Chapter 4: Communication

Introduction

- 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

Overview

- Message-Passing Protocols
  - OSI reference model
  - TCP/IP

- Higher level communication models
  - Remote Procedure Call (RPC)
  - Message-Oriented Middleware (time permitting)
  - Data Streaming (time permitting)

Introduction

- A communication network provides data exchange between two (or more) end points.

- In a computer network, the end points of the data exchange are computers and/or terminals. (nodes, sites, hosts, etc., …)

Circuit Switching vs Packet Switching

- Circuit switching is *connection-oriented* (think traditional telephone system)
  - Establish a dedicated path between hosts
  - Data can flow continuously over the connection
- Packet switching divides messages into fixed size units (packets) which are routed through the network individually.
  - different packets in the same message may follow different routes.

Protocols

- A protocol is a set of rules that defines how two entities interact.
  - For example: HTTP, FTP, TCP/IP
- Layered protocols have a hierarchical organization
- Conceptually, layer n on one host talks directly to layer n on the other host, but in fact the data must pass through all layers on both machines.

Open Systems Interconnection Reference Model (OSI)

- Supports communication between open systems
- Divides issues into 7 levels (layers) from most concrete to most abstract
- Each layer provides an interface (set of operations) to the layer immediately above
- Defines functionality – not specific protocols

Layered Protocols (1)

High level 7→

Create message, 6 → string of bits

Establish Comm. 5→

Create packets 4→

Network routing 3→

Add header/footer tag + checksum 2→

Transmit bits via 1→ comm. medium (e.g. Copper, Fiber, wireless)

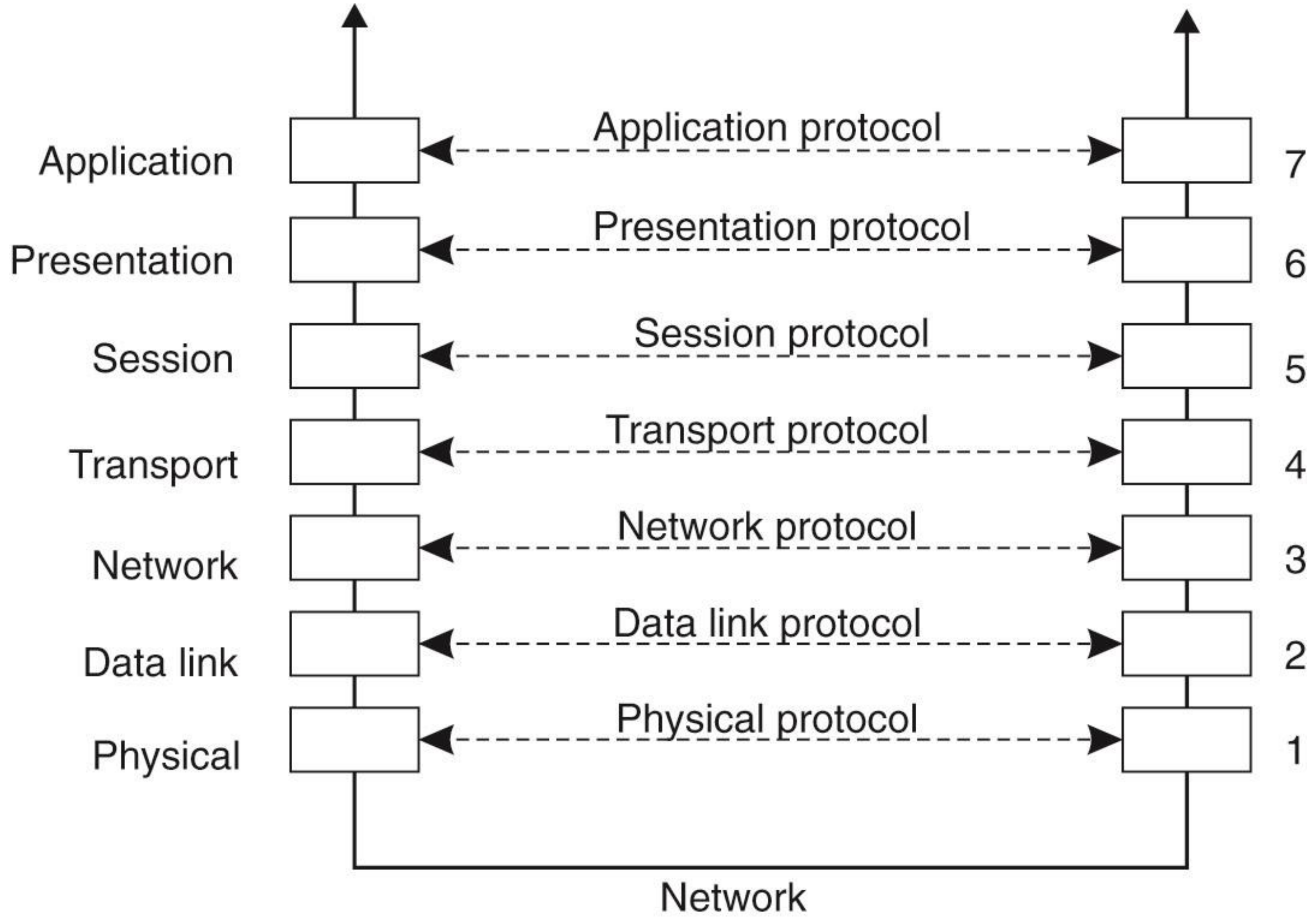


Figure 4-1. Layers, interfaces, and protocols in the OSI model.

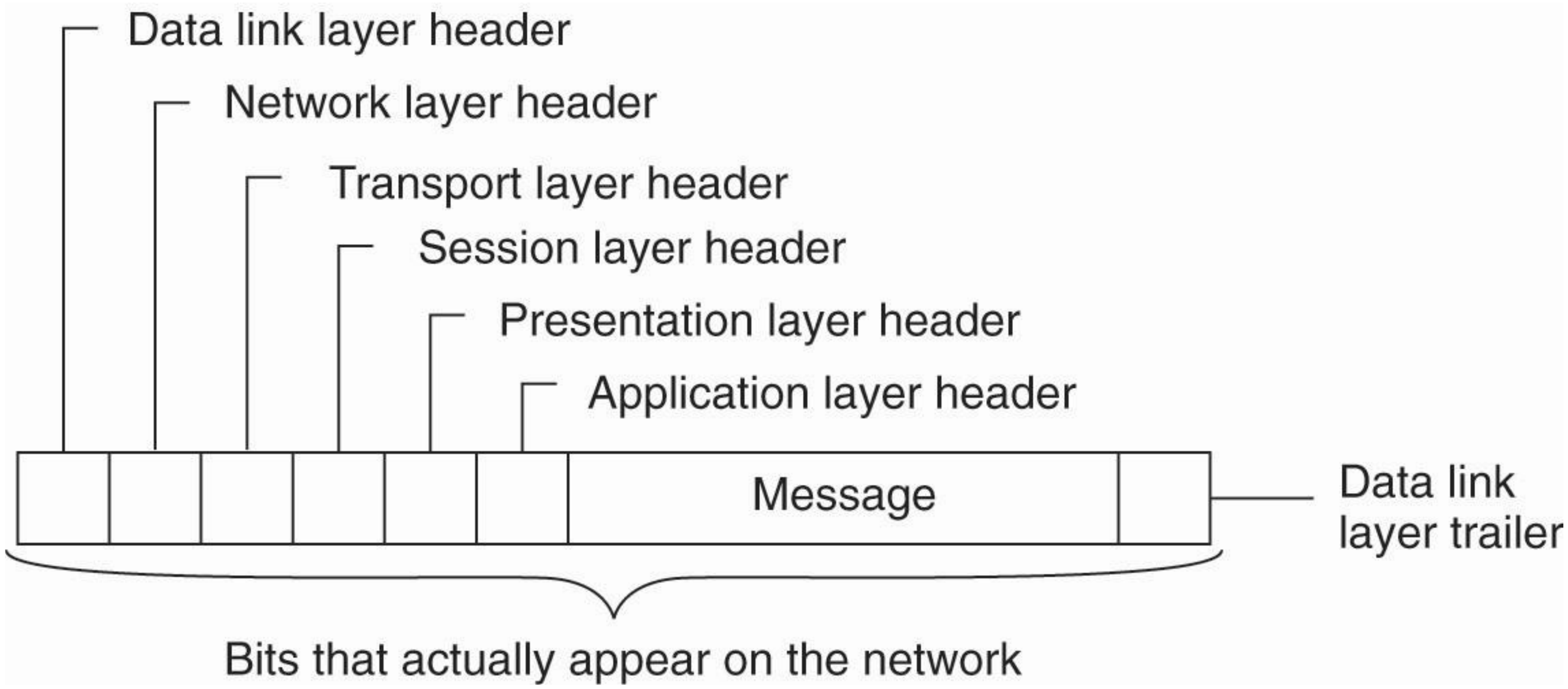


Figure 4-2. A typical message as it appears on the network

Lower-level Protocols

- **Physical**: standardizes electrical, mechanical, and signaling interfaces; e.g.,
  - # of volts that signal 0 and 1 bits
  - # of bits/sec transmitted
  - Plug size and shape, # of pins, etc.
- **Data Link**: provides low-level error checking
  - Appends start/stop bits to a frame
  - Computes and checks checksums
- **Network**: routing (generally based on IP)
  - IP packets need no setup
  - Each packet in a message is routed independently of the others

Transport Protocols

- **Transport layer, sender side**: Receives message from higher layers, divides into packets, assigns sequence #
- Reliable transport (connection-oriented) can be built on top of connection-oriented or connectionless networks
  - When a connectionless network is used the transport layer re-assembles messages in order at the receiving end.
- Most common transport protocols: TCP/IP

Higher Level Protocols

- **Session layer**: rarely supported
  - Provides dialog control;
  - Keeps track of who is transmitting
- **Presentation**: Cares about the meaning of the data
  - Record format, encoding schemes, mediates between different internal representations
- **Application**: Originally meant to be a set of basic services; now holds applications and protocols that don’t fit elsewhere

Middleware Protocols to Support Communication

- Protocols for remote procedure call (RPC)
- Protocols to
  - support message-oriented services
  - support streaming real-time data, as for multimedia applications
  - support reliable multicast service across a wide-area network

- These protocols would be built on top of low-level message passing, as supported by the transport layer.

Middleware Protocols

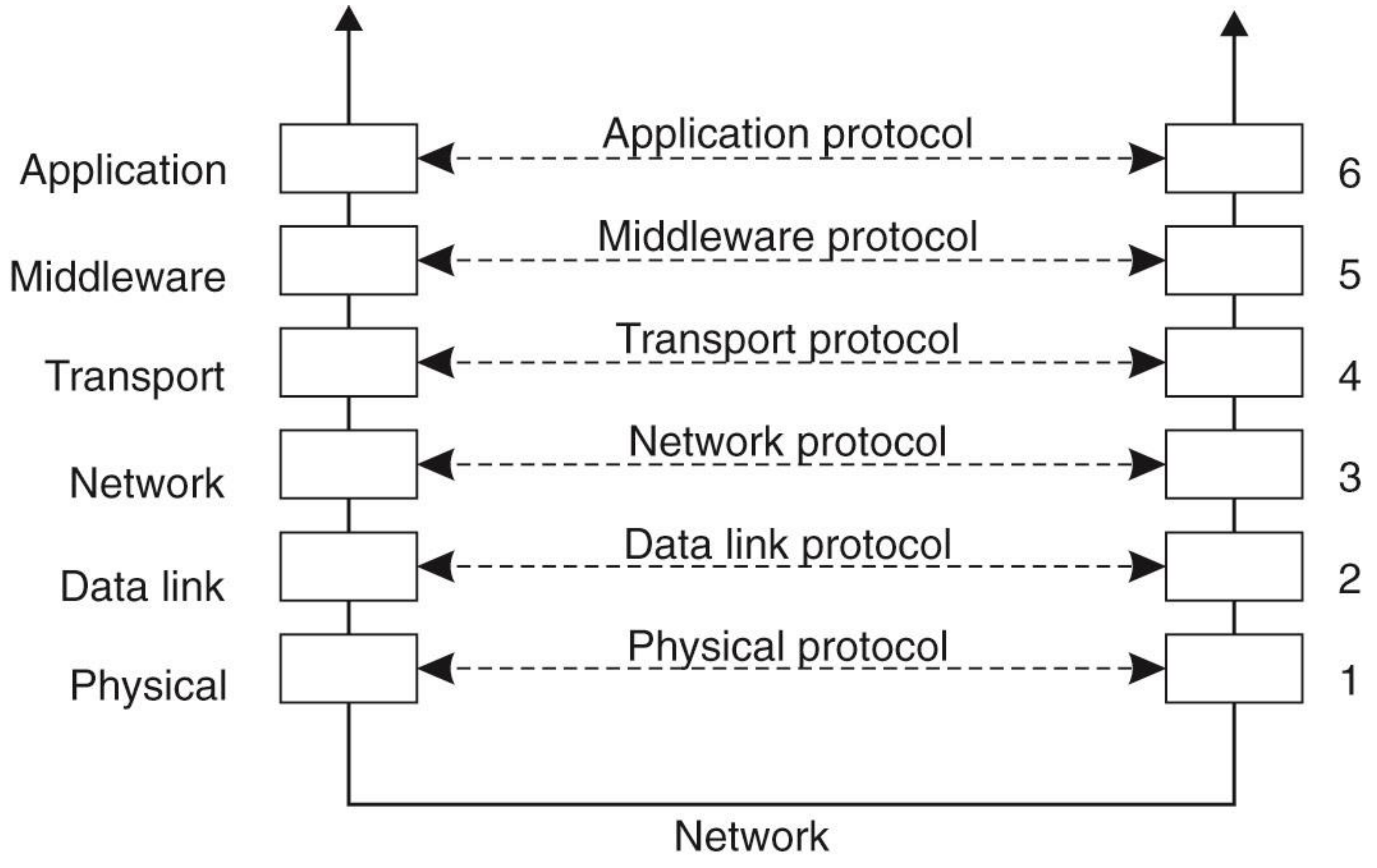


Figure 4-3. An adapted reference model for networked communication.

Messages

- Transport layer message passing consists of two types of primitives: send and receive
  - May be implemented in the OS or through add-on libraries
- Messages are composed in user space and sent via a send() primitive.
- When processes are expecting a message they execute a receive() primitive.
  - Receives are often blocking

Types of Communication

- Persistent versus Transient
- Synchronous versus Asynchronous
- Discrete versus Streaming

Persistent versus Transient Communication

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

Asynchronous v Synchronous Communication

- **Asynchronous**: (non-blocking) sender resumes execution as soon as the message is passed to the communication/middleware software
  - Message is buffered temporarily by the middleware until sent/received

- **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.

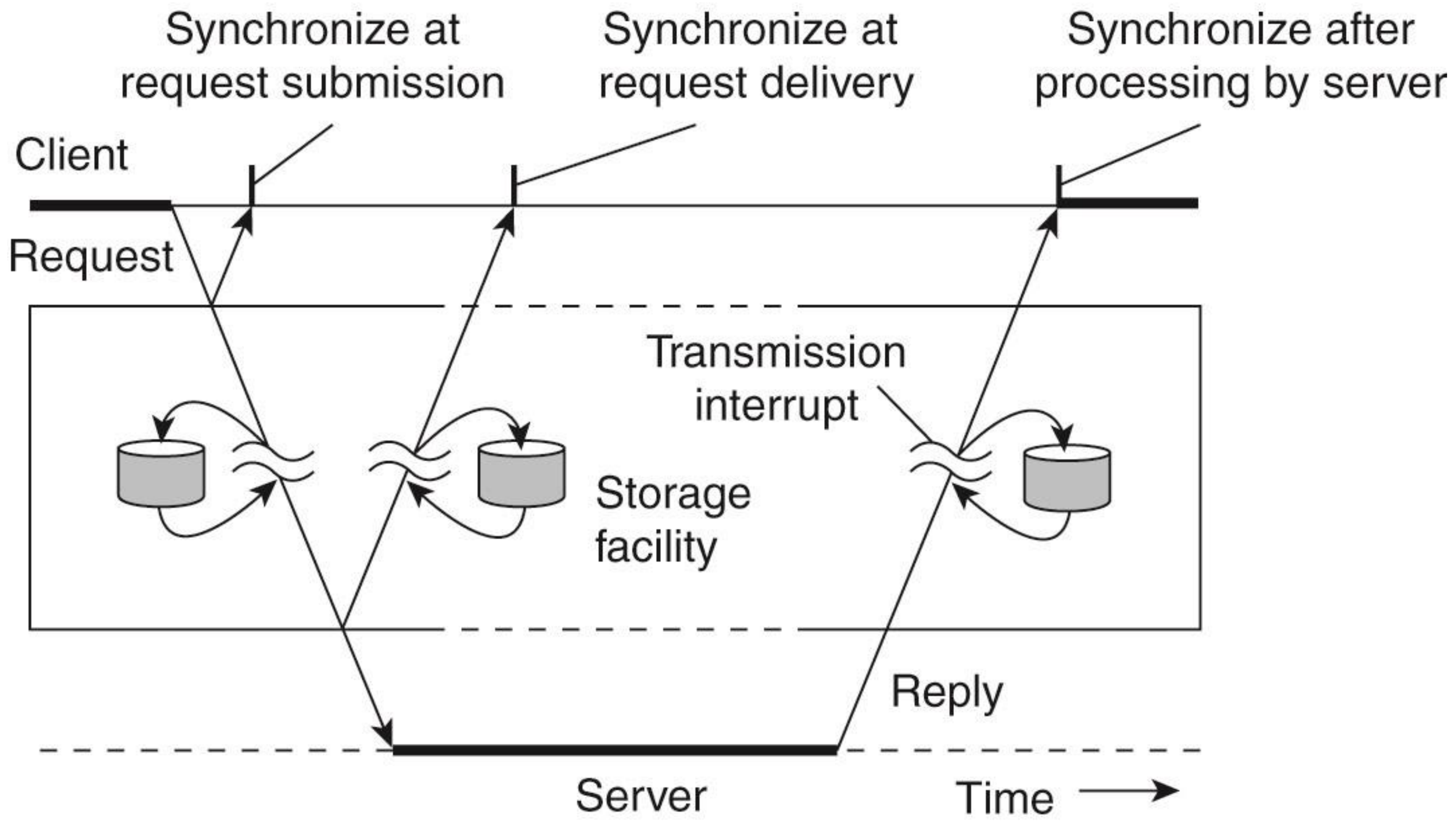

Synchronize at request submission
Synchronize at request delivery
Synchronize after processing by server
Client
Request
Transmission interrupt
Storage facility
Reply
Server
Time

Figure 4-4. Viewing middleware as an intermediate (distributed) service in application-level communication.

Evaluation

- Communication primitives that don’t wait for a response are faster, more flexible, but programs may behave unpredictably since messages will arrive at unpredictable times.
- 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.

Discrete versus Streaming Communication

- **Discrete**: communicating sections exchange discrete messages
- **Streaming**: one-way communication; a “session” consists of multiple messages from the sender that are related either by send order, temporal proximity, etc.

Middleware Communication Techniques

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

RPC - Motivation

- Low level message passing is based on *send* and *receive* primitives.
- Messages lack *access transparency*.
  - Differences in data representation, need to understand message-passing process, etc.
- Programming is simplified if processes can exchange information using techniques that are similar to those used in a shared memory environment.

The Remote Procedure Call (RPC) Model

- A high-level network communication interface
- Based on the single-process procedure call model.
- Client request: formulated as a procedure call to a function on the server.
- Server’s reply: formulated as function return

Conventional Procedure Calls

- Initiated when a process calls a function or procedure
- The caller is “suspended” until the called function completes.
- Arguments & return address are pushed onto the process stack.
- Variables local to the called function are pushed on the stack

Conventional Procedure Call

count = read(fd, buf, nbytes);

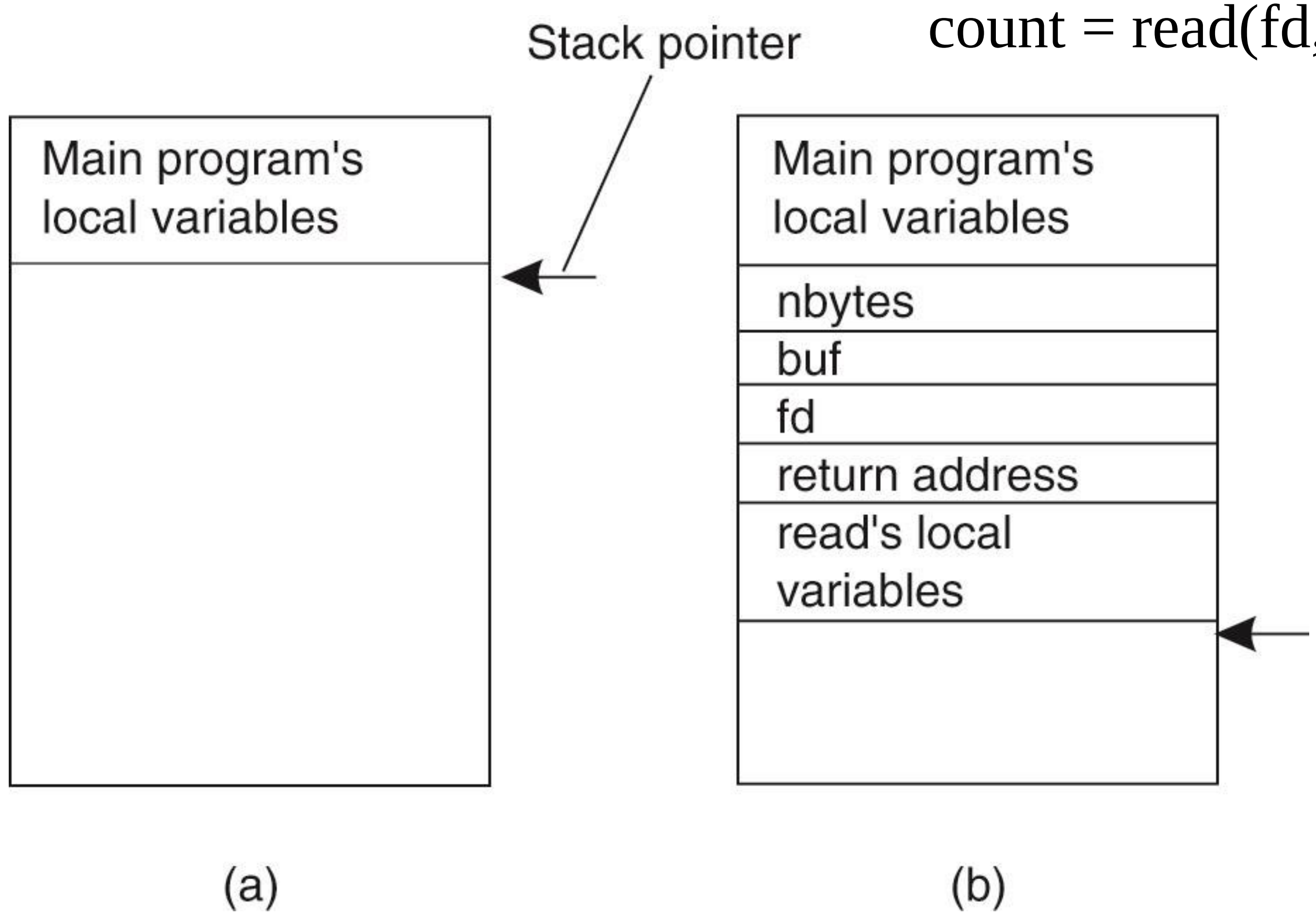


Figure 4-5. (a) Parameter passing in a local procedure call: the stack before the call to *read*. (b) The stack while the called procedure is active.

Conventional Procedure Calls

- Control passes to the called function
- The called function executes, returns value(s) either through parameters.
- The stack is popped.
- Calling function resumes executing

Remote Procedure Calls

- Basic operation of RPC parallels same-process procedure calling
- Caller process executes the remote call and is suspended until called function completes and results are returned.
- Parameters are passed to the machine where the procedure will execute.
- When procedure completes, results are passed back to the caller and the client process resumes execution at that time.

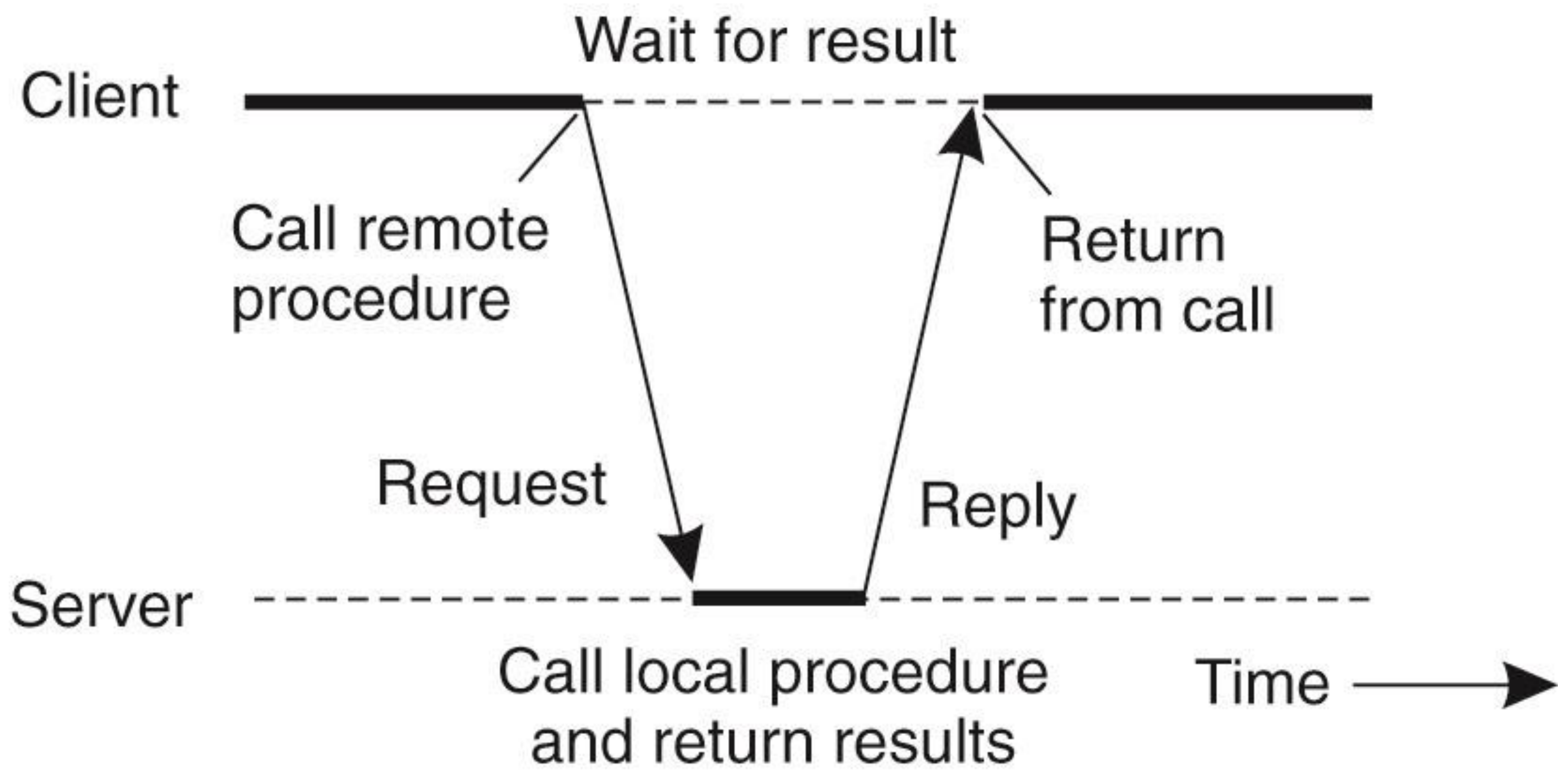


Figure 4-6. Principle of RPC between a client and server program.

RPC and Client-Server

- RPC forms the basis of most client-server systems.
- Clients formulate requests to servers as procedure calls
- Access transparency is provided by the RPC mechanism
- Implementation?

Transparency Using **Stubs**

- Stub procedures (one for each RPC)
- For procedure calls, control flows from
  - Client application to client-side stub
  - Client stub to server stub
  - Server stub to server procedure
- For procedure return, control flows from
  - Server procedure to server-stub
  - Server-stub to client-stub
  - Client-stub to client application

Client Stub

- When an application makes an RPC the stub procedure does the following:
  - Builds a message containing parameters and calls local OS to *send* the message
  - Packing parameters into a message is called **parameter marshalling**.
  - Stub procedure calls *receive( )* to wait for a reply (blocking receive primitive)

OS Layer Actions

- Client’s OS sends message to the remote machine
- Remote OS passes the message to the server stub

Server Stub Actions

- Unpack parameters, make a call to the server
- When server function completes execution and returns answers to the stub, the stub packs results into a message
- Call OS to send message to client machine

OS Layer Actions

- Server’s OS sends the message to client
- Client OS receives message containing the reply and passes it to the client stub.

Client Stub, Revisited

- Client stub unpacks the result and returns the values to the client through the normal function return mechanism
  - Either as a value, directly or
  - Through parameters

Passing Value Parameters

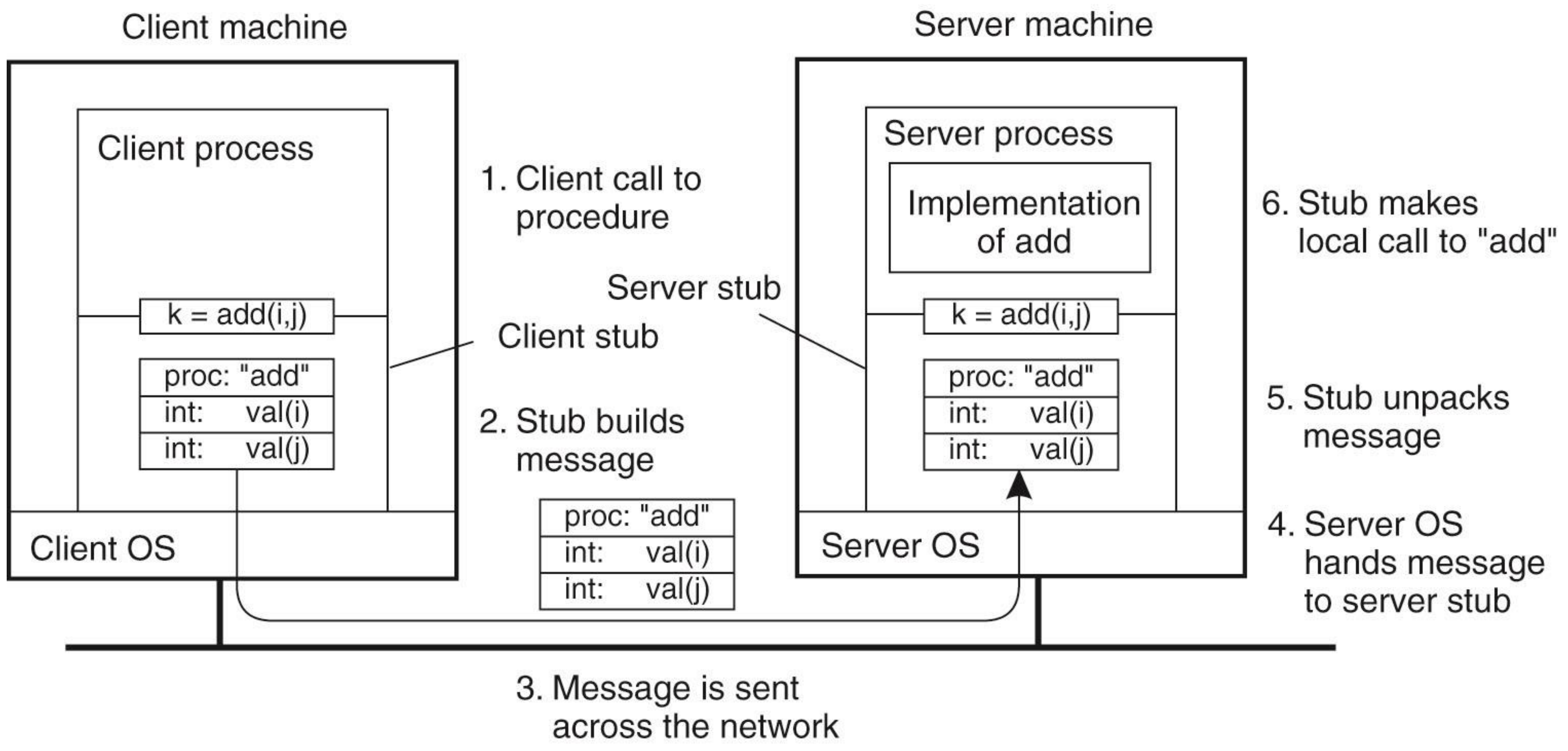


Figure 4-7. The steps involved in a doing a remote computation through RPC.

Issues

- Are parameters call-by-value or call-by-reference?
  - Call-by-value: in same-process procedure calls, parameter value is pushed on the stack, acts like a local variable
  - Call-by-reference: in same-process calls, a pointer to the parameter is pushed on the stack
- How is the data represented?
- What protocols are used?

Parameter Passing –Value Parameters

- For *value parameters*, value can be placed in the message and delivered directly, except …
    - Are the same internal representations used on both machines? (char. code, numeric rep.)
    - Is the representation big endian, or little endian? (see p. 131)

Parameter Passing – Reference Parameters

- Consider passing an array in the normal way:
  - The array is passed as a pointer
  - The function uses the pointer to directly modify the array values in the caller’s space
- Pointers = machine addresses; not relevant on a remote machine
- Solution: copy array values into the message; store values in the server stub, server processes as a normal reference parameter.

Other Issues

- Client and server must also agree on other issues
  - Message format
  - Format of complex data structures
  - Transport protocol (TCP/IP or UDP?)

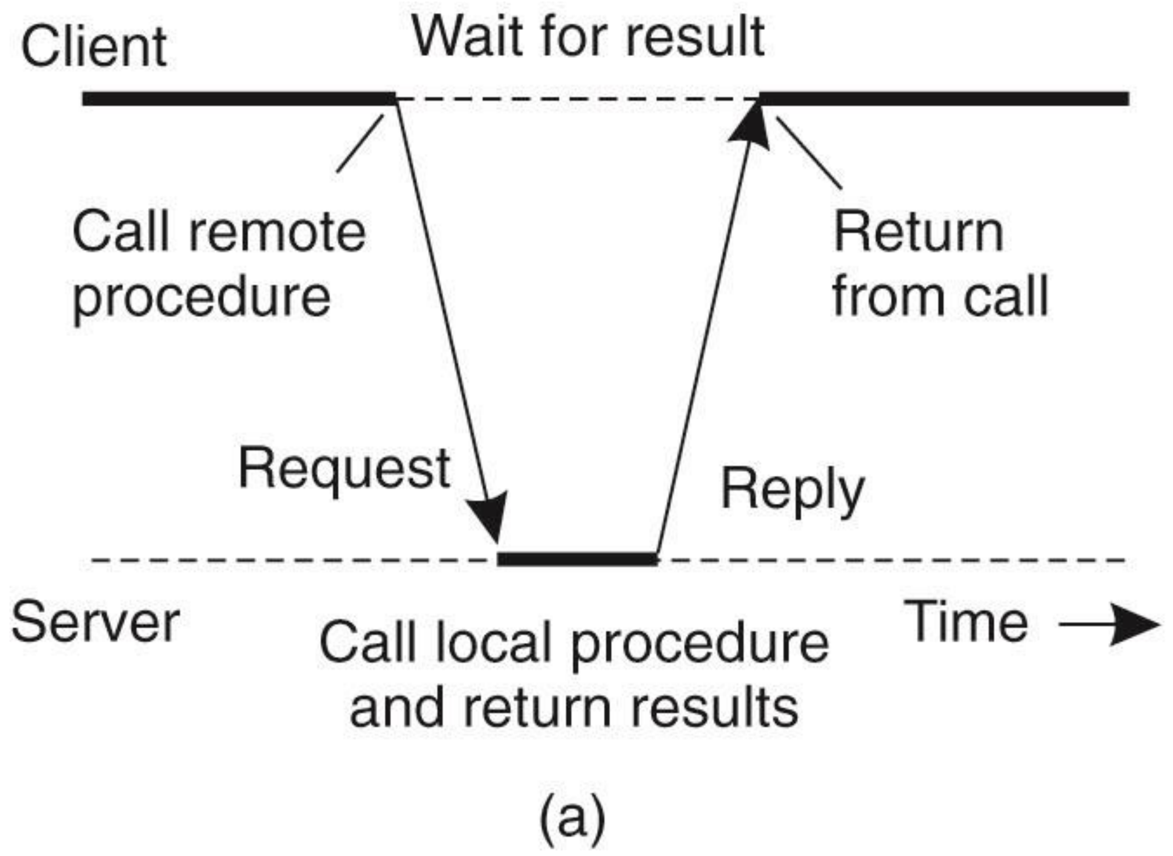
Reliable versus Unreliable RPC

- If RPC is built on a reliable transport protocol (e.g., TCP) it will behave more like a true procedure call.
- On the other hand, programmers may want a faster, connectionless protocol (e.g., UDP) or the client/server system may be on a LAN.
- How does this affect returned results?

Asynchronous RPC

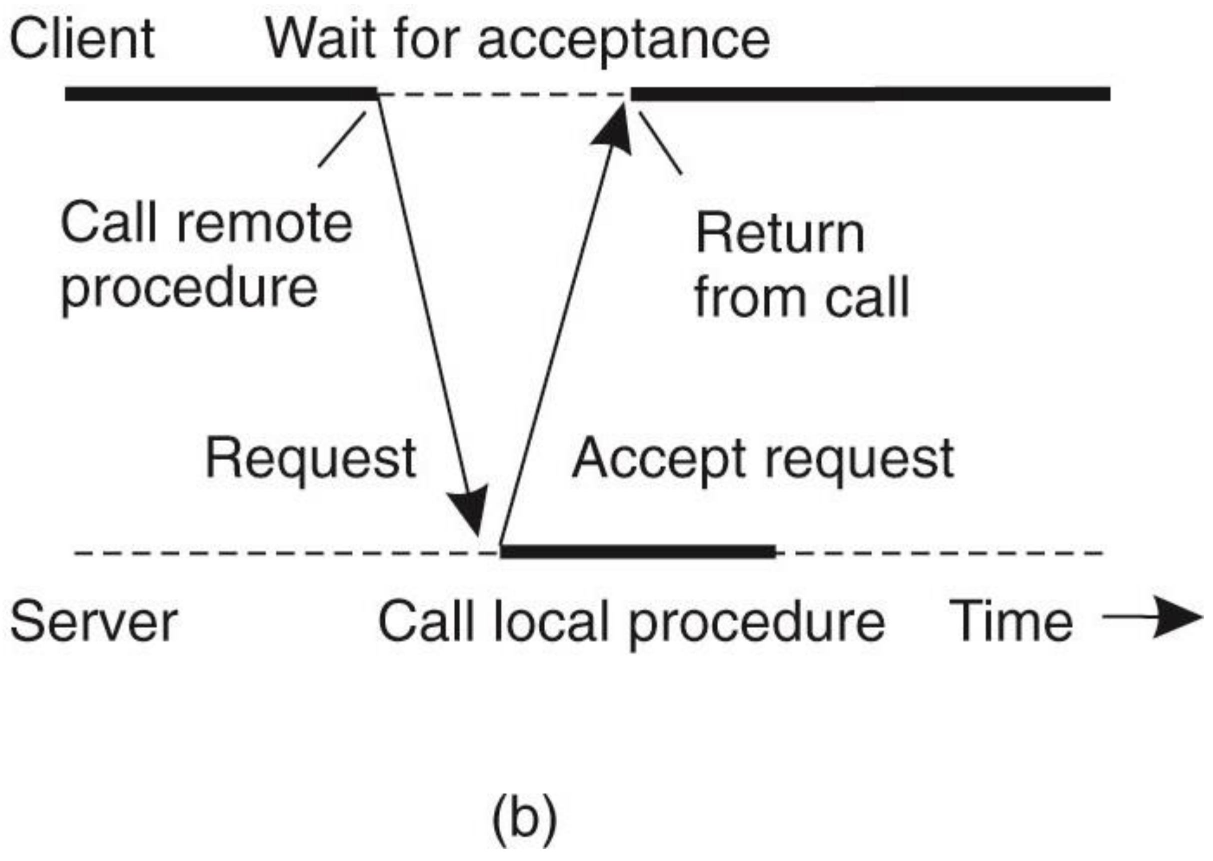
- Allow client to continue execution as soon as the RPC is issued and acknowledged, but before work is completed
  - Appropriate for requests that don’t need replies, such as a print request, file delete, etc.
  - Also may be used if client simply wants to continue doing something else until a reply is received (improves performance)
  - What are the problems with unreliable, asynchronous RPC?

Synchronous RPC



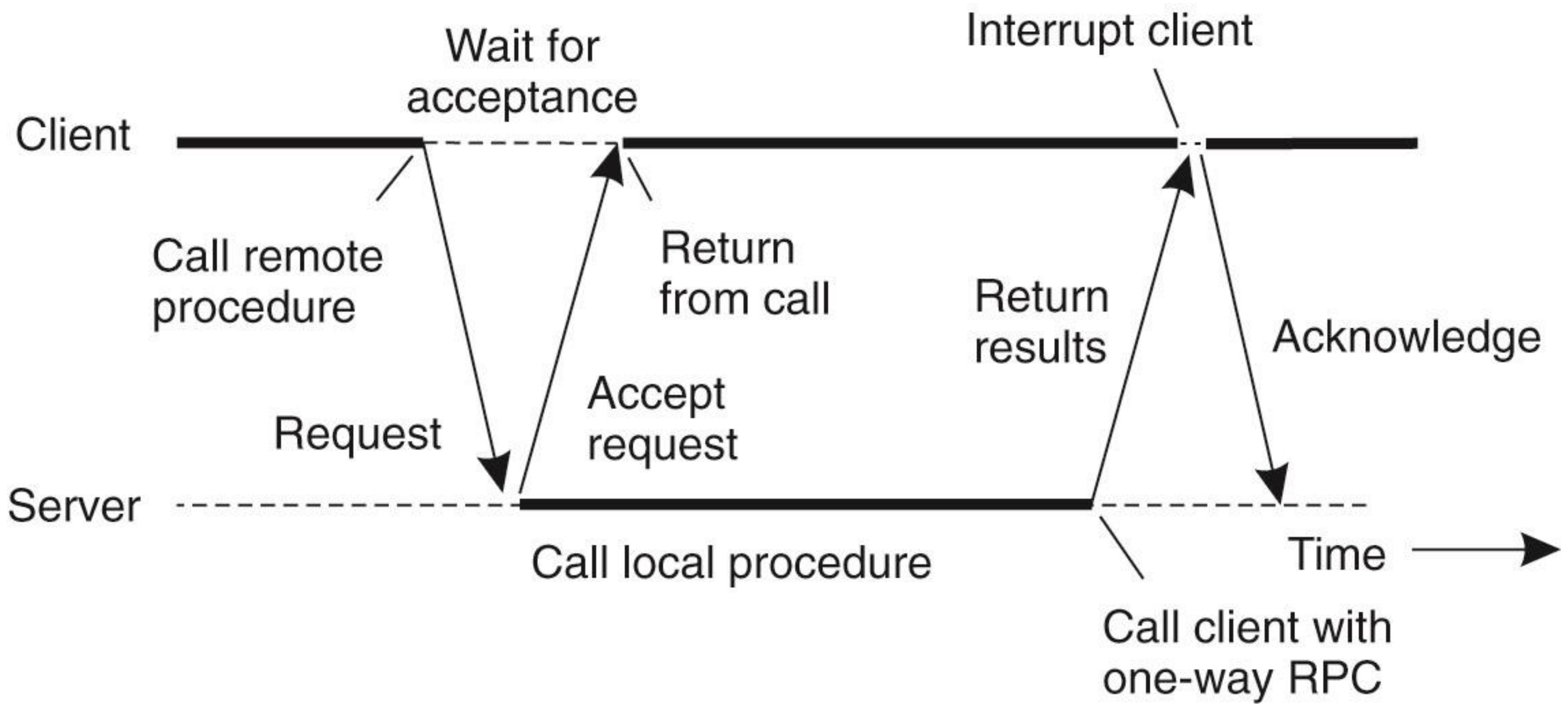
- Figure 4-10. (a) The interaction between client and server in a traditional RPC.

Asynchronous RPC



- Figure 4-10. (b) The interaction using asynchronous RPC.

Asynchronous RPC



- Figure 4-11. A client and server interacting through two asynchronous RPCs.

Synchronous or Asynchronous?

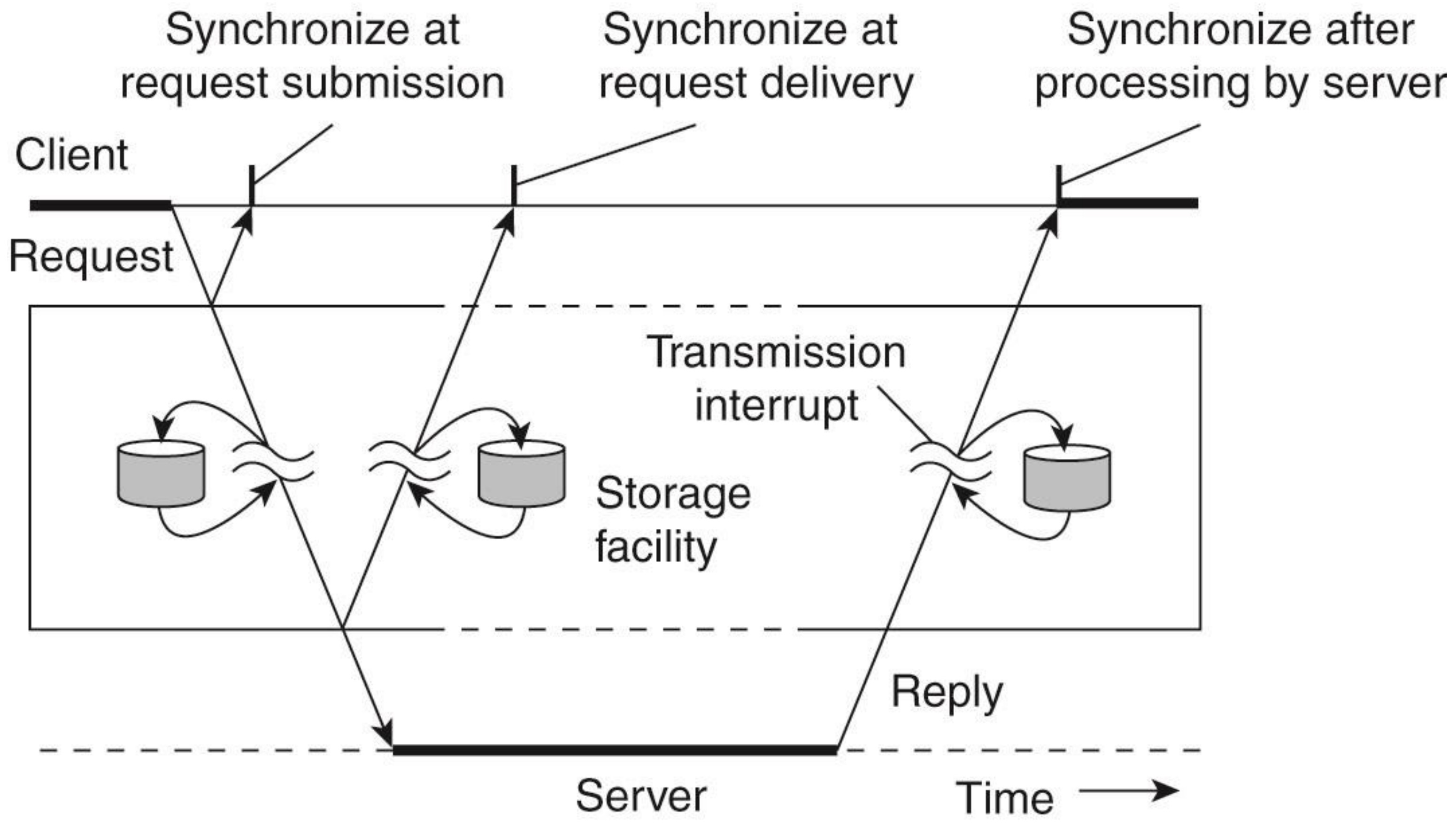

Synchronize at request submission
Synchronize at request delivery
Synchronize after processing by server
Client
Request
Transmission interrupt
Storage facility
Reply
Server
Time

Figure 4-4. Viewing middleware as an intermediate (distributed) service in application-level communication.

Message Oriented Communication

- RPC support access transparency, but aren’t always appropriate
- Message-oriented communication is more flexible
- Built on transport layer protocols.

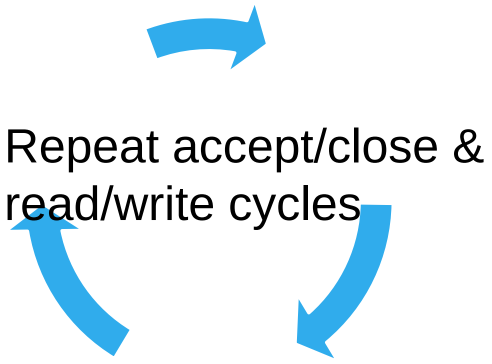
Sockets

- A communication endpoint used by applications to write and read to/from the network.
- Sockets provide a basic set of primitive operations
- Sockets are an abstraction of the actual communication endpoint used by local OS
- Socket address: IP# + port#

Primitive	**Meaning**
Socket	Create new communication end point
Bind	Attach a local address to a socket
Listen	Willing to accept *connections* (non-blocking)
Accept	Block caller until 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

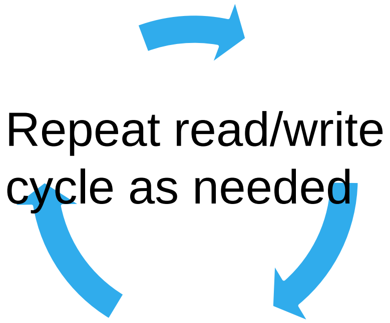
How a Server Uses Sockets

System Calls	Meaning
• Socket	• Create socket descriptor
• Bind	• Bind local IP address/ port # to the socket
• Listen	• Place in passive mode, set up request queue
• Accept	• Get the next message
• Read	• Read data from the network
• Write	• Write data to the network
• Close	• Terminate connection



How a Client Uses Sockets

System Calls	Meaning
• Socket	• Create socket descriptor
• Connect	• Connect to a remote server
• Write	• Write data to the network
• Read	• Read data from the network
• Close	• Terminate connection



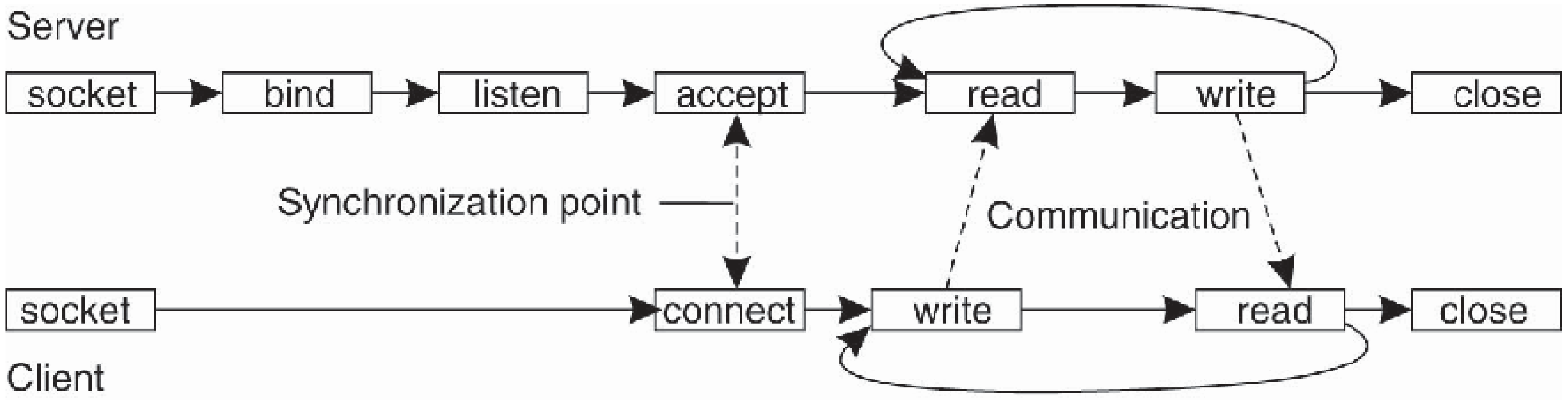


Figure 4-15. Connection-oriented communication pattern using sockets.

Socket Communication

- Using sockets, clients and servers can set up a connection-oriented communication session.
- Servers execute first four primitives (socket, bind, listen, accept) while clients execute socket and connect primitives)
- Then the processing is client/write, server/read, server/write, client/read, all close connection.

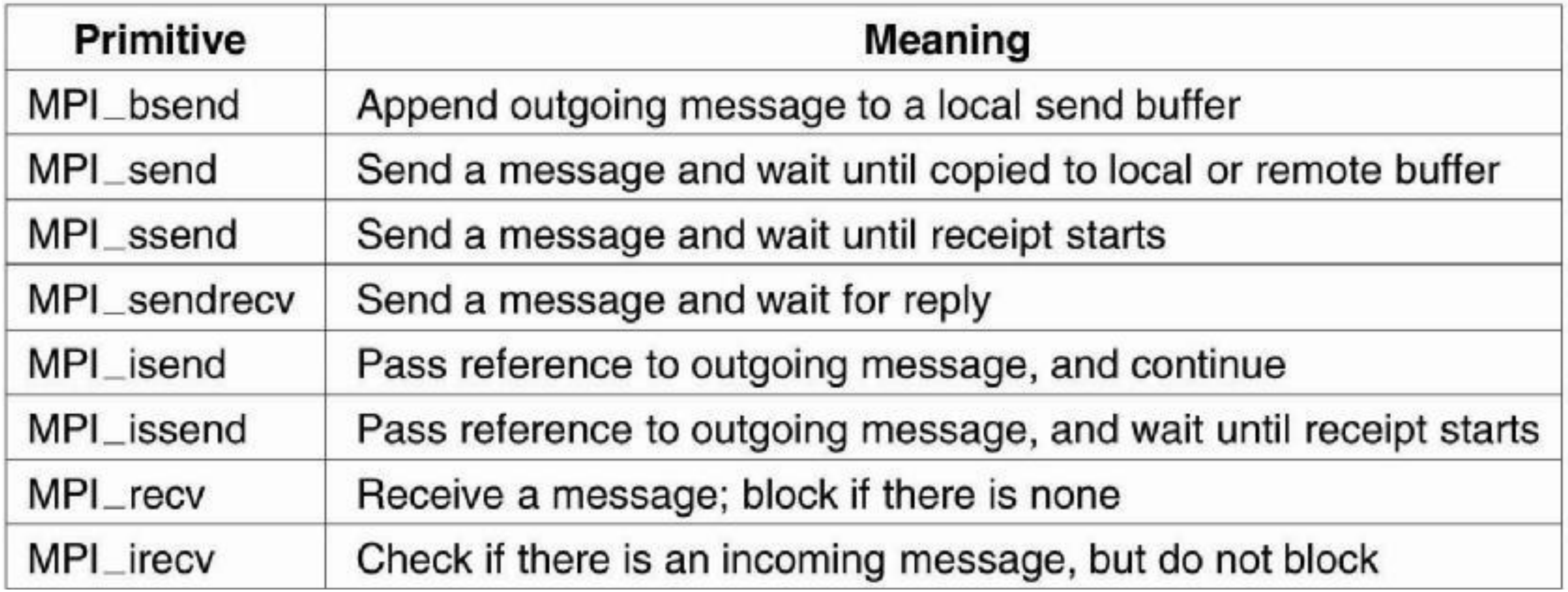
Message-Passing Interface (MPI)

- Sockets provide a low-level (send, receive) interface to wide-area (TCP/IP-based) networks
- Distributed systems that run on high-speed networks in high-performance cluster systems need more advanced protocols
- A need to be hardware/platform independent eventually led to the development of the MPI standard for message passing.

MPI

- Designed for parallel applications using transient communication
- Assumes communication is among a group of processes that know about each other
- Assign groupID to group, processID to each process in a group
- (groupID, processID) serves as an address

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

Message-Oriented Persistent Communication

- Processes communicate through message queues
  - sender appends to queue, receiver removes from queue
- MPI and sockets support transient communication, message queuing allows messages to be stored temporarily (minutes versus milliseconds).
  - Neither the sender nor receiver needs to be on-line when the message is transmitted.
- Designed for messages that take minutes to transmit.

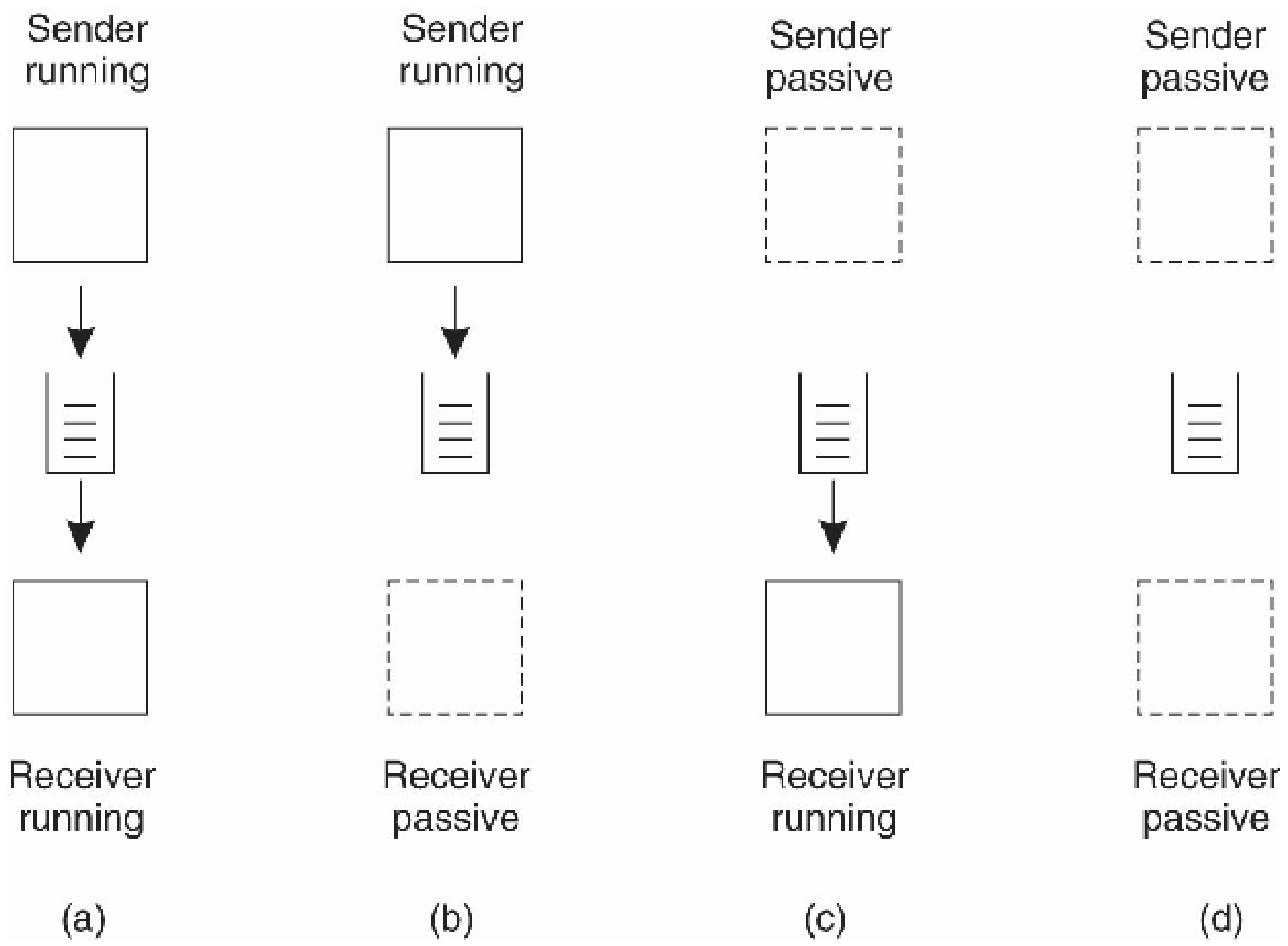
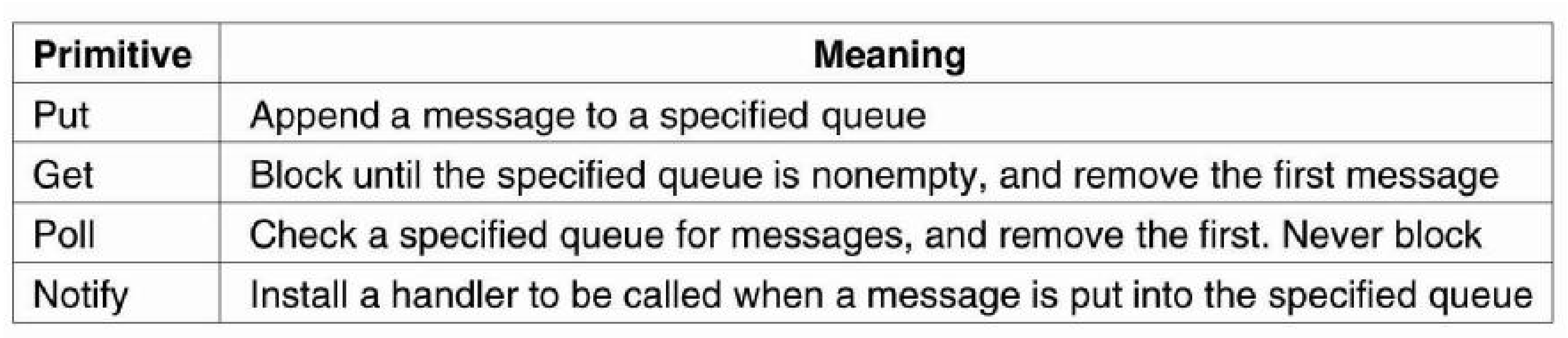


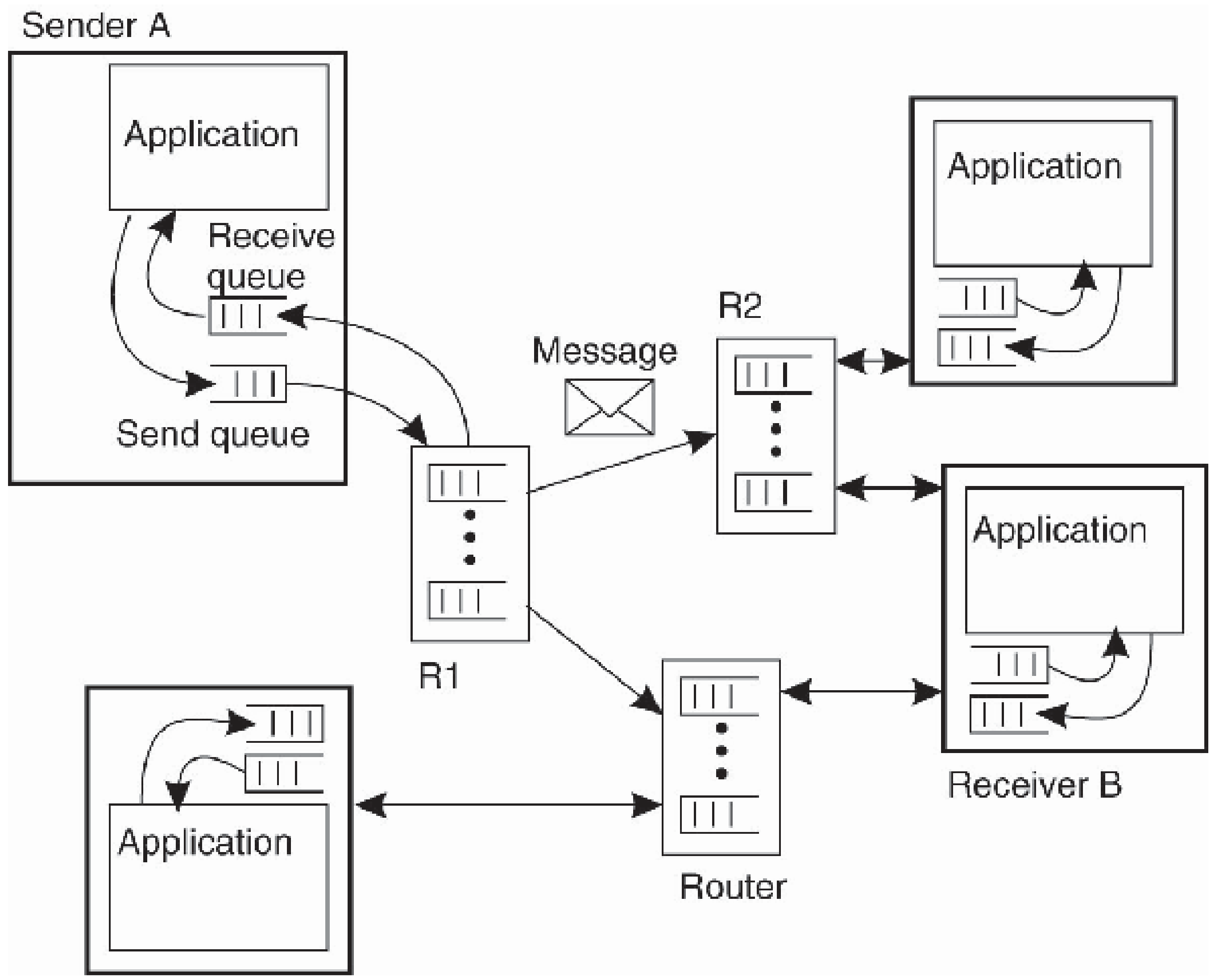
Figure 4-17. Four combinations for loosely-coupled communications using queues.

Message-Queuing Model

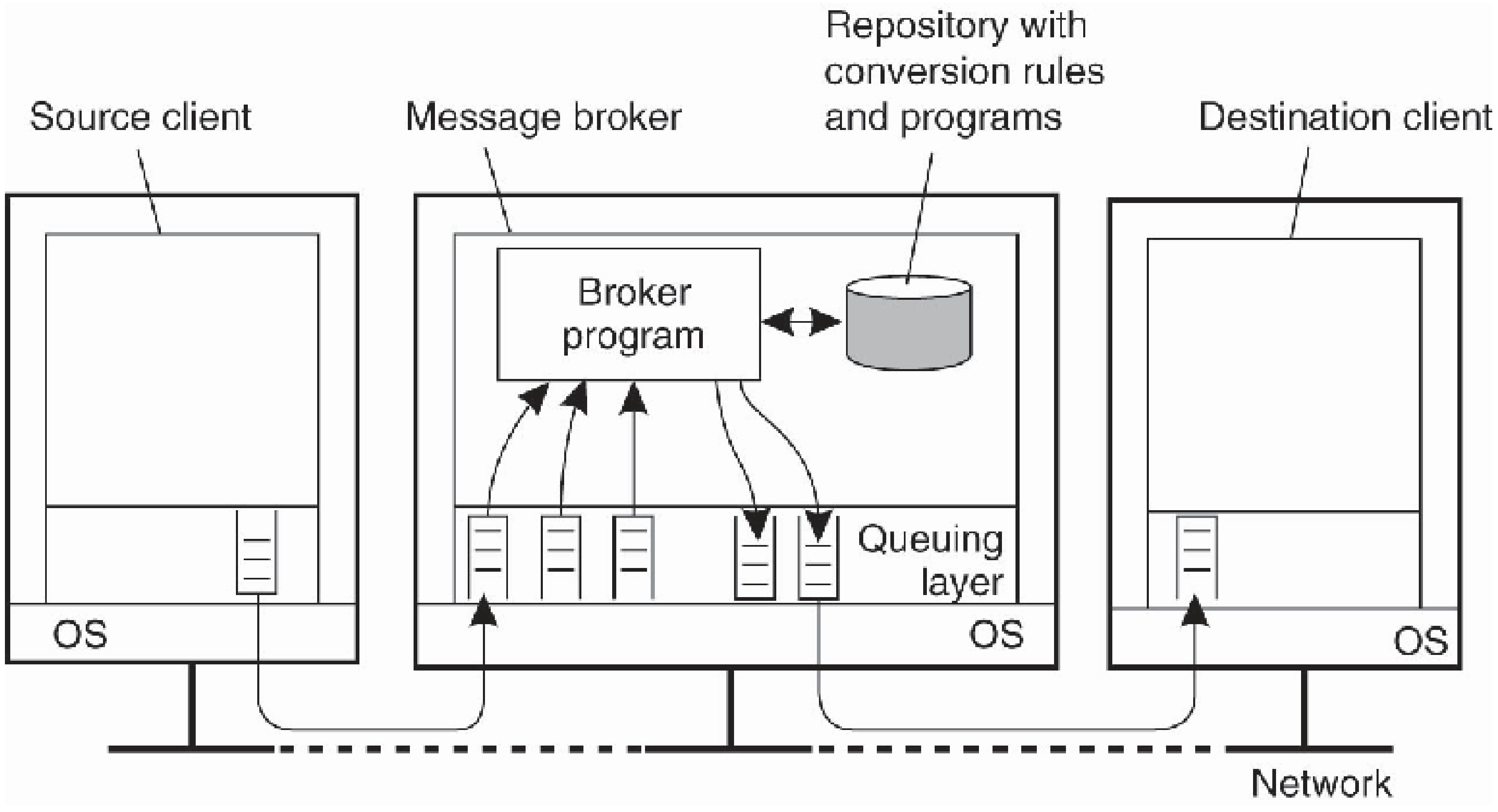


Primitive	Meaning
Put	Append a message to a specified queue
Get	Block until the specified queue is nonempty, and remove the first message
Poll	Check a specified queue for messages, and remove the first. Never block
Notify	Install a handler to be called when a message is put into the specified queue

General Architecture of a Message-Queuing System



Message Brokers



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

Data Streams

- Data stream = sequence of data items
- Can apply to discrete, as well as continuous media
- Audio and video require continuous data streams between file and device.

- **Asynchronous transmission mode:** the order is important, and data is transmitted one after the other. (file trans.)
- **Synchronous transmission** mode transmits each data unit with a guaranteed upper limit to the delay for each unit. (sensors)
- **Isochronous transmission** mode have a maximum and minimum delay. (video & audio)
  - Not too slow, but not too fast either

- Simple streams have a single data sequence
- Complex streams have several substreams, which must be synchronized with each other; for example a movie with
  - One video stream
  - Two audio streams (for stereo)
  - One stream with subtitles

Streams and Quality of Service

1. The required bit rate at which data should be transported.
2. The maximum delay until a session has been set up (i.e., when an application can start sending data).
3. The maximum end-to-end delay (i.e., how long it will take until a data unit makes it to a recipient).
4. The maximum delay variance.
5. The maximum round-trip delay.

Data Stream

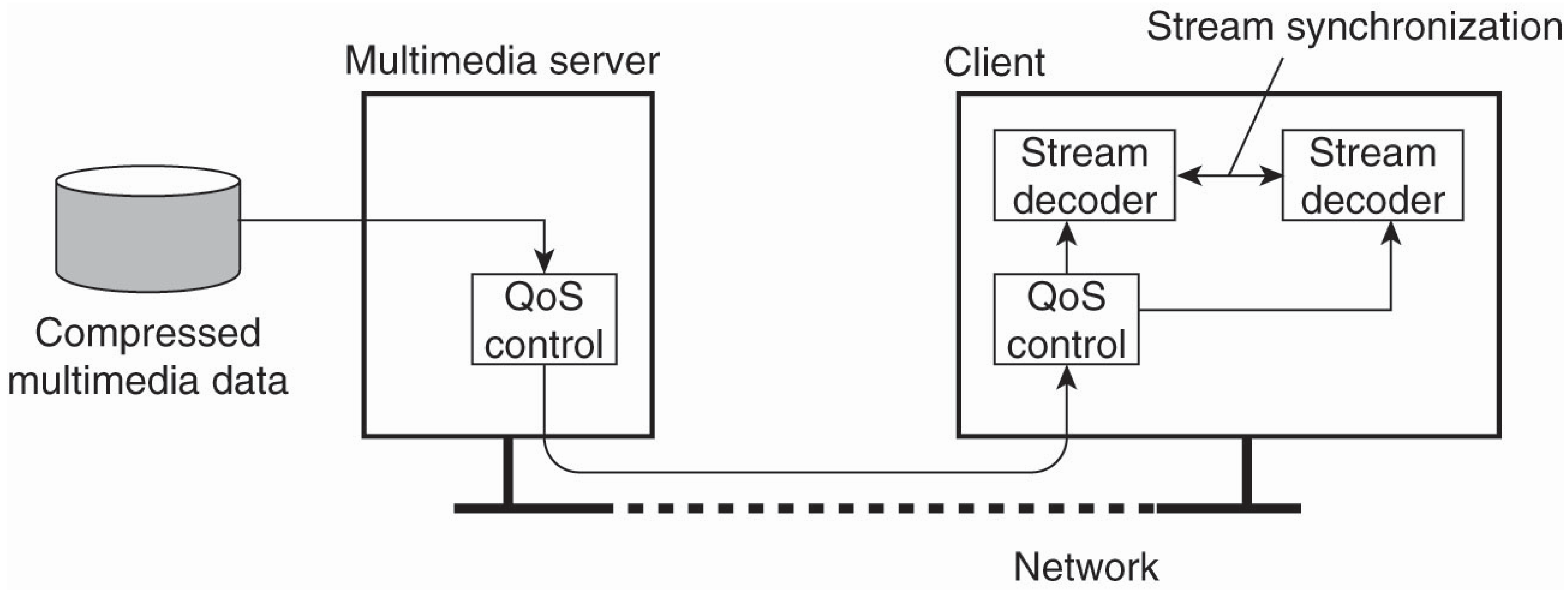


Figure 4-26. A general architecture for streaming stored multimedia data over a network.