  
#define TRUE 1  
#define MAX_PATH 255 /* maximum length of file name */  
#define BUF_SIZE 1024 /* how much data to transfer at once */  
#define FILE_SERVER 243 /* file server's network address */  
  
/* Definitions of the allowed operations */  
#define CREATE 1 /* create a new file */  
#define READ 2 /* read data from a file and return it */  
#define WRITE 3 /* write data to a file */  
#define DELETE 4 /* delete an existing file */  
  
/* Error codes. */  
#define OK 0 /* operation performed correctly */  
#define E_BAD_OPCODE -1 /* unknown operation requested */  
#define E_BAD_PARAM -2 /* error in a parameter */  
#define E_IO -3 /* disk error or other I/O error */  
  
/* Definition of the message format. */  
struct message {  
    long source; /* sender's identity */  
    long dest; /* receiver's identity */  
    long opcode; /* requested operation */  
    long count; /* number of bytes to transfer */  
    long offset; /* position in file to start I/O */  
    long result; /* result of the operation */  
    char name[MAX_PATH]; /* name of file being operated on */  
    char data[BUF_SIZE]; /* data to be read or written */  
};
```

- A sample server.

```
#include <header.h>
void main(void) {
    struct message ml, m2;          /* incoming and outgoing messages */
    int r;                          /* result code */

    while(TRUE) {                   /* server runs forever */
        receive(FILE_SERVER, &ml);   /* block waiting for a message */
        switch(ml.opcode) {          /* dispatch on type of request */
            case CREATE:           r = do_create(&ml, &m2); break;
            case READ:              r = do_read(&ml, &m2); break;
            case WRITE:              r = do_write(&ml, &m2); break;
            case DELETE:             r = do_delete(&ml, &m2); break;
            default:                r = E_BAD_OPCODE;
        }
        m2.result = r;               /* return result to client */
        send(ml.source, &m2);       /* send reply */
    }
}
```

- A client using the server to copy a file.

```
#include <header.h>          (a)
int copy(char *src, char *dst){
    struct message ml;
    long position;
    long client = 110;

    initialize();           /* prepare for execution */
    position = 0;
    do {
        ml.opcode = READ;   /* operation is a read */
        ml.offset = position; /* current position in the file */
        ml.count = BUF_SIZE; /* how many bytes to read*/
        strcpy(&ml.name, src); /* copy name of file to be read to message */
        send(FILESERVER, &ml); /* send the message to the file server */
        receive(client, &ml); /* block waiting for the reply */

        /* Write the data just received to the destination file. */
        ml.opcode = WRITE;   /* operation is a write */
        ml.offset = position; /* current position in the file */
        ml.count = ml.result; /* how many bytes to write */
        strcpy(&ml.name, dst); /* copy name of file to be written to buf */
        send(FILE_SERVER, &ml); /* send the message to the file server */
        receive(client, &ml); /* block waiting for the reply */
        position += ml.result; /* ml.result is number of bytes written */
    } while( ml.result > 0 );
    return(ml.result >= 0 ? OK : ml.result);
}
```

Application Layering

Traditional three-layered view:

1. The user-interface level

- It contains units for an application's user interface
- Clients typically implement the user-interface level
 - Simple GUI

2. The processing level

- It contains the functions of an application, i.e. without specific data
- Middle part of hierarchy -> logically placed at the processing level

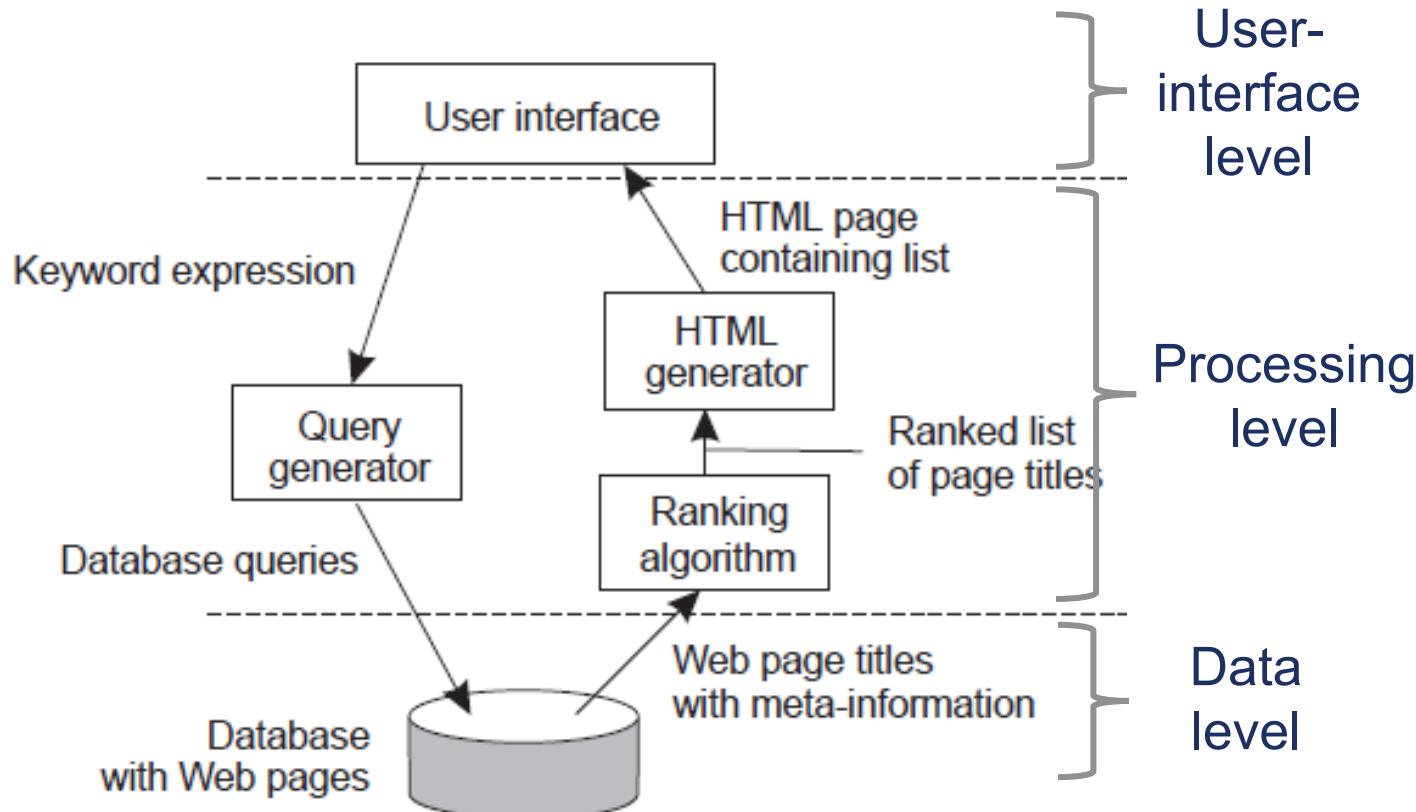
3. The data level

- It contains the data that a client wants to manipulate through the application components
- manages the actual data that is being acted on

Internet search engine- An example of Application Layering

Traditional three-layered view:

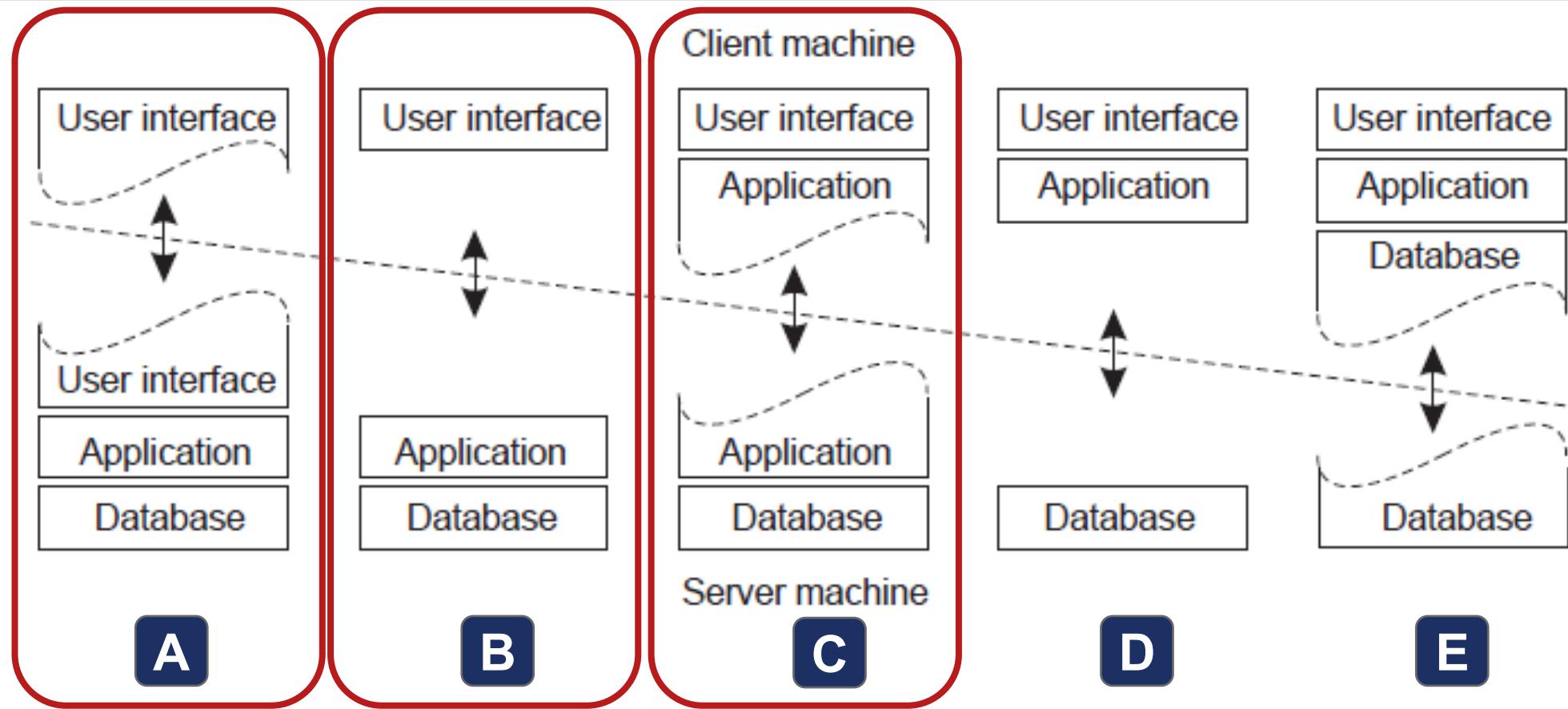
- ▶ **User-interface level:** a user types in a string of keywords and is subsequently presented with a list of titles of Web pages.
- ▶ **Data Level:** huge database of Web pages that have been prefetched and indexed.
- ▶ **Processing level:** search engine that transforms the user's string of keywords into one or more database queries.
 - Ranks the results into a list
 - Transforms that list into a series of HTML pages



Multi-Tiered Architectures

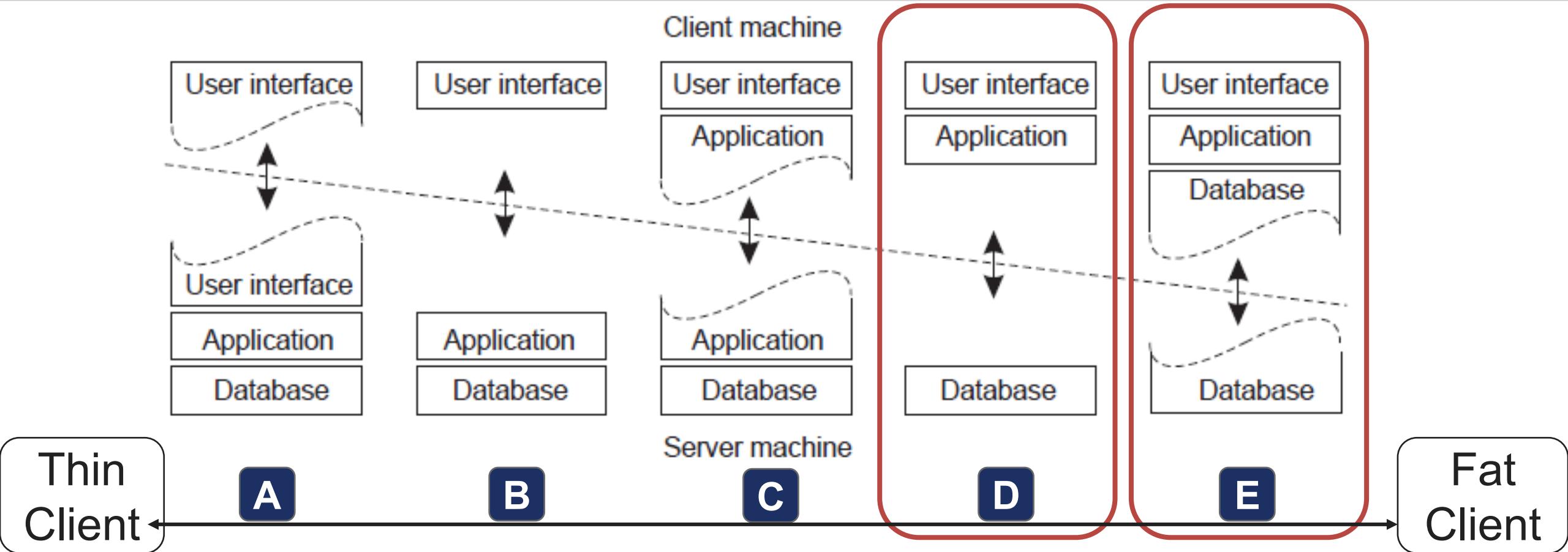
- ▶ Simplest organization - two types of machines:
 1. A client machine containing only the programs implementing (part of) the user-interface level
 2. A server machine containing the rest, that is the programs implementing the processing and data level
- ▶ Single-tiered: dumb terminal/mainframe configuration
- ▶ Two-tiered: client/single server configuration
- ▶ Three-tiered: each layer on separate machine

Two-tiered Architectures - Thin-client model and fat-client model



- ▶ Case-A: Only the terminal-dependent part of the user interface
- ▶ Case- B: Place the entire user-interface software on the client side
- ▶ Case- C: Move part of the application to the front end

Two-tiered Architectures - Thin-client model and fat-client model



- ▶ Case-D: Used where the client machine is a PC or workstation, connected through a network to a distributed file system or database. Banking Application
- ▶ Case- E: Local disk

Thin client model

- ▶ Used when legacy systems are migrated to client server architectures.
- ▶ The legacy system acts as a server in its own right with a graphical interface implemented on a client.
- ▶ A major disadvantage is that it places a heavy processing load on both the server and the network.

Fat Client Model

- ▶ More processing is delegated to the client as the application processing is locally executed.
- ▶ Most suitable for new C/S systems where the capabilities of the client system are known in advance.
- ▶ More complex than a thin client model especially for management. New versions of the application have to be installed on all clients.

Multitiered Architectures (3-Tier Architecture)

**Data Tier
(Back-End)**



Database

**Middle Tier
(Business Tier)**



**Business
Logic**

Client Tier (Front-End)

**Client
Machine**



network

**Mobile
Client**



network

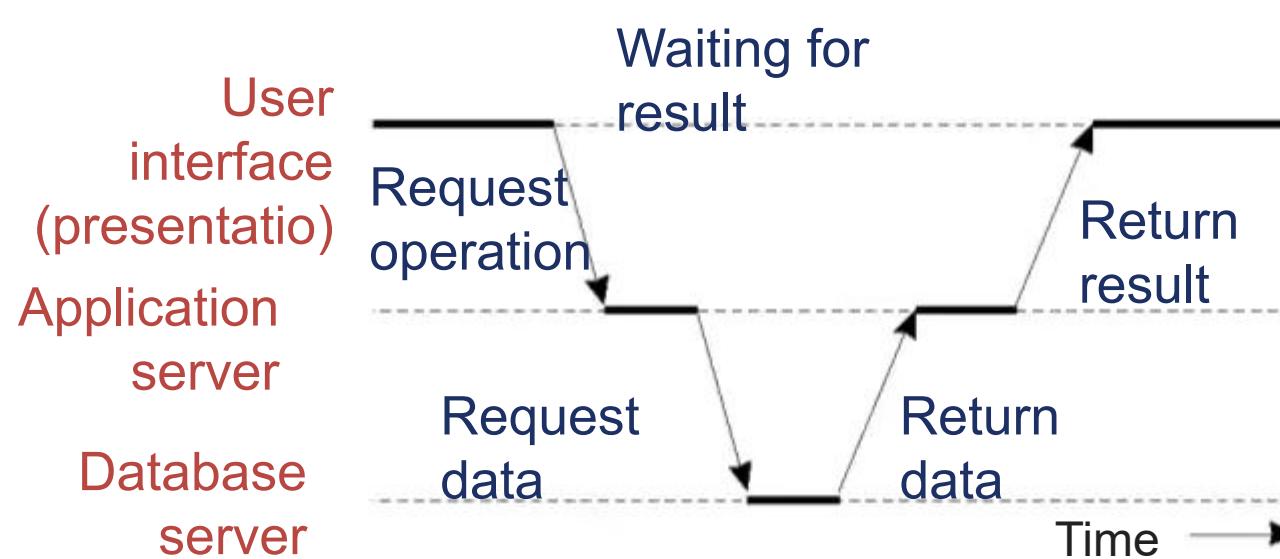
**Desktop
Client**



network

Multitiered Architectures (3-Tier Architecture)

- ▶ The server tier in two-tiered architecture becomes more and more distributed
- ▶ Distributed transaction processing
 - A single server is no longer adequate for modern information systems
- ▶ This leads to three-tiered architecture
 - Server may act as a client



An example
of a
server acting
as client

Architecture	Applications
Two-tier C/S architecture with thin clients	<p>Legacy system applications where separating application processing and data management is impractical.</p> <p>Computationally-intensive applications such as compilers with little or no data management.</p> <p>Data-intensive applications (browsing and querying) with little or no application processing.</p>
Two-tier C/S architecture with fat clients	<p>Applications where application processing is provided by off-the-shelf software (e.g. Microsoft Excel) on the client.</p> <p>Applications where computationally-intensive processing of data (e.g. data visualisation) is required.</p> <p>Applications with relatively stable end-user functionality used in an environment with well-established system management.</p>
Three-tier or multi-tier C/S architecture	<p>Large scale applications with hundreds or thousands of clients</p> <p>Applications where both the data and the application are volatile.</p> <p>Applications where data from multiple sources are integrated.</p>

Decentralized Architectures

- ▶ Multi-tiered architectures can be considered as vertical distribution
 - Placing logically different components on different machines
- ▶ An alternative is horizontal distribution (peer-to-peer systems)
 - A collection of logically equivalent parts
 - Each part operates on its own share of the complete data set, balancing the load
- ▶ Peer-to-peer architectures - how to organize the processes in an **overlay network** in which the nodes are formed by the processes and the links represent the possible communication channels (which are usually realized as TCP connections).
- ▶ Decentralized Architectures Types
 1. Structured P2P
 2. Unstructured P2P

Structured P2P Architectures

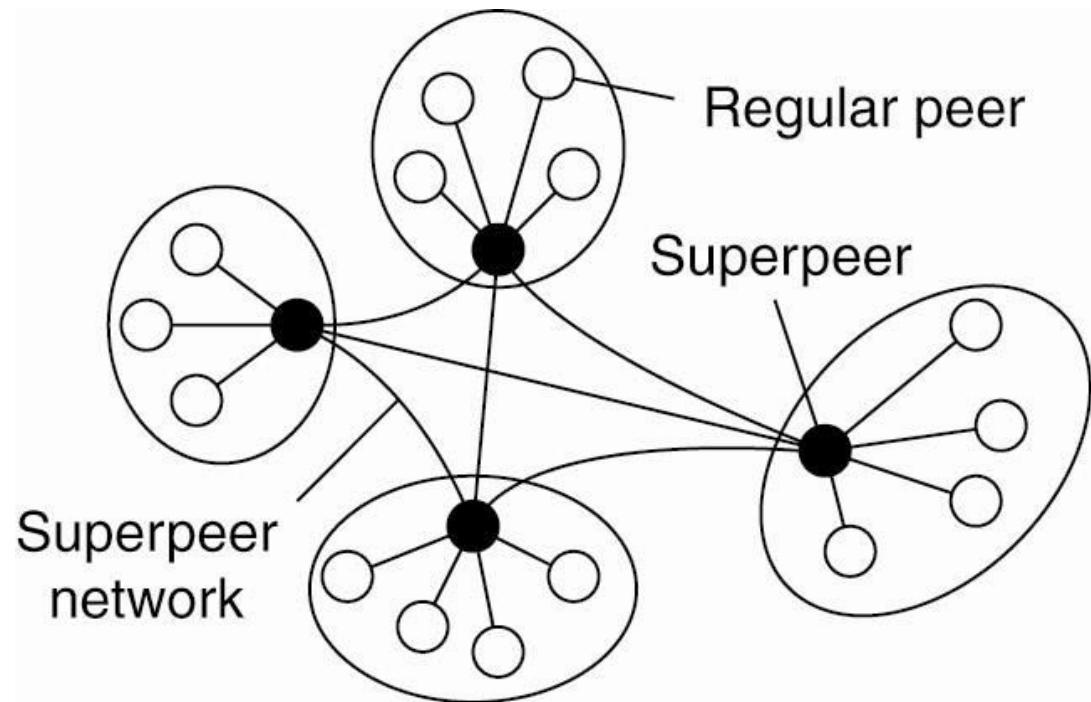
- ▶ There are links between any two nodes that know each other
- ▶ **Structured**: the overlay network is constructed in a deterministic procedure
 - Most popular: distributed hash table (DHT)
- ▶ DHT-based system
 - Data items are assigned a random key from a large identifier space, such as a 128-bit or 160-bit identifier.

Unstructured P2P Architectures

- ▶ Largely relying on randomized algorithm to construct the overlay network
 - Each node has a list of neighbors
- ▶ Many systems try to construct an overly network that resembles a random graph
 - Each node maintains a partial view, i.e., a set of live nodes randomly chosen from the current set of nodes constructed in a random way
- ▶ An unstructured P2P network is formed when the overlay links are established arbitrarily.
- ▶ Data items are randomly mapped to some node in the system & lookup is random.
- ▶ In an unstructured P2P network, if a peer wants to find a desired piece of data in the network, **the query has to be flooded** through the network in order to find as many peers as possible that share the data..

Superpeers

- ▶ Used to address the following question
 - How to find data items in unstructured P2P systems
 - Flood the network with a search query?
- ▶ An alternative is using **superpeers**
 - Nodes such as those maintaining an index are generally referred to as superpeers
 - They hold index of info. from its associated peers (i.e. selected representative of some of the peers)
 - Fixed association with superpeers



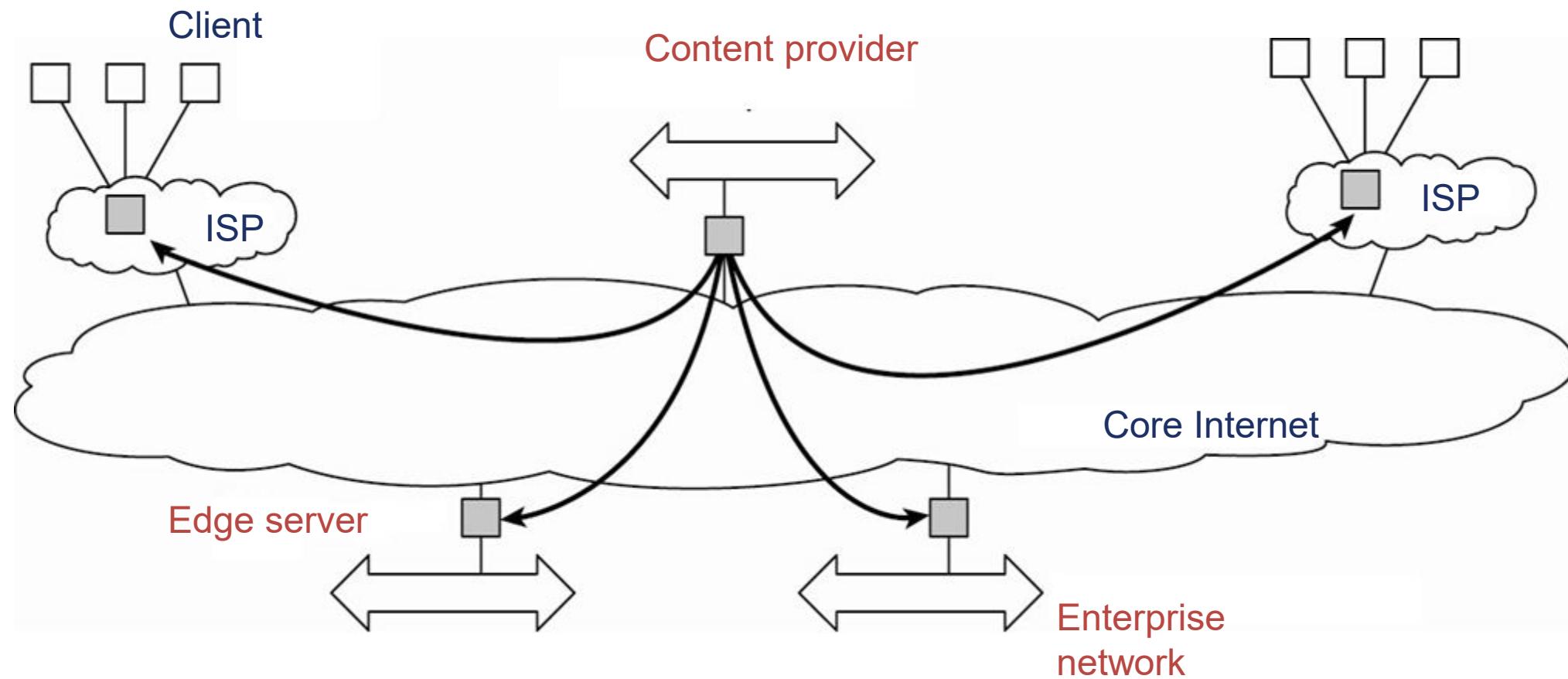
A hierarchical organization of nodes into a superpeer network

Hybrid Architectures

- ▶ Many real distributed systems combine architectural features
 - combine client-server architecture (centralized) with peer-to-peer architecture (decentralized)
- ▶ Two examples of hybrid architectures
 - Edge-server systems
 - Collaborative distributed systems

Edge-Server Systems

- ▶ Deployed on the Internet where servers are “**at the edge**” of the network.
- ▶ Each client connects to the Internet by means of an edge server.

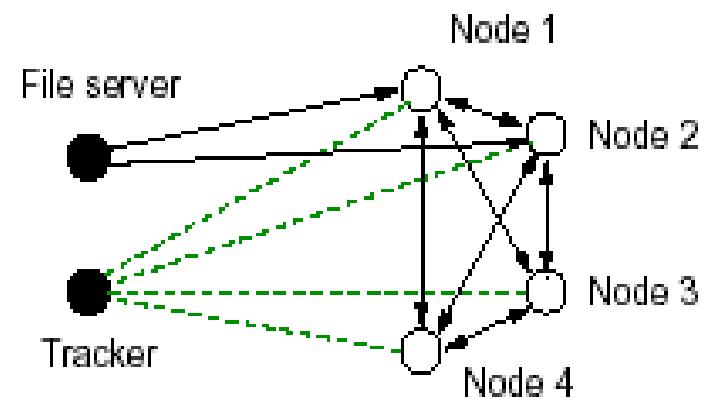


Collaborative Distributed Systems

- ▶ A hybrid distributed model that is based on mutual collaboration of various systems
 - Client-server scheme is deployed at the beginning
 - Fully decentralized scheme is used for collaboration after joining the system
- ▶ Examples of Collaborative Distributed System:
 - **BitTorrent**: is a P2P File downloading system. It allows download of various chunks of a file from other users until the entire file is downloaded
 - **Globule**: A Collaborative content distribution network. It allows replication of web pages by various web servers

Collaborative Distributed Systems

- ▶ In **collaborative distributed systems**, peers typically support each other to deliver content in a peer to peer like architecture, while they use a client server architecture for the initial setup of the network.
- ▶ Nodes requesting to download a file from a server first contact the server to get the location of a tracker.
- ▶ Tracker keeps track of **active nodes** that have chunks of file.
- ▶ Using information from the tracker, clients can download the file in chunks from multiple sites in the network.
- ▶ Nodes must then offer downloaded chunks to other nodes and are registered with the tracker, so that the other nodes can find them.



Communication in a Distributed System

- ▶ In a distributed system, processes run on different machines.
- ▶ Processes can only exchange information through message passing.
 - Harder to program than shared memory communication
- ▶ Successful distributed systems depend on communication models that hide or simplify message passing
- ▶ Communication in distributed systems is always based on low-level message passing as offered by the underlying network.

Layered Network Communication Protocols

Physical layer:

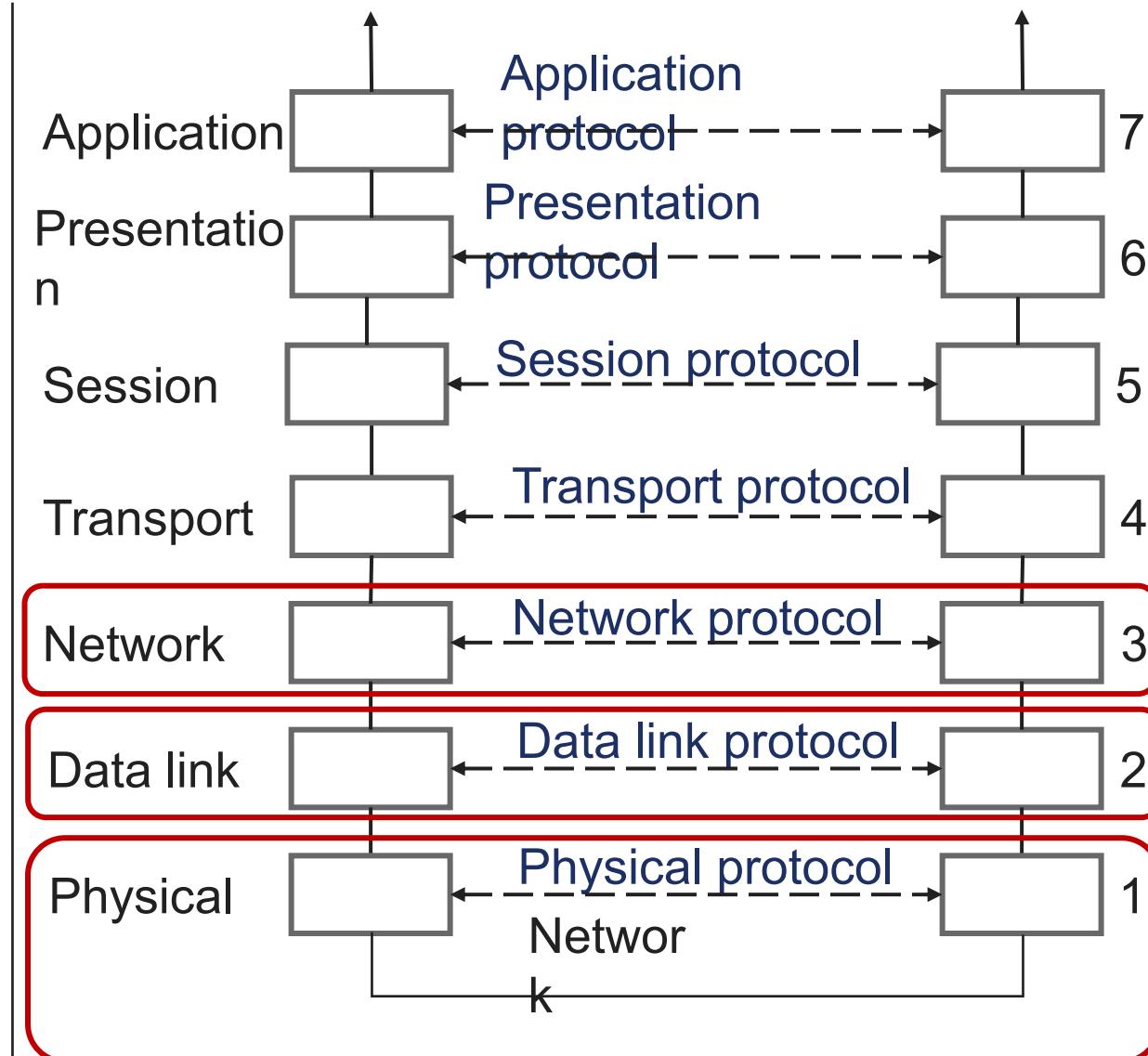
- ▶ Transmitting bits between sender and receiver
- ▶ Functions of a Physical layer: Line Configuration, Data Transmission, Topology, Signals

Data link layer:

- ▶ transmitting frames over a link, error detection and correction
- ▶ Functions of a Datalink layer: Framing, Flow Control, Error Control, Access Control

Network layer:

- ▶ Routing of packets between source host and destination host



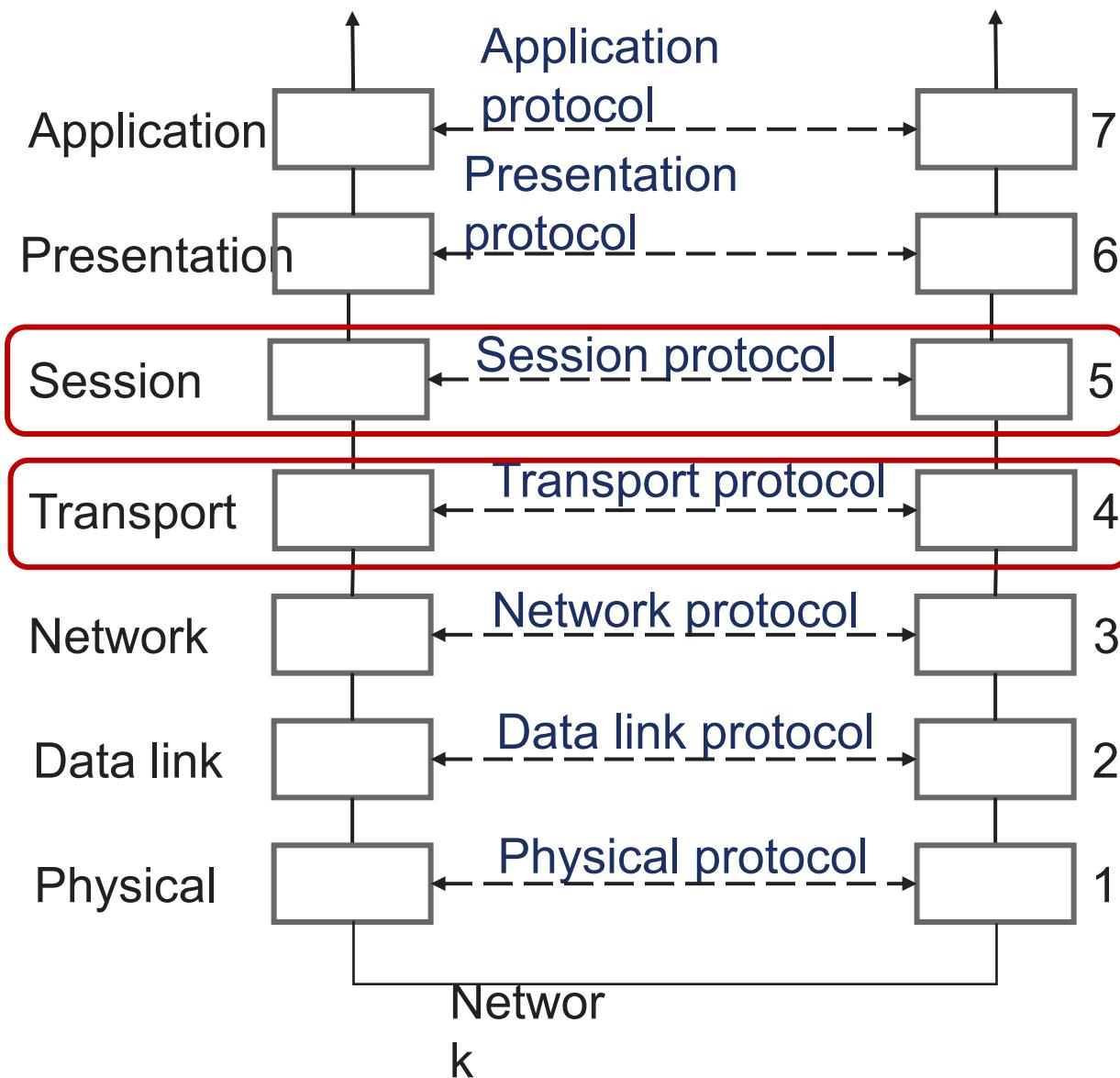
Layered Network Communication Protocols

Transport layer:

- ▶ Process-to-process communication
- ▶ **TCP and UDP** - Internet's transport layer protocols
 - **TCP**: Connection-oriented, Reliable communication
 - **UDP**: Connectionless, Unreliable communication

Session layer:

- ▶ It establishes, manages, and terminates the connections between the local and remote application.
- ▶ Functions of Session layer: Dialog control, Synchronization



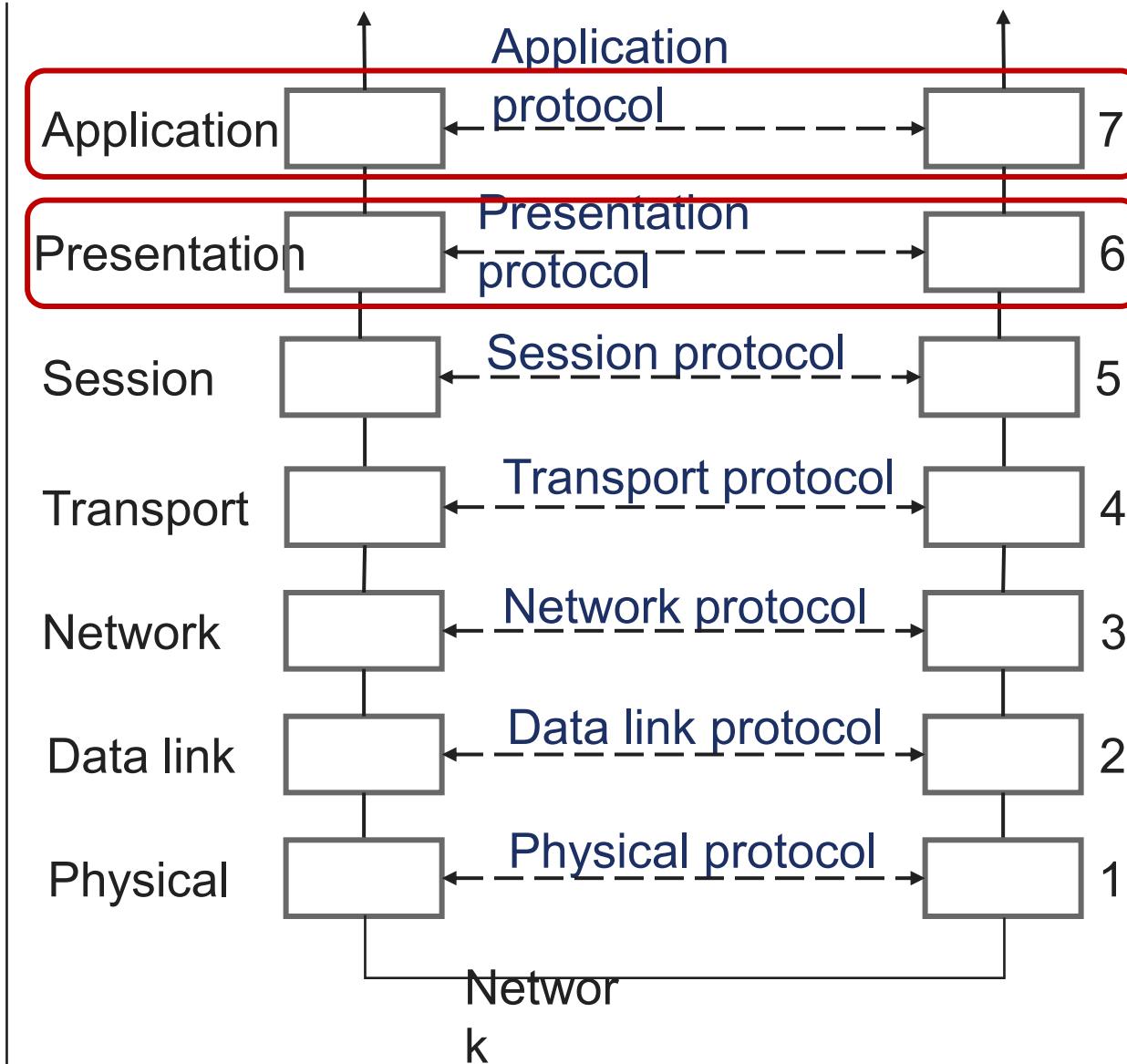
Layered Network Communication Protocols

Presentation layer:

- ▶ Transforms data to provide a standard interface for the Application layer.
- ▶ Functions of Presentation layer:
 - MIME encoding
 - Data compression
 - Data encryption

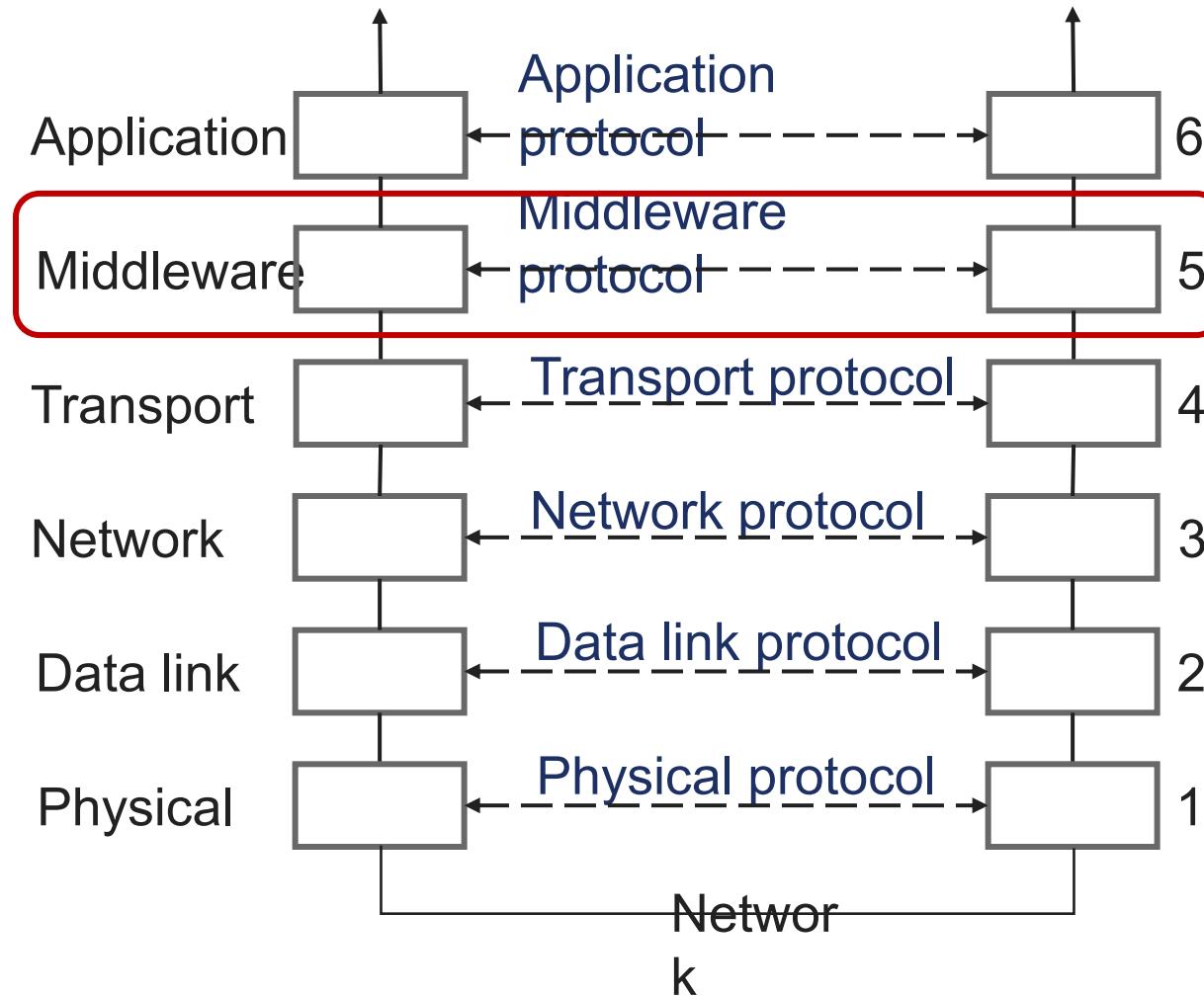
Application layer:

- ▶ Provides means for the user to access information on the network through an application
- ▶ It serves as a window for users and application processes to access network service.

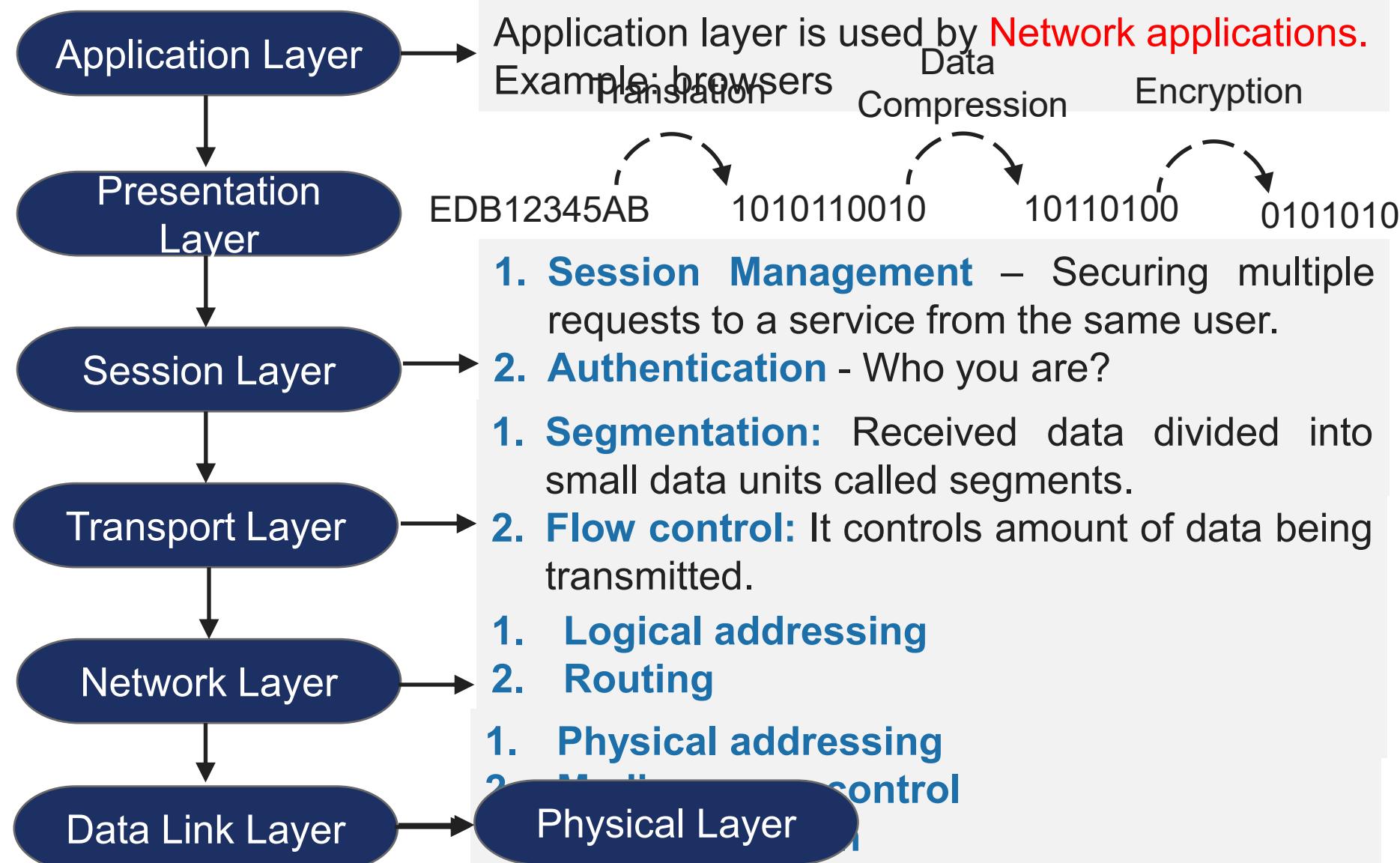


Middleware Layer

- ▶ **Middleware** provides common services and protocols that can be used by many different applications
 - High-level communication services, e.g., RPC, multicasting
 - Security protocols, e.g., authentication protocols, authorization protocols
 - Distributed locking protocols for mutual exclusion
 - Distributed commit protocols



OSI Model



OSI Model (Layers & Activities)

OSI Layers	Activities
Application	To allow access to network resources.
Presentation	To translate, compress, and encrypt/decrypt data.
Session	To establish, manage, and terminate session.
Transport	To provide reliable process-to-process message delivery and error recovery.
Network	To move packets from source to destination; To provide internetworking.
Data Link	To organize bits into frames; To provide hop-to-hop delivery.
Physical	To transmit bits over a medium; To provide mechanical and electrical specifications.

COMMUNICATION

- In distributed system processes running on separate computers cannot directly access each other's memory.

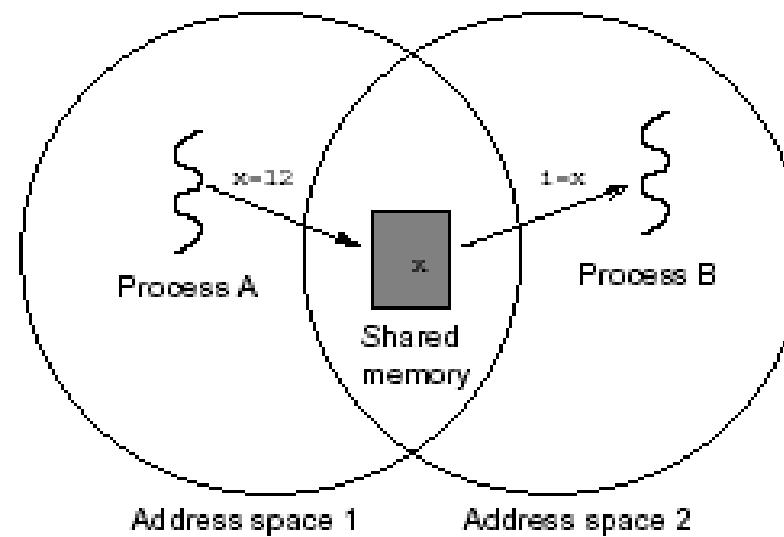
- Two approaches to communication:

Shared memory-processes must have access to some form of shared memory (i.e., they must be threads, they must be processes that can share memory, or they must have access to a shared resource, such as a file)

Message passing- processes communicate by sending each other messages

Shared Memory

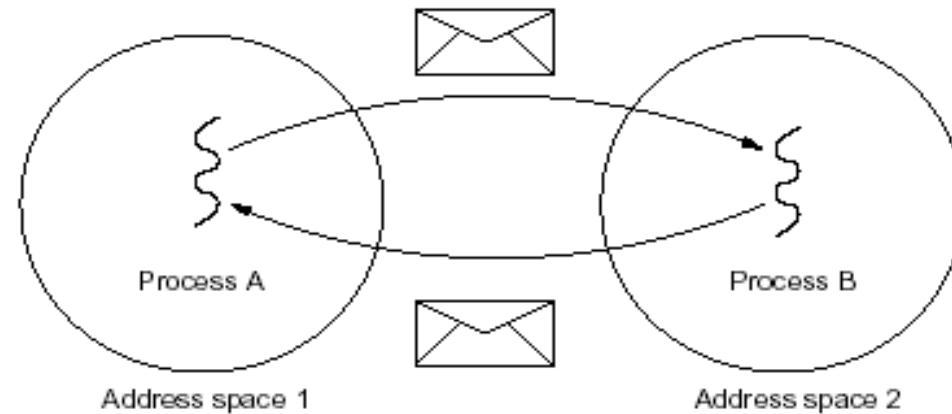
- Shared Memory:
 - There is no way to physically share memory
 - Distributed Shared Memory



Message Passing

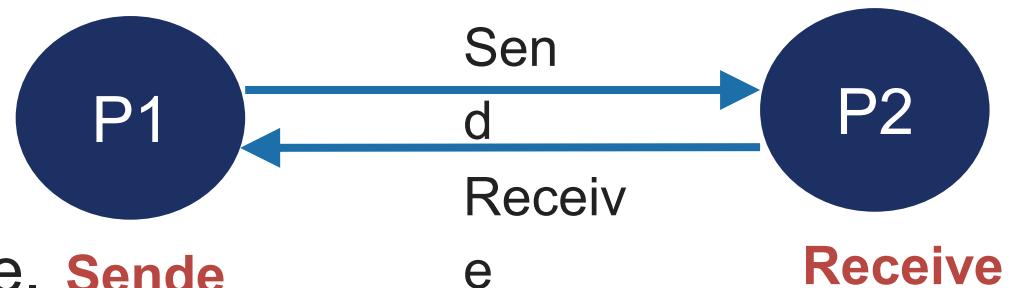
■ Message Passing:

- Over the network
- Introduces higher chances of failure
- Heterogeneity introduces possible incompatibilities



Message Passing

- ▶ It refers to means of communication between
 - Different thread with in a process .
 - Different processes running on same node.
 - Different processes running on different node. **Send** **Receive**
- ▶ In this a sender or a source process send a **r** message to a non receiver or destination process.
- ▶ Message has a predefined structure and message passing uses two system call:
Send and Receive
 - **send(name of destination process, message)**
 - **receive(name of source process, message)**



Client Server Model

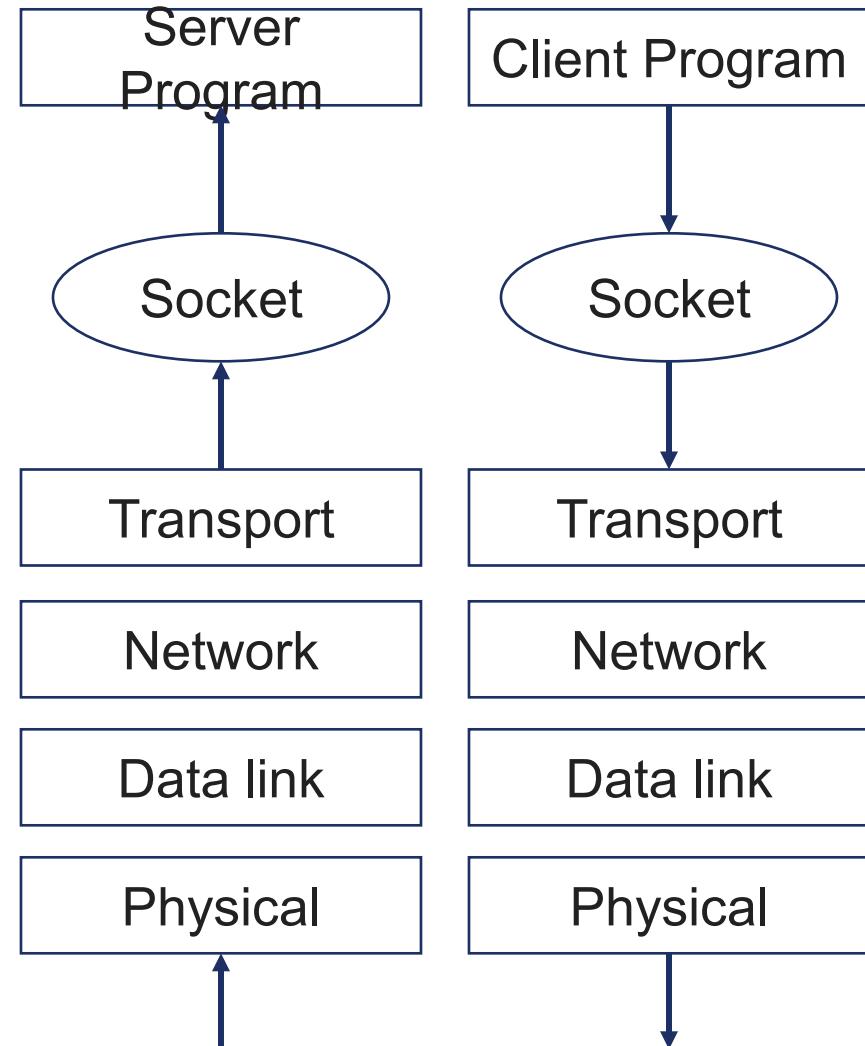
- ▶ Following are different types of packets transmitted across the network.
 - **REQ:** Request packet is used to **send the request** from the client to the server.
 - **Reply:** This message is used to **carry the result** from the server to the client.
 - **ACK:** Acknowledgement packet is used to send the **correct receipt** of the packet to the sender.
 - **Are You Alive (AYA)?:** This packet is sent in case the server takes a long time to complete the client's request.
 - **I am Alive (IAA):** The server, if active, replies with this packet.

Client Server Model Interaction

- ▶ Two processes in client-server model can interact in various ways:
 - Sockets
 - Remote Procedure Calls (RPC)

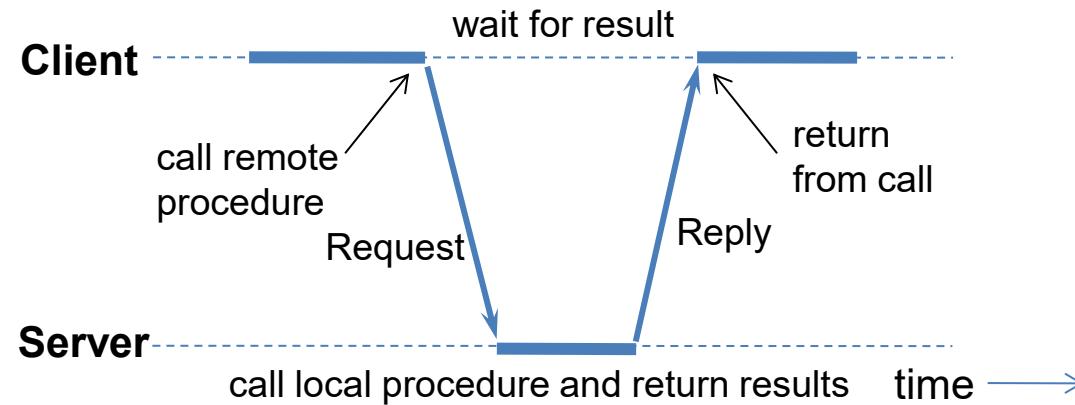
Socket

- ▶ The process acting as server opens a socket using a well-known port and waits until some client request comes.
- ▶ The second process acting as a client also opens a socket but instead of waiting for an incoming request, the client processes 'requests first'.



Remote Procedure Call (RPC)

- ▶ Low level message passing is based on send and receive primitives.
- ▶ Remote Procedure Call (RPC) is a protocol that one program can use to request a service from a program located in another computer on a network without having to understand the network's details.
- ▶ A procedure call is also sometimes known as a **function call** or a **subroutine call**.
- ▶ More sophisticated is allowing programs to call procedures located on other machines.
- ▶ RPC is a request–response protocol, i.e., it follows the **client-server model**

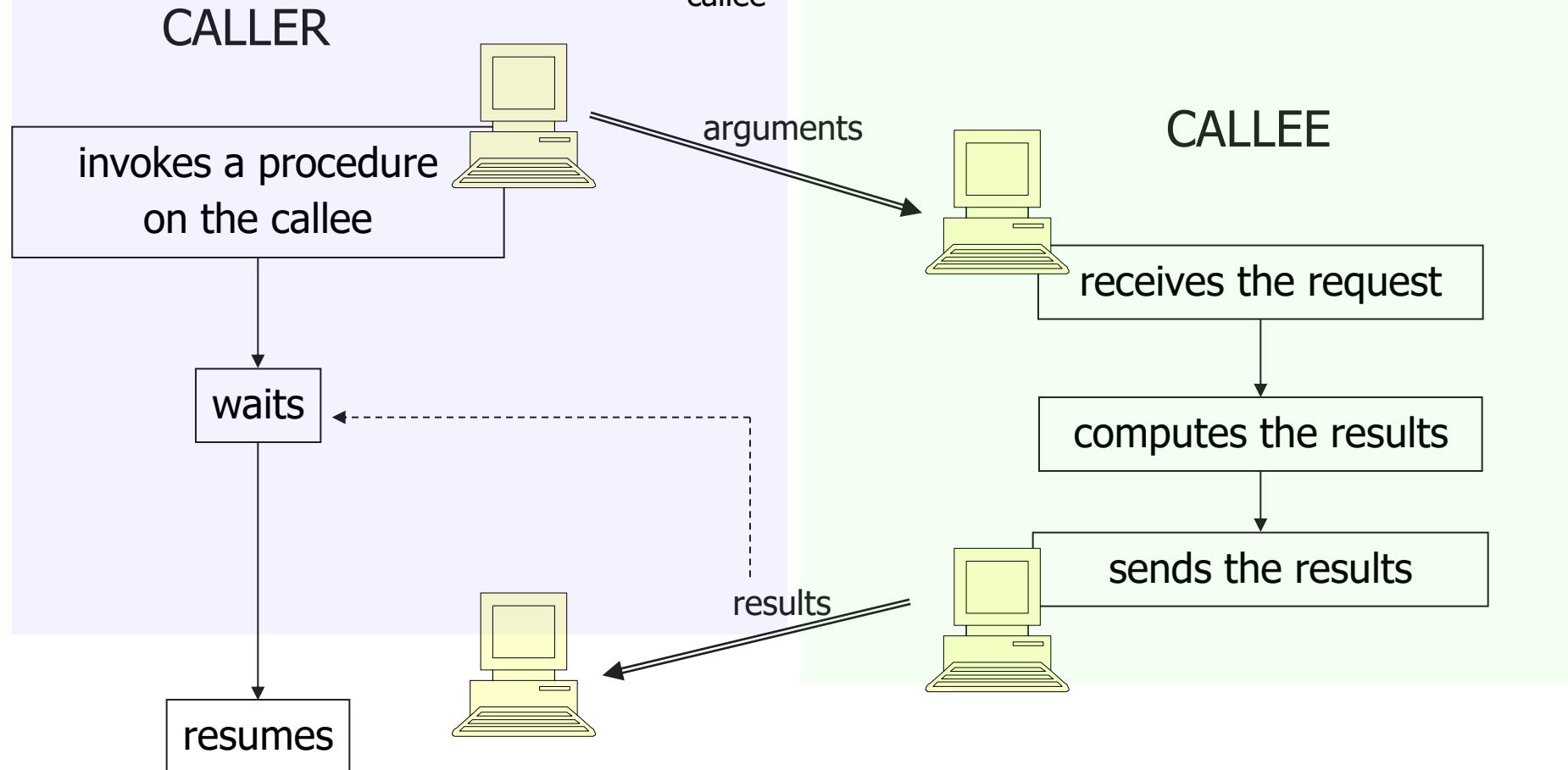


How RPC Works

1. An RPC is analogous to a function call. Like a function call, when an RPC is made, the calling arguments are passed to the remote procedure and the caller waits for a response to be returned from the remote procedure.
2. The client makes a procedure call that sends a request to the server and waits.
3. The thread is blocked from processing until either a reply is received, or it times out. When the request arrives, the server calls a local procedure that performs the requested service, and sends the reply to the client.
4. After the RPC call is completed, the client program continues. RPC specifically supports network applications.

How RPC Works

1. one procedure (**caller**) calls another procedure (**callee**)
2. the **caller waits** for the result from the callee
3. the **callee receives the request**, computes the results, and then **send them to the caller**
4. the **caller resumes** upon receiving the results from the callee



RPC Model

► It is similar to commonly used procedure call model. It works in the following manner:

1. For making a procedure call, the **caller places arguments** to the procedure in some well specified location.
2. **Control is then transferred** to the sequence of instructions that constitutes the body of the procedure.
3. The **procedure body is executed** in a newly created execution environment that includes copies of the arguments given in the calling instruction.
4. After the procedure execution is over, **control returns** to the calling point, returning a result.

Functions of RPC Elements

The Client

- ▶ It is user process which initiates a remote procedure call
- ▶ The client makes a perfectly normal call that invokes a corresponding procedure in the client stub.

The Client stub

- ▶ On receipt of a request it packs a requirements into a message and asks the local RPCRuntime to send it to the server stub.
- ▶ On receipt of a result it unpacks the result and passes it to client.

Functions of RPC Elements

RPCRuntime

- ▶ It handles transmission of messages between client and server.
- ▶ It is responsible for
 - Retransmission
 - Acknowledgement
 - Routing and Encryption

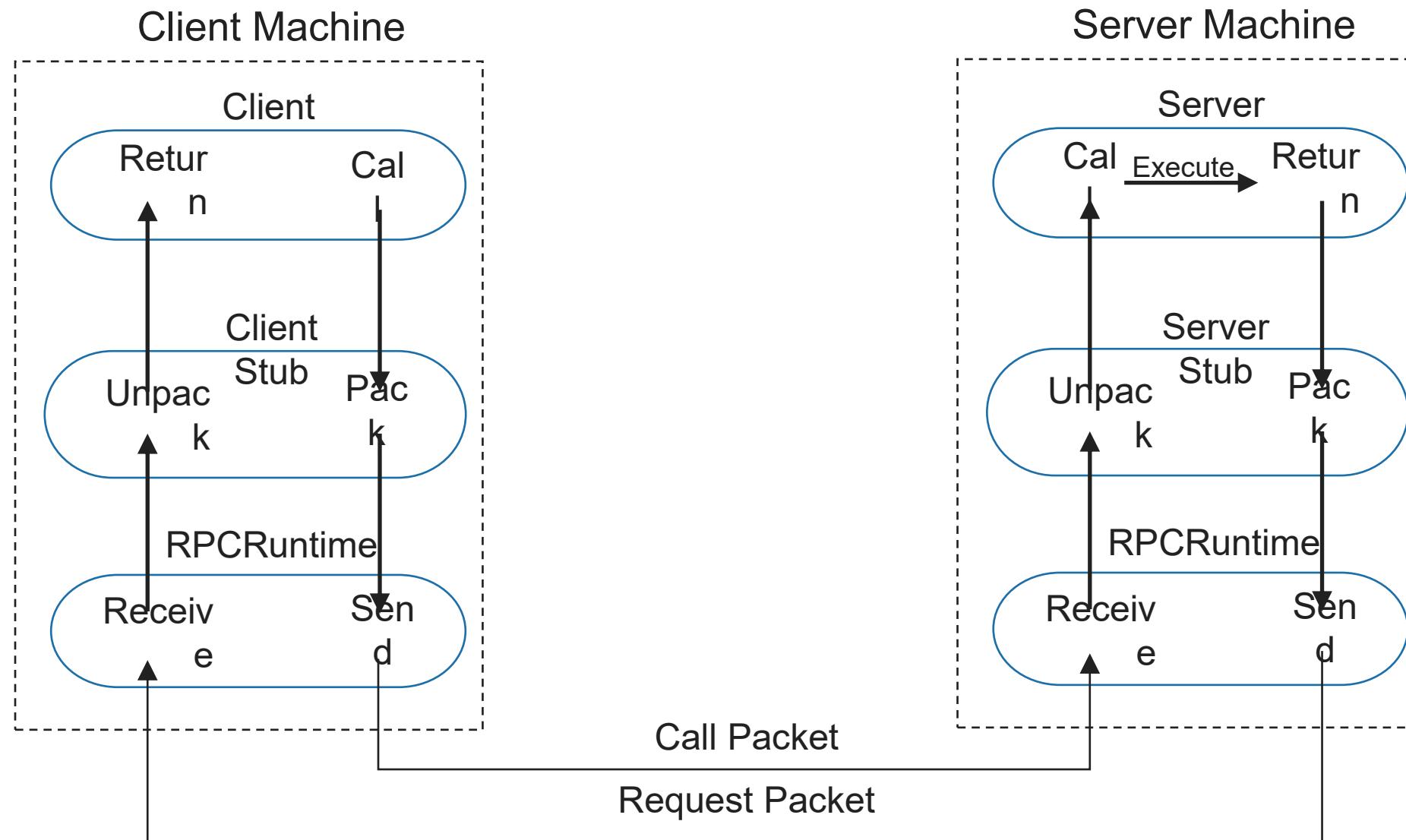
The Server stub

- ▶ It unpacks a call request and make a perfectly normal call to invoke the appropriate procedure in the server.
- ▶ On receipt of a result of procedure execution it packs the result and asks to RPCRuntime to send.

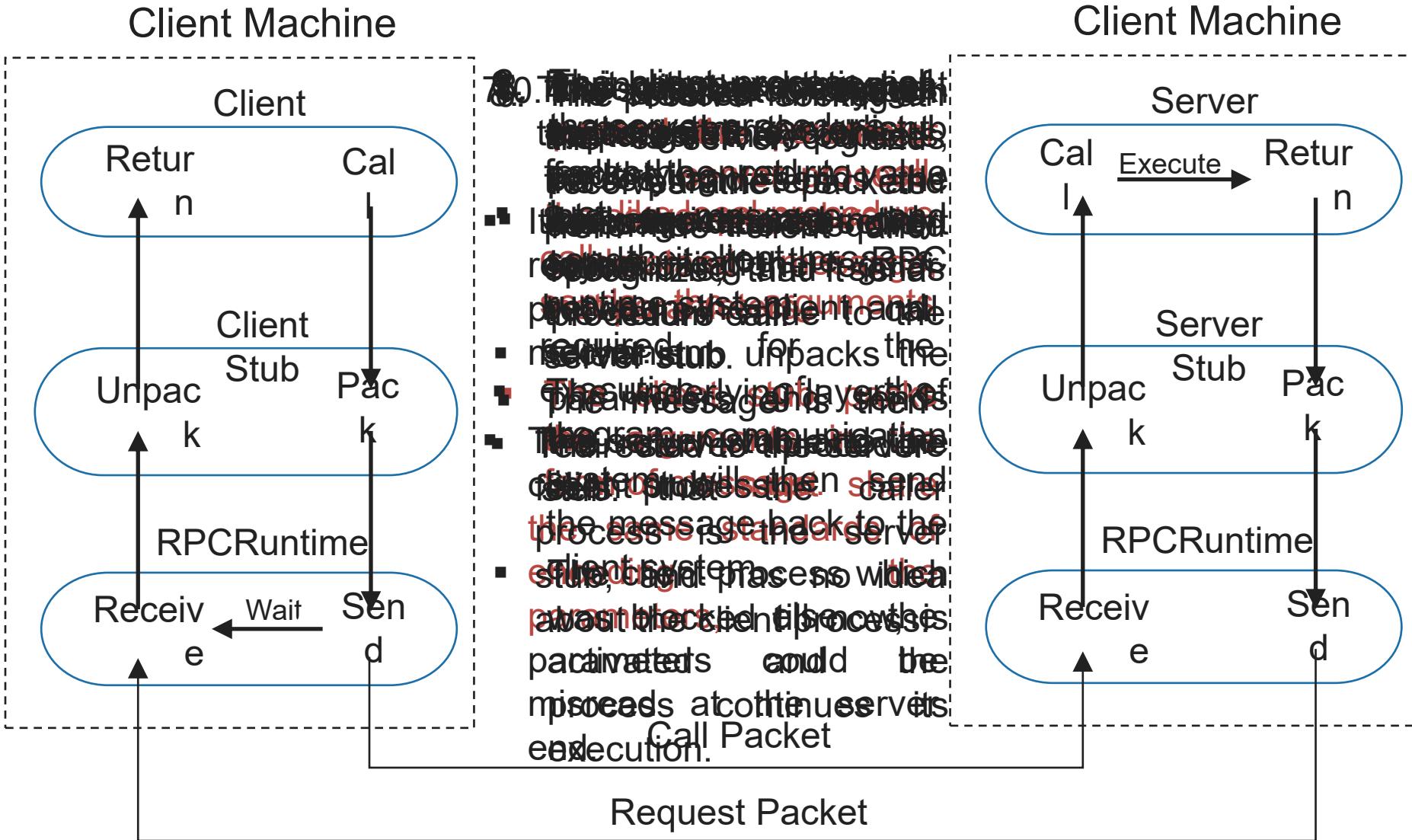
The Server

- ▶ It executes a appropriate procedure and returns the result from a server stub.

RPC Mechanism

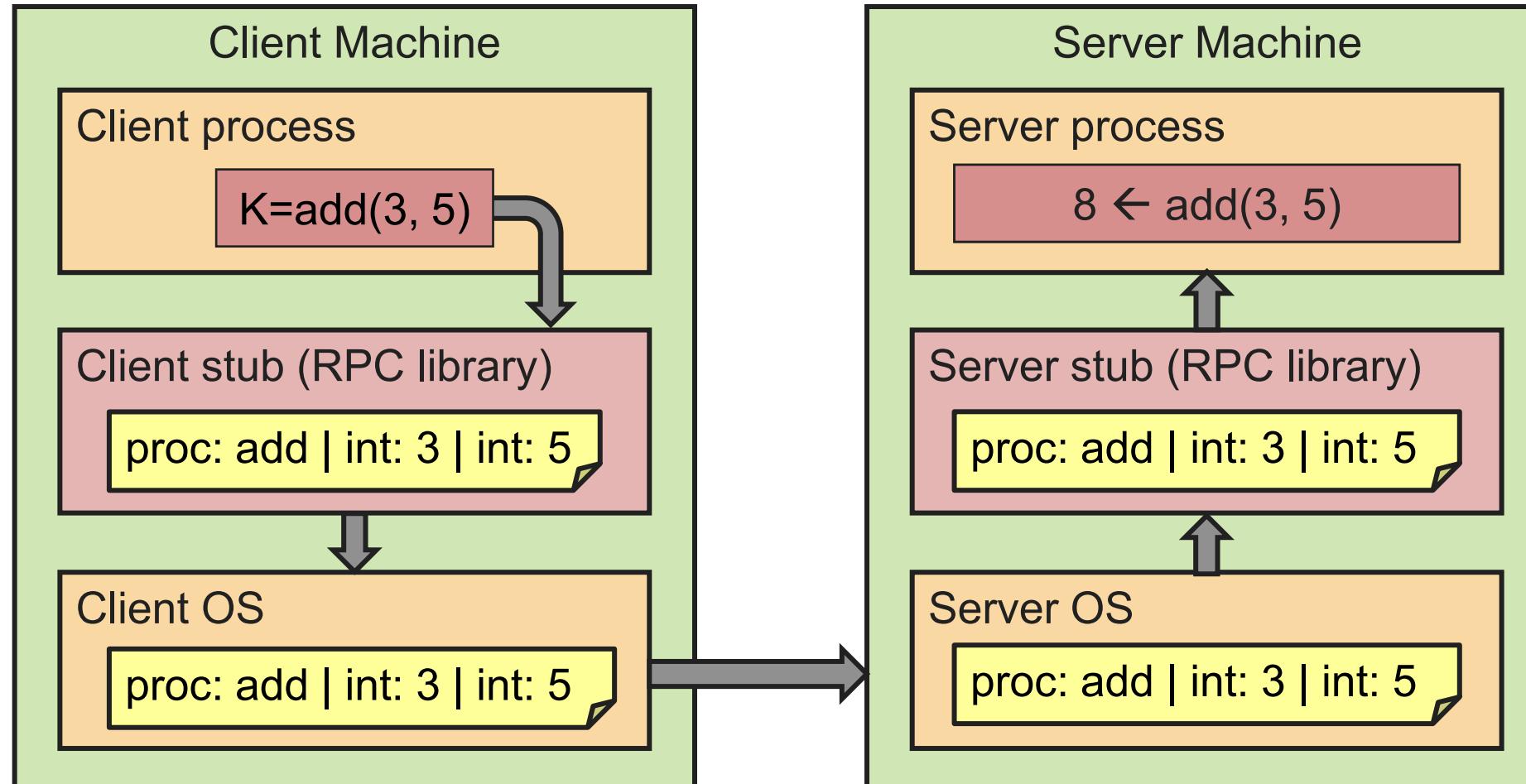


RPC Mechanism



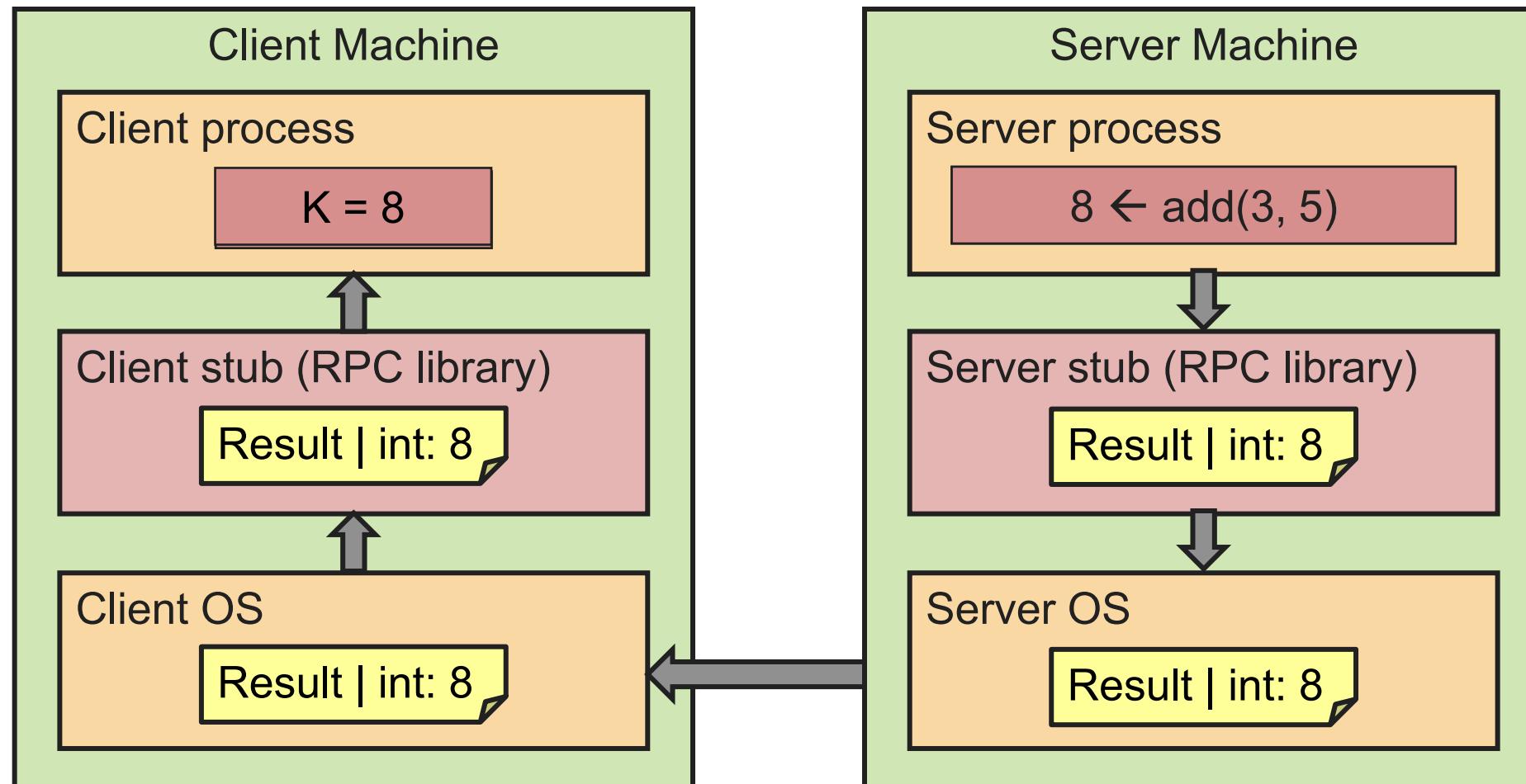
RPC Mechanism

3. Client OS takes responsibility to handle OS.



RPC Mechanism

Q. Services provided by the client OS.



Steps in a Remote Procedure Call

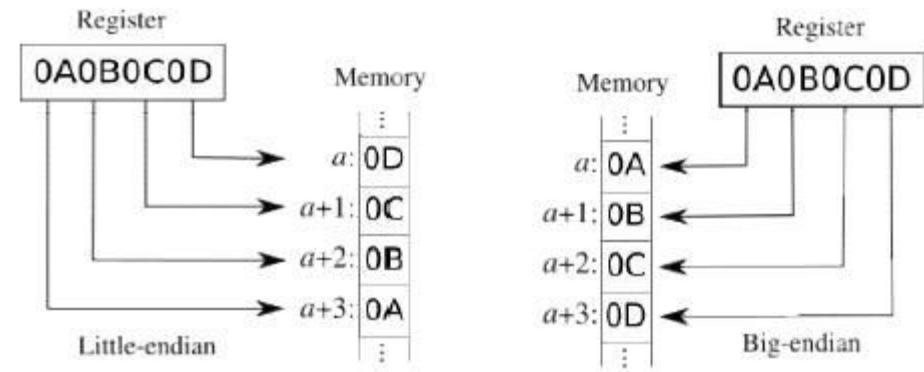
1. The client **calls a local procedure**, called the **client stub**.
2. Network messages are **sent by the client stub to the remote system** (via a system call to the local kernel using *sockets* interfaces).
3. Network messages are **transferred by the kernel to the remote system** via some protocol (either connectionless or connection-oriented).
4. A server stub, sometimes called the **skeleton**, **receives the messages on the server**. It unmarshals the arguments from the messages.
5. The server **stub calls the server function**, passing it the arguments that it received from the client.
6. When server function is finished, **it returns to the server stub** with its return values.
7. The server stub **converts the return values**, if necessary, and marshals them into one or more network messages to send to the client stub.

Steps in a Remote Procedure Call

8. Messages get sent back across the network to the client stub.
9. The client stub reads the messages from the local kernel.
10. The client stub then returns the results to the client function, converting them from the network representation to a local one if necessary.

Passing Value Parameters

- ▶ Packing parameters into a message is called **parameter marshaling**.
- ▶ Client and server machines may have different data representations (think of byte ordering)
- ▶ Wrapping a parameter means transforming a value into a sequence of bytes
- ▶ Client and server have to agree on the same encoding:
 - How are basic data values represented (integers, floats, characters)
 - How are complex data values represented (arrays, unions)
- ▶ Client and server need to properly interpret messages, transforming them into machine-dependent representations.
- ▶ Possible problems:
 - IBM mainframes use EBCDIC char code and IBM PC uses ASCII code
 - Integer: one's complement and 2's complement
 - Little endian and big endian



Passing Value Parameters

	3	2	1	0
0	0	0	5	
L	L	I	J	

Original message
on the Pentium

0	1	2	3
5	0	0	0
4	5	6	7

J I L L

The message after
receipt on the
SPARC

0	1	2	3
0	0	0	5
4	5	6	7

L L I J

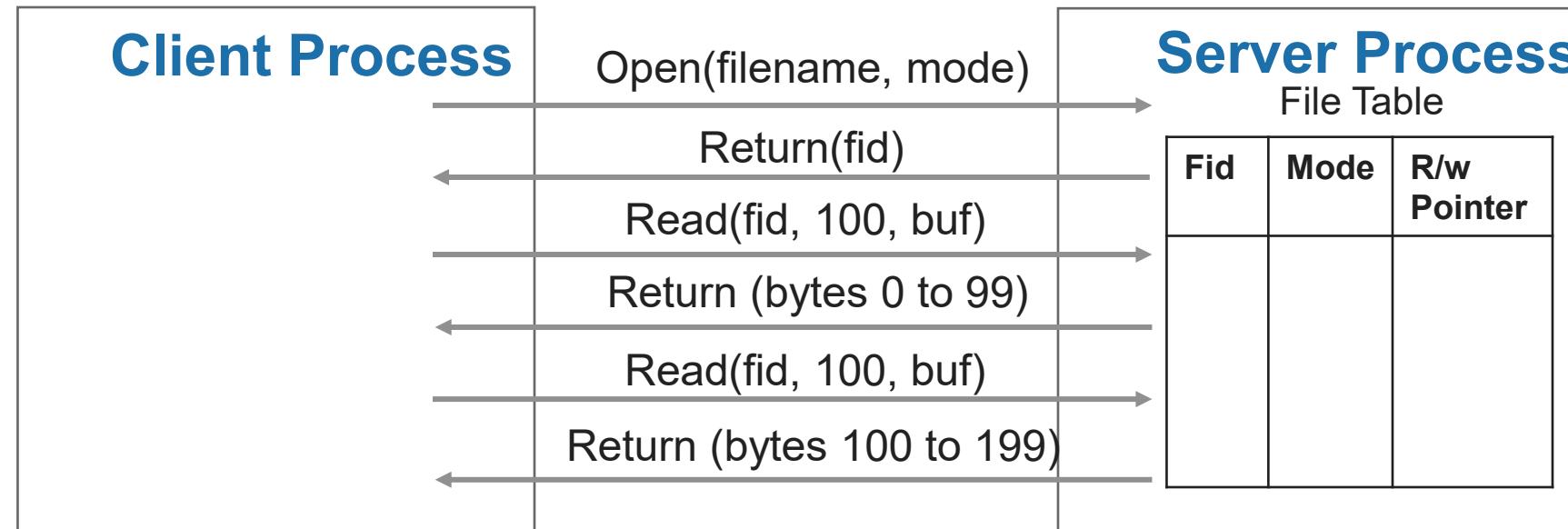
The message after
being inverted

Server Management

Stateful File Servers

- ▶ A stateful server **maintains client's state information** from one remote procedure call to the next.
- ▶ These clients state information is subsequently used at the time of executing the second call.
- ▶ To illustrate how a stateful file server works, let us consider a file server for byte-stream files that allows the following operations on files:
 - Open(filename, mode)
 - Read(fid, n, buffer)
 - Write(fid, n, buffer)
 - Seek(fid, position)
 - Close(fid)

Stateful File Server

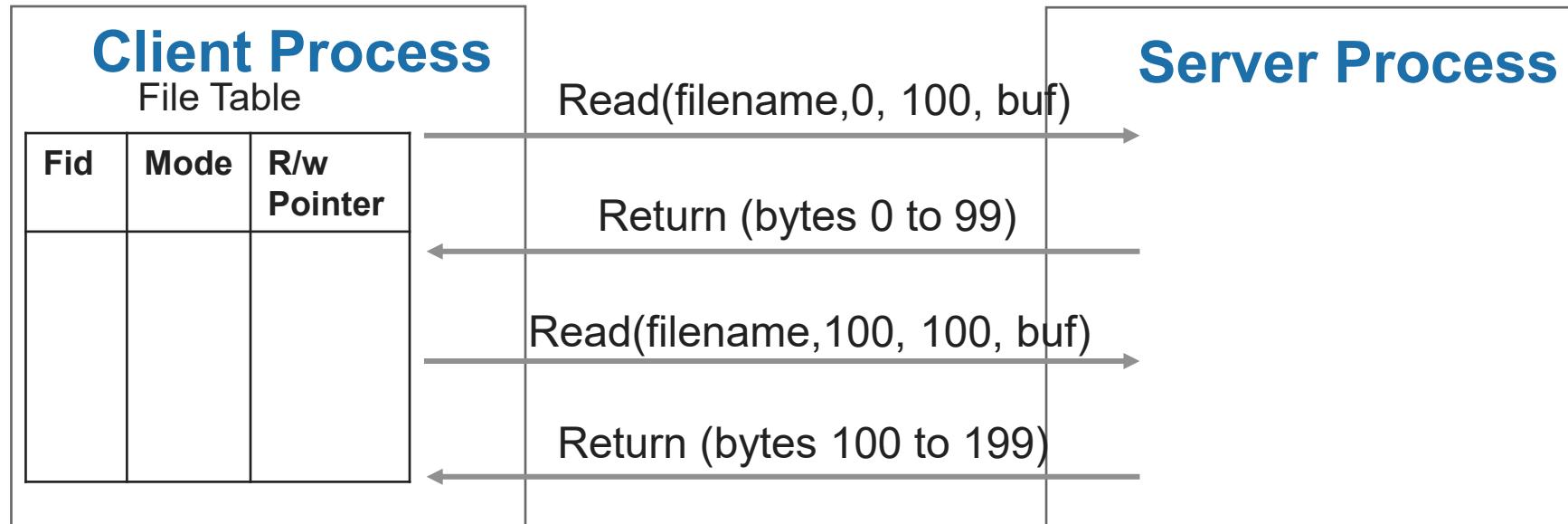


- After opening a file, if a client makes two subsequent Read (fid, 100, buf) requests, for the first request the first 100 bytes (bytes 0 to 99) will be read and for the second request the next 100 bytes (bytes 100 to 199) will be read.

Stateless File Server

- ▶ A stateless file server **does not maintain any client state information.**
- ▶ Therefore every request from a client must be accompanied with all the necessary parameters to successfully carry out the desired operation.
- ▶ Each request identifies the file and the position in the file for the read/write access.
- ▶ Operations on files in Stateless File server:
 - *Read(filename, position, n, buffer)*: On execution, the server returns n bytes of data of the file identified by filename.
 - *Write(filename, position, n, buffer)*: On execution, it takes n bytes of data from the specified buffer and writes it into the file identified by filename.

Stateless File Server



- ▶ This file server does not keep track of any file state information resulting from a previous operation.
- ▶ Therefore, if a client wishes to have similar effect as previous figure, the following two read operations must be carried out:
 - Read(filename, 0, 100, buffer)
 - Read(filename, 100, 100, buffer)

Difference between Stateful & Stateless

Parameters	Stateful	Stateless
State	A Stateful server remember client data (state) from one request to the next.	A Stateless server does not remember state information.
Programmin	Stateful server is harder to code.	Stateless server is straightforward to code.
Efficiency	More because clients do not have to provide full file information every time they perform an operation.	Less because information needs to be provided.
Crash recovery	Difficult due to loss of information.	Can easily recover from failure because there is no state that must be restored.
Information transfer	The client can send less data with each request.	The client must specify complete file names in each request.
Operations	Open, Read, Write, Seek, Close	Read, Write

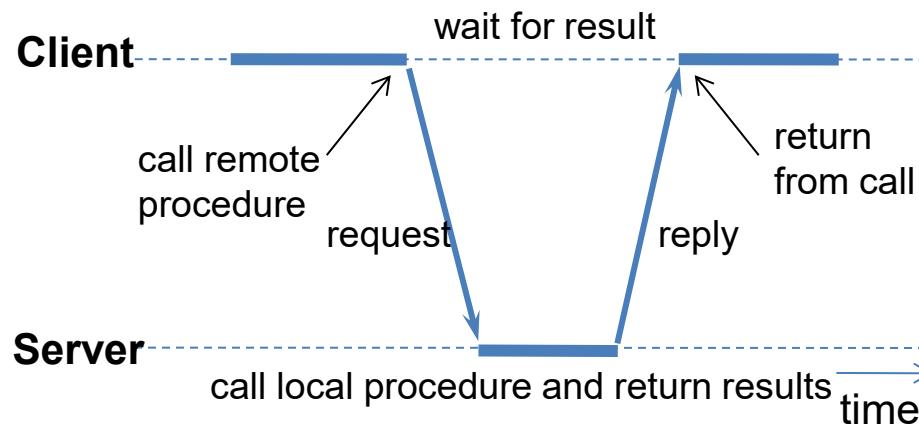
Asynchronous RPC

- ▶ A shortcoming of the original model: no need of blocking for the client in some cases.
- ▶ There are two cases
 1. If there is no result to be returned
 2. If the result can be collected later

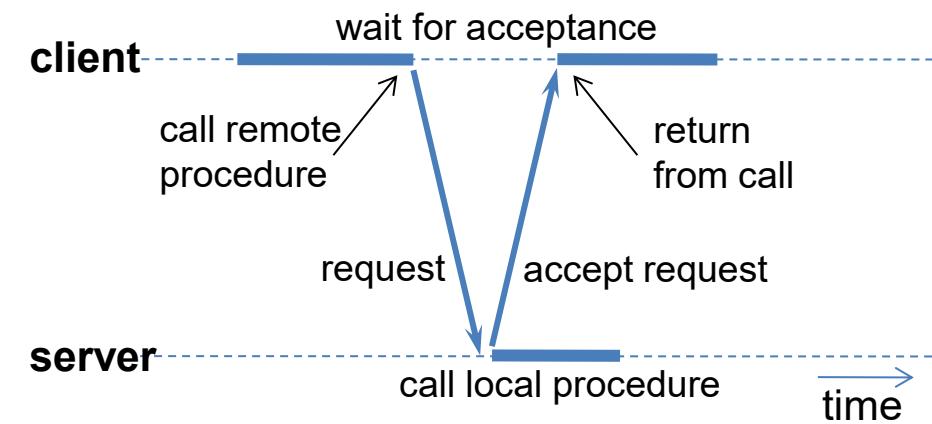
Asynchronous RPC

1. If there is no result to be returned

- e.g., inserting records in a database, ...
- The server immediately sends an ack promising that it will carryout the request



The interconnection between client and server in a traditional RPC

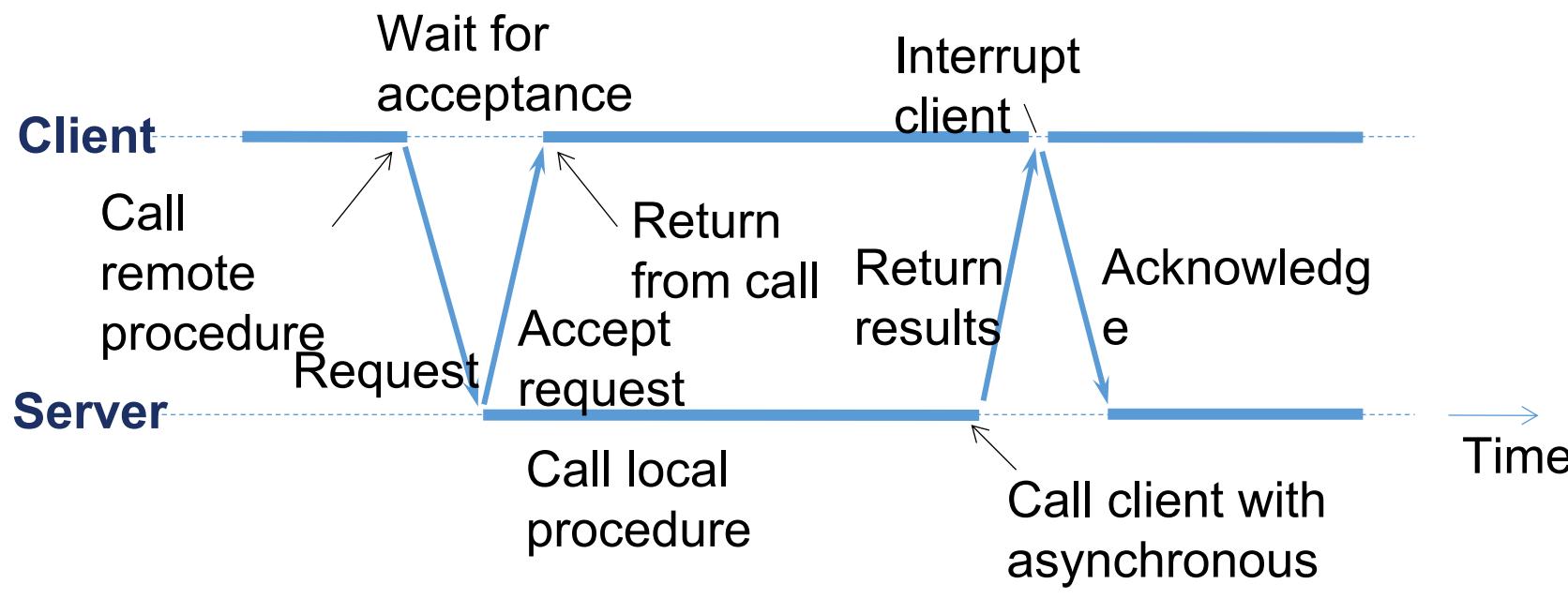


The interaction using **asynchronous RPC**

Asynchronous RPC

1. If the result can be collected later

- Example: prefetching network addresses of a set of hosts, ...
- The server immediately sends an ACK promising that it will carryout the request
- The client can now proceed without blocking
- The server later sends the result



A client and server interacting through two asynchronous RPCs

RPC Application Development

To develop an RPC application the following steps are needed:

- Specify the protocol for client server communication
- Develop the client program
- Develop the server program

The programs will be compiled separately. The communication protocol is achieved by generated stubs and these stubs and rpc (and other libraries) will need to be linked in.

The *rpcgen* Protocol Compiler

- ▶ The easiest way to define and generate the [protocol](#) is to use a [protocol compiler](#) such as [rpcgen](#).
- ▶ [rpcgen](#) provides programmers a simple and direct way to write distributed applications.
- ▶ [rpcgen](#) is a compiler. It accepts a remote program interface definition written in a language, [called RPC Language](#), which is similar to C.

- ▶ It produces a C language output which includes **stub versions of the client routines**, a **server skeleton**, **XDR filter routines** for both parameters and results, and a **header file** that contains common definitions.
- ▶ The **client stubs** interface with the RPC library and effectively hide the network from their callers.
- ▶ The **server stub** similarly hides the network from the server procedures that are to be invoked by remote clients.

The output of rpcgen is:

- ▶ A header file of definitions common to the server and the client
- ▶ A set of XDR routines that translate each data type defined in the header file
- ▶ A stub program for the server
- ▶ A stub program for the client

RPC specification

- ▶ A file with a ``.x" suffix acts as a remote procedure specification file. It defines **functions** that will be remotely executed.
- ▶ Multiple functions may be defined at once. They are numbered from one upward, and any of these may be remotely executed.
- ▶ The specification defines a program that will run remotely.
- ▶ **The program has a name, a version number and a unique identifying number** (chosen by you).

RPC versions and numbers

- Each RPC procedure is uniquely identified by a *program number, version number, and procedure number*.
- The program number identifies a group of related remote procedures, each of which has a different procedure number.
- Program numbers are given out in groups of hexadecimal 20000000.
0 - 1fffffff defined by Sun,
20000000 - 3fffffff defined by user
- Version numbers are incremented when functionality is changed in the remote program.
- More than **one version of a remote program** can be defined and a version can have more than one procedure defined.

- Writing protocol specification in RPC language to describe the remote version of calcu_pi.

- /* pi.x: Remote pi calculation protocol */

```
program PIPROG {  
    version CALCULATORS {  
        double CALCULATORS() = 1;  
    } = 1;  
} = 0x39876543;
```

The program PIPROG is 0x39876543, Version number referred to symbolically as CALCULATORS is 1.

In this example,

→ CALCU_PI procedure is declared to be:

- the procedure 1,
- in version 1 of the remote program

→ PIPROG, with the program number 0x39876543.

▶ Notice that the **program and procedure names** are declared with all capital letters. This is not required, but is a good convention to follow.

► To compile a **.x file** using

rpcgen -a -C pi.x

where:

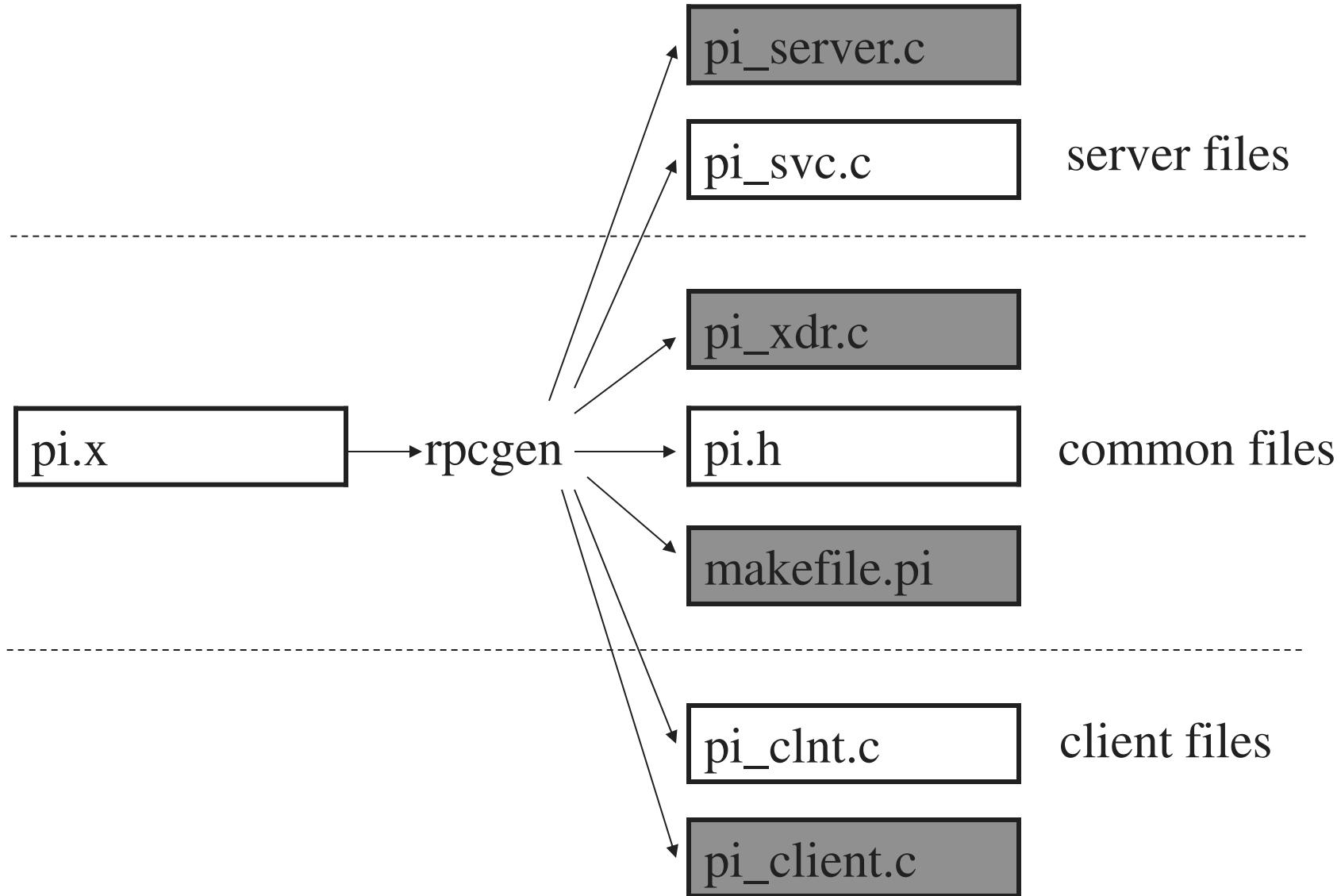
option **-a** tells rpcgen to generate all of the supporting files

option **-C** indicates ANSI C is used

This will generate the files:

- ▶ `pi_clnt.c` - the client stub - usually is not modified
- ▶ `pi_svc.c` - the server stub - usually is not modified
- ▶ `pi.h` - the header file that contains all of the XDR types generated from the specification
- ▶ `makefile.pi` - makefile for compiling all of the client and server code
- ▶ `pi_client.c` - client skeleton, need to be modified
- ▶ `pi_server.c` – server skeleton, need to be modified
- ▶ `pi_xdr.c`- Contains XDR filters needed by the client and server stubs – usually is not modified

rpcgen Files



- Make changes to pi_client.c file to give actual arguments to call(if required) and to display the output. then save the changes and run the following command.
- Write the logic at the location indicated and save the changes. Now, run the last command.

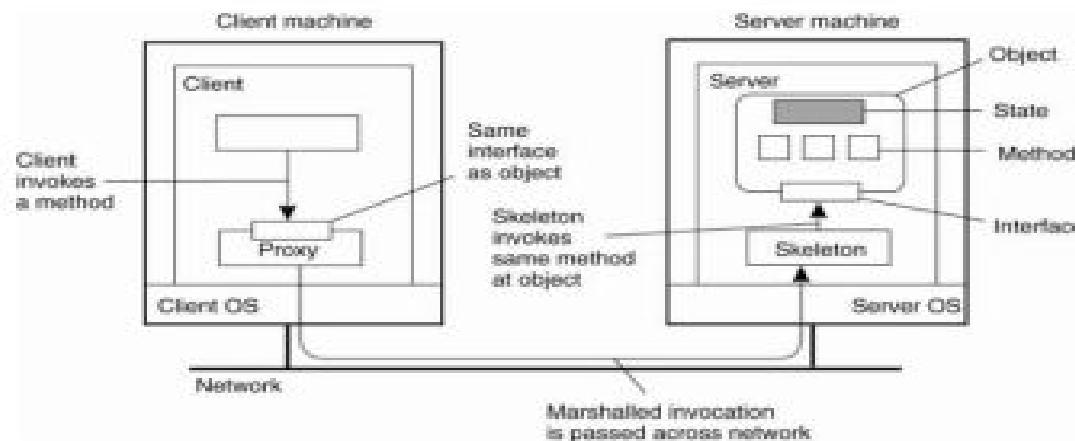
```
make -f Makefile.add
```

What is an Object ?

- *Objects* are units of data with the following properties:
 - **typed and self-contained**
Each object is an instance of a *type* that defines a set of *methods* (signatures) that can be invoked to operate on the object.
 - **encapsulated**
The only way to operate on an object is through its methods; the internal representation/implementation is hidden from view.
 - **dynamically allocated/destroyed**
Objects are created as needed and destroyed when no longer needed.
 - **uniquely referenced**
Each object is uniquely identified during its existence by a name/reference(pointer) that can be held/passed/stored/shared.

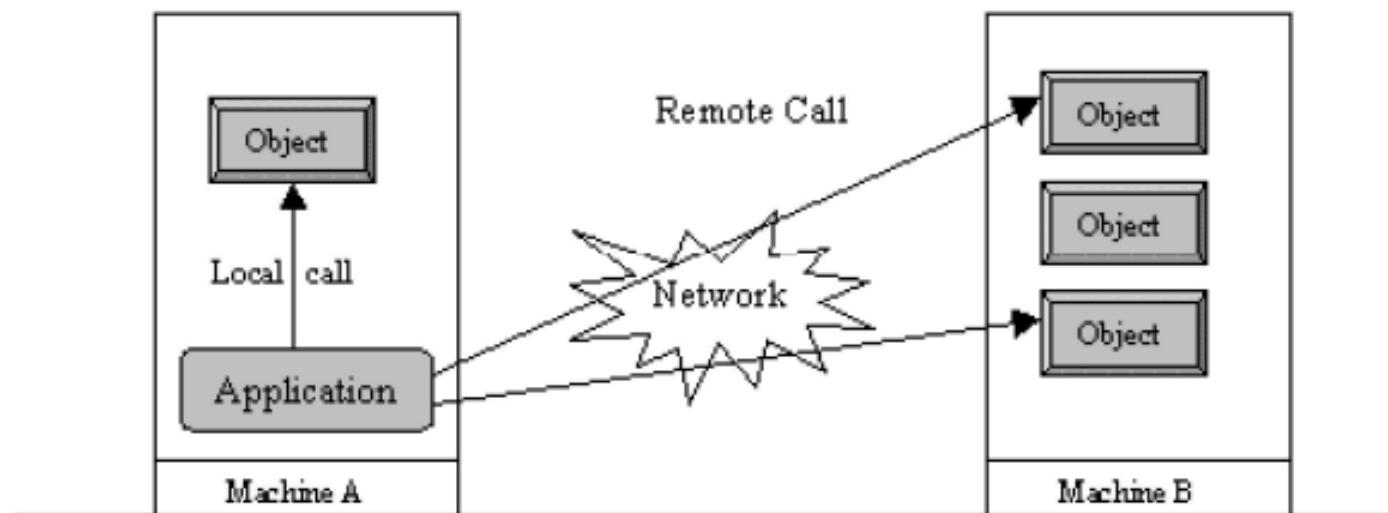
Distributed Objects

- Object encapsulates data – called state
- Operations on those data- called methods
- Methods are available through interface
- Distributed object – interface on one machine and object on another machine



- When client binds to a distributed object, an implementation of the object's interface, called a proxy is loaded into the client's address space.
- Proxy = client stub in RPC
- Skeleton = server stub
- Proxy/Skeleton – marshals method invocations into messages and unmarshal reply messages
- State is not distributed, interfaces implemented by the object are made available on different machines
- Compile time vs runtime objects
- Persistent and transient objects

Local v. Remote method invocation



What is (Java)RMI?

- The **RMI** (Remote Method Invocation) is an API that provides a mechanism to create distributed application in java. The RMI allows an object to invoke methods on an object running in another JVM.
- The RMI provides remote communication between the applications using two objects *stub* and *skeleton*.
- RMI uses **stub** and **skeleton** object for communication with the remote object.
- A **remote object** is an object whose method can be invoked from another JVM.

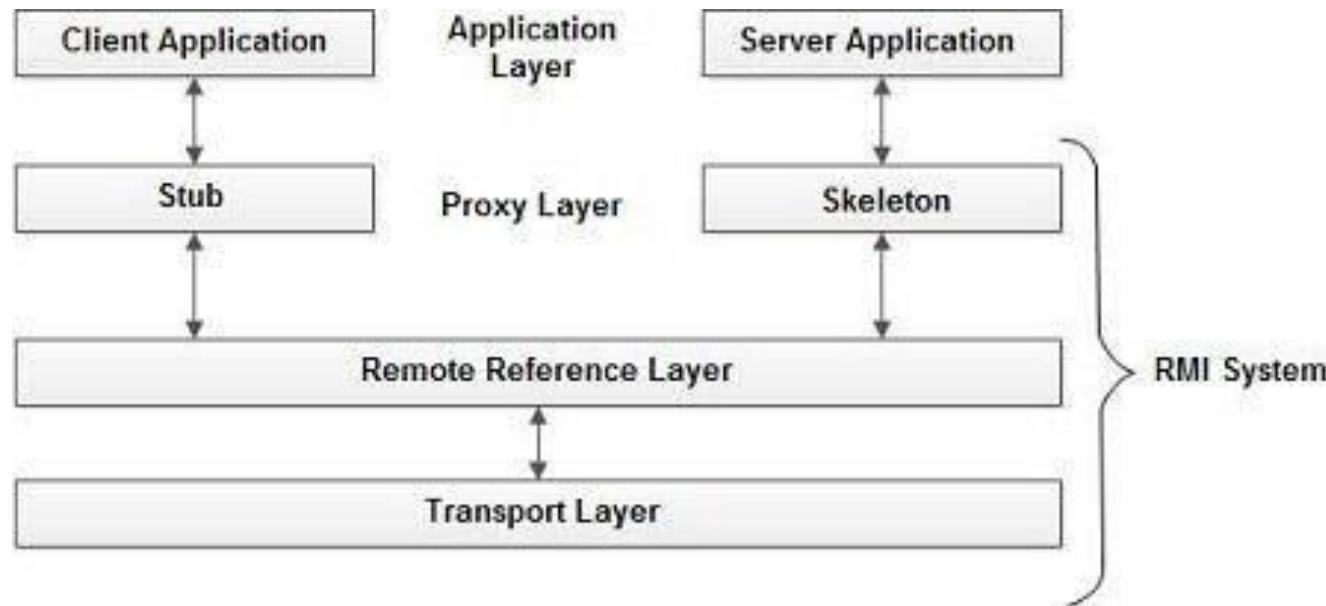
Goals and Features of RMI

- seamless object remote invocations
- callbacks from server to client
- distributed garbage collection
- all objects must be written in Java

RMI Architecture

Built in three layers

- Stub/Skeleton layer
- Remote reference layer
- Transport layer



Archileture of RMI

The Stub/Skeleton layer

- The interface between the application layer and the rest of the system
- Stubs and skeletons are generated using the RMIC compiler
- This layer transmits data to the remote reference layer via the abstraction of marshal streams (that use object serialization)
- This layer doesn't deal with the specifics of any transport

Stub

The stub is an object, acts as a gateway for the client side.

All the outgoing requests are routed through it. It resides at the client side and represents the remote object. When the caller invokes method on the stub object, it does the following tasks:

- Lives on client
- Pretends to be remote object
- It initiates a connection with remote Virtual Machine (JVM),
- It writes and transmits (marshals) the parameters to the remote Virtual Machine (JVM),
- It waits for the result
- It reads (unmarshals) the return value or exception, and
- It finally, returns the value to the caller.

Skeleton

The skeleton is an object, acts as a gateway for the server side object. All the incoming requests are routed through it. When the skeleton receives the incoming request, it does the following tasks:

- It reads the parameter for the remote method
- It invokes the method on the actual remote object, and
- It writes and transmits (marshals) the result to the caller.

The Remote Reference Layer

- The middle layer
- Provides the ability to support varying remote reference or invocation protocols independent of the client stub and server skeleton

The Transport Layer

- A low-level layer that ships serialized objects between different address spaces
- Responsible for:
 - Setting up connections to remote address spaces
 - Managing the connections
 - Listening to incoming calls
 - Maintaining a table of remote objects that reside in the same address space
 - Setting up connections for an incoming call
 - Locating the dispatcher for the target of the remote call

How does RMI work

- Defining a remote interface
- Implementing the remote interface
- Creating Stub and Skeleton objects from the implementation class using rmic (RMI compiler)
- Start the rmiregistry
- Create and execute the server application program
- Create and execute the client application program.

Distributed Garbage Collection

- RMI provides a distributed garbage collector that deletes remote objects no longer referenced by a client
- Uses a reference-counting algorithm to keep track of live references in each Virtual Machine
- RMI keeps track of VM identifiers, so objects are collected when no local or remote references to them

Advantages

- **Enables use of Design Patterns**
 - Use the full power of object oriented technology in distributed computing, such as two- and three-tier systems (pass behavior and use OO design patterns)
- **Safe and Secure**
 - e.g Java RMI uses built-in Java security mechanisms
- **Easy to Write/Easy to Use**
 - e.g. in Java RMI a remote interface is an actual Java interface
- **Distributed Garbage Collection**
 - Collects remote server objects that are no longer referenced by any client in the network

COMMUNICATION ABSTRACTIONS

- Number of communication abstractions that make writing distributed applications easier.

Provided by higher level APIs:

- Remote Procedure Call (RPC)
- Remote Method Invocation (RMI)
- Message-Oriented Communication
- Group Communication
- Streams

Types of Communication

- ▶ Transient Vs. Persistent communication
- ▶ Synchronous Vs. Asynchronous communication

Transient Vs. Persistent communication

Transient Communication

- ▶ Here, Sender and receiver run at the same time based on request/response protocol, the message is expected otherwise it will be discarded.
- ▶ Middleware discards a message if it cannot be delivered to receiver immediately
- ▶ Messages exist only while the sender and receiver are running.
- ▶ Communication errors or inactive receiver cause the message to be discarded.
- ▶ Transport-level communication is transient

Persistent Communication

- ▶ Messages are stored by middleware until receiver can accept it
- ▶ Receiving application need not be executing when the message is submitted.
- ▶ Example: Email

Synchronous Vs Asynchronous Communication

Synchronous Communication

- ▶ Sender blocks until its request is known to be accepted
- ▶ Sender and receiver must be active at the same time
- ▶ Sender execution is continued only if the previous message is received and processed.

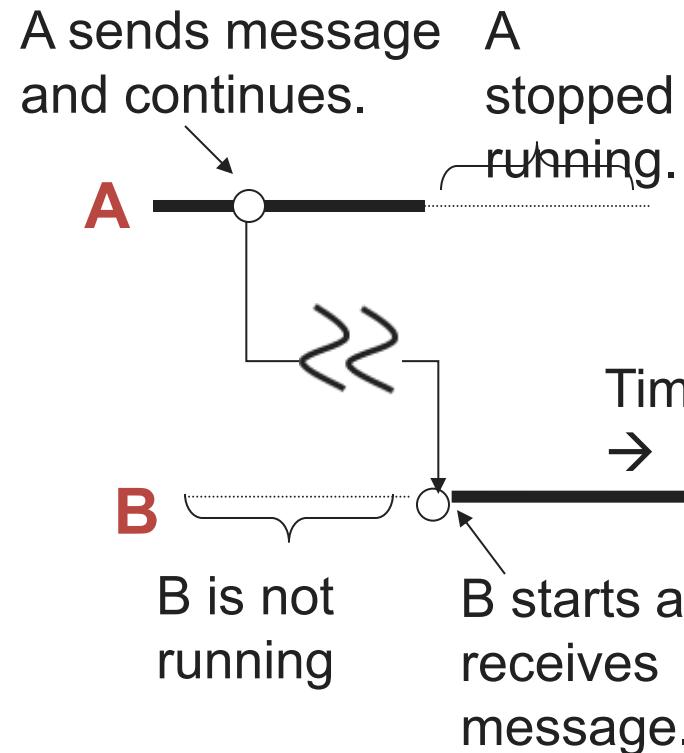
Asynchronous Communication

- ▶ Sender continues execution immediately after sending a message
- ▶ Message stored by middleware upon submission
- ▶ Message may be processed later at receiver's convenience

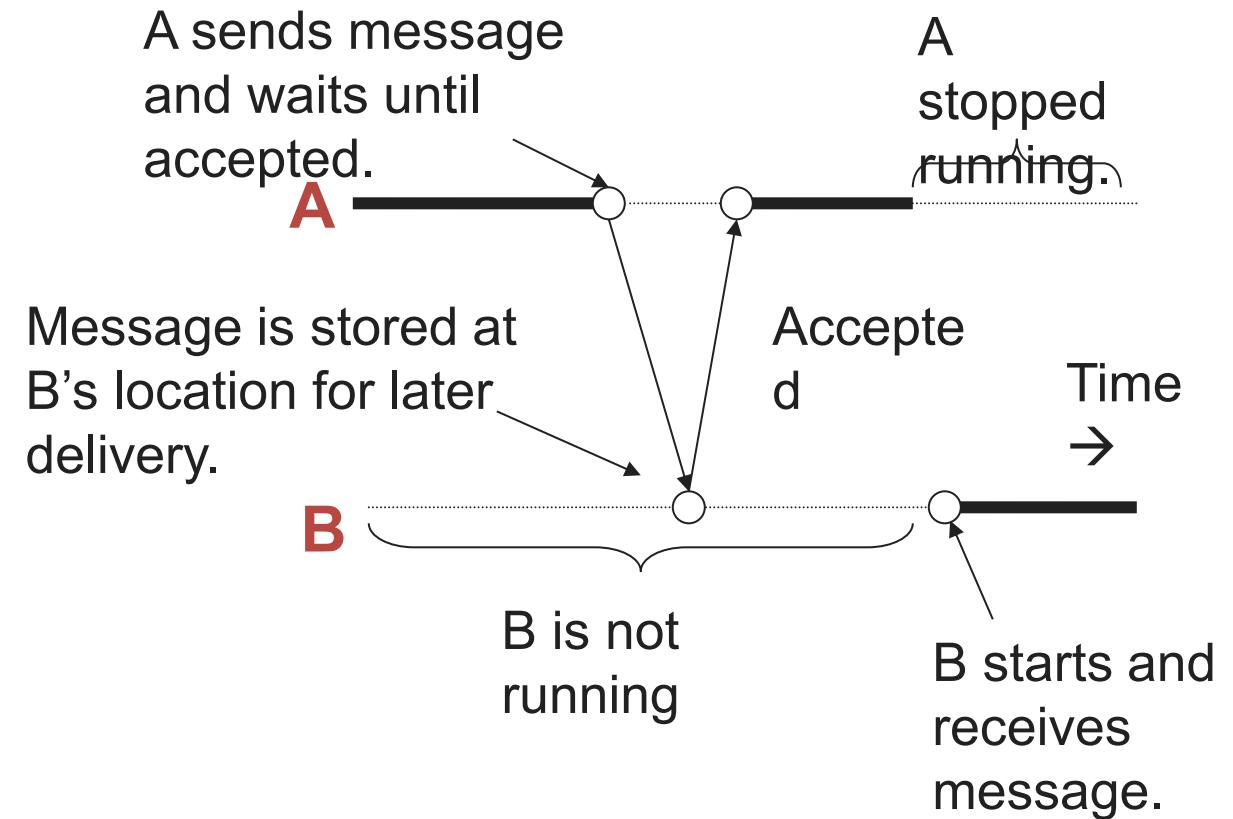
Message Oriented Communication

1. Persistent asynchronous communication.
2. Persistent synchronous communication.
3. Transient asynchronous communication.
4. Transient synchronous communication.
 - Receipt-based transient synchronous communication.
 - Delivery-based transient synchronous communication at message delivery.
 - Response-based transient synchronous communication.

Message Oriented Communication



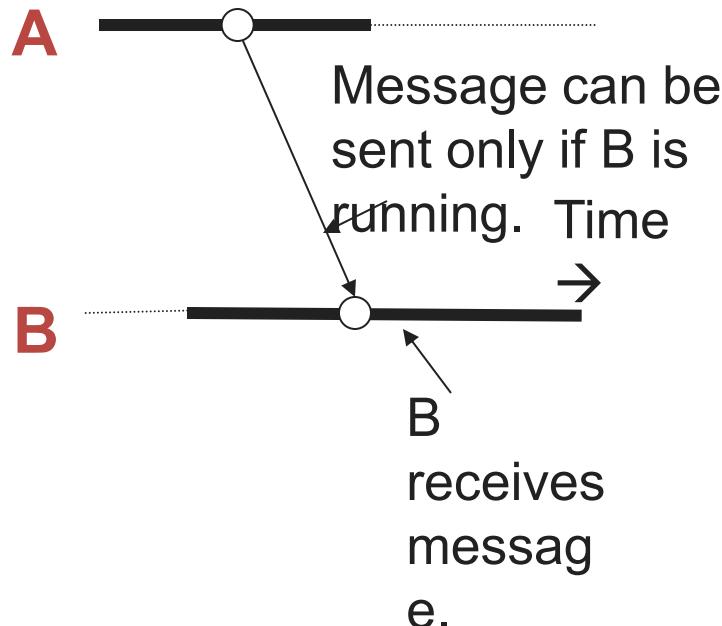
Persistent asynchronous communication



Persistent synchronous communication

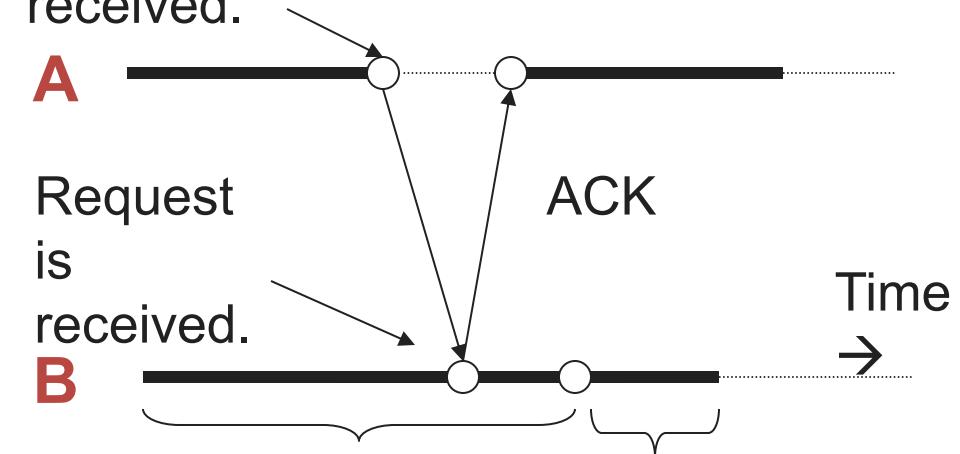
Message Oriented Communication

A sends message and continues.



Transient asynchronous communication

A sends message and waits until received.



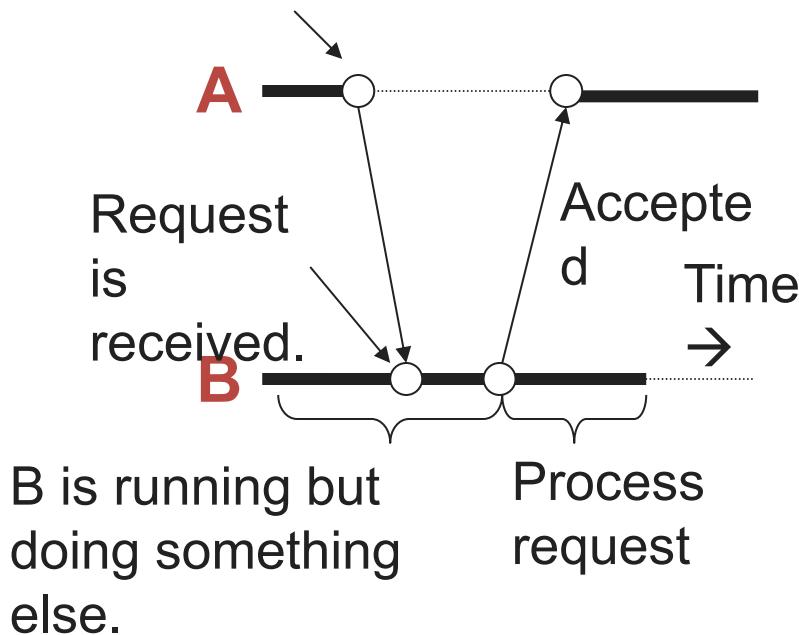
B is running but doing something else.

B processes request.

Receipt-based synchronous communication

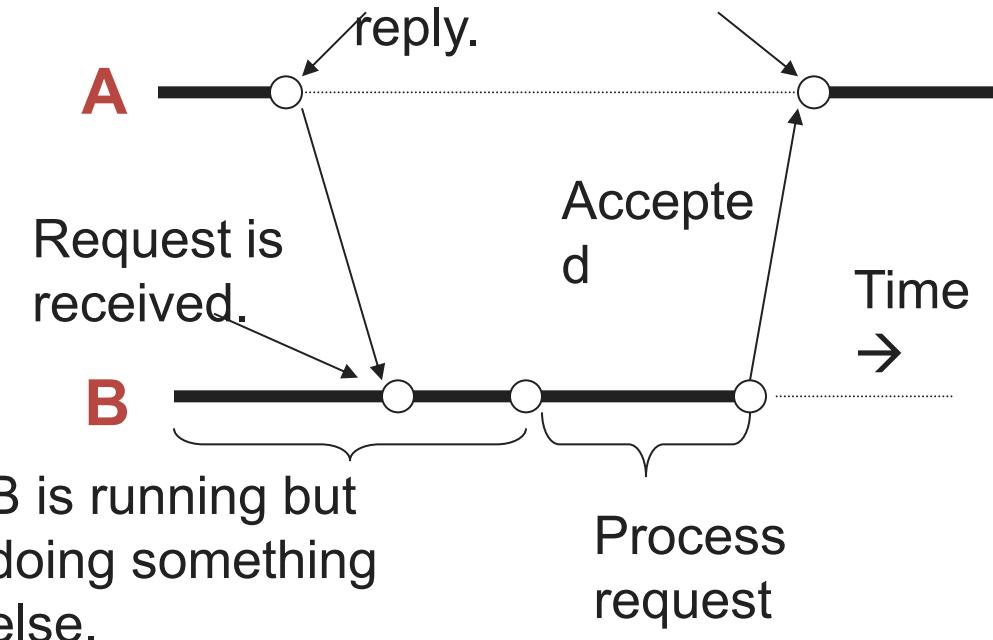
Message Oriented Communication

A sends request and waits until accepted.



Delivery-based synchronous communication

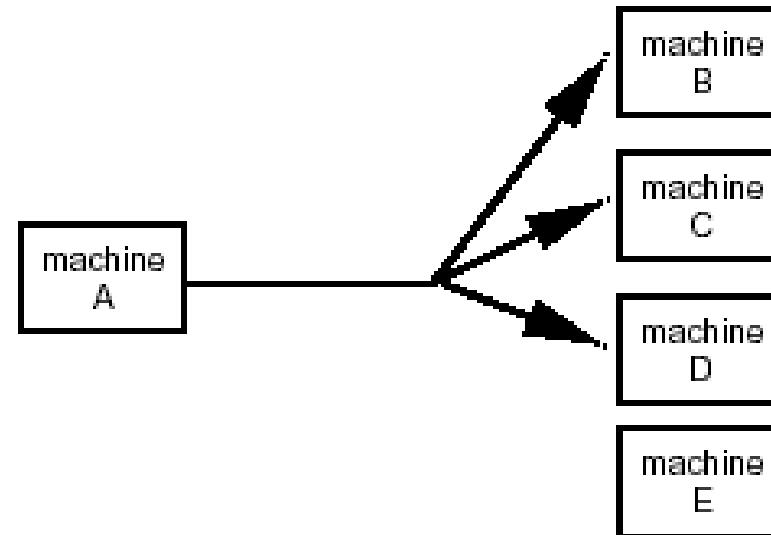
A sends request and waits for reply.



Response-based synchronous communication

GROUP COMMUNICATION

- ▶ Sender performs a single send()



GROUP COMMUNICATION

- Two kinds of group communication:
 - Broadcast (messgae sent to everyone)
 - Multicast (message sent to specific group)
- Used for:
 - Replication of services
 - Replication of data
 - Service discovery
- Issues:
 - Reliability
 - Ordering
- Example:
 - IP multicast

Strems

- Support for Continuous Media
 - 1. Between applications
 - 2. Between devices
 - 3. Data represented as single stream rather than discrete chunks

Continuous Media

- ▶ All communication facilities discussed so far are essentially based on a discrete, that is time independent exchange of information
- ▶ Continuous media: Characterized by the fact that values are time dependent:
 - Audio
 - Video
 - Animations
 - Sensor data (temperature, pressure, etc.)

Stream-Oriented Communication

- ▶ RPC, RMI, message-oriented communication are based on the exchange of discrete messages
 - Timing might affect performance, but not correctness
- ▶ In stream-oriented communication the message content must be delivered at a certain rate, as well as correctly.
 - e.g., music or video
- ▶ Audio and video are time-dependent data streams – if the timing is off, the resulting “output” from the system will be incorrect.
- ▶ Time-dependent information – known as “continuous media” communications.

Transmission Modes in Stream-Oriented Communication

- ▶ **Asynchronous transmission mode** – the data stream(Sequence of data units) is transmitted in order, but there's **no timing constraints** placed on the actual delivery (e.g., File Transfer). no restrictions with respect to *when* data is to be delivered.
- ▶ **Synchronous transmission mode** – define a maximum end-to-end delay for individual data packets (but data can travel faster).
- ▶ **Isochronous transmission mode** – data transferred “**on time**” – there's a maximum and minimum end-to-end delay (known as “**bounded jitter**”).
 - Known as “**streams**” – isochronous transmission mode is very useful for multimedia systems.
 - e.g., audio & video

Types of Streams in Stream-Oriented Communication

- ▶ Simple Streams – consists of a single flow of data, e.g., audio or video
- ▶ Complex Streams – several sequences of data (substreams) that are “related” by time.
 - Think of a lip synchronized movie, with sound and pictures, together with subtitles
 - This leads to data synchronization problems ... which are not at all easy to deal with
 - multiple data flows, e.g., audio or combination audio/video

Process Management

References:

- Pradeep K. Sinha, “Distributed Operation Systems : Concepts and Design” , PHI.

Process management

- Conventional(Centralied) OS:
 - deals with the mechanisms and policies for sharing the processor of the system among all processes
- Distributed operating system:
 - To make best possible use of the processing resources of the entire system by sharing them among all processes

Process management contd...

- Three concepts to achieve this goal:
 - Processor allocation
 - Deals with the process of deciding which process should be assigned to which processor
 - Process migration
 - Deals with the movement of a process from its current location to the processor to which it has been assigned
 - Threads
 - Deals with parallelism for better utilization of the processing capability of the system

Process Migration

- Process Migration:
 - Relocation of a process from its current location (*the source node*) to another node (*the destination node*).
 - A process may be migrated
 - either before it starts executing on its source node or
 - during the course of its execution.

Process Migration Contd...

- Process migration enables:
 - **dynamic load distribution**, by migrating processes from overloaded nodes to less loaded ones,
 - **fault resilience**, by migrating processes from nodes that may have experienced a partial failure,
 - **improved system administration**, by migrating processes from the nodes that are about to be shut down or otherwise made unavailable, and
 - **data access locality**, by migrating processes closer to the source of some data.

Goals

- The goals of process migration are closely tied with the type of applications that use migration.
 - **Accessing more processing power** is a goal of migration when it is used for load distribution. Migration is particularly important in the *receiver-initiated distributed scheduling algorithms*, where a lightly loaded node announces its availability and initiates process migration from an overloaded node.
 - **Resource sharing** is enabled by migration to a specific node with a special hardware device, large amounts of free memory, or some other unique resource.
 - **Fault resilience** is improved by migration from a partially failed node.

Goals Contd...

- **System administration** is simplified if long-running computations can be temporarily transferred to other machines. **For example**, an application could migrate from a node that will be shutdown, and then migrate back after the node is brought back up.
- **Mobile computing** also increases the demand for migration. Users may want to migrate running applications from a host to their mobile computer as they connect to a network.

System Requirements for Migration

- To support migration effectively, a system should provide the following types of functionality:
 - **Exporting/importing the process state-**
 - The system must provide some type of export/import interfaces that allow the process migration mechanism *to extract a process's state from the source node and import this state on the destination node.*
 - These interfaces may be provided by the underlying operating system, the programming language, or other elements of the programming environment.
 - **Naming/accessing the process and its resources-**
 - After migration, the migrated process should be accessible by the same name and mechanisms.

System Requirements for Migration

Contd..

- **Cleaning up the process's non-migratable state-**

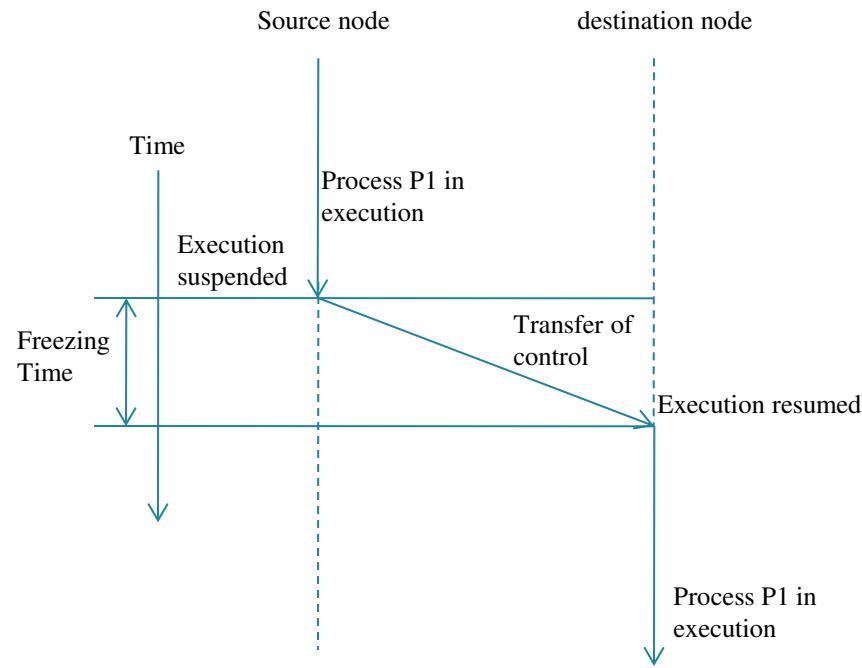
- The migrated process has associated system state that is not migratable.
- Migration must wait until the process finishes.
- If the operation can be arbitrarily long, it is typically aborted and restarted on the destination node. OR migration can wait for the completion of local file operations for limited time frame.

Process Migration contd...

- Two types:
 - Preemptive process migration
 - Process may be migrated during the course of its execution.
 - This type of process migration is **relatively expensive**, since it involves recording, migration and recreation of the process state as well as the reconstructing of any inter-process communication channels to which the migrating process is connected.
 - Non preemptive process migration
 - Process may be migrated before it starts executing on its source node.
 - This type of process migration is **relatively cheap**, since relatively little administrative overhead is involved.

Process Migration contd...

- The flow of execution of a migrating process:



Process Migration contd...

- Involves three steps:
 - Selection of a process that should be migrated
 - Selection of the destination node to which the selected process should be migrated
 - Actual transfer of the selected process to the destination node

Some desirable features

- **Transparency**
 - *Object access level.*
 - *System call and interprocess communication level*
- **Minimal interference**
 - Should cause minimal interference to progress of process and system.
 - Can be done by minimizing freezing time.
 - **Freezing time:** a time period for which the execution of the process is stopped for *transferring its information* to the destination node.

Some desirable features contd...

- **Minimal residual dependencies**
 - Migrated process should not continue to depend on its previous node once it has started executing on new node.
 - Otherwise problems will occur:
 - Migrated process continue to impose a load on its previous node
 - A failure or reboot of the previous node will cause the process to fail

Some desirable features contd...

- **Efficiency**
 - *Time* required of migrating a process.
 - The *cost of locating an object*.
 - The *cost of supporting remote execution* once the process is migrated.
- **Robustness**
 - The failure of a node other than the one on which a process is currently running *should not affect the execution of that process*.

Process migration mechanisms

- Process migration involves proper handling of several **sub-activities** to meet the good process migration mechanism requirements.
- Four major **sub-activities**
 - Freezing the process on its source node and restarting it on its destination node
 - Transfer of process's address space from source to its destination node
 - Forwarding messages intended for the migrant process
 - Handling communication between cooperating processes that have been separated (placed on different nodes) as a result of process migration.

Process migration mechanisms

- **Mechanisms for Freezing and Restarting a Process**

For preemptive process migration, the process is to take a “*snapshot*” of the process’s state on its source node and restore the snapshot on the destination node.

Steps-

- 1st step: freeing the process
 - Execution of the process is suspended and all external interactions are postponed
- 2nd step: state information is transferred to its destination node
- 3rd step: process is restarted using state information on destination node

Process migration mechanisms

- **Immediate and Delayed blocking of the process**
 - Before process can be frozen, its execution must be blocked
 - may be blocked immediately or delayed
 - if the process is not executing a system call --- can be immediately blocked
- **Interruptible and Uninterruptible States:**
 - if the process is executing a system call and at an interruptible priority waiting for a kernel event to occur, it can be immediately blocked from further execution
 - and at non-interruptible priority waiting for a kernel event to occur, it cannot be blocked, has to be delayed until the system call is complete.

Process migration mechanisms

- *Fast and Slow I/O Operations*
- *Information about Open Files*
- *Restoring the Process on its Destination Node.*

Address Space Transfer Mechanisms

- **Address Space Transfer Mechanisms**
 - Information to be transferred from source node to destination node:
 - **Process's state information**
 - Consists contents of register, scheduling information, memory tables, I/O buffers, interrupt signals, I/O states, process identifier, information about I/O files
 - **Process's address space**
 - Consists code, data, stack of the program
 - Difference between the size of process's state information (few kilobytes) and address space (several megabytes)
 - Time taken to transfer address space is more
 - Possible to transfer the address space without stopping its execution
 - Not possible to resume execution until the state information is fully transferred

Address Space Transfer Mechanisms

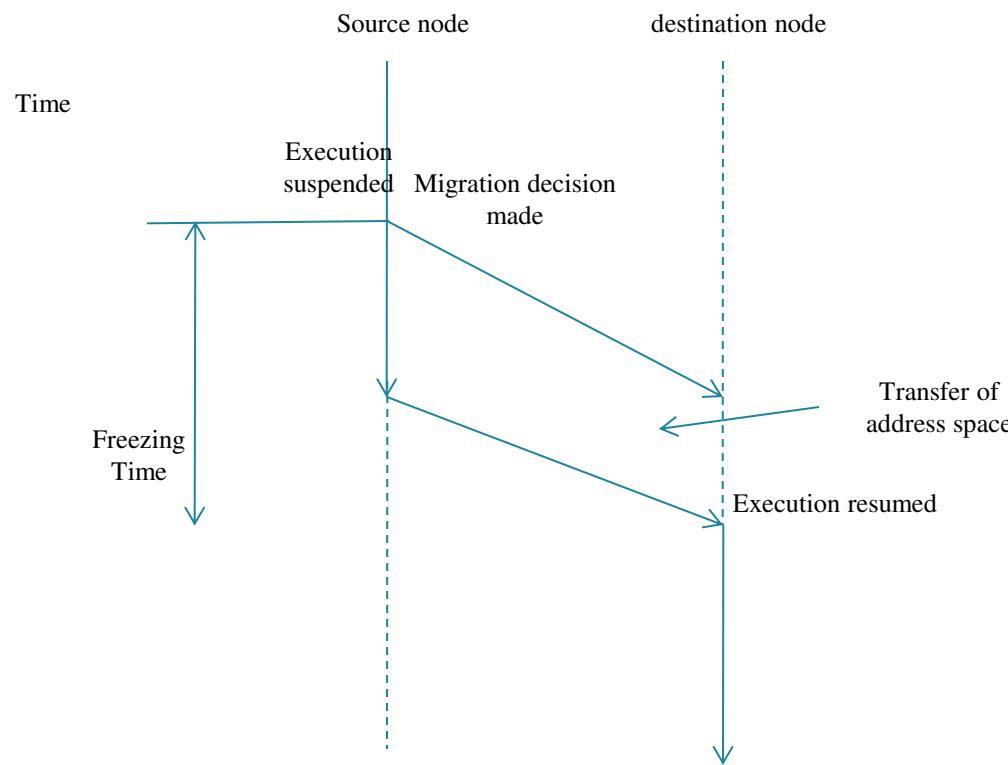
- Three methods for address space transfer
 - Total Freezing
 - Pretransferring
 - Transfer on reference

Address Space Transfer Mechanisms

- Total Freezing:
 - Process execution is stopped while its address space is being transferred
 - Simple and easy to implement
 - Disadvantages:
 - Process is suspended for a long time during migration
 - Not suitable for interactive process, the delay can occur

Address Space Transfer Mechanisms

- Total Freezing:

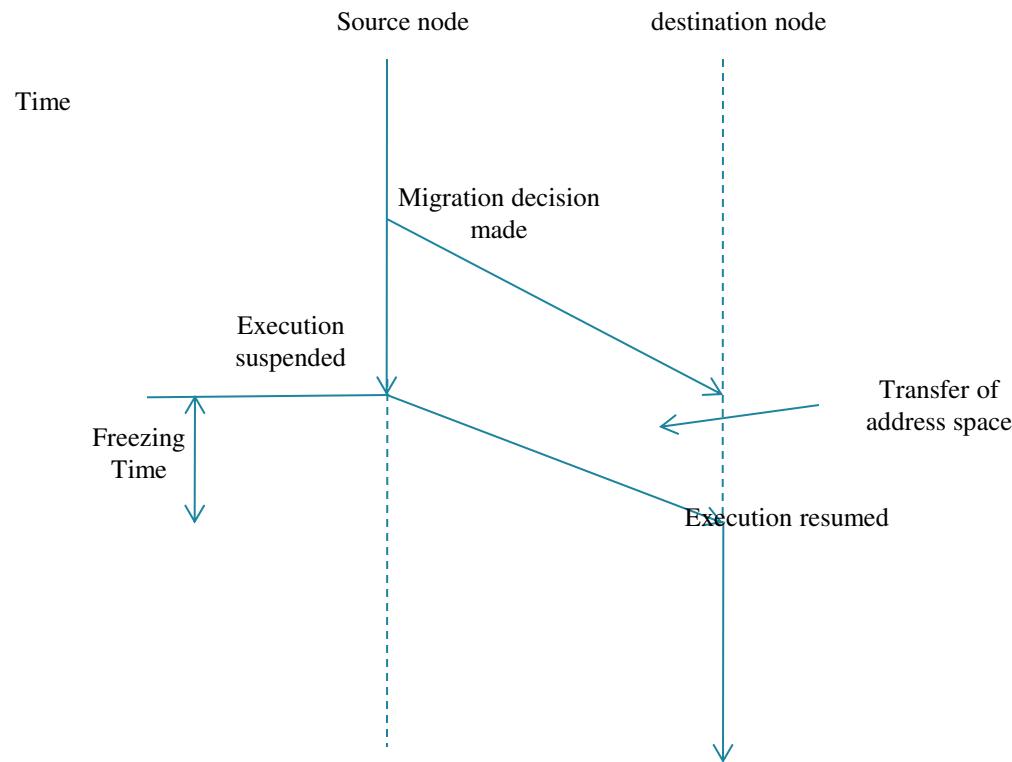


Address Space Transfer Mechanisms

- Pretransferring (precopying):
 - Address space is transferred while the process is still running on the source node.
 - Initial transfer of the complete address space followed by repeated transfers of the pages modified during previous transfer.
 - Remaining modified pages are retransferred after the process is frozen for transferring its **state information**.
 - Pretransfer operation is executed at a higher priority than all other programs on the source node.

Address Space Transfer Mechanisms

- Pretransferring (precopying):



Address Space Transfer Mechanisms

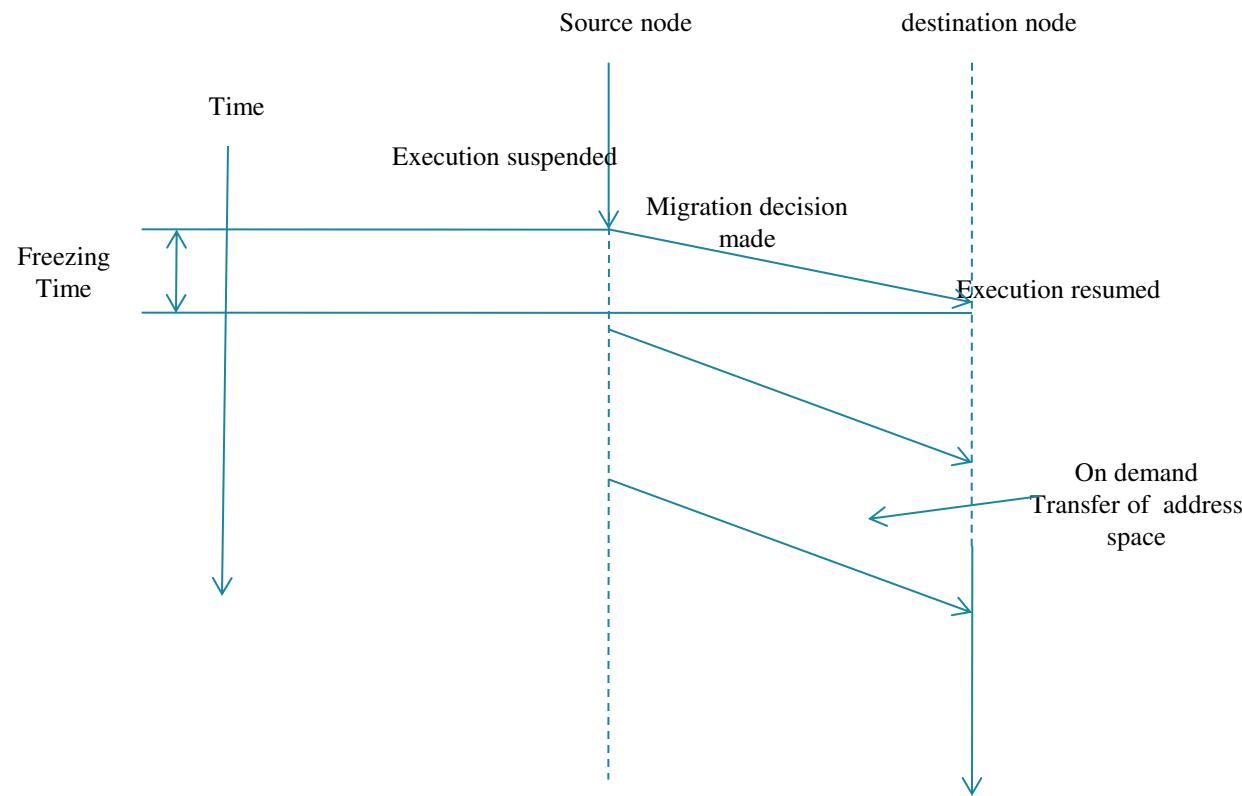
- Advantage:
 - freezing time is reduced
- Disadvantage:
 - Total time of migration is increased due to the possibility of redundant page transfers.
“Redundant pages are pages that are transferred more than once during pretransferring”

Address Space Transfer Mechanisms

- Transfer on reference
 - Based on the assumption that the process *tends* to use only a *relatively small part of their address space* while executing.
 - A page of the address space is transferred from its source node to destination node only when referenced.
 - Demand-driven copy-on-reference approach.
 - Switching time is very short and *independent of the size* of the address space.

Address Space Transfer Mechanisms

- Transfer on reference



Address Space Transfer Mechanisms

- Disadvantages:
 - Not efficient in terms of cost
 - Imposes continued load on process's source node
 - Results in failure if source node fails or reboots

Message forwarding mechanisms

- Ensures that all pending, en-route and future messages arrive at the process's new location
- **Classification of the messages to be forwarded:**
 - **Type 1:** Messages received at the source node after the process's execution has been stopped on its source node and process's execution has not yet been started on its destination node
 - **Type 2:** Message received at the source node after the process's execution has started on its destination node
 - **Type 3:** Messages that are to be sent to the migrant process from any other node after it has started executing on the destination node

Message forwarding mechanisms

- **Mechanism of Resending the Message**
 - Used in V-system and Amoeb for handling all three type messages

Message Type	V-System	Amoeba
Type 1 or Type 2	<ul style="list-style-type: none">• Simply dropped the message and sender is prompted to resend it to the process's new node• Sender will retry until successful receipt of a reply	<p>Source node's kernel sends the message to sender</p> <p>Type 1: "try again later, this process is frozen"</p> <p>Type 2: "this process is unknown at this node"</p>
Type 3	Sender does a "broadcast mechanism" to find the new location of the process	

Message forwarding mechanisms

- **Origin Site Mechanism**
 - There is process's origin site (or home node)
 - Origin site is responsible for keeping information about the current location of all the processes created on it
 - Messages are sent to the origin site first and from there they are forwarded to the current location
 - Drawbacks:
 - not good from reliability point of view
 - Failure of origin site will disrupt the message-forwarding mechanisms
 - continuous load on migrant process's original site

Message forwarding mechanisms

- **Link-Traversal Mechanism**

- Uses message queue for storing messages of type 1 on the source node
- Use of link (a forwarding address) for messages of type 2 and 3
 - Link is left at the source node pointing to the destination node
- Link has two components:
 - Process identifier: origin node
 - last known location of the process
- Migrated process is located by traversing a series of links

Message forwarding mechanisms

- Link-Traversal Mechanism
- Drawbacks:
 - poor efficiency
 - Several links may have to be traversed to locate a process from a node
 - poor reliability
 - If any node in the chain of links fails, the process cannot be located

Mechanisms for handling co-processes

- Important issue is the necessity to provide efficient communication between a process (parent) and its sub-processes (children)
- Two different mechanisms
 - Disallowing separation of Co-processes
 - home node or origin site concept

Mechanisms for handling co-processes

- Disallowing separation of Co-processes
 - By disallowing the migration of processes that wait for one or more of their children to complete.
 - By ensuring that when a parent process migrates, its children processes will be migrated along with it
 - Concept of logical host
 - Address spaces and their associated process's are grouped into logical host
 - Migration of a process is actually migration of logical host
- Drawback:
 - Does not allow parallelism within jobs, which is achieved by assigning various tasks of a job to the different nodes of the system and execute them simultaneously
 - Overhead is large when logical host contains several processes

Mechanisms for handling co-processes

- **Home node or origin site concept**
 - Complete freedom of migrating a process or its sub-processes independently and executing them on different nodes
 - All communications between a parent process and children processes take place via **home node**
- Drawback:
 - The message traffic and the communication cost increase
 - Lost communication if origin node fails

Advantages of process migration

- Reducing average response time of processes
 - avg. response time is increase as load increase on to the node
 - Process is migrated from heavily loaded node to a ideal node
- Speeding up individual jobs
 - Execute tasks of a job concurrently
 - To migrate a job to a node having faster CPU
- Gaining higher throughput
 - In a system that does not support process migration, CPUs of all nodes are not fully utilized.
 - In a system that support process migration, CPUs of all nodes can be better utilized by using suitable load balancing policy

Advantages of process migration

- Utilizing resources effectively
 - Capabilities of the various resources such as CPU, printers, storage devices etc.
 - Depending on the nature of the process, it can be migrated to the most suitable node
- Reducing network traffic
 - Migrate the process closer to the resources
 - To migrate and cluster two or more processes which frequently communicate with each other, on the same node

Advantages of process migration

- Improving system reliability
 - Migrating critical process to more reliable node
- Improving system security
 - A sensitive process may be migrated and run on the secure node that is not directly accessible to general users thus improve the security of that process

Distributed Scheduling

References:

- Mukesh Singhal, Niranjan Shivaratri. “Advanced concepts in operating systems”. Tata McGraw Hill publication.

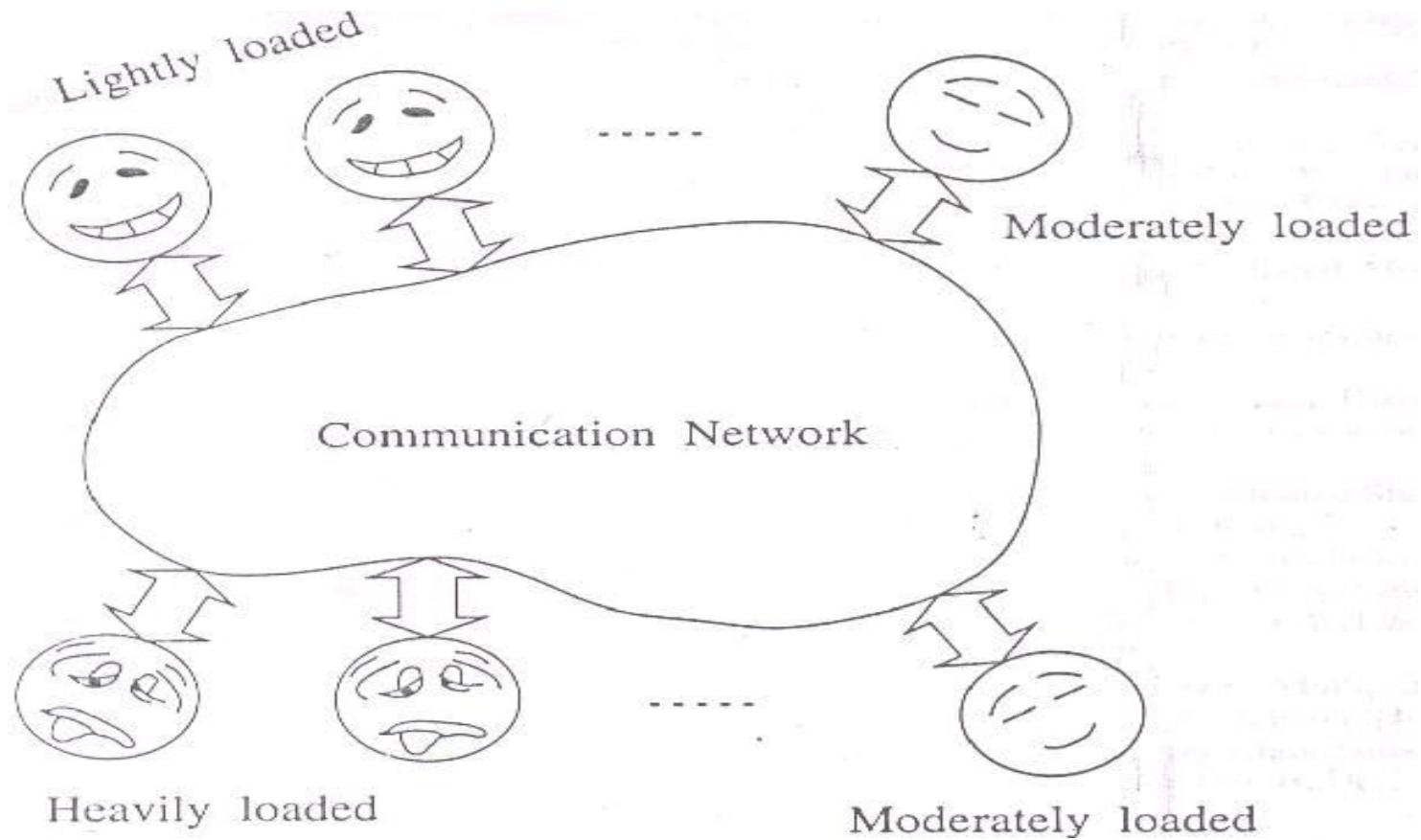
Introduction

- Distributed systems offer a tremendous processing capacity.
- however, in order to realize this tremendous computing capacity, and take full advantage of it, good resource allocation schemes are needed.
- A distributed scheduler is a resource management components of a distributed operating system that focuses on redistributing the load of the system among the computers such that overall performance of the system is maximized.

Introduction

- Due to random arrival of tasks and their random CPU service time requirements, there is a good possibility that several computers are **heavily loaded** (hence suffering from performance degradation), while others are **idle or lightly loaded**.

Introduction



Introduction

- **Goal:** To improve the performance of distributed system by appropriately transferring work from heavily loaded computer to idle or lightly loaded computer.
- Performance of a system
 - One of the metric is **average response time** of task which is the length of the time interval between its origination and completion.
 - Minimizing the **average response time** is often the goal of load distributing.
- Defining a proper characterization of a load at node is very important, as load distributing decisions are based on the load measured at one or more nodes.
- The mechanism used to measure load imposes minimal overhead.

Issues in Load Distributing

Several central issues in load distributing

1. Load:

- CPU queue length are good indicators of load .
- As a result , when all the tasks that the node has accepted have arrived , the node can become overloaded and require further task transfers to reduce its load.
- Prevention- Artificially increment CPU queue length at a node whenever the node accepts a remote task.
- Use timeout to avoid anomalies when task transfers fail.
- After the timeout, if the task has not yet arrived, the CPU queue length is decremented.

Issues in Load Distributing

2. Classification of load distributing algorithms:

Broadly characterized as:

- Static
 - Decision is hard wired in the algorithm, using a prior knowledge of the system.
 - They make no use of system state information (the loads at nodes).

Classification of load distributing algorithms

- Dynamic
 - Make use of system state information to make load distributing decisions
 - They collect, store and analyze the system state information.
 - Impose overhead in collection, storage and analysis of the system state information
- Adaptive
 - Special class of dynamic algorithm
 - they adapt their activities by **dynamically changing the parameters of the algorithm** to suit the changing system state

Issues in Load Distributing

3. Load distributed algorithms can further be classified as **Load balancing** vs. **Load sharing algorithms**

- Both types of algorithms attempt to reduce the likelihood of an *unshared state* by transferring tasks to lightly loaded nodes.
- Unshared state : A state in which one computer lies idle while at the same time tasks argue for service at another computer
- Load balancing algo:
 - Attempt to equalize the loads at all computers

Load balancing vs. Load sharing

- Task transfer are not **instantaneous** because of **communication delays**.
- Delays in transferring a task increase the duration of unshared state.
- To avoid lengthy unshared states, *anticipatory* task transfers from overloaded computers to computers that are **likely to become idle shortly** can be used.
- *Anticipatory* task transfers increase the task transfer rate of a load sharing algorithm.

Issues in Load Distributing

4. Preemptive vs. Non-preemptive transfers:

- Preemptive:
 - Transfer of a task that is partially executed
- Non preemptive:
 - Transfer of a task that has not yet started execution
- Information about the environment in which the task will execute must be transferred to the receiving node. Ex: user's current working directory, privileges inherited by the task

Components of Load Distributing algorithm

- Four components
 - Transfer policy
 - Determines whether a node is in a suitable state to participate in a task transfer
 - Selection policy
 - determines which task should be transferred
 - Location policy
 - determines to which node a task selected for transfer should be sent
 - Information policy
 - responsible for motivating the collection of system state information

Transfer policy

- A large number of the transfer policies are threshold policies.
 - Better to decide the **sender** and **receiver**.
 - when a new task originates at a node, and the load at the node **exceeds** a threshold T , the transfer policy decides that the node is **sender**.
 - If the load at a node **falls** below, the transfer policy decides that the node can be a **receiver** for a remote task.

Selection Policy

- Selects a task for transfer, once the transfer policy decides that the node is a sender
- Simplest approach: to select the newly originated task
- Overhead incurred in task transfer should be compensated by the reduction in the response time realized by the task
- Bryant and Finkel propose another approach - a task is selected for transfer only if its response time will be improved upon transfer.
- Factors to consider:
 - Overhead incurred by transfer should be minimal. For example, a task of small size carries less overhead.
 - Number of location dependent system calls made by the selected task should be minimal

Location Policy

- To find suitable nodes to share load (sender or receiver)
- Widely used method : polling- a node polls another node to find out whether it is suitable node for load sharing
 - Either randomly or on a nearest-neighbor basis
- Alternative to polling
 - Broadcast a query to find out if any node is available for load sharing

Information Policy

- To decide when, where and what information about the states of other nodes on the system should be collected
- One of three types:
 - Demand driven
 - Node collects the state of the other nodes only when it becomes either a sender or a receiver
 - dynamic policy
 - can be sender-initiated , receiver-initiated or symmetrically initiated

Information Policy

- Periodic
 - Nodes exchange load information periodically
- State change driven
 - Nodes distribute state information whenever their state changes by a certain degree
 - Centralized policy- nodes send the state information to a centralized collection points and decentralized policy- nodes send the information to peers.

Load Distributing Algorithms

- Sender-initiated
- Receiver-initiated
- Symmetrically initiated

Sender-Initiated algorithms

- Load distributing activity is initiated by an **overloaded node (sender)** that attempts to send a task to an underloaded node (receiver).
- **Transfer Policy:** A Threshold policy based on CPU queue length **used by the algorithms**
 - A node is identified as a **sender** if a new task originating at the node makes the queue length exceed a **threshold T**.
- **Selection Policy:** These sender-initiated algorithms consider only newly arrived tasks will be transferred.

Sender-Initiated algorithms

- **Location policy:**

1. **Random :**

- Node is randomly selected & task is simply transferred.
- No information exchange to assist in decision making
- **Problem** – useless task transfers may occur when a task is transferred to a node that is already heavily loaded.
- **Solution** – limit the number of times a task can be transferred
- It provides significant performance improvement over no load sharing

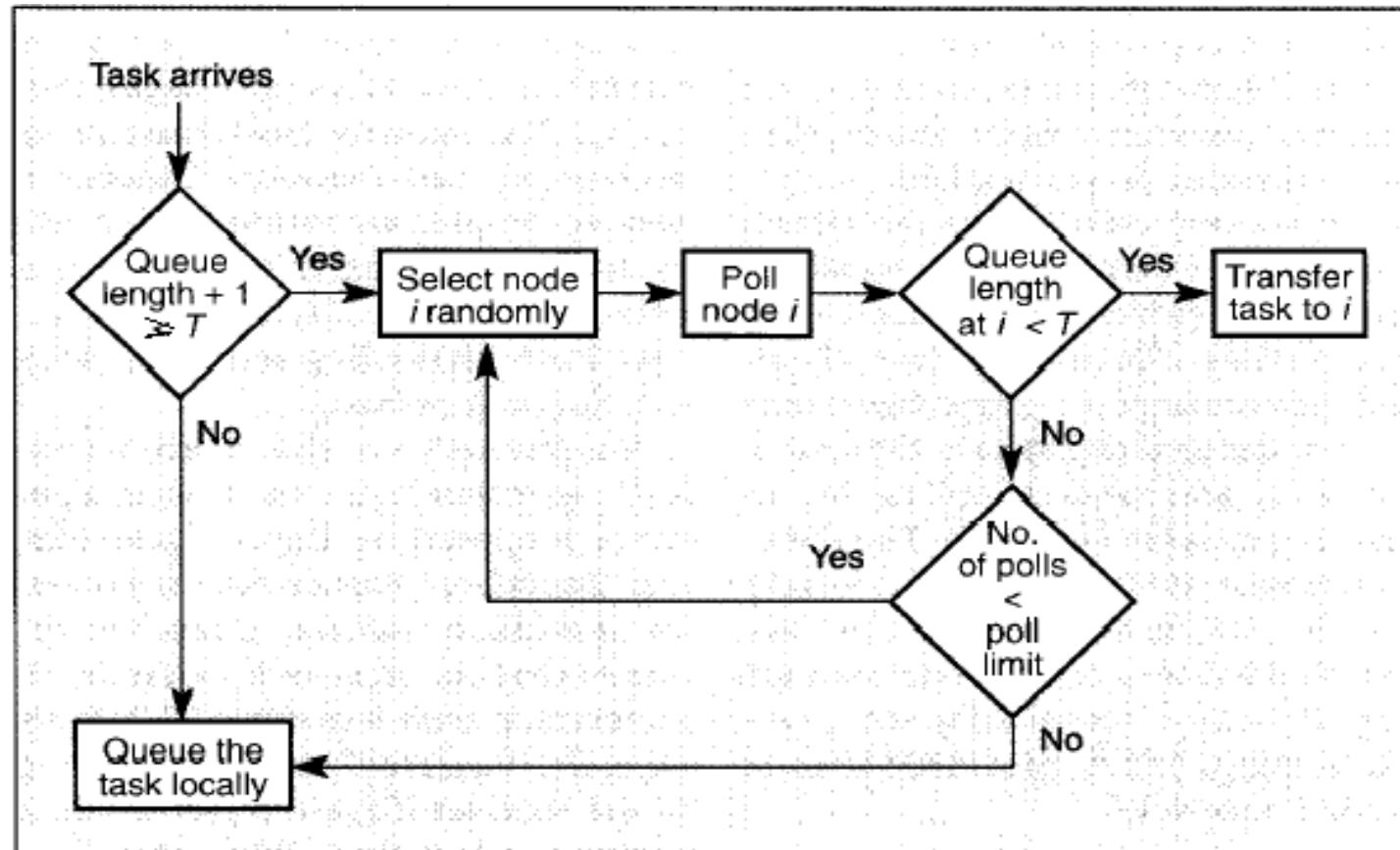
Sender-Initiated algorithms

- **Location policy:**

- 2. **Threshold :**

- The problem of useless task transfer can be avoided using polling (selected a random)
 - Node is polled to determine whether it is a receiver
 - The number of polls is limited by a parameter called ***PollLimit*** to keep the overhead low
 - A sender node will not poll any node more than once during one searching session of ***PollLimit*** polls
 - If no suitable node is found, then the sender node must execute the task
 - It provides significant performance improvement over the random location policy
 - Flowchart

Sender-Initiated algorithms



Sender-Initiated algorithms

- Location policy:

3. Shortest :

- It makes effort to choose the **best receiver** for a task
- Number of nodes (=PollLimit) are selected at random and are polled to determine their queue length
- Node with shortest queue length is selected
- The destination node will execute the task regardless of its queue length at the time of arrival of the transferred task.
- Performance improvement is marginal indicating that more detailed state information does not necessarily result in significant improvement in system performance

Sender-Initiated algorithms

- Information Policy
 - Demand driven
- Stability
 - All three approaches for location policy cause system instability at high system loads.
 - In highly loaded system, probability of finding receiver is very low,
 - When the load **due to work arriving** and due to the **load sharing activity exceeds** the system's serving capacity, instability occurs.

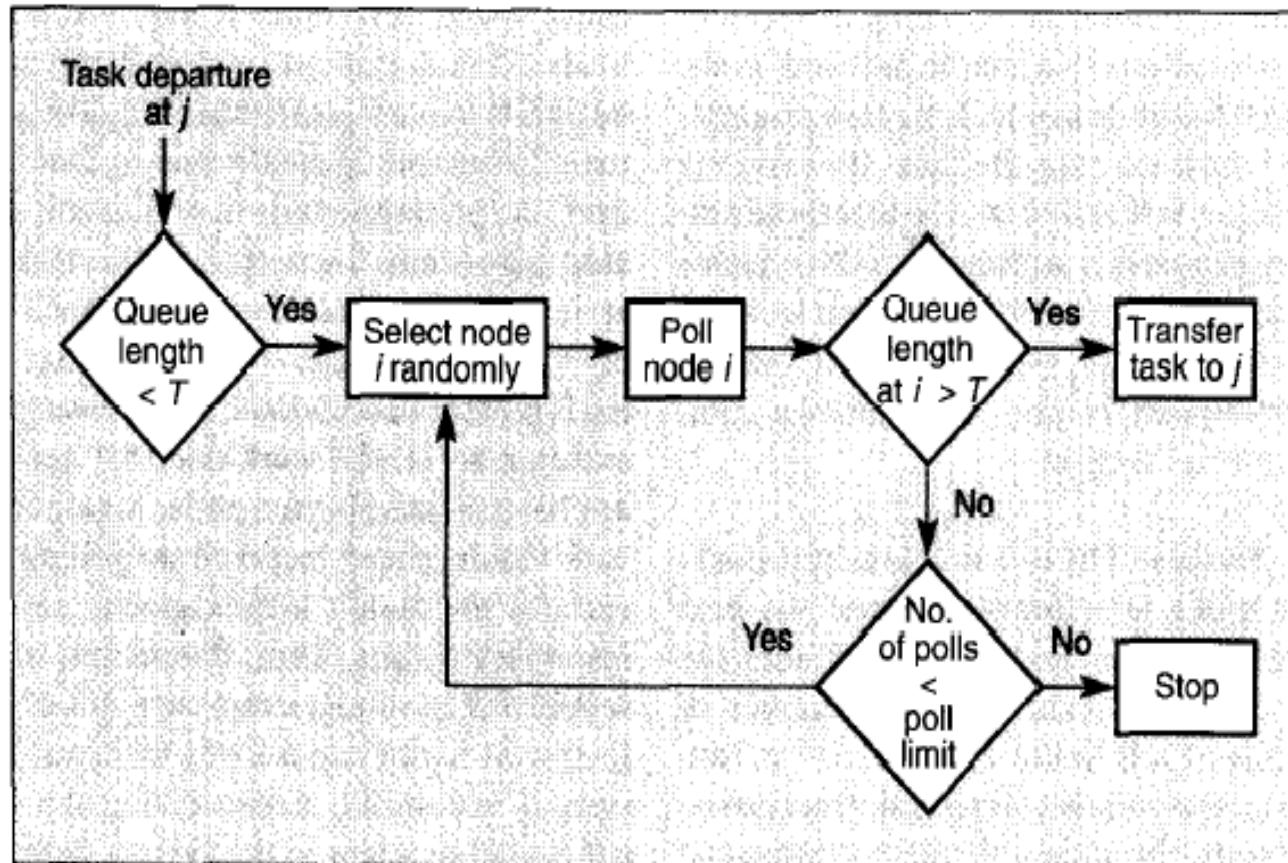
Receiver-Initiated Algorithms

- Load distributing activity is initiated from an under-loaded node (receiver) that is trying to obtain a task from an overloaded node (sender).
- **Transfer Policy:**
 - Threshold policy based on CPU queue length
 - **Transfer policy is triggered when a task departs**
- **Selection Policy:**
 - Any of the approaches as discussed earlier.

Receiver-Initiated Algorithms

- **Location Policy:**
 - Randomly selected node is polled
 - Above procedure is repeated until a **node that can transfer a task (i.e., a sender) is found** or a static PollLimit number of tries have failed to find a sender
 - If all polls fail to find a sender, the node waits until another task completes or until a predetermined period is over before initiating the search for a sender

Receiver-Initiated Algorithms



Receiver-Initiated Algorithms

- **Information Policy:**
 - Demand driven because the polling activity starts only after a node becomes a receiver
- **Stability:**
 - They do not cause system instability
 - Very little wastage of CPU cycles at high system loads
 - At low system loads, more receiver initiated polls do not cause system instability as spare CPU cycles are available
- **Drawback:**
 - Most transfers are preemptive and therefore expensive

Symmetrically Initiated Algorithms

- In symmetrically initiated algorithm, both senders and receivers search for receivers and senders, respectively, for task transfers.
- These algorithms have the advantages of both sender and receiver initiated algorithms.
- At low system loads, the sender-initiated component is more successful in finding under-loaded nodes. At high system loads, the receiver-initiated component is more successful in finding overloaded nodes.
- As in sender-initiated algorithms, cause system instability at high system loads.
- Receiver-initiated algorithms, a preemptive task transfer is necessary.

The Above-Average Algorithms

- **Type of Symmetrically Initiated Algorithm**
- **Transfer Policy:**
 - Threshold policy which uses two adaptive thresholds
 - Thresholds (two) are equidistant from the **node's estimate** of the **average load across all nodes**
- Example
 - Node's estimate of the average load = 2
 - Lower threshold = 1
 - Upper threshold = 3
- A node whose load is greater than upper threshold → a sender
- A node whose load is less than lower threshold → a receiver.
- Nodes that have loads between these thresholds lie within the acceptable range, so they are neither senders nor receivers.

Symmetrically Initiated Algorithms

- **Location Policy:** It has two components
 - Sender-Initiated Components
 - Receiver Initiated Components
- **Sender-initiated component:**
 - **Sender:**
 - Broadcast TooHigh message
 - Set TooHigh timeout alarm
 - Listen for an Accept message until timeout expires
 - **Receiver:**
 - Receive TooHigh message
 - Send Accept message to sender
 - Sets AwaitingTask timeout alarm
 - Increases its load value before accepting a task. Why????
 - If Awaiting Task timeout expires without the arrival of transferred task , the load value at the receiver is decreased.

The Above-Average Algorithms

- On receiving an *Accept* message, if the node is still a sender, it chooses the best process to transfer and transfer it to the node that responded.
- On expiration of TooHigh timeout, if no Accept message is received,
 - Sender guesses that its estimate of the average system load is too low (since no node has a load much lower)
 - Hence, it broadcasts a ChangeAverage message to increase the average load estimate at the other nodes.

The Above-Average Algorithms

- Receiver-initiated component
 - broadcasts a TooLow message
 - Sets a TooLow timeout alarm
 - listening for a TooHigh message
- If a TooHigh message is received, the receiver performs the same actions that it does under sender-initiated negotiation
- If the TooLow timeout expires before receiving any TooHigh message, the receiver broadcasts a ChangeAverage message to decrease the average load estimate at the other nodes

The Above-Average Algorithms

- **Selection policy:** This algorithm can make use of any of the approaches discussed earlier.
- **Information policy:** The information policy is demand-driven.
- **Key Points:**
 1. The average system load is determined individually at each node, imposing little overhead.
 2. Acceptable range determines the responsiveness of the algorithm.

Requirements for Load Distributing

- Scalability: It should work in large distributed system
- Location transparency: It should hide the location of processes.
- Determinism: A transferred process must produce the same results it would produce if it was not transferred.
- Preemption: Remotely executed processes should be preempted and migrated elsewhere on demand.
- Heterogeneity: It should be able to distinguish among different processors.