Distributed Systems

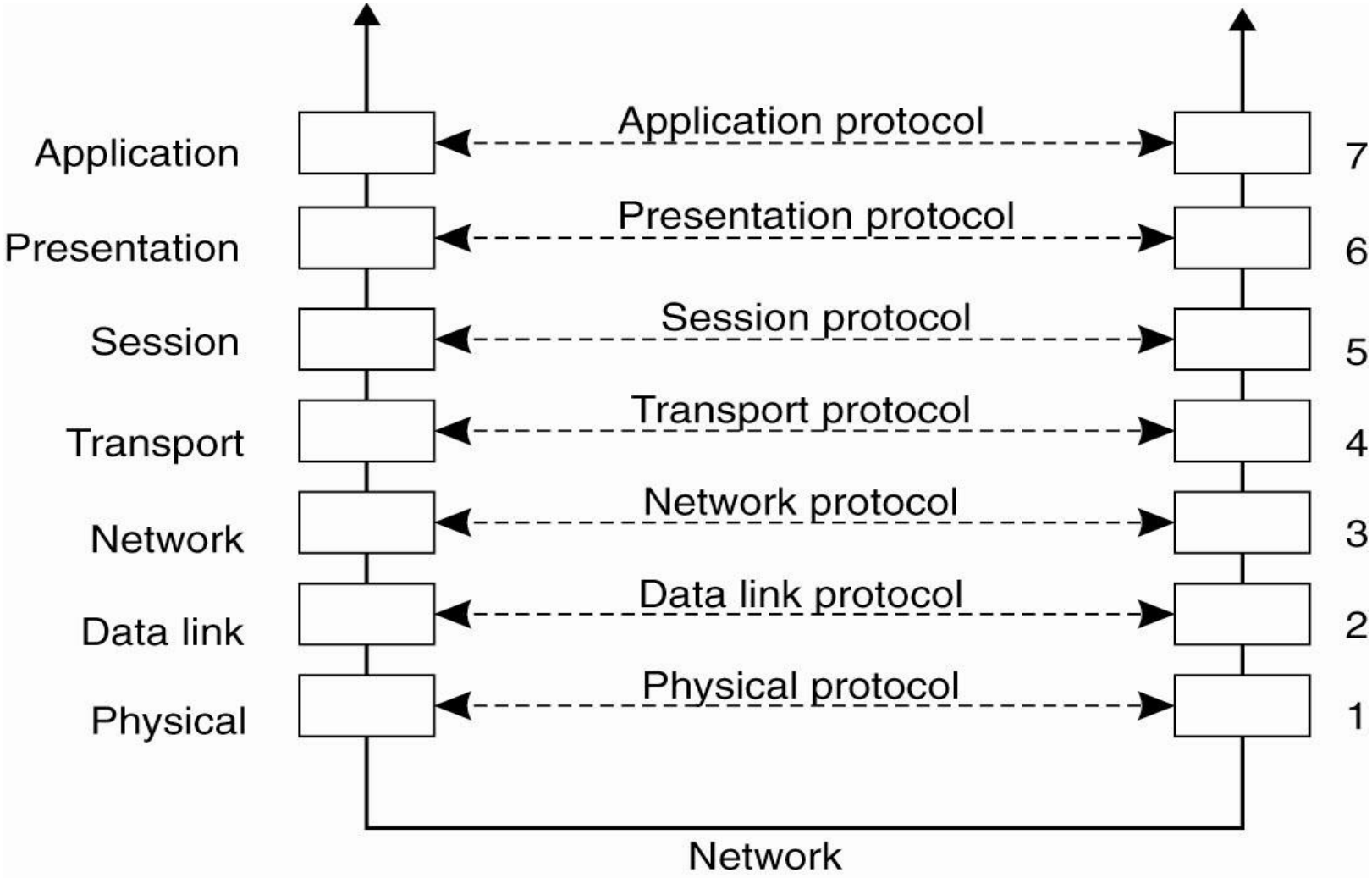
Communication

Chapter 4

Communication in DS

- Interprocess communication is at the heart of all distributed systems. It makes no sense to study distributed systems without carefully examining the ways that processes on different machines can exchange information.
- Communication in distributed systems has traditionally always been based on low-level message passing as offered by the underlying network.
- Expressing communication through message passing is harder than using primitives based on shared memory, as available for non -distributed platforms.
- Modern distributed systems often consist of thousands or even millions of processes scattered across a network

Layers, interfaces, and protocols in the OSI model



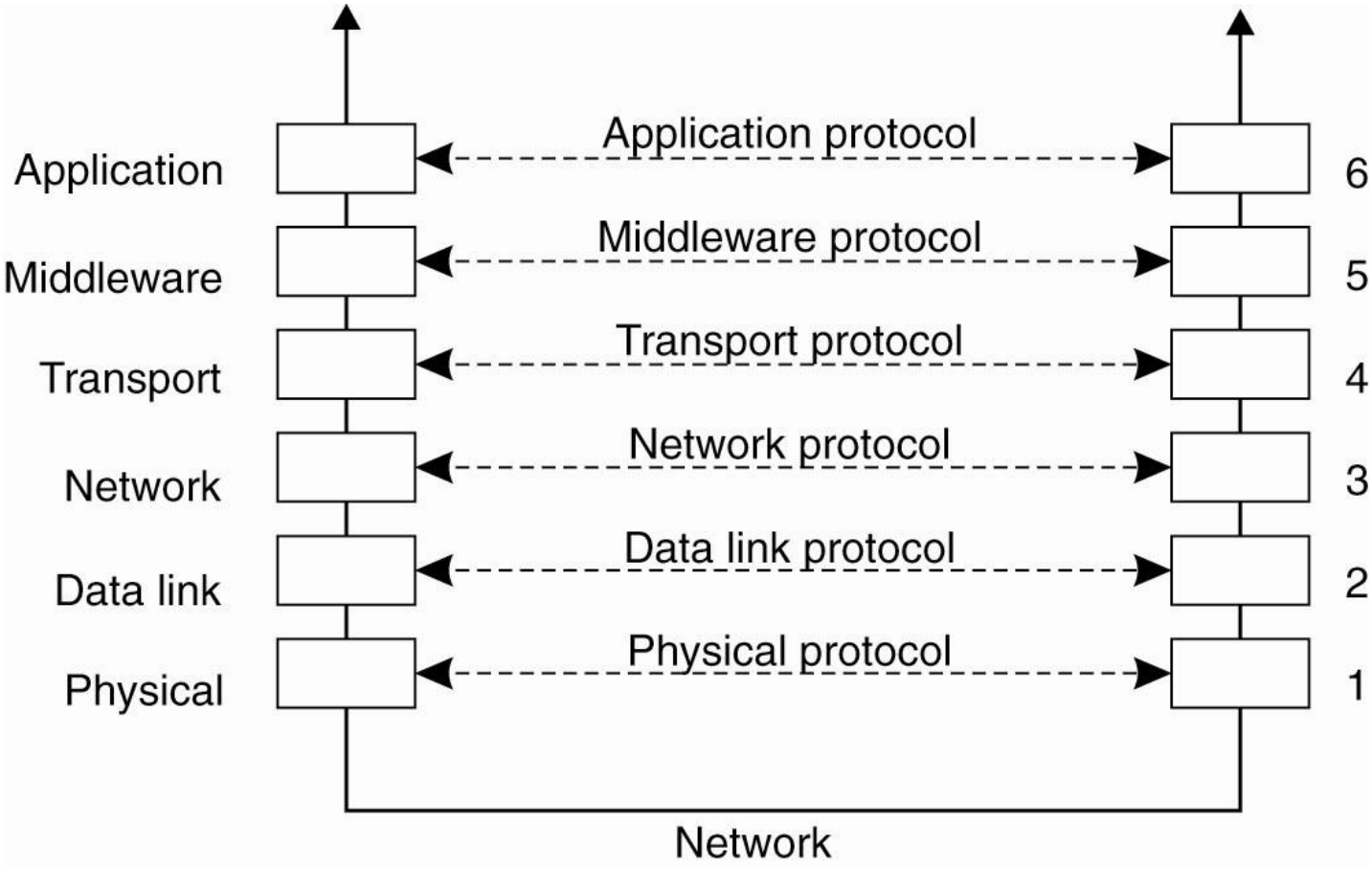
ISO Layers (1)

1. **Physical layer** – handles the mechanical/electrical details of the physical transmission of a bit stream.
2. **Data-link layer** – handles the *frames*, or fixed-length parts of packets, including any error detection and recovery that occurred in the physical layer.
3. **Network layer** – provides connections and routes packets in the communication network, including handling the address of outgoing packets, decoding the address of incoming packets, and maintaining routing information for proper response to changing load levels.

ISO Layers (2)

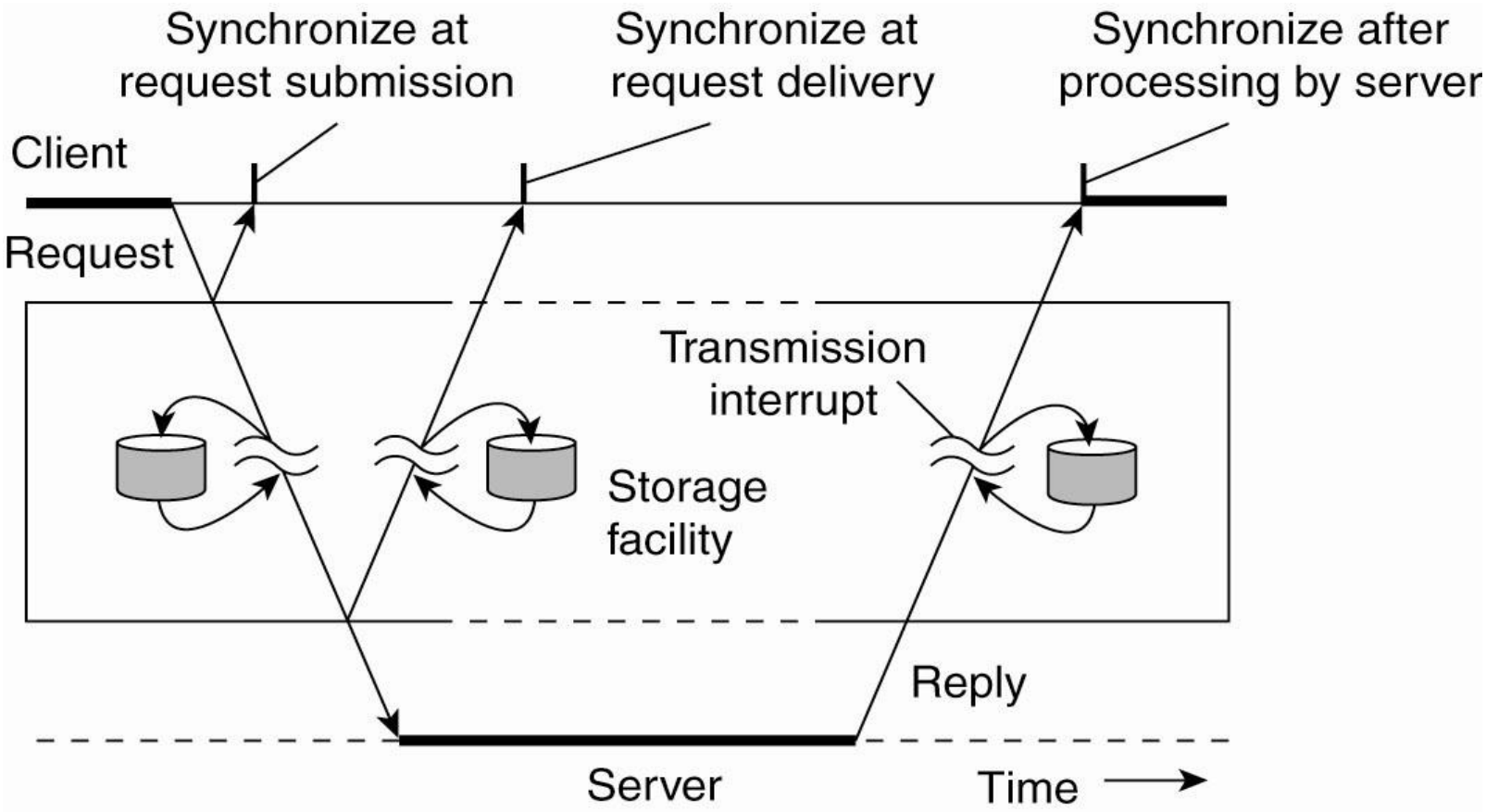
4. **Transport layer** – responsible for low-level network access and for message transfer between clients, including partitioning messages into packets, maintaining packet order, controlling flow, and generating physical addresses.
5. **Session layer** – implements sessions, or process-to-process communications protocols.
6. **Presentation layer** – resolves the differences in formats among the various sites in the network, including character conversions, and half duplex/full duplex (echoing).
7. **Application layer** – interacts directly with the users' deals with file transfer, remote-login protocols and electronic mail, as well as schemas for distributed databases.

Middleware Protocols



An adapted reference model for networked communication

Types of Communication



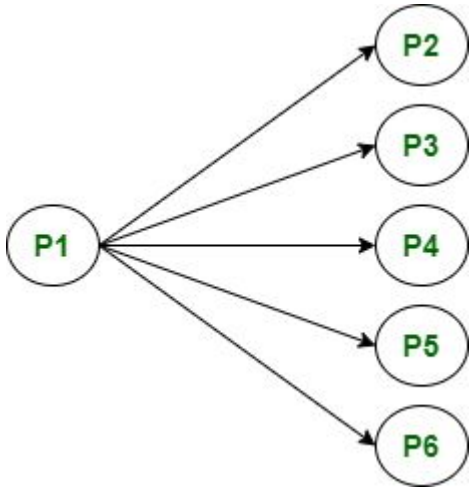
Viewing middleware as an intermediate (distributed) service in application-level communication

Group Communication

- A group is a collection of interconnected processes with abstraction. This abstraction is to hide the message passing so that the communication looks like a normal procedure call. Group communication also helps the processes from different hosts to work together and perform operations in a synchronized manner, therefore increases the overall performance of the system.

Broadcast Communication:

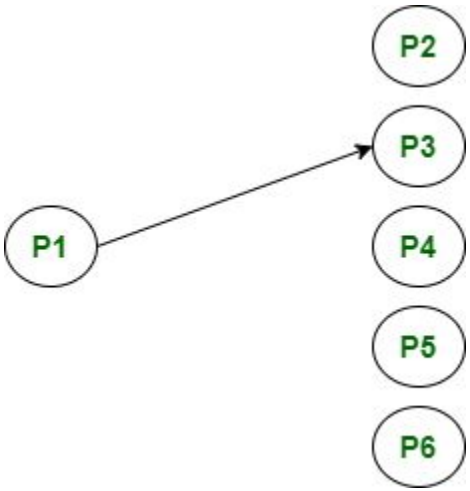
When the host process tries to communicate with every process in a distributed system at same time.



Group Communication

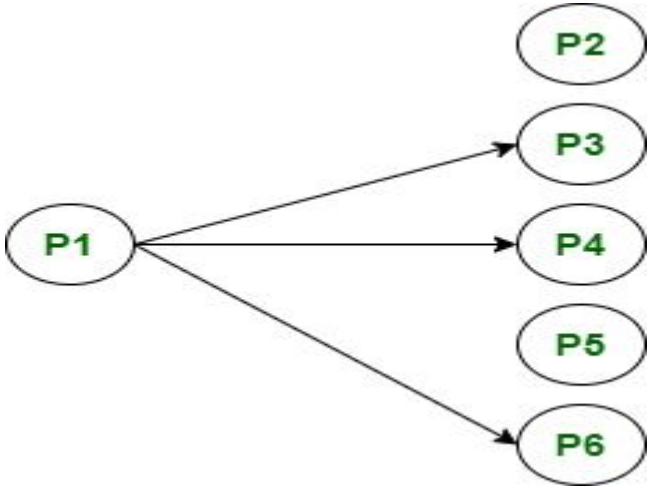
Unicast Communication :

When the host process tries to communicate with a single process in a distributed system at the same time.



Group Communication

- **Multicast Communication :**
- When the host process tries to communicate with a designated group of processes in a distributed system at the same time.



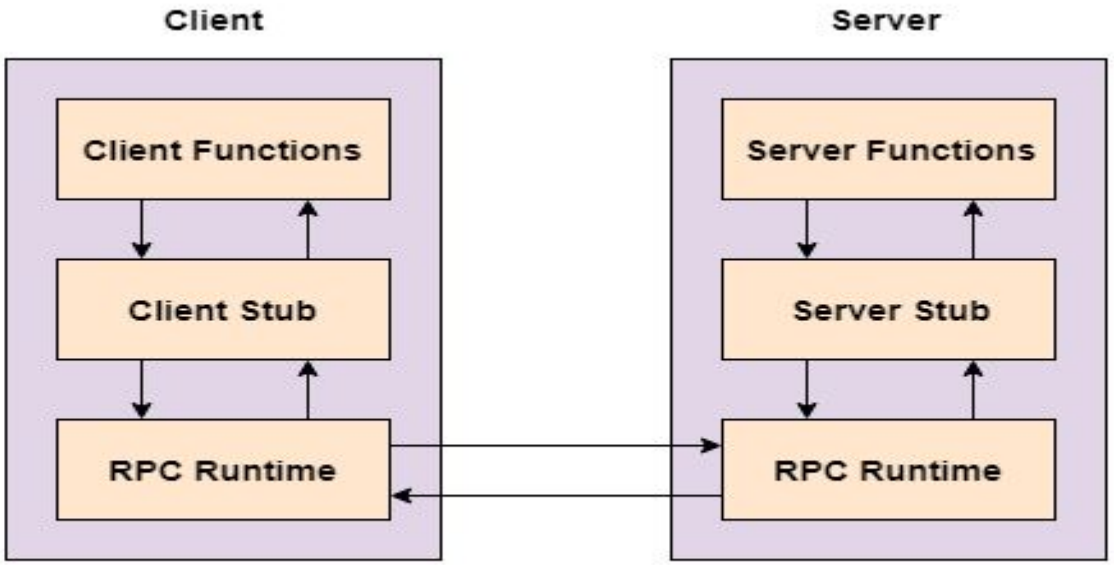
RPC: Remote Procedure Call

- A remote procedure call is an interprocess communication technique that is used for client-server based applications.
- It is also known as a subroutine call or a function call.
- A client has a request message that the RPC translates and sends to the server. This request may be a procedure or a function call to a remote server. When the server receives the request, it sends the required response back to the client. The client is blocked while the server is processing the call and only resumed execution after the server is finished

RPC: Remote Procedure Call

- The sequence of events in a remote procedure call are given as follows −
- The client stub is called by the client.
- The client stub makes a system call to send the message to the server and puts the parameters in the message.
- The message is sent from the client to the server by the client's operating system.
- The message is passed to the server stub by the server operating system.
- The parameters are removed from the message by the server stub.
- Then, the server procedure is called by the server stub.

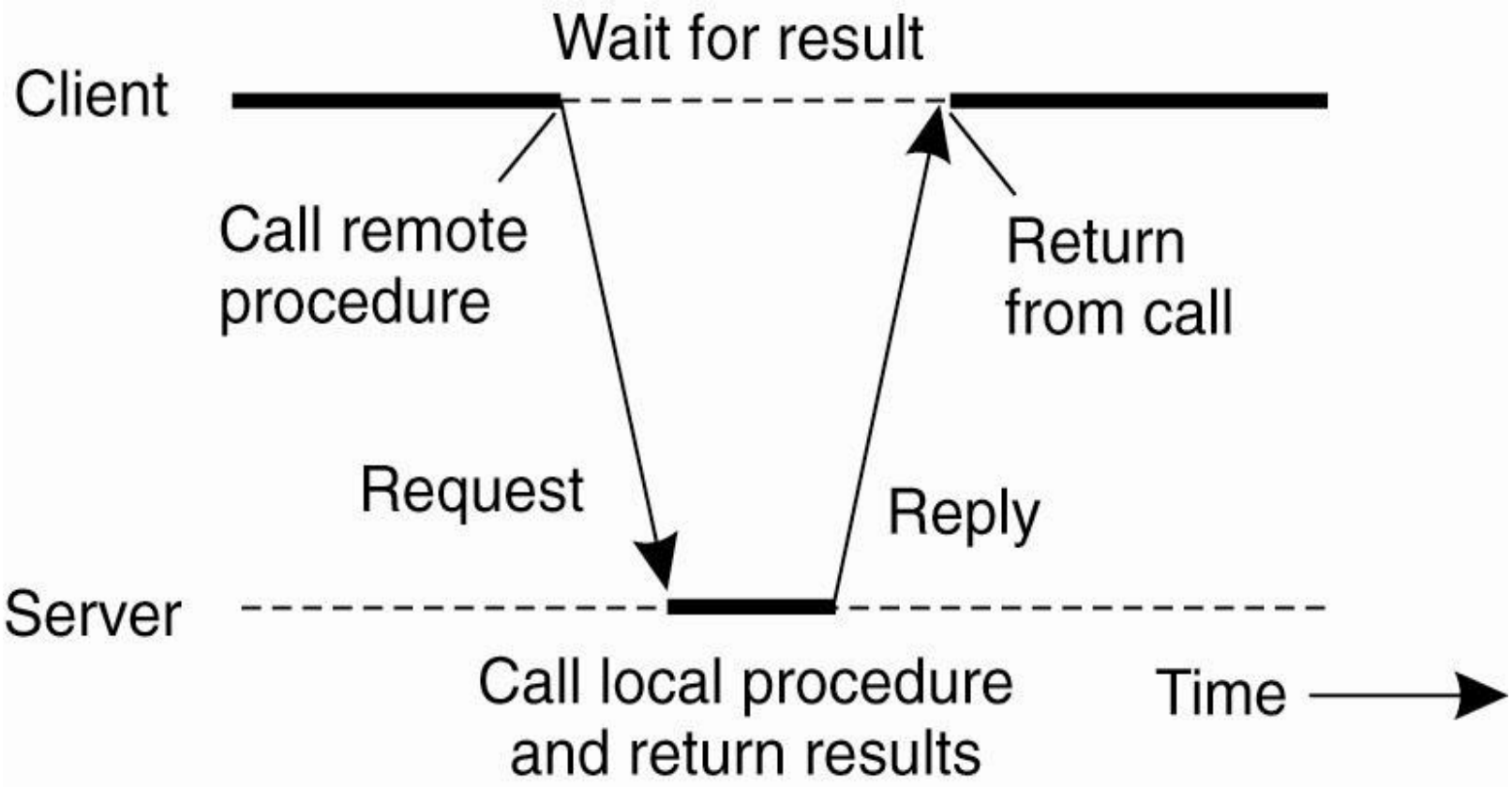
RPC: Remote Procedure Call



How RPC Works: Part 2

- The procedure is "split" into two parts:
  1. The CLIENT "stub" – implements the interface on the local machine through which the remote functionality can be invoked.
  2. The SERVER "stub" – implements the actual functionality, i.e., does the real work!
- Parameters are "marshaled" by the client prior to transmission to the server.

Client and Server Stubs

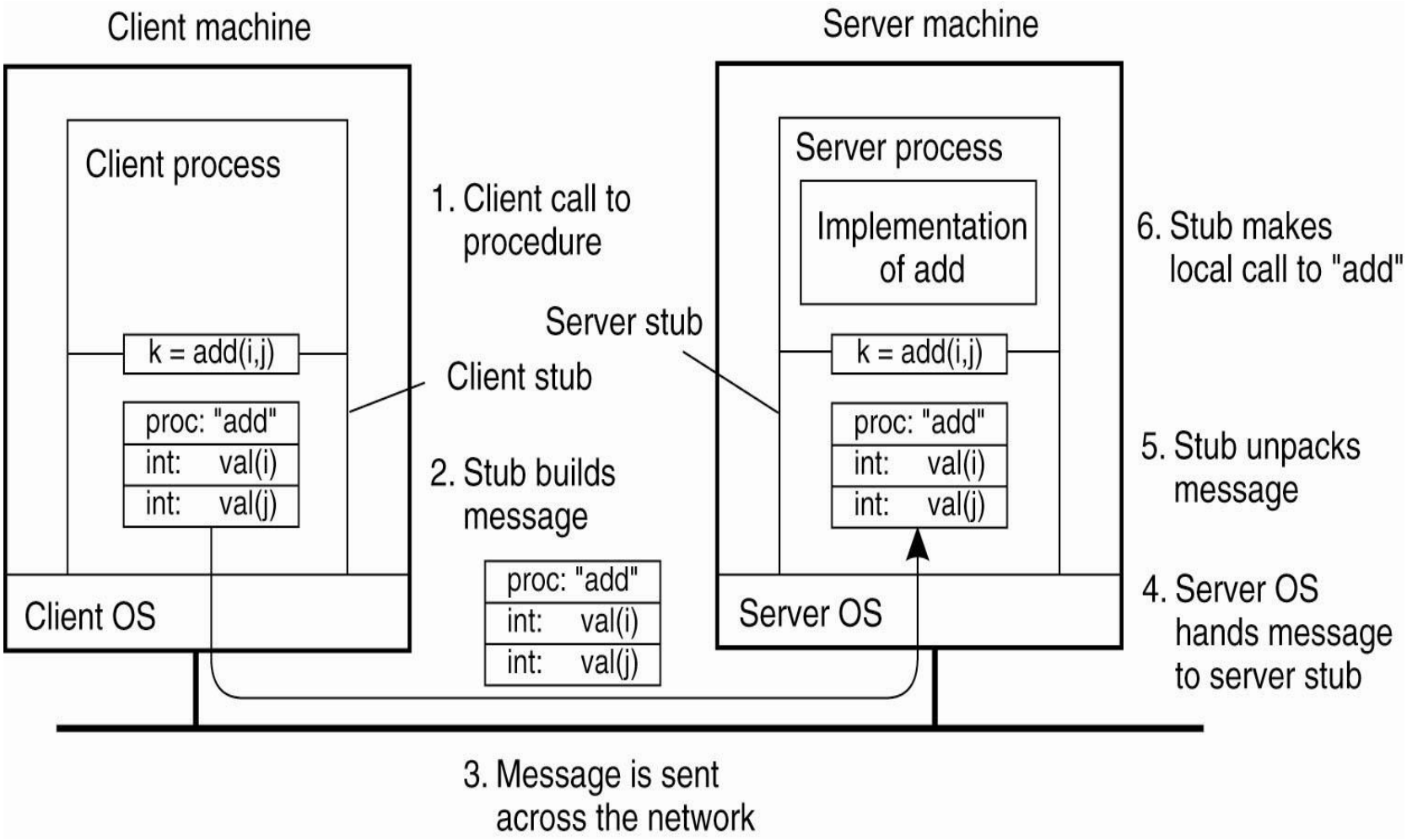


Principle of RPC between a client and server program

The 10 Steps of a RPC

1. Client procedure calls client stub in normal way
2. Client stub builds message, calls local OS
3. Client's OS sends message to remote OS
4. Remote OS gives message to server stub
5. Server stub unpacks parameters, calls server
6. Server does work, returns result to the stub
7. Server stub packs it in message, calls local OS
8. Server's OS sends message to client's OS
9. Client's OS gives message to client stub
10. Stub unpacks result, returns to client

The steps involved in a doing a remote computation through RPC



Issues in RPC

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

- **On the client side**, the stub handles the interface between the client's local procedure call and the run-time system, marshaling and unmarshalling data, invoking the RPC run-time protocol, and if requested, carrying out some of the binding steps.
- **On the server side**, the stub provides a similar interface between the run-time system and the local manager procedures that are executed by the server.

Issues in RPC

3. Binding:

How does the client know who to call, and where the service resides?

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

- **Binding consists of two parts:**
- Naming:
- Locating:
- **A Server** having a service to offer exports an interface for it. Exporting an interface registers it with the system so that clients can use it.
- **A Client** must import an (exported) interface before communication can begin.

Issues in RPC

- **4. The call semantics associated with RPC :**
  It is mainly classified into following choices-
- **Retry request message –**
  Whether to retry sending a request message when a server has failed or the receiver didn't receive the message.
- **Duplicate filtering –**
  Remove the duplicate server requests.
- **Retransmission of results –**
  To resend lost messages without re-executing the operations at the server side

Advantages

- **ADVANTAGES :**
- RPC provides **ABSTRACTION** i.e message-passing nature of network communication is hidden from the user.
- RPC often omits many of the protocol layers to improve performance. Even a small performance improvement is important because a program may invoke RPCs often.
- RPC enables the usage of the applications in the distributed environment, not only in the local environment.
- With RPC code re-writing / re-developing effort is minimized.

Disadvantages

- There is increase in cost because of remote procedure call.
- The remote procedure call is a concept that can be implemented in different ways.It is not a standard.
- There is no flexibility in RPC for hardware architecture.

Message-oriented communication

- Remote procedure calls and remote object invocations contribute to hiding communication in distributed systems, that is, they enhance access transparency.
- Unfortunately, neither mechanism is always appropriate.
- Likewise, the inherent synchronous nature of RPCs, by which a client is blocked until its request has been processed, may need to be replaced by something else. That something else is messaging.

DS Communications Terminology

Persistent Communications:

- An electronic mail system is a typical example in which communication is persistent.
- With **persistent communication**, a message that has been sub- mitted for transmission is stored by the communication middleware as long as it takes to deliver it to the receiver.
- As a consequence, it is not necessary for the sending application to continue execution after submitting the message.
- the receiving application need not be executing when the message is submitted.

DS Communications Terminology

Transient Communications:

In contrast, with **transient communication**, a message is stored by the communication system only as long as the sending and receiving application are executing.

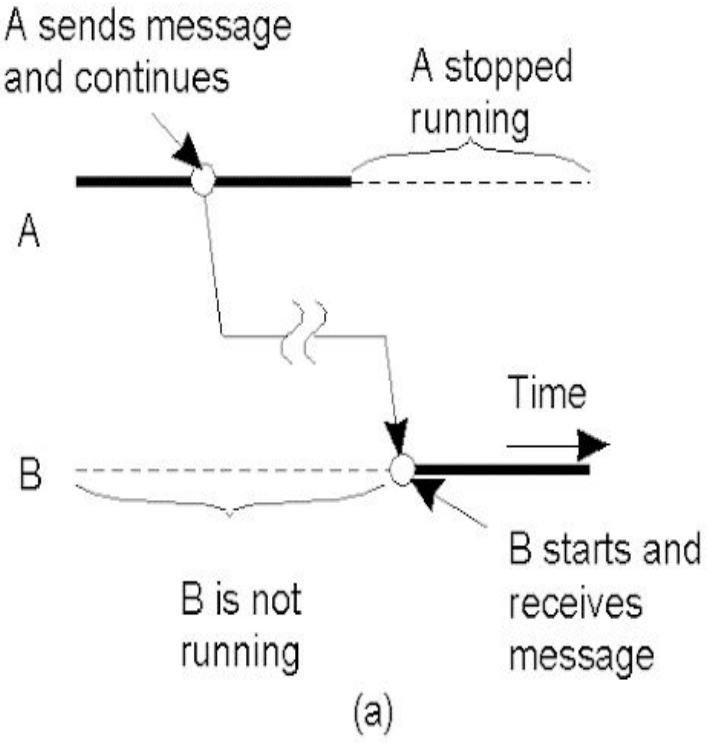
if the middleware cannot deliver a message due to a transmission interrupt, or because the recipient is currently not active, it will simply be discarded.

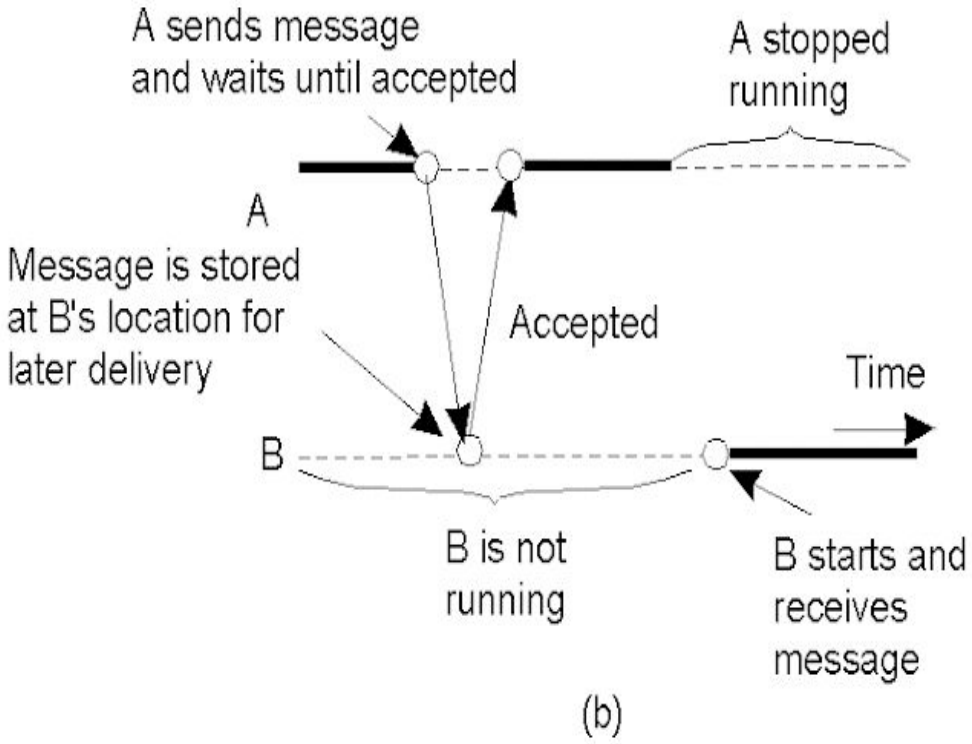
Exmple:TCP/IP communication.

DS Communications

- *Asynchronous Communications:*
  - A sender **continues** with other work immediately upon sending a message to the receiver.
- *Synchronous Communications:*
  - A sender **blocks**, **waiting** for a reply from the receiver before doing any other work. (This tends to be the default model for RPC/RMI technologies).

Classifying Distributed Communications

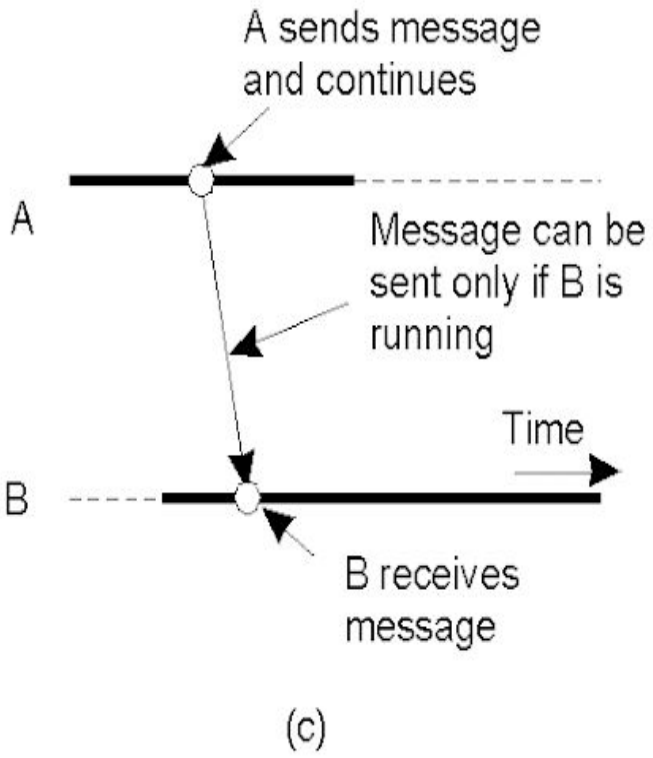


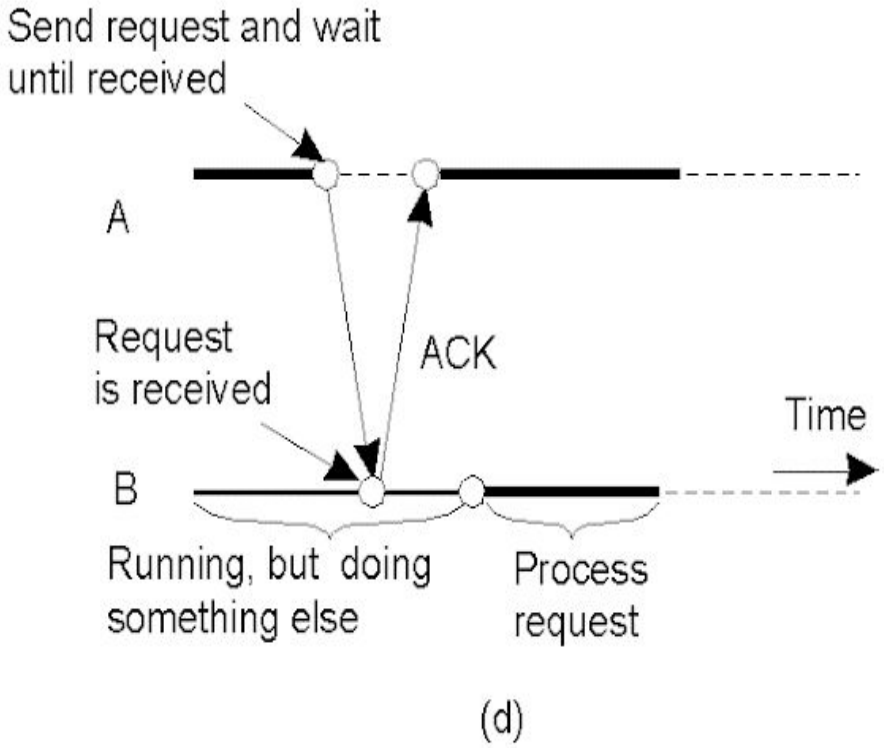


a) Persistent asynchronous communication.

b) Persistent synchronous communication.

Classifying Distributed Communications (2)

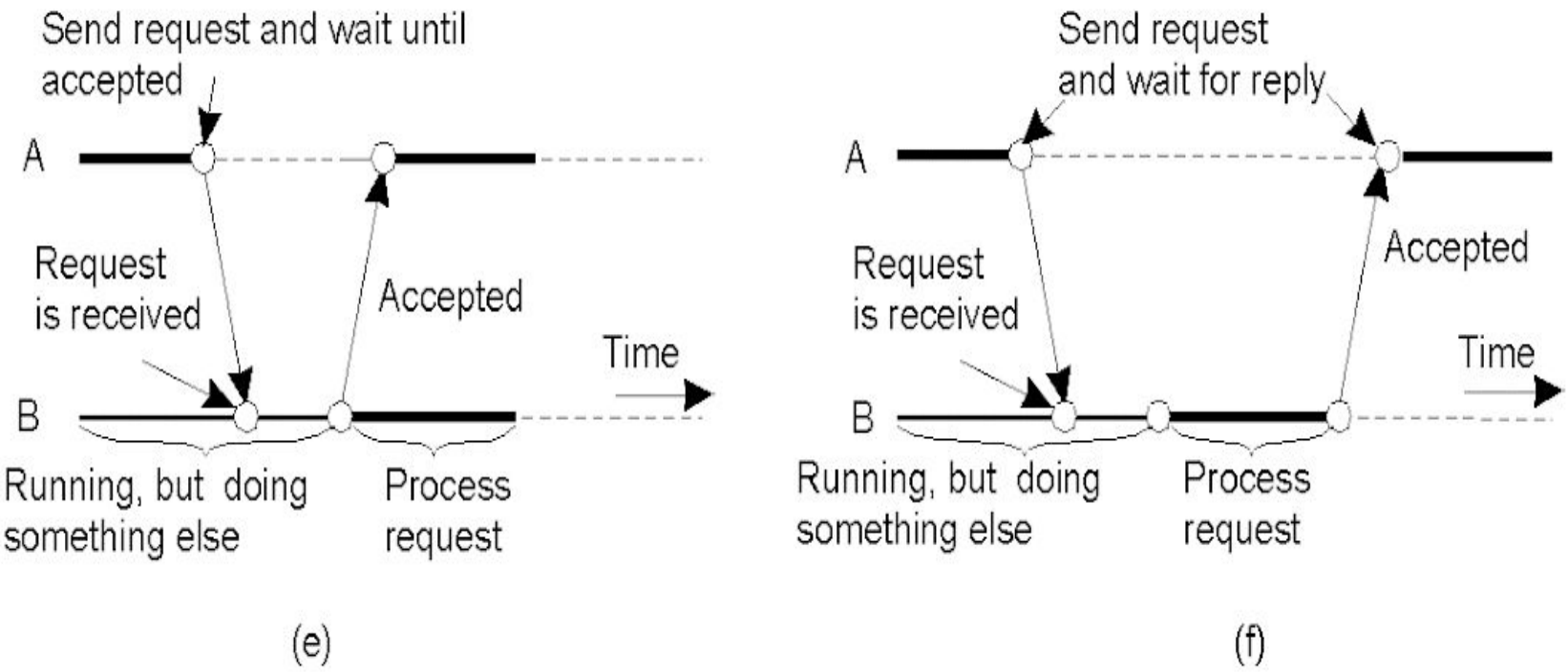




c) Transient asynchronous communication.

d) Receipt-based transient synchronous communication.

Classifying Distributed Communications (3)



e) Delivery-based transient synchronous communication at message delivery.

f) Response-based transient synchronous communication.

Message-oriented communication

- Remote procedure calls and remote object invocations contribute to hiding communication in distributed systems, that is, they enhance access transparency.
- Unfortunately, neither mechanism is always appropriate. In particular, when it cannot be assumed that the receiving side is executing at the time a request is issued, alternative communication services are needed. That something else is messaging

Berkeley socket primitives

- Conceptually, a **socket** is a communication end point to which an application can write data that are to be sent out over the underlying network, and from which incoming data can be read. A socket forms an abstraction over the actual port that is used by the local operating system for a specific transport portocal.

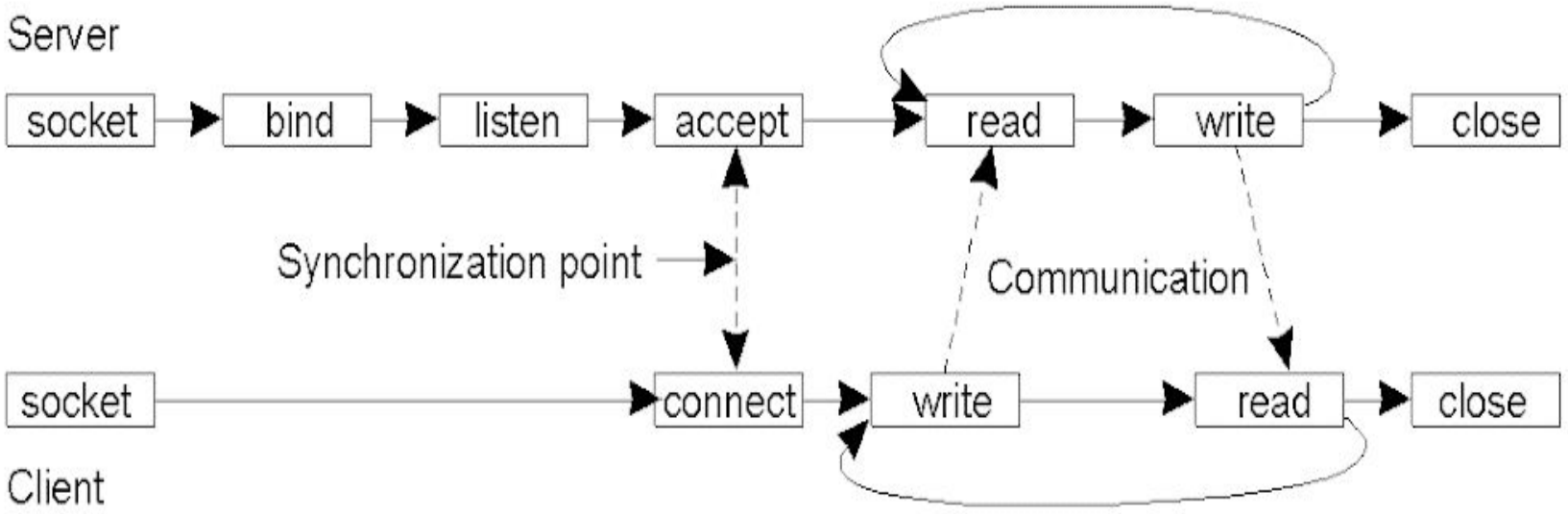
Berkeley Sockets

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

The socket primitives for TCP/IP

Message-Oriented Transient Communications

Initial efforts relied on connection-oriented communication patterns using Sockets (API).



However, DS developers rejected Sockets:

- Wrong level of abstraction (only "send" and "receive").
- Too closely coupled to TCP/IP networks – not diverse enough.

The Message-Passing Interface

- Sockets were deemed insufficient for two reasons. First, they were at the wrong level of abstraction by supporting only simple send and receive operations. Second, sockets had been designed to communicate across networks using general-purpose protocol stacks such as TCP/IP. They were not considered suitable for the proprietary protocols developed for high-speed interconnection networks.
- The result was that most interconnection networks and high-performance multi-computers were shipped with proprietary communication libraries. These libraries offered a wealth of high-level and generally efficient communication operations.

The Message-Passing Interface

- MPI is designed for parallel applications and as such is tailored to transient communication. It makes direct use of the underlying network.
- MPI assumes communication takes place within a known group of processes. Each group is assigned an identifier. Each process within a group is also assigned a (local) identifier. A (groupID, processID) pair therefore uniquely identifies the source or destination of a message, and is used instead of a transport-level address.

The Message-Passing Interface (2)

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

Some of the most intuitive message-passing primitives of MPI

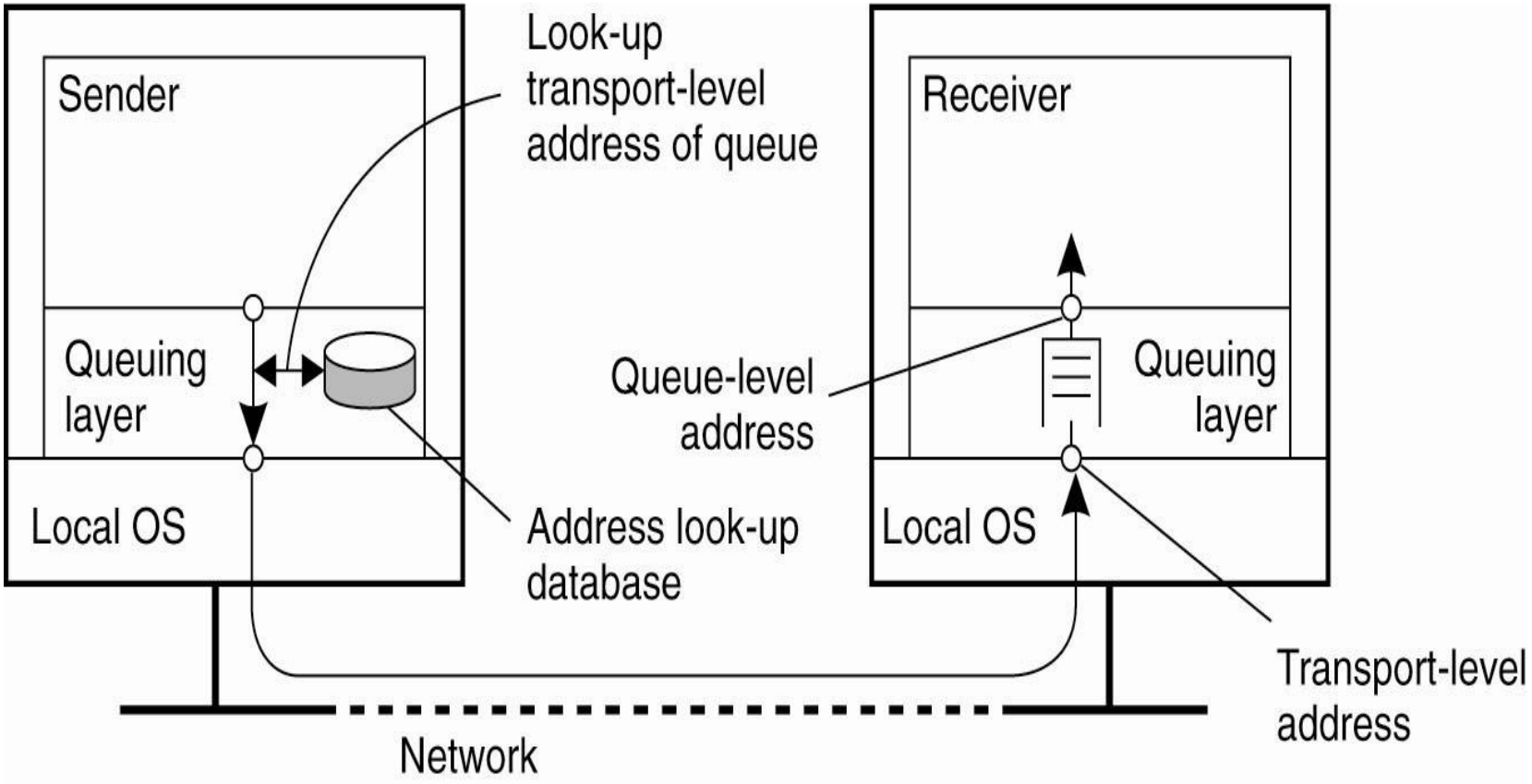
message-queuing systems

- Also known as: “message-queuing systems”.
- They support persistent asynchronous communications.
- Typically, transport can take minutes (hours?) as opposed to seconds/milliseconds.
- **The basic idea**: applications communicate by putting messages into and taking messages out of “message queues”.
- Only guarantee: your message will eventually make it into the receiver’s message queue.
- This leads to “loosely-coupled” communications.

Message-Queuing System Architecture

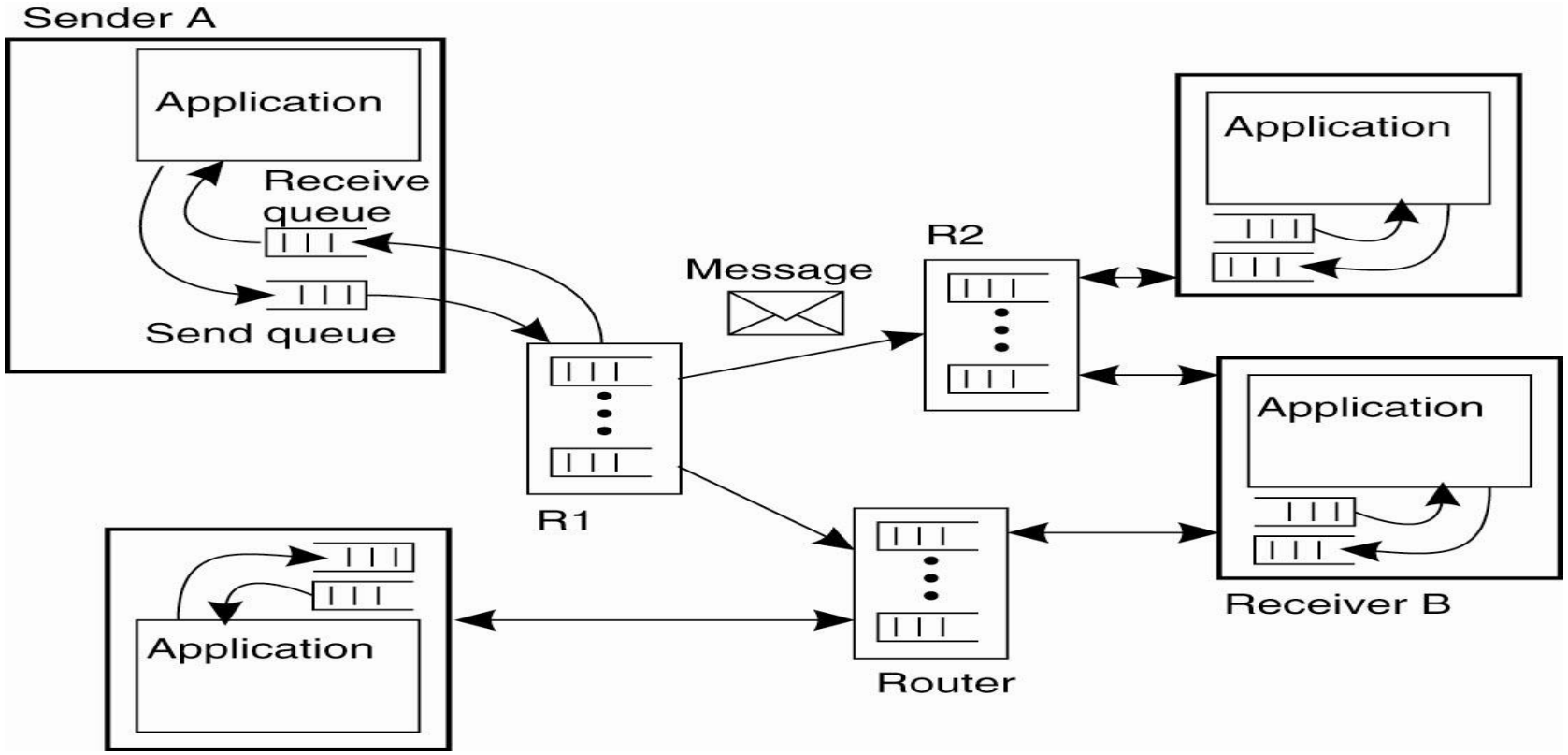
- Messages are “put into” a *source queue*.
- They are then “taken from” a *destination queue*.
- Obviously, a mechanism has to exist to move a message from a source queue to a destination queue.
- This is the role of the *Queue Manager*.
- These are message-queuing “relays” that interact with the distributed applications and with each other.

General Architecture of a Message-Queuing System (1)



The relationship between queue-level addressing and network-level addressing

General Architecture of a Message-Queuing System (2)



The general organization of a message-queuing system with routers

Multicast communication

An important topic in communication in distributed systems is the support for sending data to multiple receivers.