COMMUNICATION

Sunita Sahu
Assistant professor

Module 2: Syllabus

2 Communication 4

2.1 Interprocess communication (IPC):
Remote Procedure Call (RPC),
Remote Method Invocation (RMI).

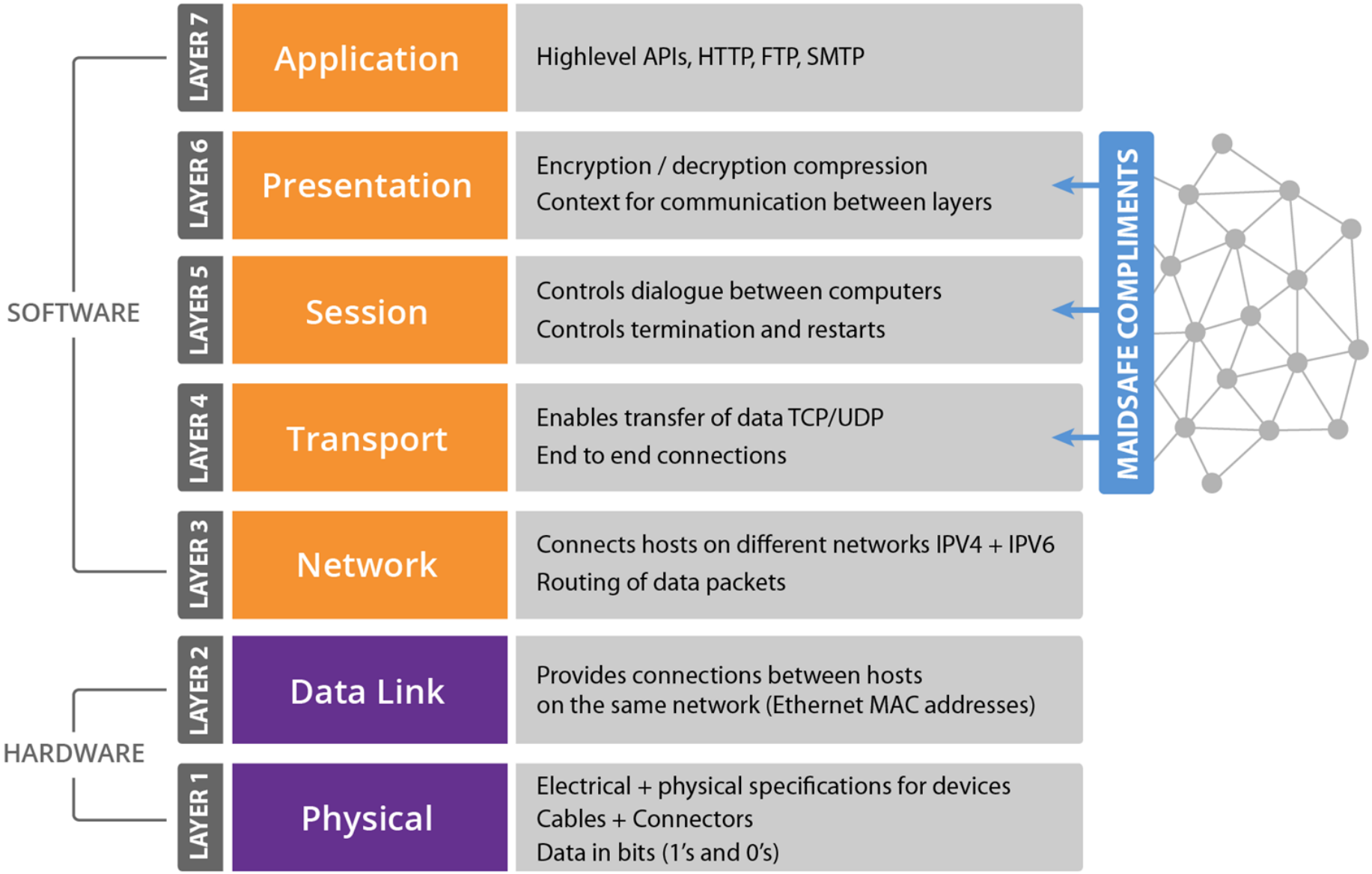
2.2 Message-Oriented Communication,
Stream Oriented Communication,
Group Communication

Desirable Features of a Good Message Passing System

a) Simplicity
b) Uniform Semantics
c) Efficiency
d) Reliability
e) Correctness
f) Flexibility
g) Security
h) Portability

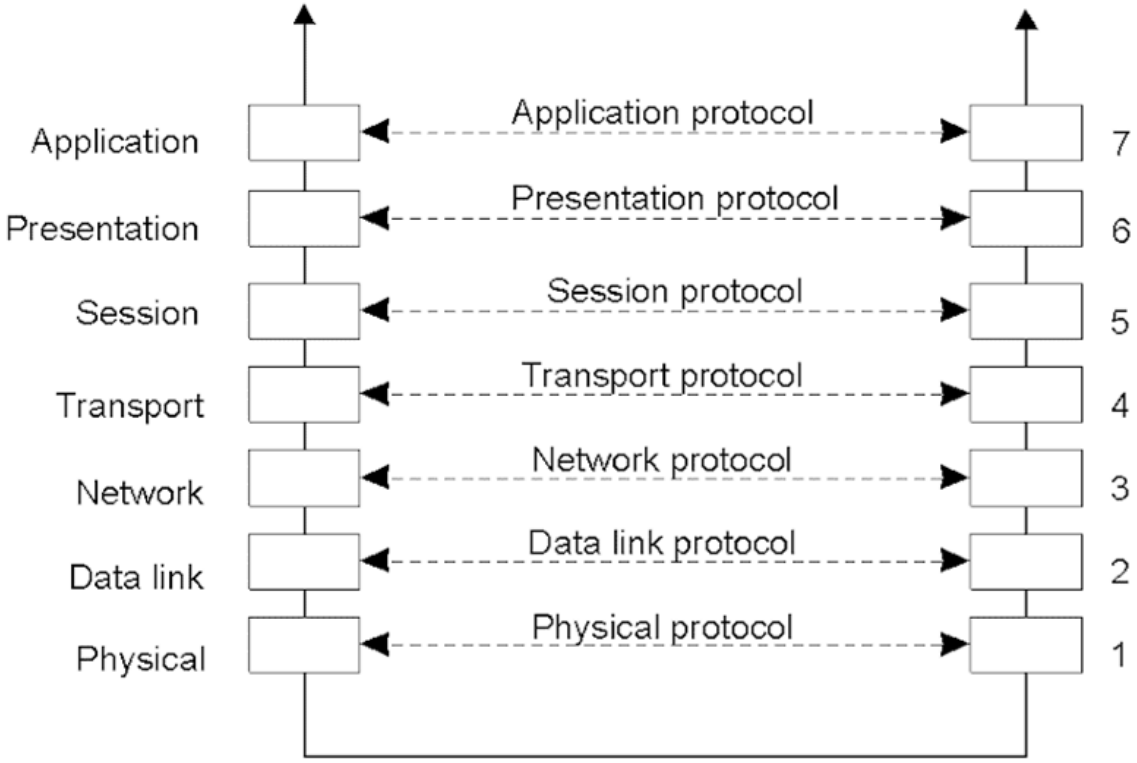
Issues in IPC by Message Passing

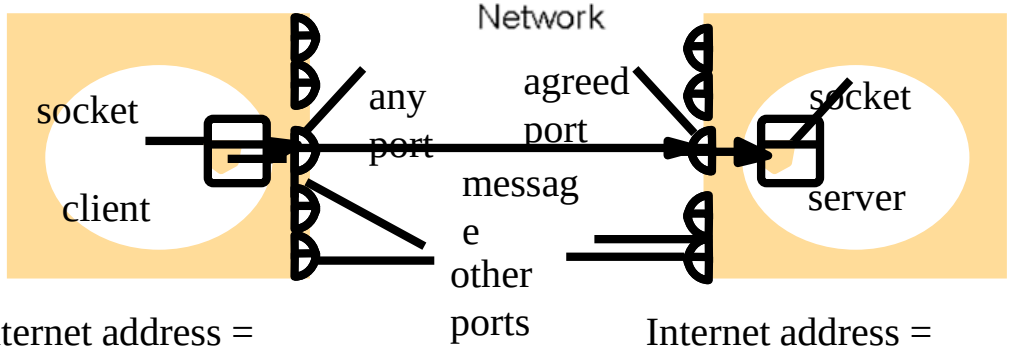
a) Address
b) Sequence No.
c) Structure information
d) Synchronization
e) Buffering


SOFTWARE
HARDWARE
LAYER 7
Application
Highlevel APIs, HTTP, FTP, SMTP
LAYER 6
Presentation
Encryption / decryption compression
Context for communication between layers
LAYER 5
Session
Controls dialogue between computers
Controls termination and restarts
LAYER 4
Transport
Enables transfer of data TCP/UDP
End to end connections
LAYER 3
Network
Connects hosts on different networks IPV4 + IPV6
Routing of data packets
LAYER 2
Data Link
Provides connections between hosts
on the same network (Ethernet MAC addresses)
LAYER 1
Physical
Electrical + physical specifications for devices
Cables + Connectors
Data in bits (1's and 0's)
MAIDSAFE COMPLIMENTS

Layered Protocols (1)

del.





What is Middleware

- Middleware is software that lies between an operating system and the applications running on it.
- Essentially functioning as hidden translation layer, middleware enables communication and data management for distributed applications.
- It's sometimes called plumbing, as it connects two applications together so data and databases can be easily passed between the "pipe."

Remote Procedure Calls

Introduction:RPC

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

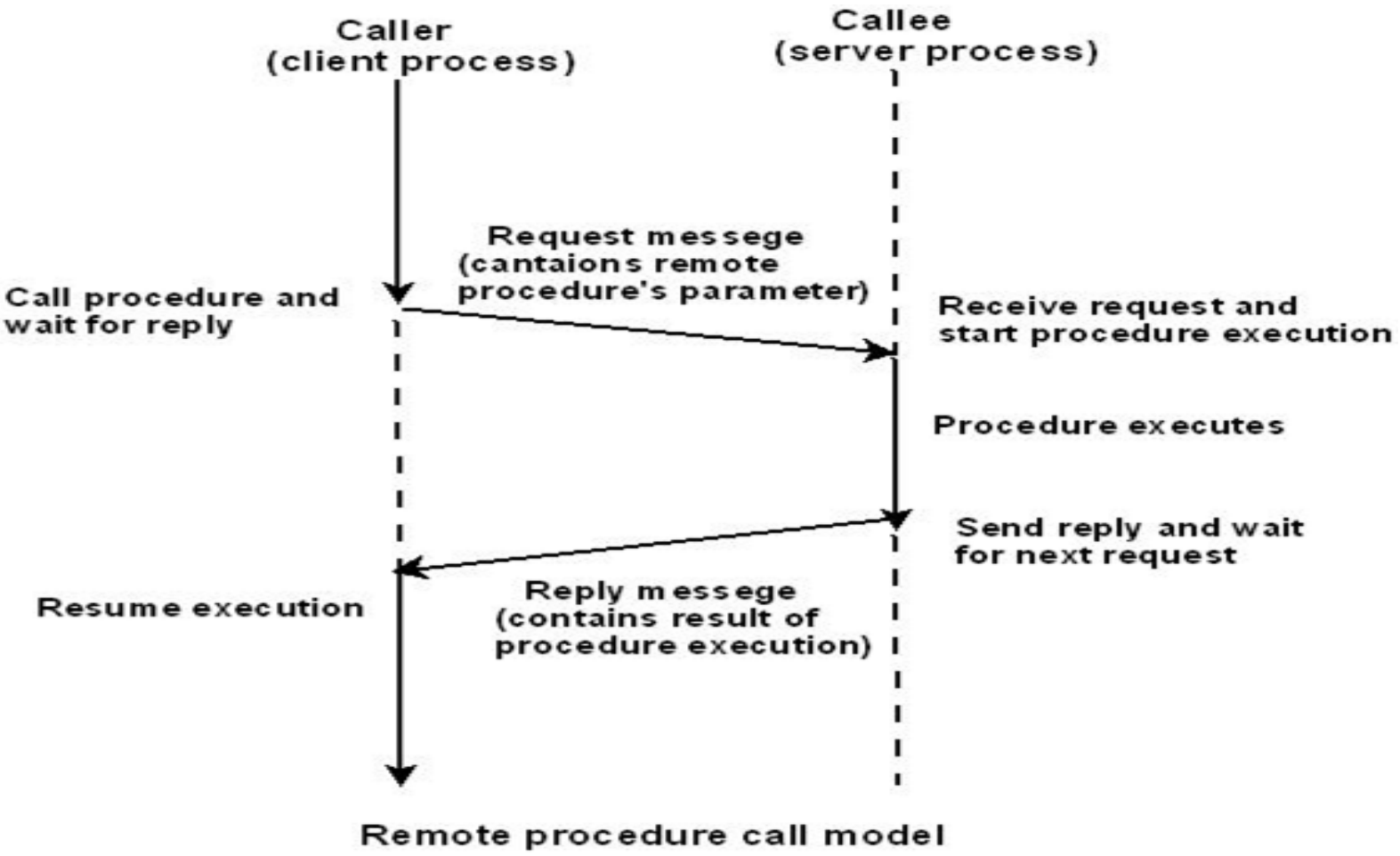
RPC Model

- It is similar to commonly used procedure call model. It works in the following manner:
  - For making a procedure call, the caller places arguments to the procedure in some well specified location.
  - Control is then transferred to the sequence of instructions that constitutes the body of the procedure.
  - The procedure body is executed in a newly created execution environment that includes copies of the arguments given in the calling instruction.
  - After the procedure execution is over, control returns to the calling point, returning a result.

Cont...

- The RPC enables a call to be made to a procedure that does not reside in the address space of the calling process.
- Since the caller and the callee processes have disjoint address space, the remote procedure has no access to data and variables of the callers environment.
- RPC facility uses a message-passing scheme for information exchange between the caller and the callee processes.
- On arrival of request message, the server process
  - extracts the procedure's parameters,
  - computes the result,
  - sends a reply message, and
  - then awaits the next call message.

Cont...



Remote procedure call model

Cont...

- Only one of the two processes is active at any given time.
- It is not always necessary that the caller gets blocked.
- There can be RPC implementations depending on the parallelism of the caller and the callee's environment.
- The RPC could be asynchronous, so that the client may do useful work while waiting for the reply from the server.
- Server could create a thread to process an incoming request so that server is free to receive other requests.

Cont...

Difference between remote procedure calls and local procedure calls:

- Unlike local procedure calls, with remote procedure calls,
  - Disjoint Address Space
  - Absence of shared memory.
  - Meaningless making call by reference, using addresses in arguments and pointers.

- RPC's are more vulnerable to failure because of:
  - Possibility of processor crashes or
  - communication problems of a network.

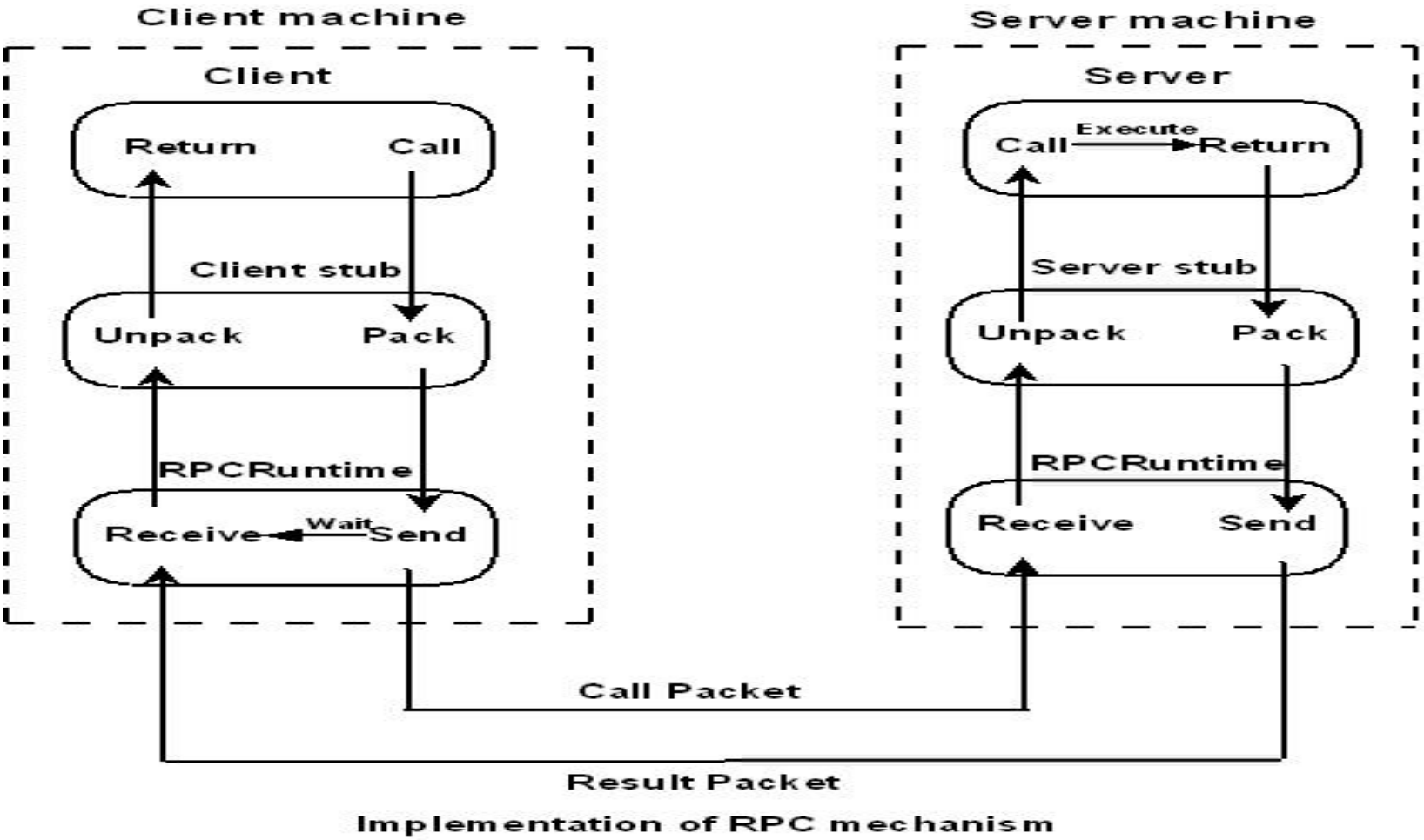
- RPC's are much more time consuming than LPC's due to the involvement of communication network.

Implementing RPC Mechanism

- Implementation of RPC involves the five elements of program:
  - Client
  - Client Stub
  - RPC Runtime
  - Server stub
  - Server
- The client, the client stub, and one instance of RPCRuntime execute on the client machine.
- The server, the server stub, and one instance of RPCRuntime execute on the server machine.
- As far as the client is concerned, remote services are accessed by the user by making ordinary LPC

Stubs

- Provide a normal / local procedure call abstraction by concealing the underlying RPC mechanism.
- A separate stub procedure is associated with both the client and server processes.
- To hide the underlying communication network, RPC communication package known as RPC Runtime is used on both the sides.



Implementation of RPC mechanism

Client Stub

- It is responsible for the following two tasks:
  - On receipt of a call request from the client,
    - it packs a specifications of the target procedure and the arguments into a message and
    - asks the local RPC Runtime to send it to the server stub.
  - On receipt of the result of procedure execution, it unpacks the result and passes it to the client.

RPCRuntime

- It handles transmission of messages across the network between Client and the server machine.

- It is responsible for
  - Retransmission,
  - Acknowledgement,
  - Routing and
  - Encryption.

Server Stub

- It is responsible for the following two tasks:
- On receipt of a call request message from the local RPCRuntime, it unpacks it and makes a perfectly normal call to invoke the appropriate procedure in the server.
- On receipt of the result of procedure execution from the server, it unpacks the result into a message and then asks the local RPCRuntime to send it to the client stub.

Marshalling Arguments and Results

- The arguments and results in RPC are language-level data structures, which are transferred in the form of message data.
- Transfer of data between two computers requires encoding and decoding of the message data.
- Encoding and decoding of messages in RPC is known as **marshaling & Unmarshaling**.

Working

To summarize, a remote procedure call occurs in the following steps:

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

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

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

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

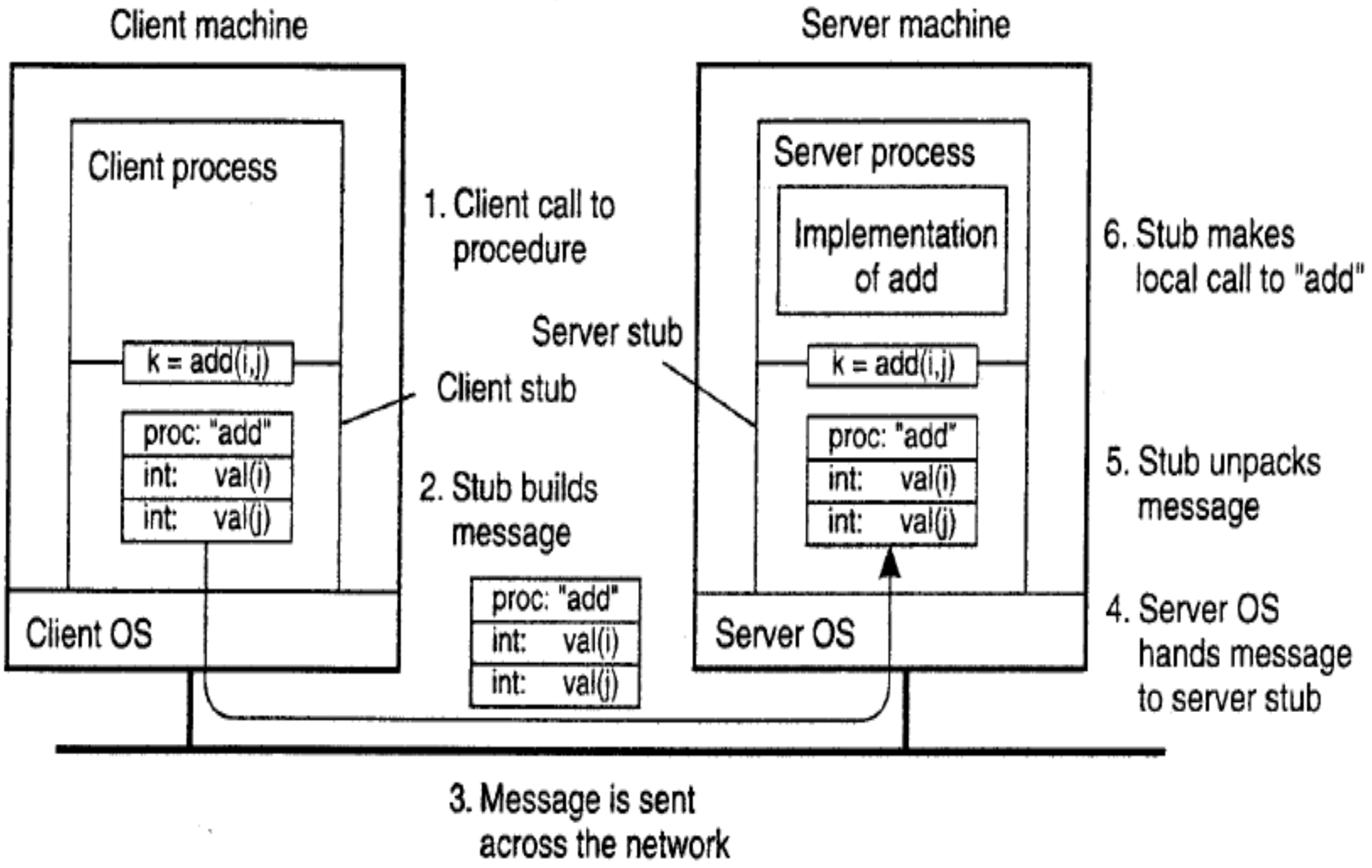
5. The server stub unpacks the parameters and calls the server.

6. The server does the work and returns the result to the stub.
7. The server stub packs it in a message and calls its local as.
8. The server's as sends the message to the client's as.
9. The client's as gives the message to the client stub.
10. The stub unpacks the result and returns to the client.

Call-by-value

- In the call-by-value method, all parameters are copied into a message that is transmitted from the client to the server through the network.
- Packing parameters into a message is called parameter marshaling.
- As a very simple example, consider a remote procedure, add(i, j), that takes two integer parameters *i and j and returns their arithmetic sum as a result.*
- The client stub takes its two parameters and puts them in a message as indicated, It also puts the name or number of the procedure to be called in the message because the server might support several different calls, and it has to be told which one is required.

Call-by-value....



Call-by-value.....

- As long as the client and server machines are identical and all the parameters and results are scalar types. such as integers, characters, and Booleans, this model works fine.
- However, in a large distributed system, it is common that multiple machine types are present.
- Each machine often has its own representation for numbers, characters, and other data items. For example, IRM mainframes use the EBCDIC character code, whereas IBM personal computers use ASCII. As a consequence, it is not possible to pass a character parameter from an IBM PC client to an IBM mainframe server using the simple scheme

Call-by-reference

- Most RPC mechanisms use the call-by-reference semantics for parameter passing
  - The client and the server exist in different address space.

- Remember that a pointer is meaningful only within the address space of the process in which it is being used.
- For example, if the second parameter (the address of the buffer) happens to be 1000 on the client, one cannot just pass the number 1000 to the server and expect it to work.
- Address 1000 on the server might be in the middle of the program text or some thing else.

Call-by-reference...

- One solution is just to forbid pointers and reference parameters in general.
- The client stub knows that the second parameter points to an array of characters. Suppose, for the moment, that it also knows how big the array is.
- One strategy then becomes apparent: copy the array into the message and send it to the server. The server stub can then call the server with a pointer to this array, even though this pointer has a different numerical value than the second parameter of read has.
- Changes the server makes using the pointer (e.g., storing data into it) directly affect the message buffer inside the server stub.
- When the server finishes, the original message can be sent back to the client stub, which then copies it back to the client. In effect, call-by-reference has been replaced by copy/restore.

Some special types or RPCs

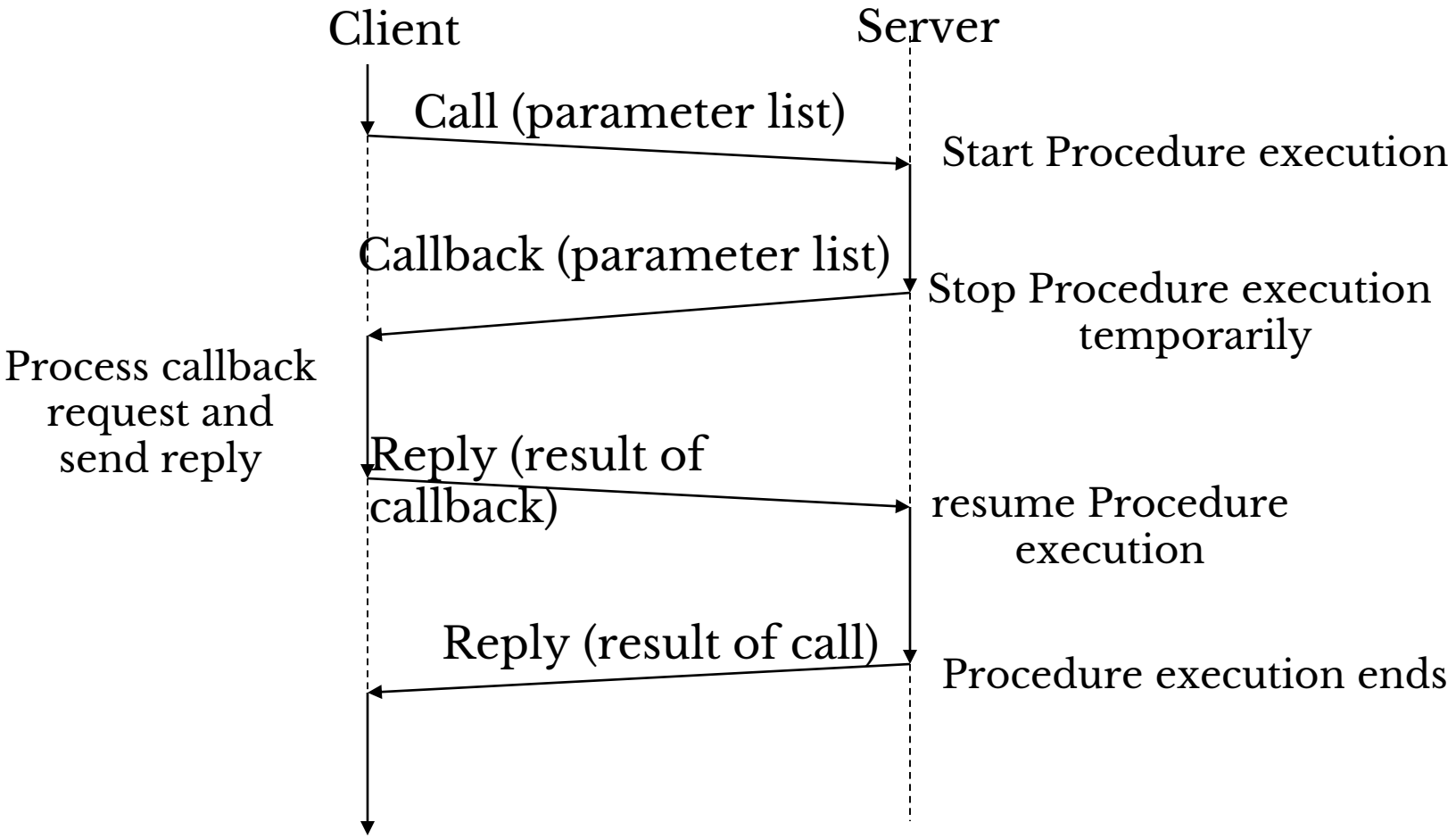
- Callback RPC
- Broadcast RPC
- Batch-mode RPC

Callback RPC

- It facilitates a peer-to-Peer paradigm among participating processes.
- It allows a process to be both a client and a server.
- Remotely processed interactive applications that need user input from time to time or under special conditions for further processing require this type of facility.
- The client process makes an RPC to the concerned server process, and during procedure execution for the client, the server process makes a callback RPC to the client process.

- The client process takes necessary action based on the server's request and returns a reply for the call back RPC to the server process.
- On receiving this reply, the server resumes the execution of the procedure and finally returns the result of the initial call to the client.
- Note that the server may make several callbacks to the client before returning the result of the initial call to the client process.
- Issues
  - Providing server with clients handle
  - Making the client process wait for the callback RPC
  - Handling callback deadlocks

Cont...


Client
Server
Call (parameter list)
Start Procedure execution
Callback (parameter list)
Stop Procedure execution temporarily
Process callback request and send reply
Reply (result of callback)
resume Procedure execution
Reply (result of call)
Procedure execution ends

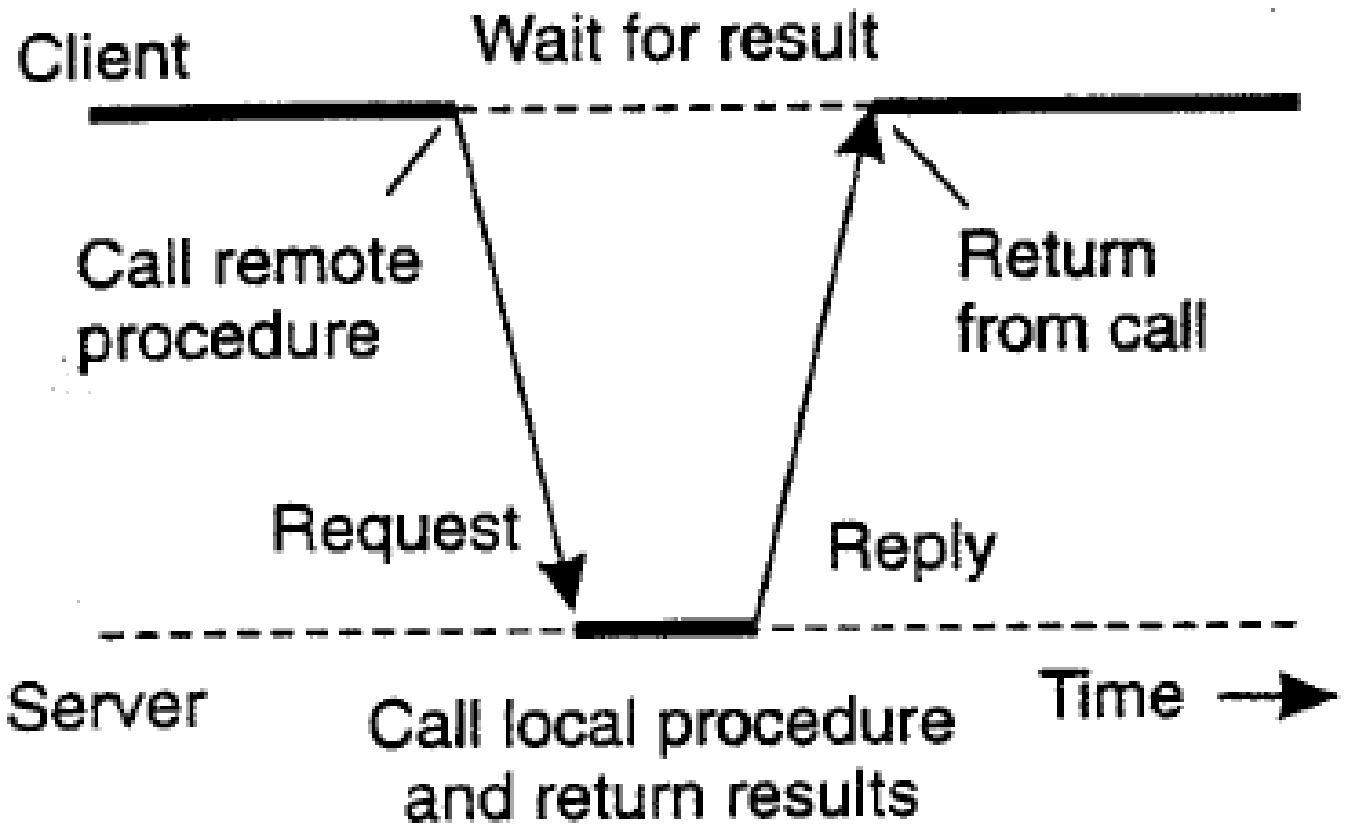
Broadcast RPC

- A client's request is broadcast on the network and is processed by all servers that have the procedure for processing that request.
- Methods for broadcasting a client's request:
- Use of broadcast primitive to indicate that the client's request message has to be broadcasted.
- Declare broadcast ports.
- Back-off algorithm can be used to increase the time between retransmissions.
- It helps in reducing the load on the physical network and computers involved.

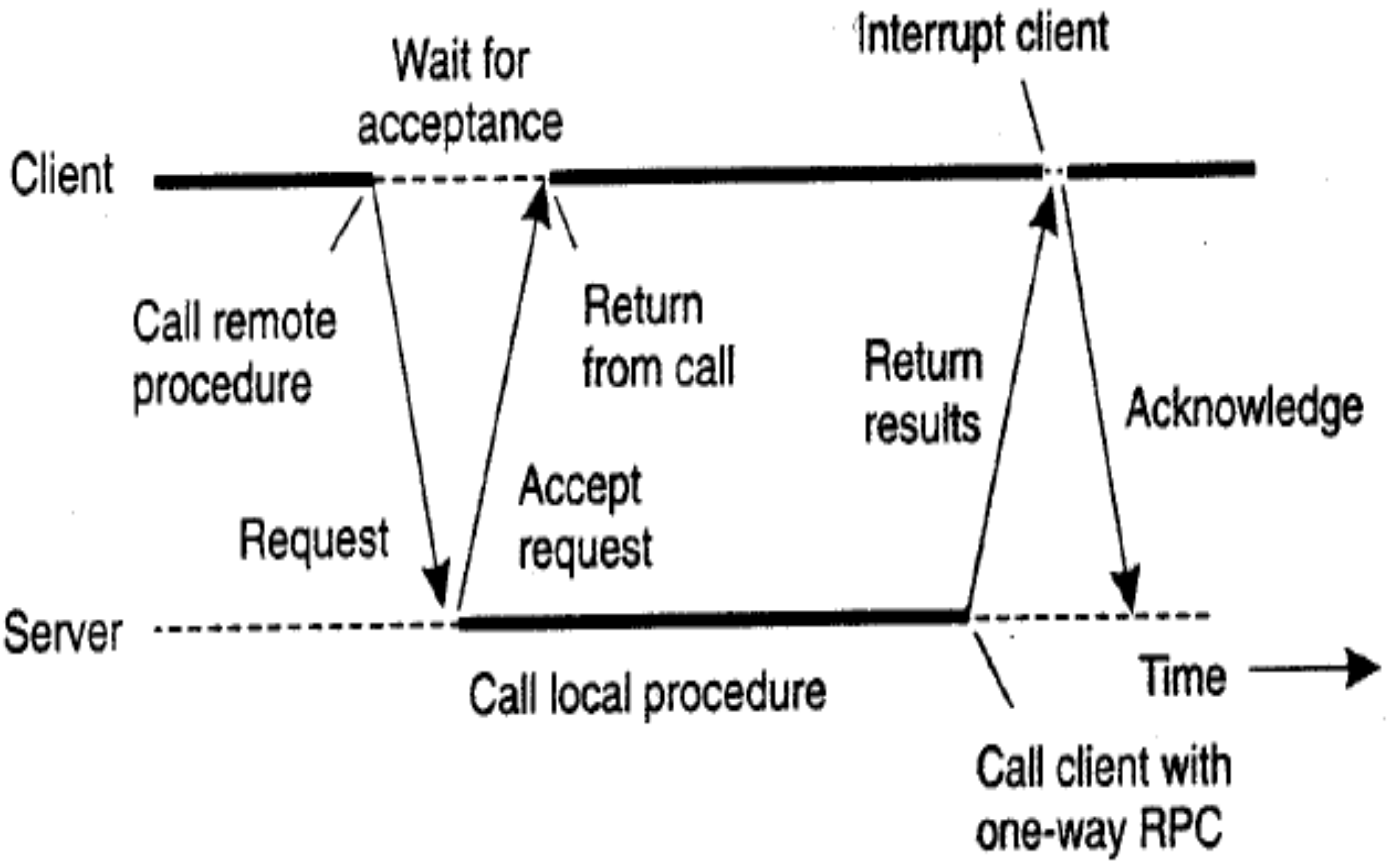
Batch-mode RPC

- Batch-mode RPC is used to queue separate RPC requests in a transmission buffer on the client side and then send them over the network in one batch to the server.
- It reduces the overhead involved in sending requests and waiting for a response for each request.
- It is efficient for applications requiring lower call rates and client doesn't need a reply.
- It requires reliable transmission protocol.

Synchronous RPC



Asynchronous RPC



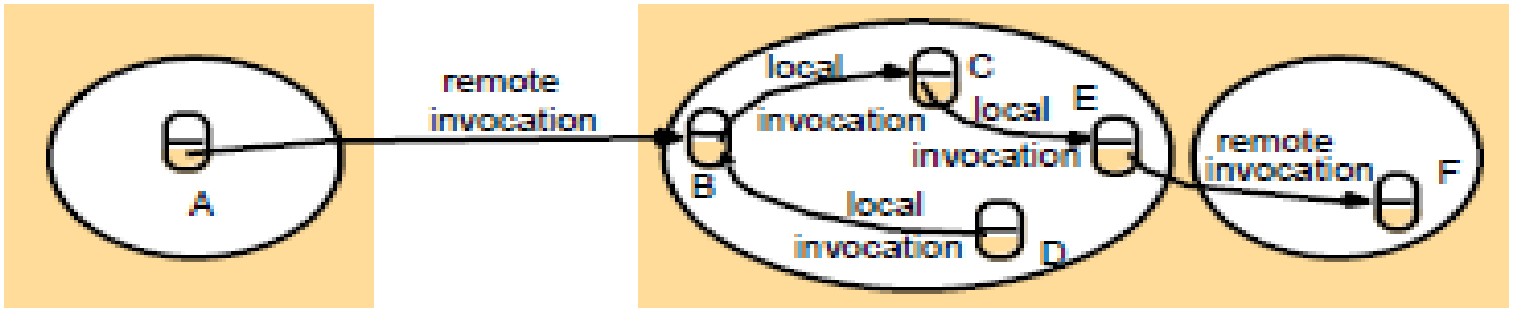
Distributed Objects

Remote Method Invocation

Distributed Objects

- The idea of distributed objects is an extension of the concept of remote procedure calls.
- In a system for distributed objects, the unit of distribution is the object.
- Key feature of an object:: it encapsulates data, the state, and the operations on those data, the methods.
- Methods are made available through an interface. The separation between interfaces and the objects implementing these interfaces is crucial for distributed systems.

Remote and local method invocations



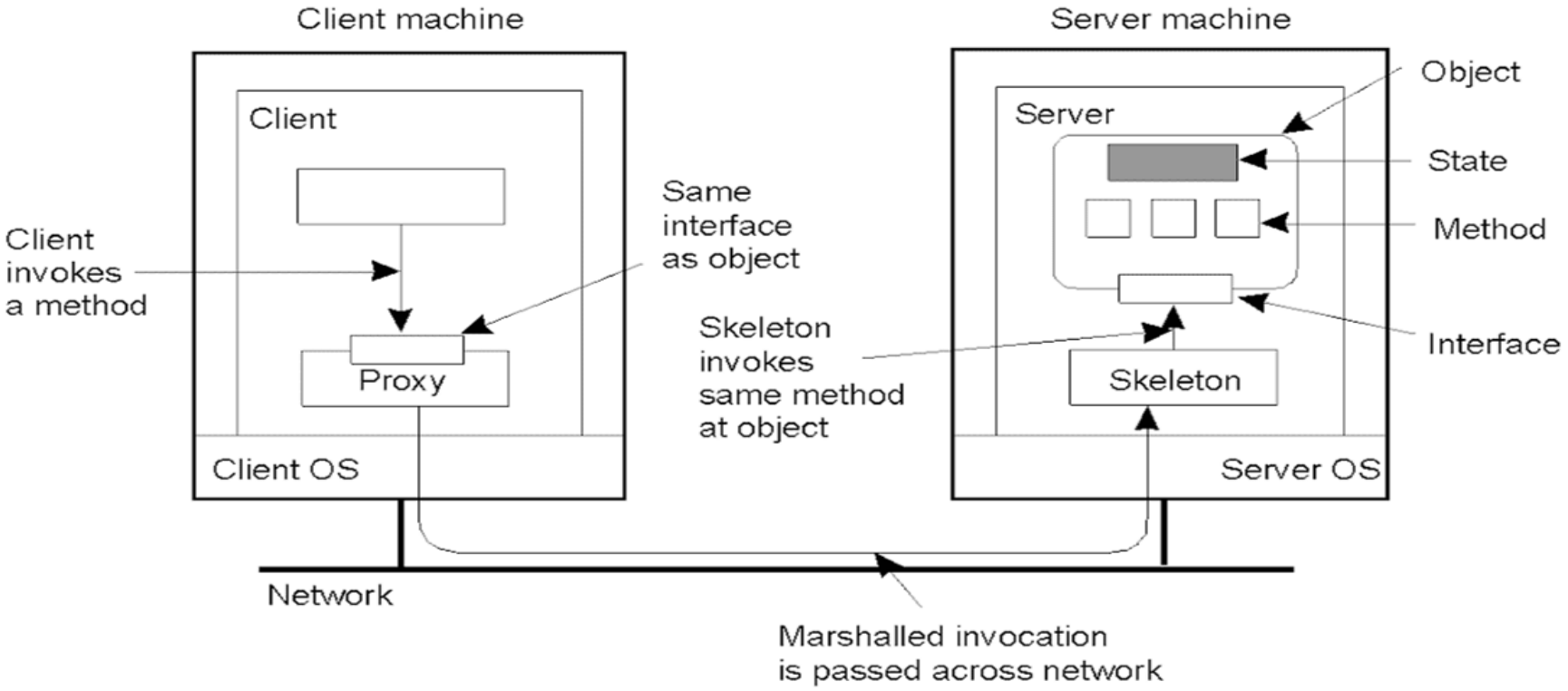
Distributed Objects

- When a client binds to a distributed object, an implementation of the object's interface, a proxy, is loaded into the client's address space.

- The proxy marshals method invocations into messages and unmarshals reply messages to return the result of the method invocation to the client.

Distributed Objects

Common organization of a remote object with client-side proxy.



Distributed Objects

- The actual object resides at a server.
- Incoming invocation requests are first passed to a server stub, a skeleton, that unmarshals them to proper method invocations at the object's interface at the server.
- The skeleton also marshals replies and forwards replies to the client-side proxy.

Remote Method Invocation

- After a client is bound to an object, it can invoke the object's method through the proxy.
- Such a *remote method invocation (RMI)* *is similar to a RPC with respect to marshaling and parameter passing.*
- *The difference is that RMI supports systemwide object references.*
- *An interface definition language is used in RMI to specify the object's interface.*
- *Static invocation* *implies using object-based languages (e.g., Java) to predefine interface definitions.*
- *Dynamic invocation* *permits composing a method invocation at run-time*

RPC & RMI: Similarities

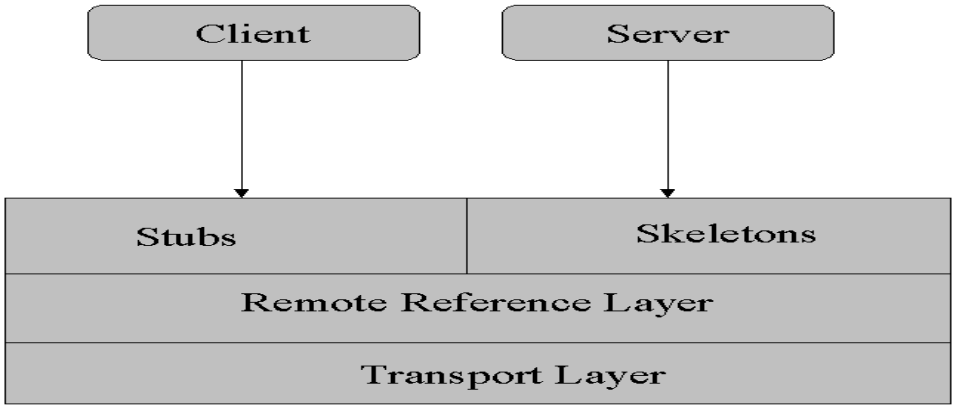
- The commonalities between RMI and RPC are as follows:
  - **They both support programming with interfaces,**
  - **They are both typically constructed on top of request-reply protocols**
  - **They both offer a similar level of transparency – that is, local and remote calls** employ the same syntax but remote interfaces typically expose the distributed nature of the underlying call, for example by supporting remote exceptions.

RPC & RMI: Differences

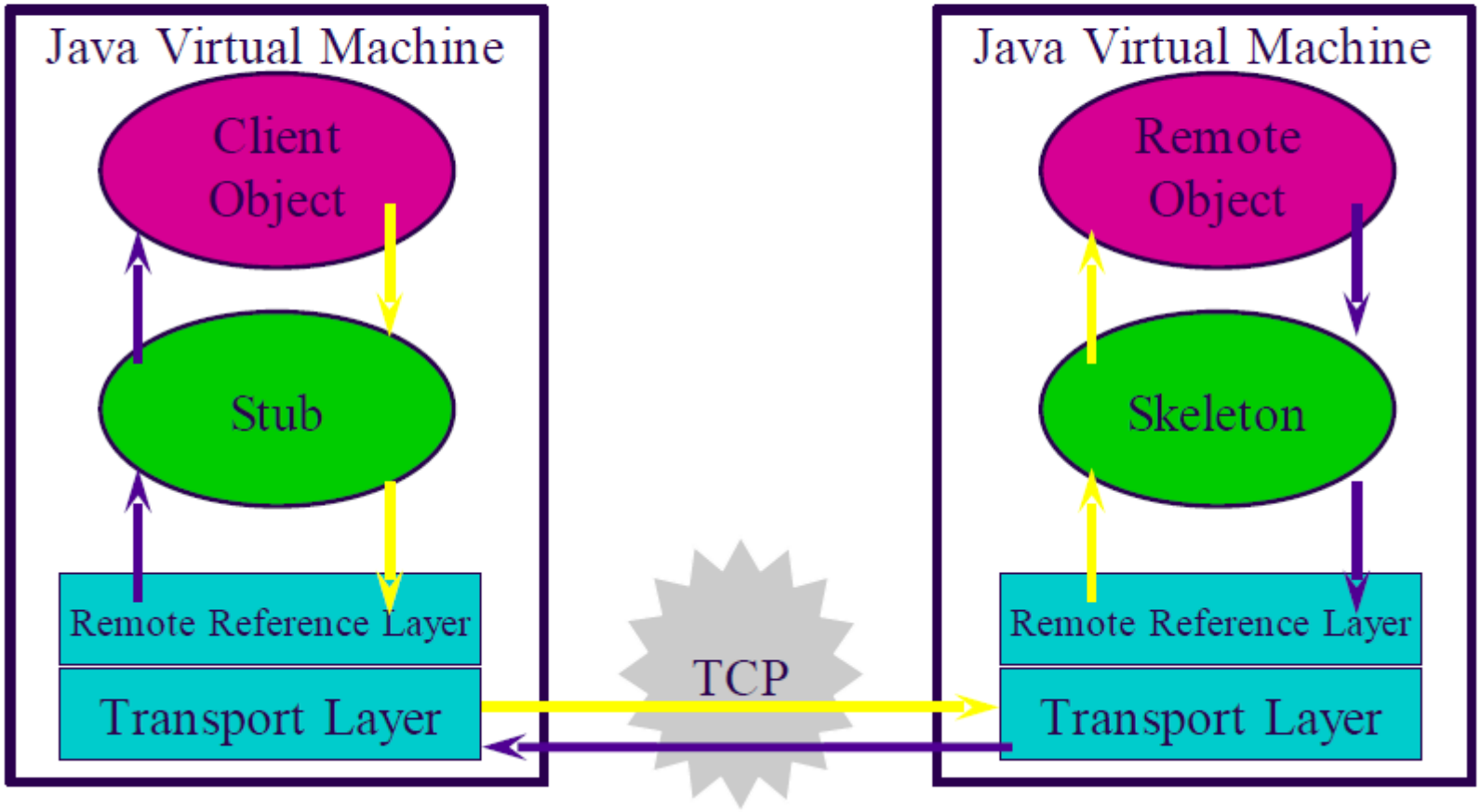
- The programmer is able to use the full expressive power of object-oriented programming in the development of distributed systems software, including the use of objects, classes and inheritance, and can also employ related objectoriented design methodologies and associated tools.
- **Building on the concept of object identity in object-oriented systems, all objects** in an RMI-based system have unique object references (whether they are local or remote), such object references can also be passed as parameters, thus offering significantly richer parameter-passing semantics than in RPC.

RMI System Architecture

- Built in three layers (they are all independent):
  - Stub/Skeleton layer
  - Remote reference layer
  - Transport layer



RMI Layers


Java Virtual Machine
Client Object
Stub
Remote Reference Layer
Transport Layer
TCP
Java Virtual Machine
Remote Object
Skeleton
Remote Reference Layer
Transport Layer

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

The Stub/Skeleton I

- Client stub responsible for:
  - Initiate remote calls
  - Marshal arguments to be sent
  - Inform the remote reference layer to invoke the call
  - Unmarshaling the return value
  - Inform remote reference the call is complete
- Server skeleton responsible for:
  - Unmarshaling incoming arguments from client
  - Calling the actual remote object implementation
  - Marshaling the return value for transport back to client

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
- Example: the unicast protocol provides point-to-point invocation, and multicast provides invocation to replicated groups of objects, other protocols may deal with different strategies…
- Not all these features are supported….

The Transport Layer

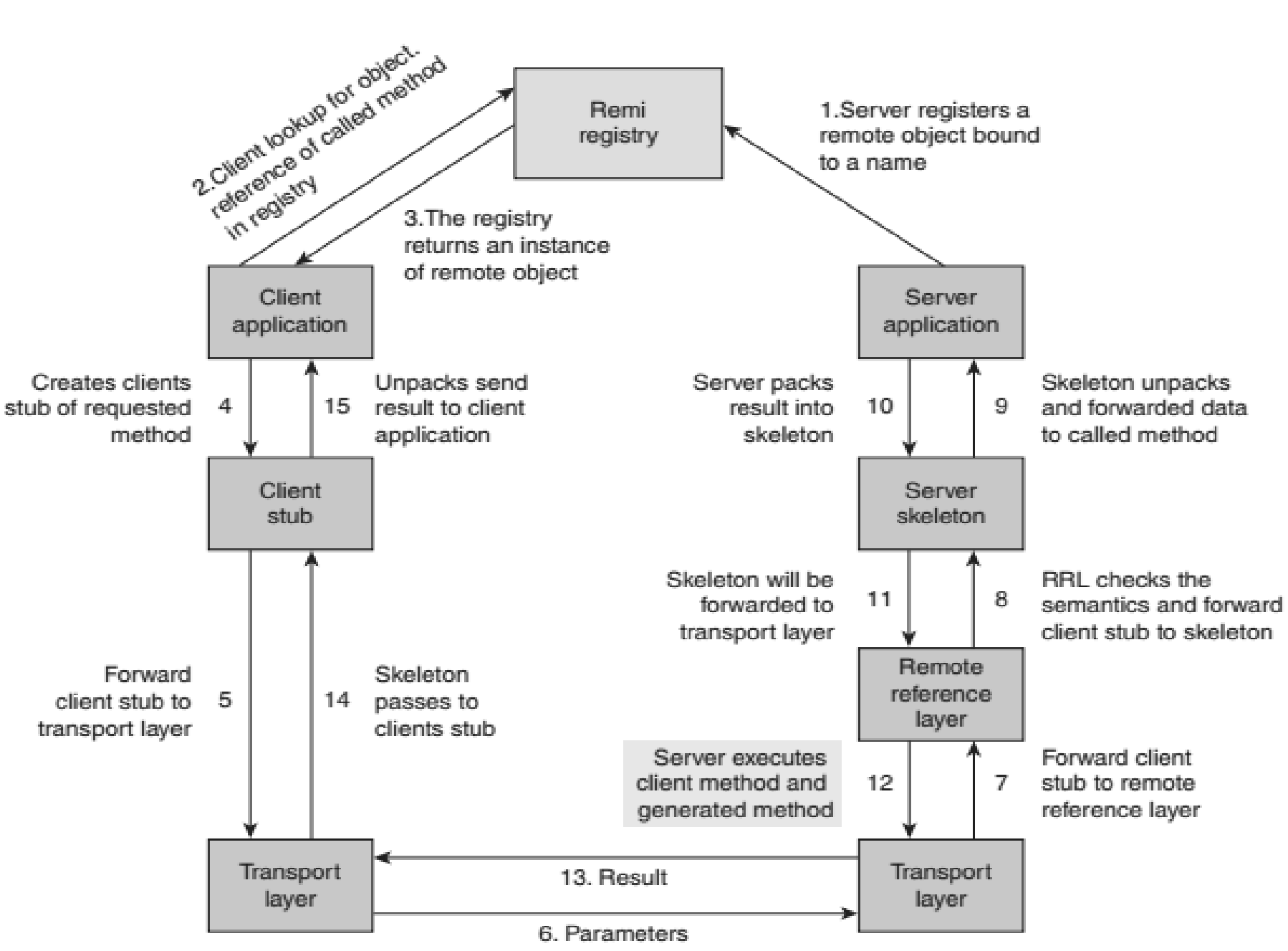
- A low-level layer that ships serialized objects between different address spaces
- Responsible for:
  - Setting up connections to remote address spaces
  - Managing the connections
  - Listening to incoming calls
  - Maintaining a table of remote objects that reside in the same address space
  - Setting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

How does RMI work?

- An invocation will pass through the stub/skeleton layer, which will transfer data to the remote reference layer
- The semantics of the invocation are carried to the transport layer
- The transport layer is responsible for setting up the connection

The Naming Registry

- The remote object must register itself with the RMI naming registry
- A reference to the remote object is obtained by the client by looking up the registry


Remi registry
1.Server registers a remote object bound to a name
2.Client lookup for object reference of called method in registry
3.The registry returns an instance of remote object
Client application
Server application
Creates clients stub of requested method
4
15
Unpacks send result to client application
Server packs result into skeleton
10
9
Skeleton unpacks and forwarded data to called method
Client stub
Server skeleton
Skeleton will be forwarded to transport layer
11
8
RRL checks the semantics and forward client stub to skeleton
Forward client stub to transport layer
5
14
Skeleton passes to clients stub
Remote reference layer
Server executes client method and generated method
12
7
Forward client stub to remote reference layer
Transport layer
13. Result
6. Parameters
Transport layer

Java RMI

- Distributed objects have been integrated in the Java language with a goal of a high degree of distribution transparency.
- Java adopts remote objects as the only form of distributed objects. *[i.e., objects whose state only resides on a single machine]*
- *Java allows each object to be constructed as a monitor by declaring a method to be synchronized.*

Server Management

- **Server States**
  - **Stateful**
  - **Stateless**
- **Server Creation Semantics**
  - **Instance - Per - Call**
  - **Instance - Per – Session**
  - **Persistent Servers**

Parameter Passing

- **Call – by – value**
- **Call by Reference**

Call Semantics

- **May- be**
- **Last-one call**
- **Last – of –many**
- **At – least once**
- **Exactly once**

Communication Protocols

a) The Request Protocol (may-be)

b) The Request/Reply Protocol (at-least once)

c) The Request/Reply/Acknowledge Protocol

d) RPC involving Long-duration calls /gaps

- Periodic probing of the server by the client

(server crash/link failure)

- Periodic generation of an acknowledgement by the server

e) RPC involving arguments and or results that are large to fit in a single datagram packet.

Multiple RPC (multiple ack.) / Multidatagram (single ack)

Middleware Communication Techniques

- Remote Procedure Call
- RMI
- Message-Oriented Communication
- Stream-Oriented Communication
- Multicast Communication

Types of Communication

- Persistent versus transient
- Synchronous versus asynchronous

Persistent vs Transient Communication

- **Persistent**: messages are held by the middleware comm. service until they can be delivered (e.g., email)
  - Sender can terminate after executing send
  - Receiver will get message next time it runs

- **Transient**: Message is stored only so long as ending /receiving application are executing
- – Discard message if it can't be delivered to next server/receiver
  - – Example: transport-level communication services offer transient communication
  - – Example: Typical network router – discard message if it can't be delivered next router or destination
  - Example: TCP/UDP

Asynchronous vs Synchronous Communication

- **Synchronous**: sender is blocked until
  - The OS or middleware notifies acceptance of the message, or
  - The message has been delivered to the receiver, or
  - The receiver processes it & returns a response
- **Asynchronous**: (non-blocking)
  - sender resumes execution as soon as the message is passed to the communication/middleware software

Evaluation

- Fully synchronous primitives may slow processes down, but program behavior is easier to understand

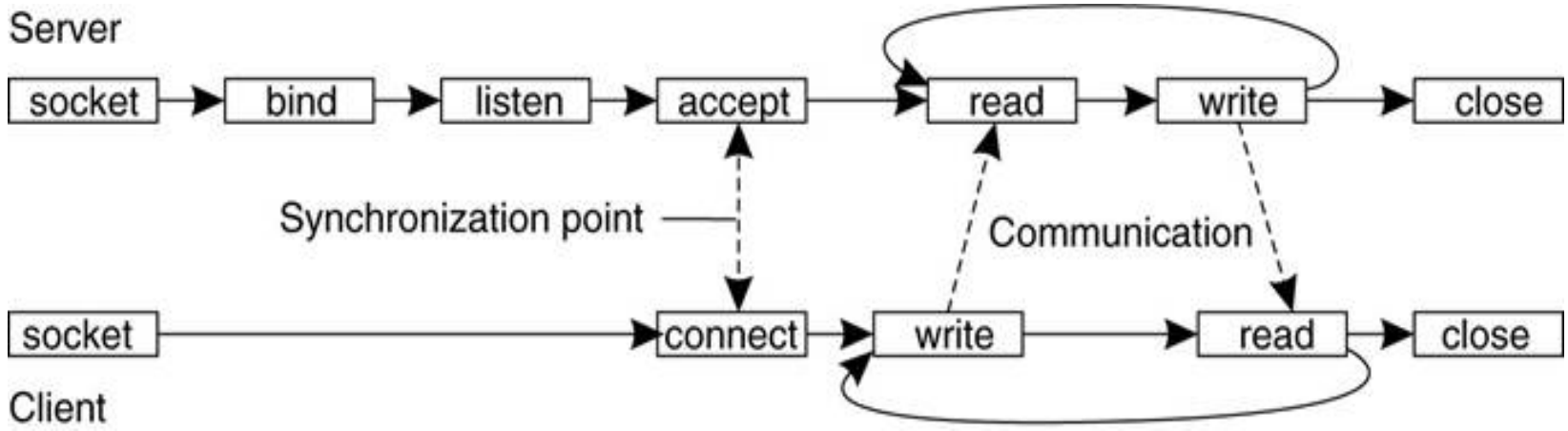
- In multithreaded processes, blocking is not as big a problem because a special thread can be created to wait for messages

Message Oriented Transient Communication

- In this , messages are stored only as long as sending and receiving applications are executing.
- Message is discarded if it can not be delivered to next server or receiver
- Implemented through-
  - Sockets
  - MPI

Socket

- Socket is an end point of an inter process communication
- Socket address is the combination of ip address + port number



Socket Primitives

Primitive	Meaning
SOCKET	Create a new communication end point
BIND	Attach a local address to a socket
LISTEN	Announce willingness to accept connections; give queue size
ACCEPT	Block the caller until a connection attempt arrives
CONNECT	Actively attempt to establish a connection
SEND	Send some data over the connection
RECEIVE	Receive some data from the connection
CLOSE	Release the connection

Sockets are insufficient

- First, they were at the wrong level of abstraction by supporting only simple send and receive primitives.
- Second, sockets had been designed
  - general-purpose protocol stacks such as TCPIIP.
  - They were not considered suitable for the proprietary protocols developed for high-speed interconnection networks, such as those used in high-performance server clusters.
  - Those protocols required an 'interface that could handle more advanced features, such as different forms of buffering and synchronization

Message Passing Interface(MPI)

- MPI
  - Hardware Independent
  - Designed for parallel applications
- Communication between group of processes
- Support most of the forms of transient communication

MPI Primitives

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 are none
MPI_irecv	Check if there is an incoming message, but do not block

Message Oriented Persistent Communication

- Message Queuing systems or Message Oriented Middleware(MOM)

  - Supports Asynchronous Persistent Communication

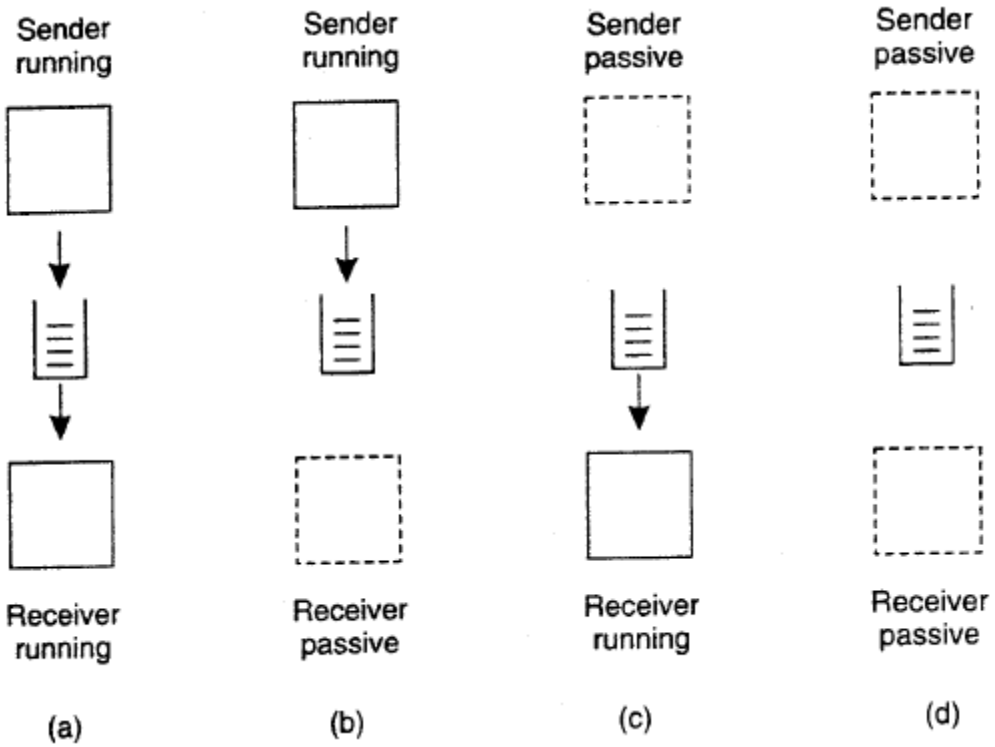
  -Intermediate Storage for messages while sender/receiver are inactive

- Communicate by inserting messages in queues
- Sender is only guaranteed that message will be eventually inserted in recipients queue

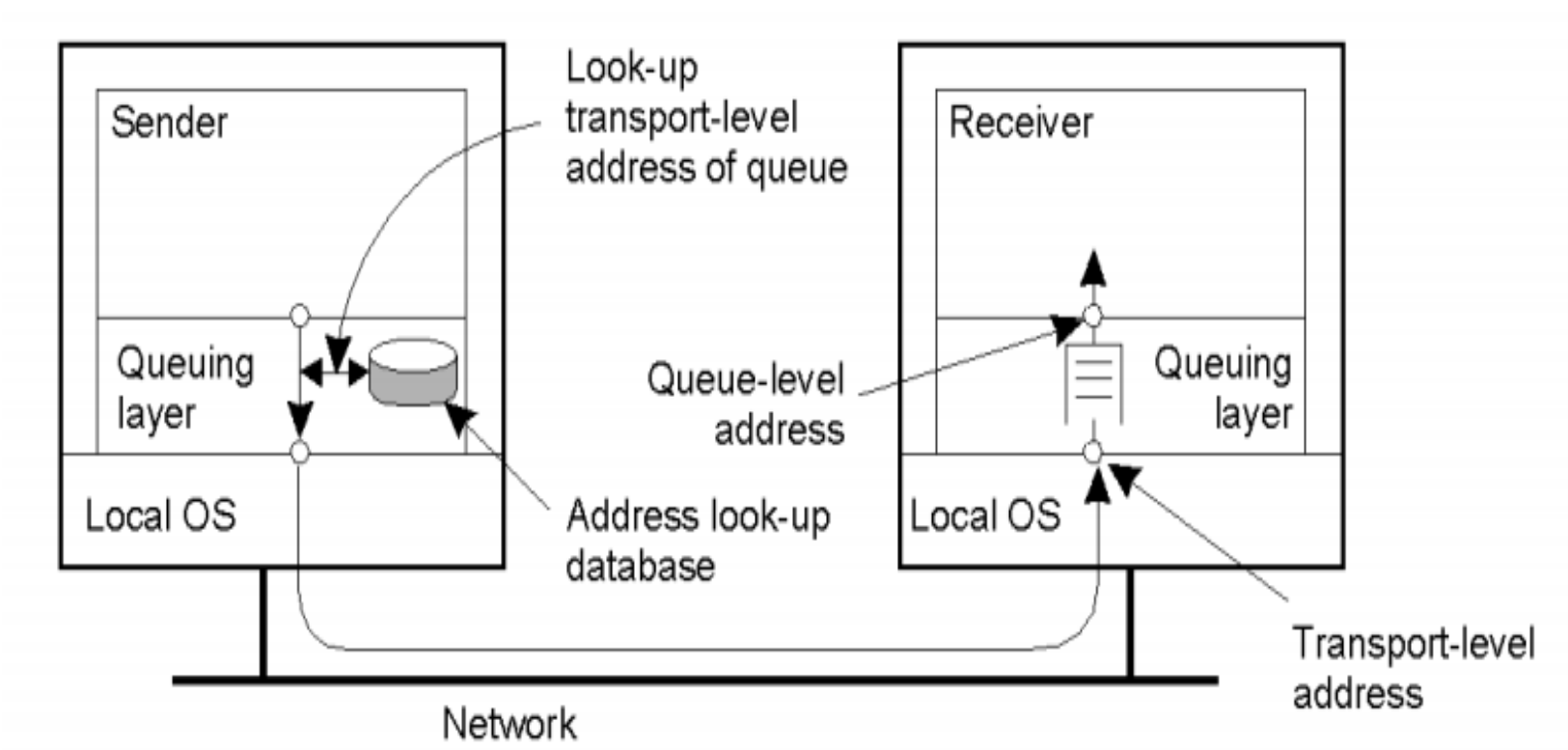
Message-Oriented Middleware (MOM) - Persistent

- Processes communicate through message queues
  - *Queues are maintained by the message-queuing system*
  - Sender appends to queue, receiver removes from queue
  - Neither the sender nor receiver needs to be on-line when the message is transmitted.
- In principle, each application has its own private queue to which other applications can send messages.
- A queue can be read only by its associated application, but it is also possible for multiple applications to share a single queue.

- *sender is generally given only the guarantees that its message will eventually be inserted in the recipient's queue.*
- *No guarantees are given about when, or even if the message will actually be read, which is completely determined by the behavior of the recipient.*



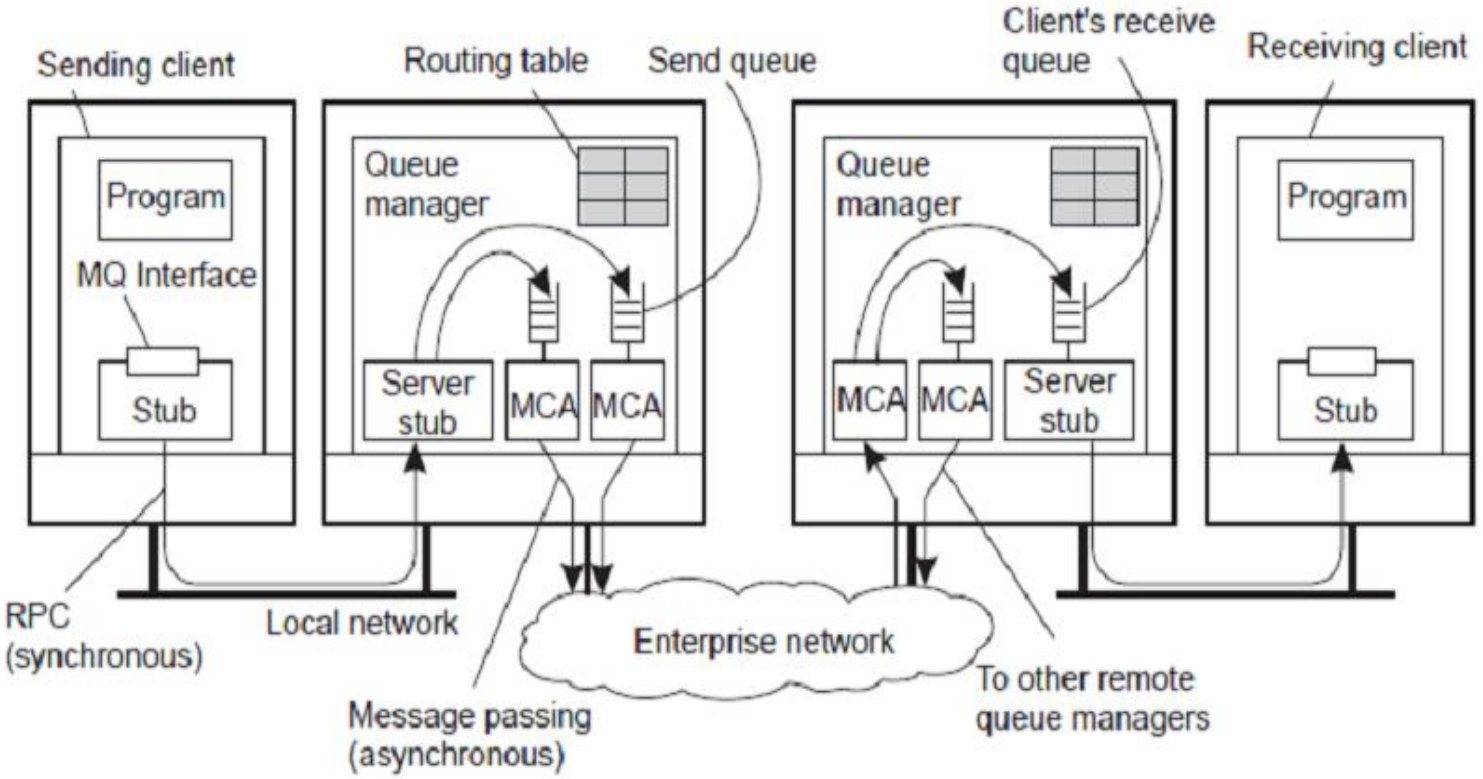
Message Queuing Model



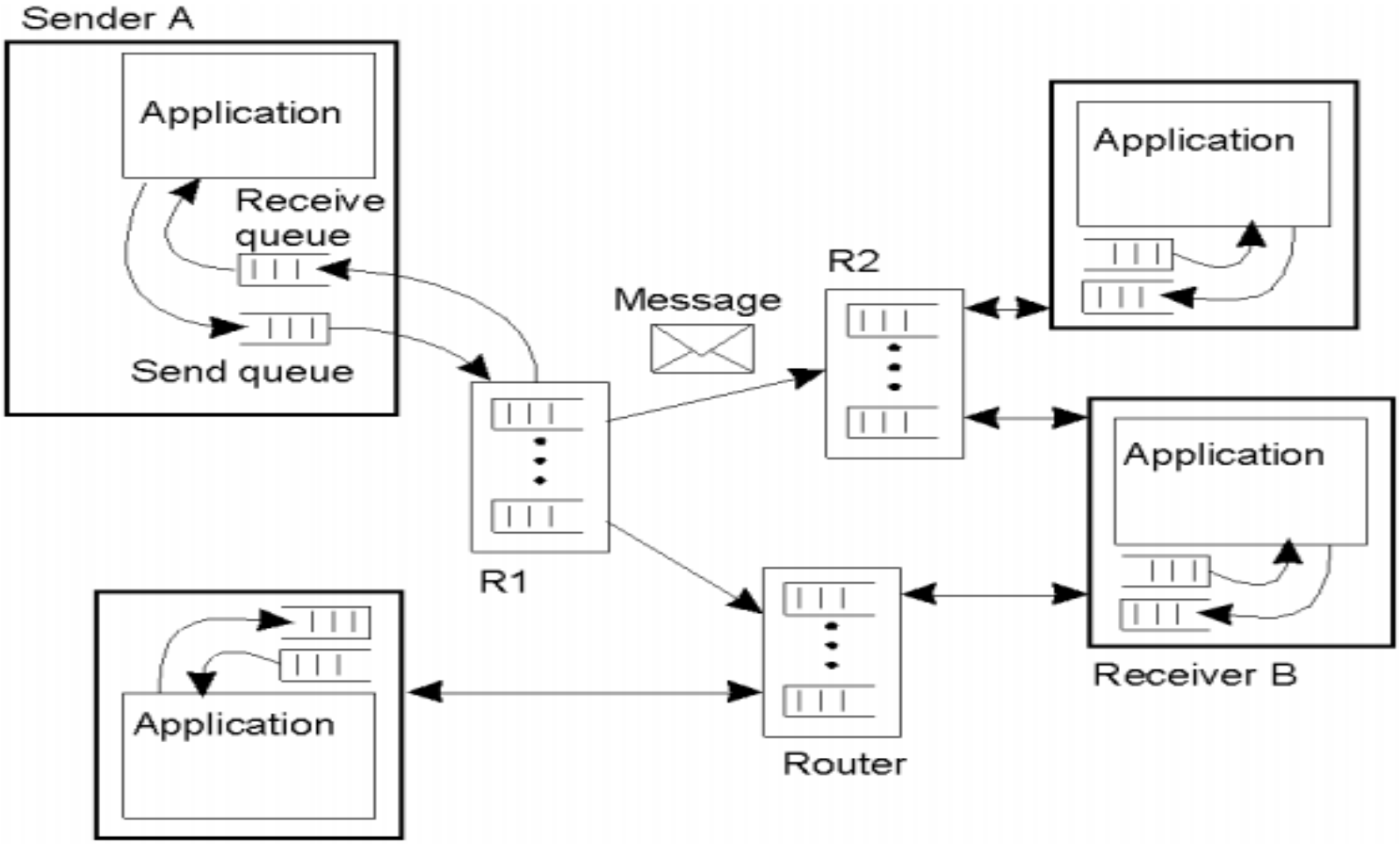
MOM primitive

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

IBM's WebSphere MQ



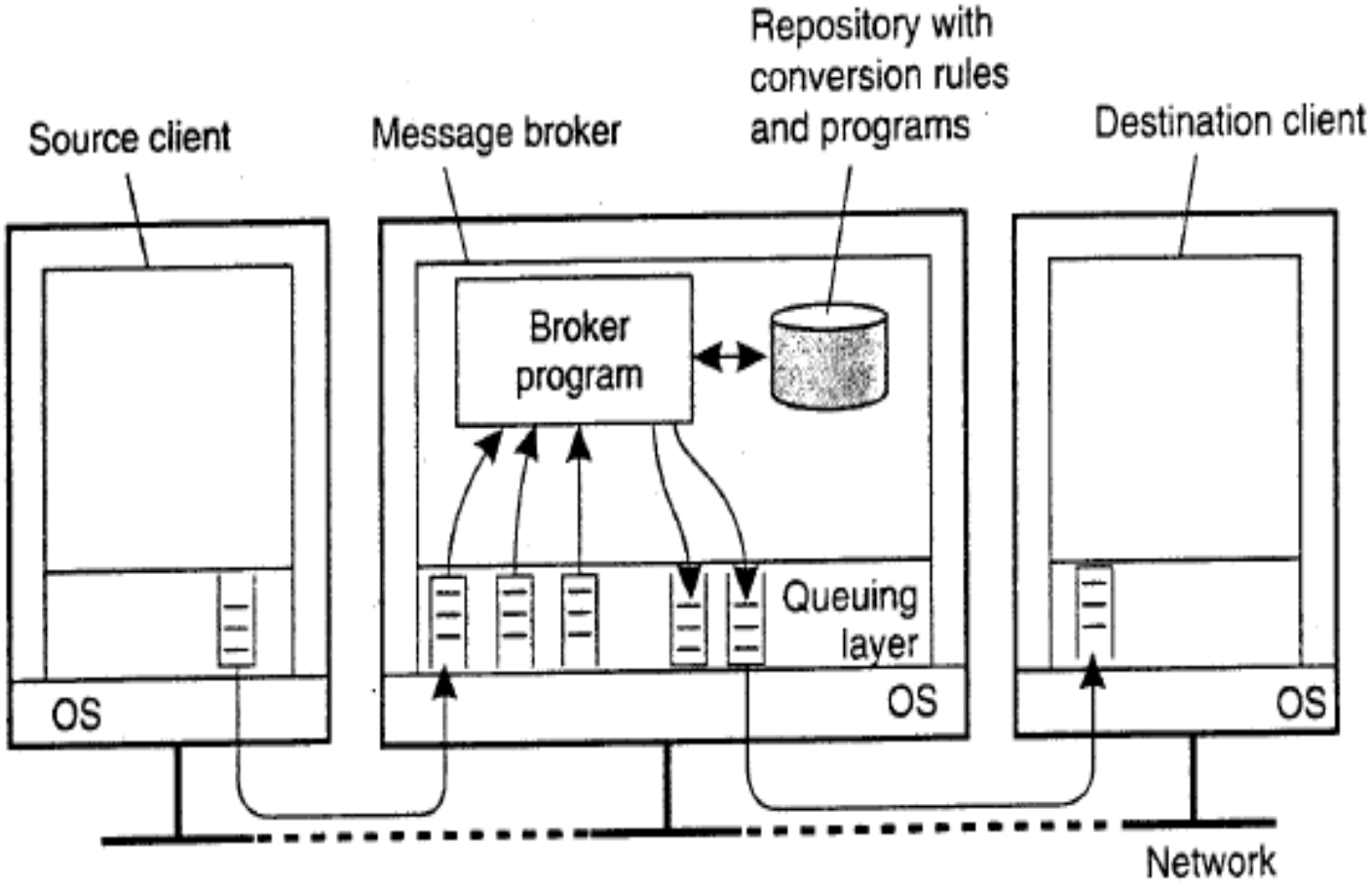
Message Queuing System



Message Brokers

- Message Brokers are building blocks of Message Oriented Middleware
- A message broker acts as an application-level gateway in a message-queuing system.
- Its main purpose is to convert incoming messages so that they can be understood by the destination application
- Architectural pattern of message validation, message transformation and message routing
  - Transforms incoming messages to target format, possibly using intermediate representation

Message Brokers



Stream-Oriented Communication

- RPC and 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 (multimedia streams) must be delivered at a certain rate,
- as well as correctly
  - e.g., music or video

Discrete and Continuous Media

- Media: means by which information is conveyed
- Types of media
  - Discrete media
    - No temporal dependence between data items
    - ex) text, still images, object code or executable files
  - Continuous media
    - Temporal dependence between data items
    - ex) Motion - series of images

Data Stream

- To capture the exchange of time-dependent information, distributed systems generally provide support for data streams.
- A data stream is nothing but a sequence of data units. Data streams can be applied to discrete as well as continuous media.
- TCP/IP connections are typical examples of (byte-oriented) discrete data streams.
- Playing an audio file typically requires setting up a continuous data stream between the file and the audio device.

Transmission modes: CONSIDERING TIMING ASPECTS

- **Asynchronous transmission mode**: the data items in a stream are transmitted one after the other, but there are no further timing constraints on when transmission of items should take place.
- This is typically the case for discrete data streams.
- For example, a file can be transferred as a data stream, but it is mostly irrelevant exactly when the transfer of each item completes.

Transmission modes: cntd...

- **Synchronous transmission mode: there is a maximum end-to-end delay** defined for each unit in a data stream.
- If a data unit is transferred much faster than the maximum tolerated delay is not important.
- For example, a sensor may sample temperature at a certain rate and pass it through a network to an operator. In that case, it may be important that the end-to-end propagation time through the network is guaranteed to be lower than the time interval between taking samples, but it cannot do any harm if samples are propagated much faster than necessary.

Transmission modes: cntd...

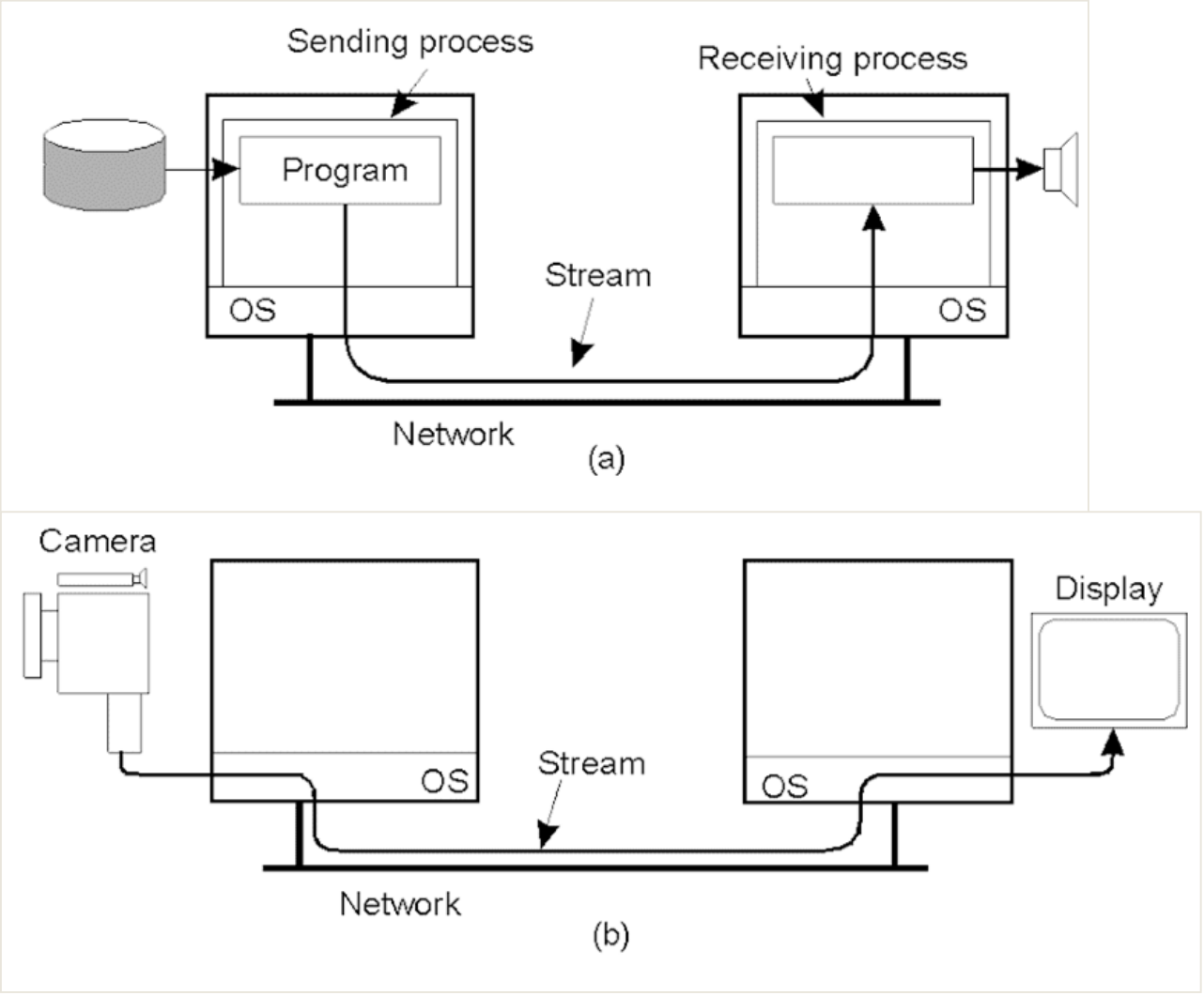
- **Isochronous transmission mode: it is necessary that data units are** transferred on time.
- This means that data transfer is subject to a maximum *and* minimum end-to-end delay, also referred to as bounded (delay) jitter.
- Isochronous transmission mode is particularly interesting for distributed multimedia systems, as it plays a crucial role in representing audio and video.

Support for Continuous Media(1)

- Types of stream
  - Simple stream
    - consist of only a single sequence of data
  - Complex stream
    - consist of several related simple stream
    - ex) stereo audio, movie
  - Substream
    - related simple stream
    - ex) stereo audio channel

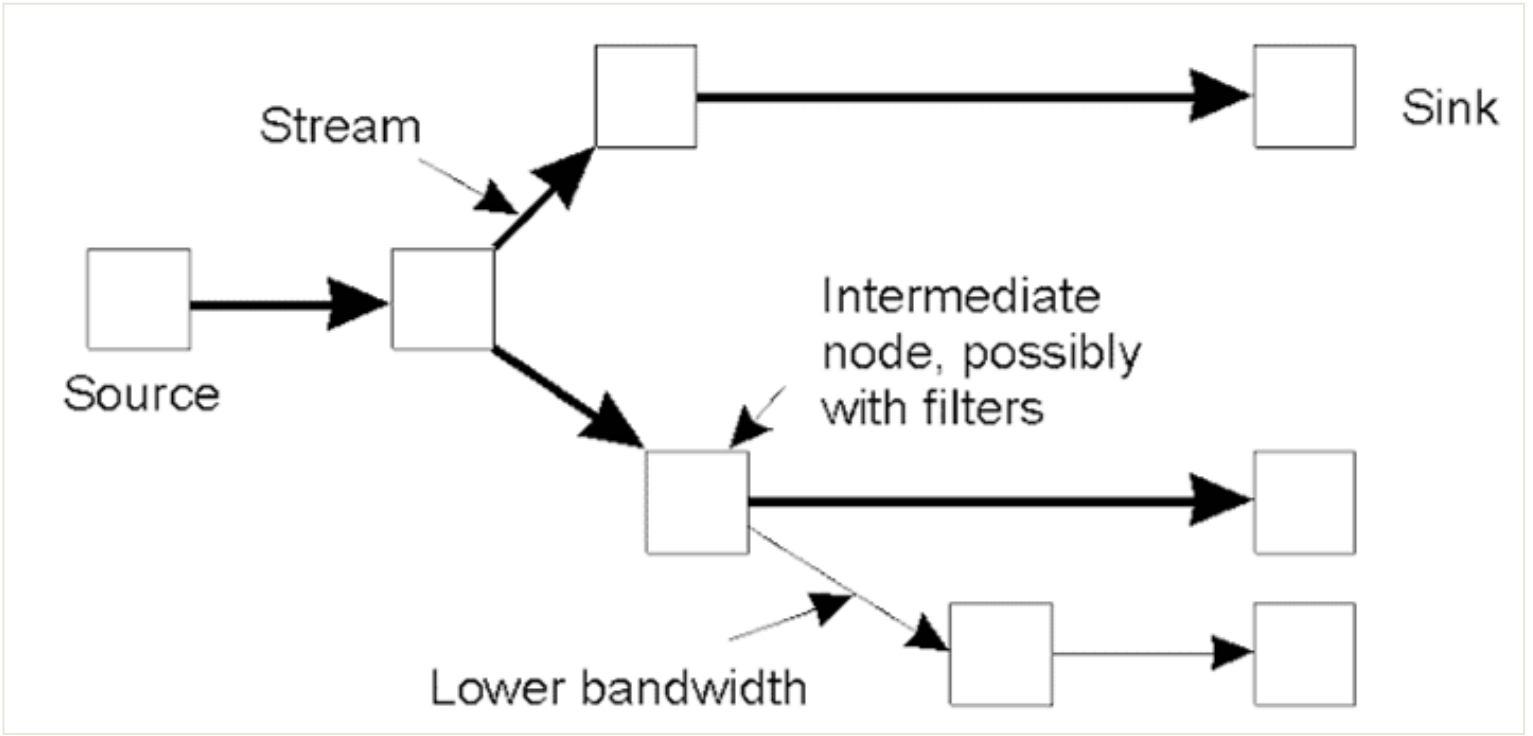
Support for Continuous Media(2)

Figure. 2-35
(a) Setting up a stream between two processes across a network
(b) Setting up a stream directly between two devices



Support for Continuous Media(3)

Figure. 2-36
An example of multicasting a stream to several receivers



Streams and Quality of Service(1)

- Specifying QoS(1)
  - Flow specification
  - To provide a precise factors(bandwith, transmission rates and delay, etc.)
  - Example of flow specification developed by Partridge

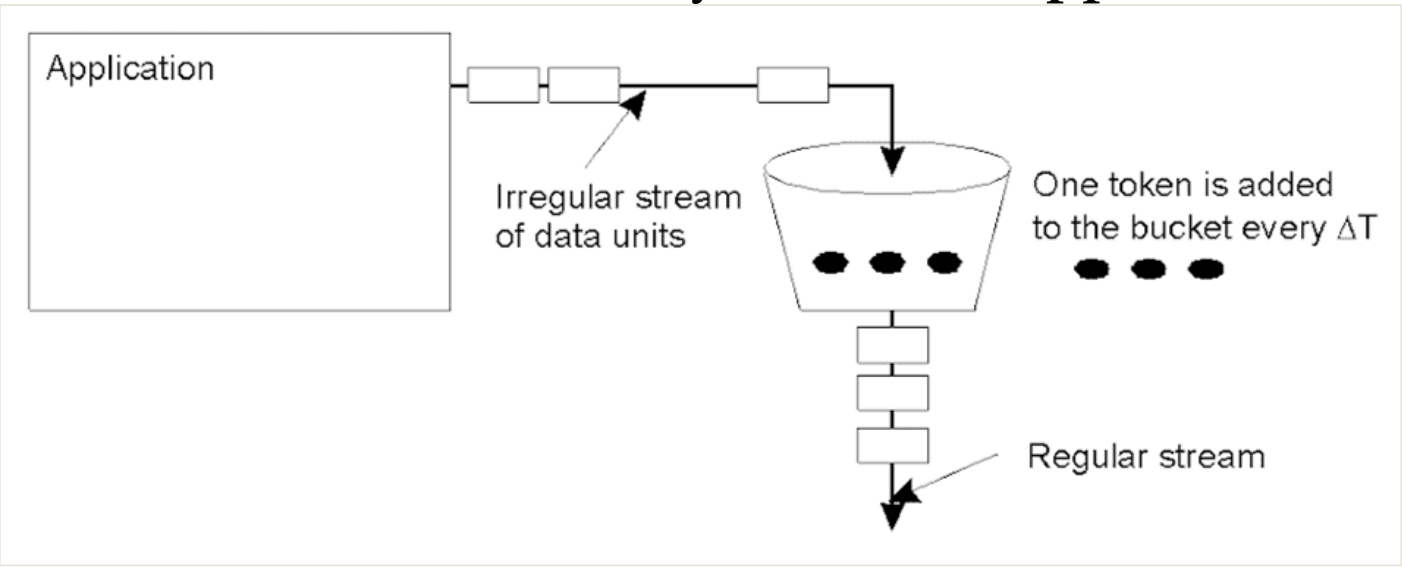
Figure. 2-37 A flow specification

Characteristics of the Input	Service Required
Maximum data unit size (bytes) Token bucket rate (bytes/sec) Token bucket size (bytes) Maximum transmission rate (bytes/sec)	Loss sensitivity (bytes) Loss interval (microsec) Burst loss sensitivity (data units) Minimum delay noticed (microsec) Maximum delay variation (microsec) Quality of guarantee

Streams and Quality of Service(2)

- Specifying QoS(2)
  - Token bucket algorithms
    - Tokens are generated at a constant rate
    - Token is fixed number of bytes that an application

Figure. 2-38 is *The Principle of a token bucket algorithm*



Streams and Quality of Service(3)

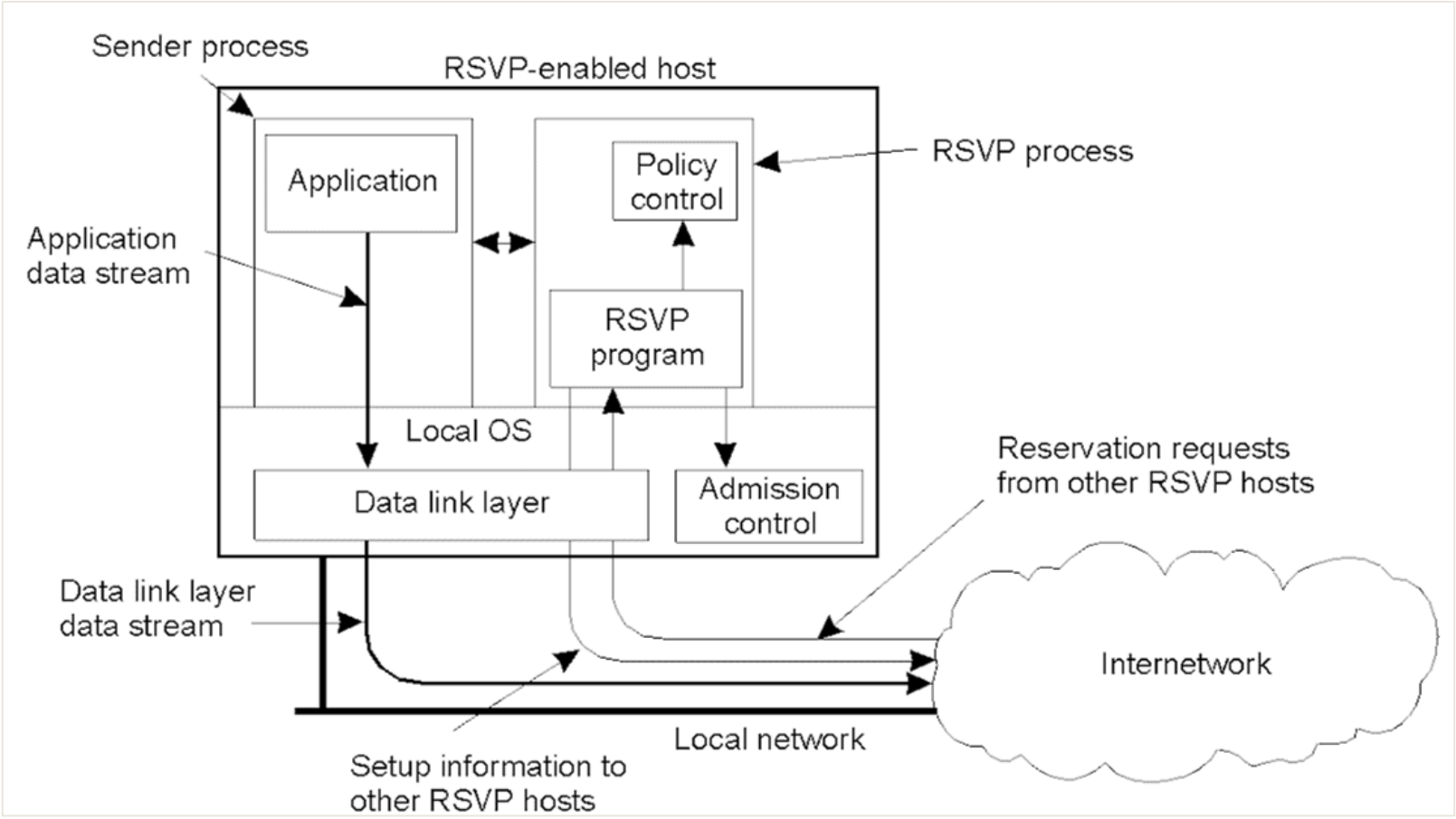
- Setting up a stream
  - Resource reSerVation Protocol(RSVP)
    - Transport-level control protocol for enabling resource reservation in network router
    - Used to provide QoS for continuous data streams by reserving resources (bandwidth, delay, jitter and so on)
    - Issue: How to translate QoS parameters to resource usage?
      - Two ways to translate

        1. RSVP translates QoS parameters into data link layer parameters

        2. Data link layer provides its own set of parameters (as in ATM)

Streams and Quality of Service(4)

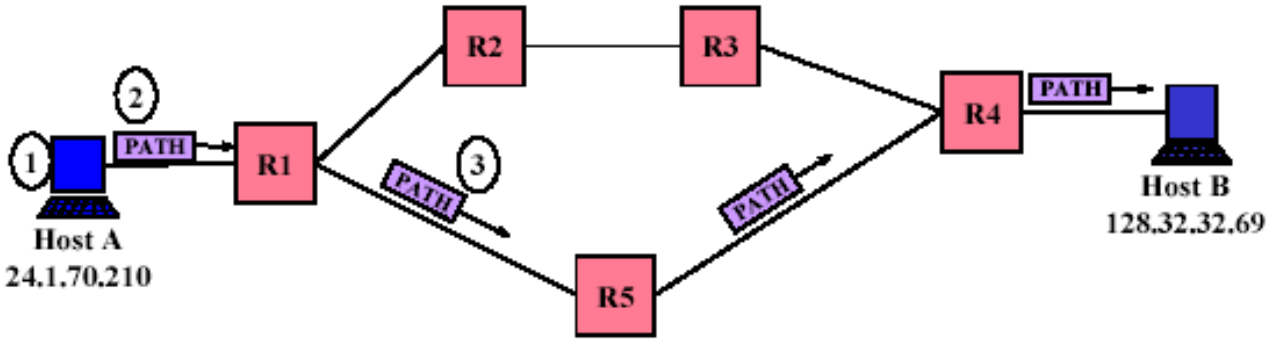
Figure. 2-39
The basic organization of RSVP for resource reservation in a distributed system



Streams and Quality of Service(5)

RSVP Reservation

- Senders advertise using PATH message
- Receivers reserve using RESV message
  - Travels upstream in reverse direction of Path message



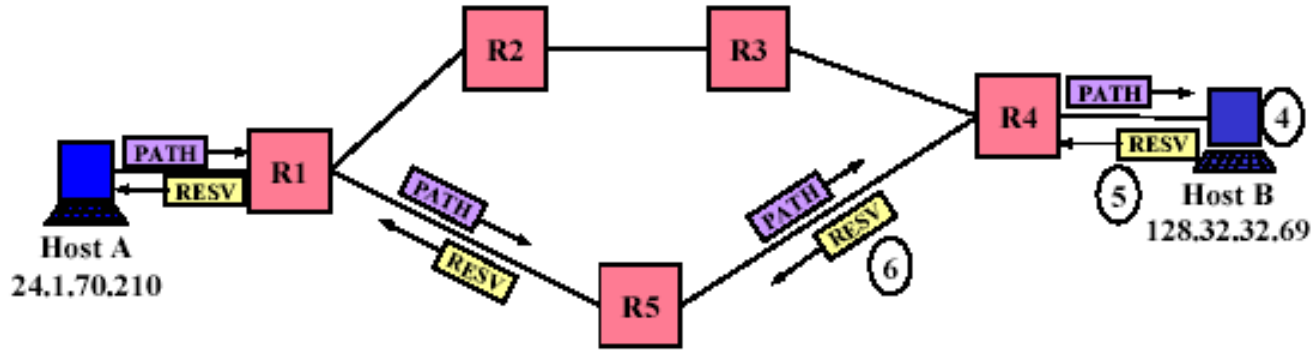
1. An application on Host A creates a session, 128.32.32.69/4078, by communicating with the RSVP daemon on Host A.

2. The Host A RSVP daemon generates a PATH message that is sent to the next hop RSVP router, R1, in the direction of the session address, 128.32.32.69.

3. The PATH message follows the next hop path through R5 and R4 until it gets to Host B. Each router on the path creates soft session state with the reservation parameters.

Streams and Quality of Service(6)

RSVP UDP Reservation



4. An application on Host B communicates with the local RSVP daemon and asks for a reservation in session 128.32.32.69/4078. The daemon checks for and finds existing session state.

5. The Host B RSVP daemon generates a RESV message that is sent to the next hop RSVP router, R4, in the direction of the source address, 24.1.70.210.

6. The RESV message continues to follow the next hop path through R5 and R1 until it gets to Host A. Each router on the path makes a resource reservation.

Stream Synchronization(1)

- Basic ideas
  - Synchronize transmission of data units
  - Synchronization take place when the data stream is made up
    - Stereo audio with CD quality(16 bit samples)
      - Sampling rate 44.1 KHz -> synchronize 22.6 micro sec
  - Synchronization between audio stream and video stream for lip sync.
    - NTSC 30Hz(a frame every 33.33ms), CD Quality sound
      - Synchronized every 1470 sound samples

Stream Synchronization(2)

- Synchronization Mechanisms(1)

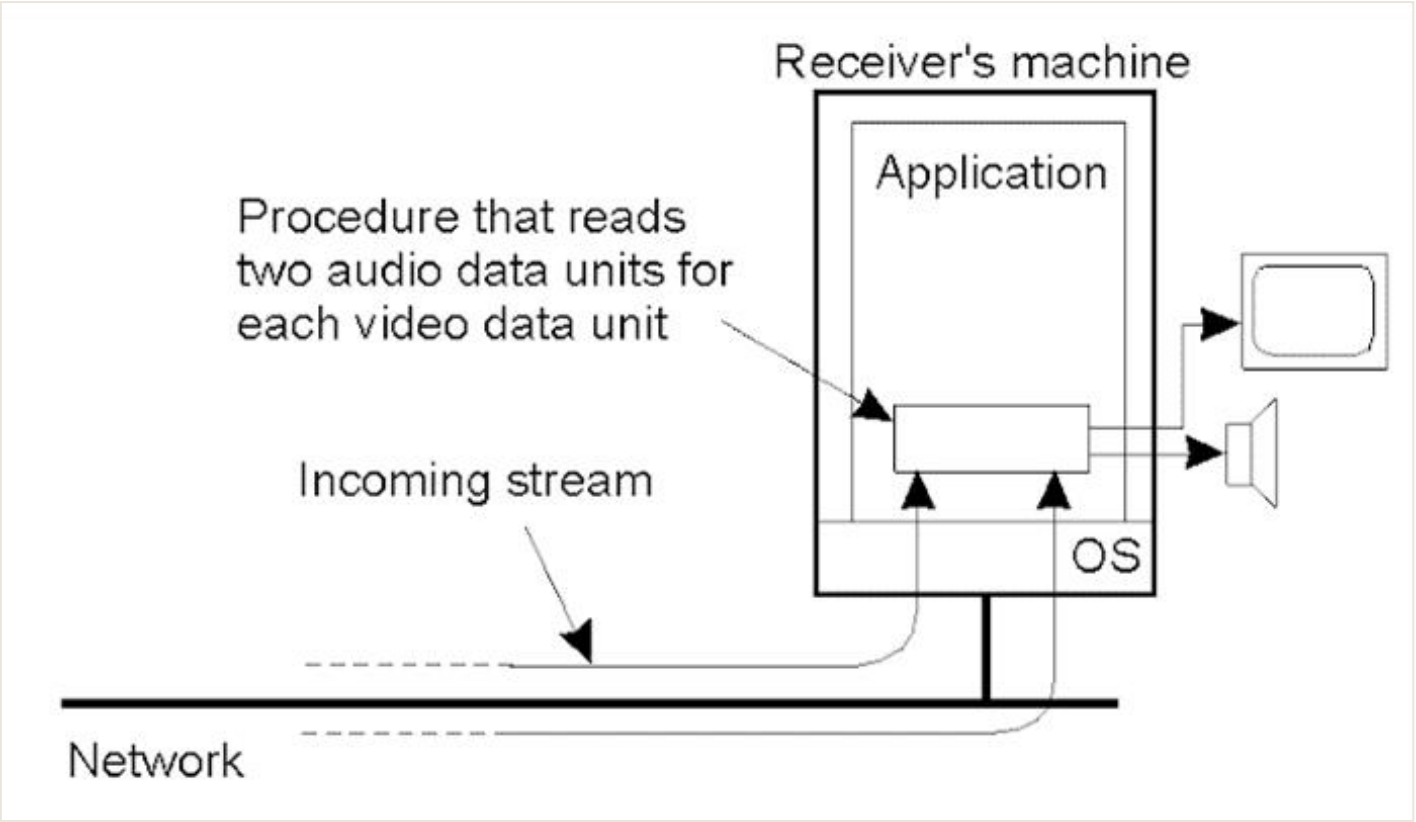


Figure. 2-40
The principle of explicit synchronization on the level data units

Stream Synchronization(3)

- Synchronization Mechanisms(2)

Figure. 2-41 *The principle of synchronization as supported by high-level interfaces*