DISTRIBUTED SYSTEMS
Principles and Paradigms

Second Edition

ANDREW S. TANENBAUM
MAARTEN VAN STEEN
By: Dr. Faramarz Safi
Islamic Azad University
Najafabad Branch

Chapter 4
Communication

Layered Protocols (1)

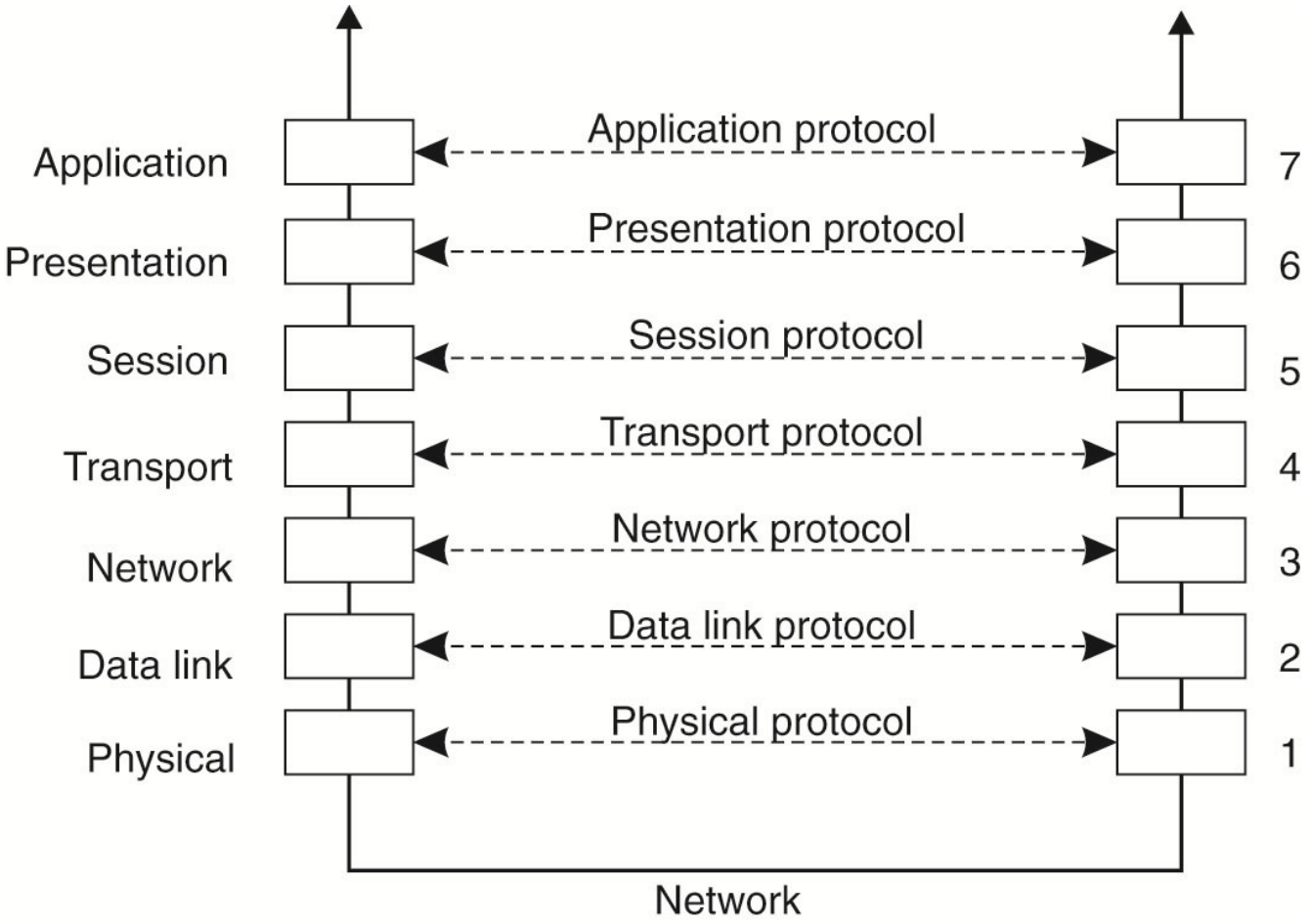


Figure 4-1. Layers, interfaces, and protocols in the OSI model.

Layered Protocols (2)

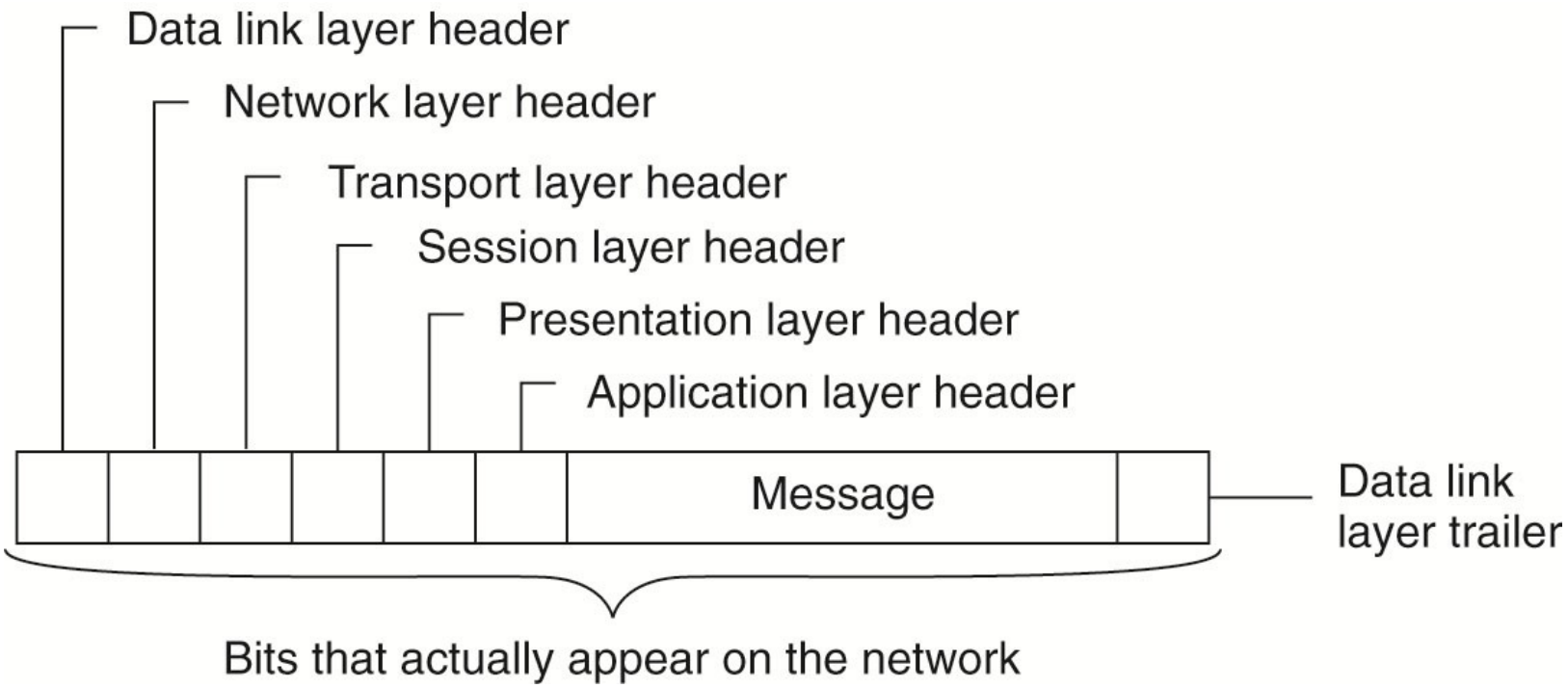


Figure 4-2. A typical message as it appears on the network.

Middleware Protocols

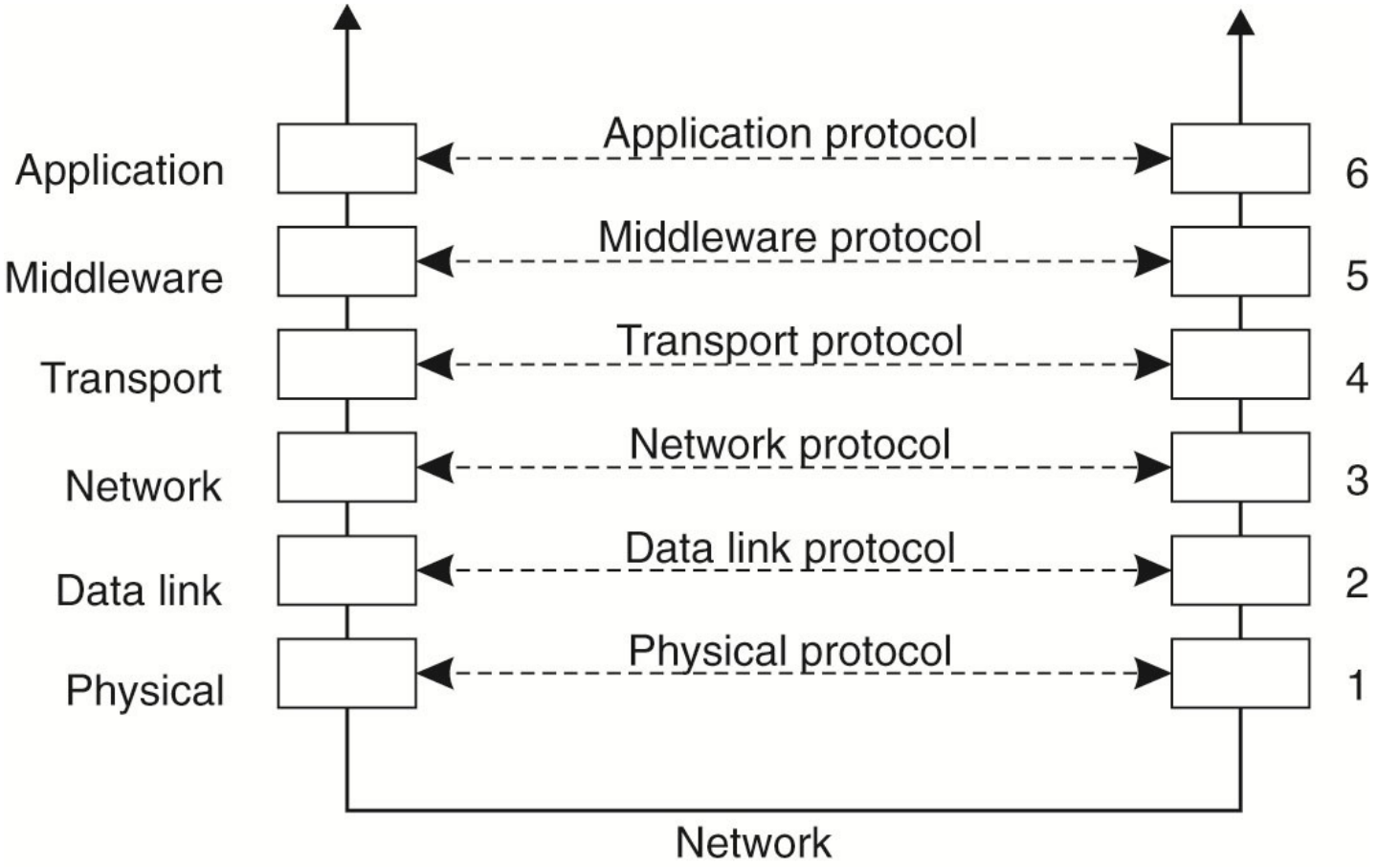


Figure 4-3. An adapted reference model for networked communication.

Types of Communication

Consider, for example an electronic mail system. In principle, the core of the mail delivery system can be seen as a middleware communication service.

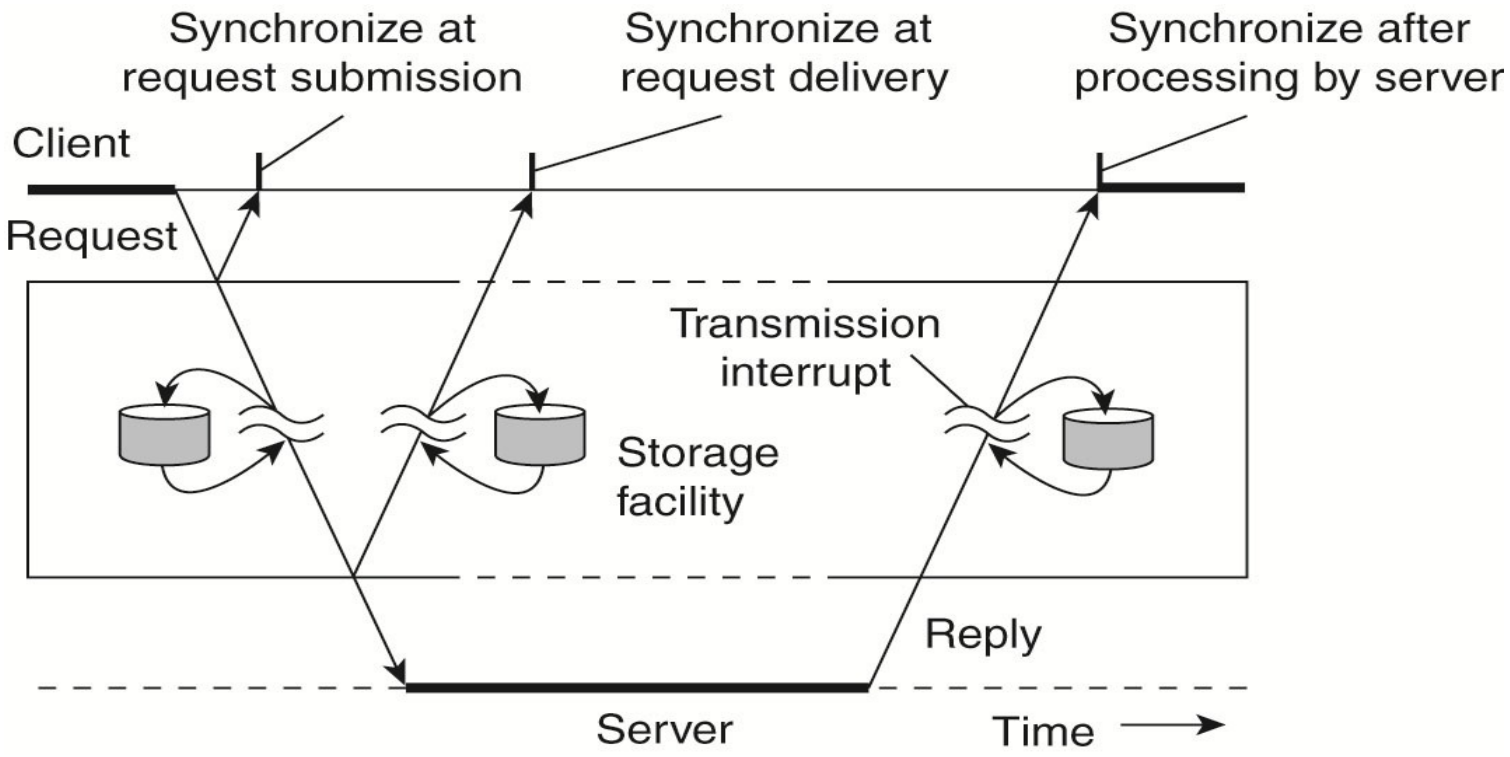


Figure 4-4. Viewing middleware as an intermediate (distributed) service in application-level communication.

Persistent vs. Transient

With **persistent** communication, a message that has been submitted for transmission is stored by the communication middleware as long as it takes to deliver it to the receiver. In this case, the middleware will store the message at one or several of the storage facilities. As a consequence, it is not necessary for the sending application to continue execution after submitting the message. Likewise, the receiving application need not be executing when the message is submitted.

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

Synchronous vs. Asynchronous

The characteristic feature of **asynchronous** communication is that a sender continues immediately after it has submitted its message for transmission. This means that the message is (temporarily) stored immediately by the middleware upon submission. With **synchronous** communication, the sender is blocked until its request is known to be accepted. There are essentially three points where synchronization can take place.

First, the sender may be blocked until the middleware notifies that it will take over transmission of the request.

Second, the sender may synchronize until its request has been delivered to the intended recipient.

Third, synchronization may take place by letting the sender wait until its request has been fully processed, that is, up the time that the recipient returns a response.

Synchronous vs. Asynchronous

Various combinations of persistence and synchronization occur in practice. Popular ones are persistence in combination with synchronization at request submission, which is a common scheme for many message-queuing systems, which we discuss later in this chapter. Likewise, transient communication with synchronization after the request has been fully processed is also widely used.

Discrete vs. Streaming

We should also make a distinction between discrete and streaming communication. The examples so far all fall in the category of discrete communication: the parties communicate by messages, each message forming a complete unit of information. In contrast, streaming involves sending multiple messages, one after the other, where the messages are related to each other by the order they are sent, or because there is a temporal relationship. Remote Procedure Call can be considered as this type of communication.

Conventional Procedure Call

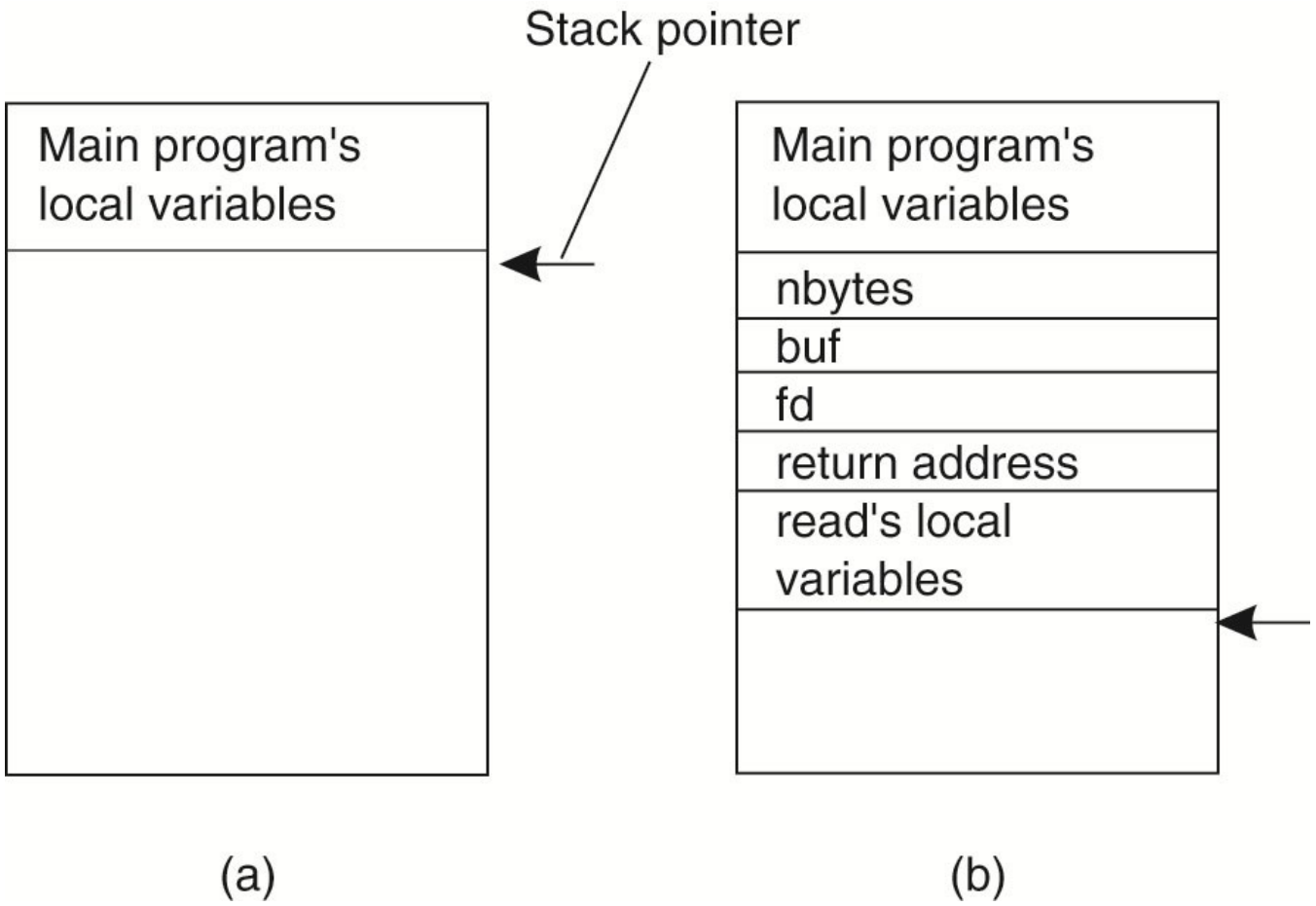


Figure 4-5. (a) Parameter passing in a local procedure call: the stack before the call to read. (b) The stack while the called procedure is active.

Client and Server Stubs

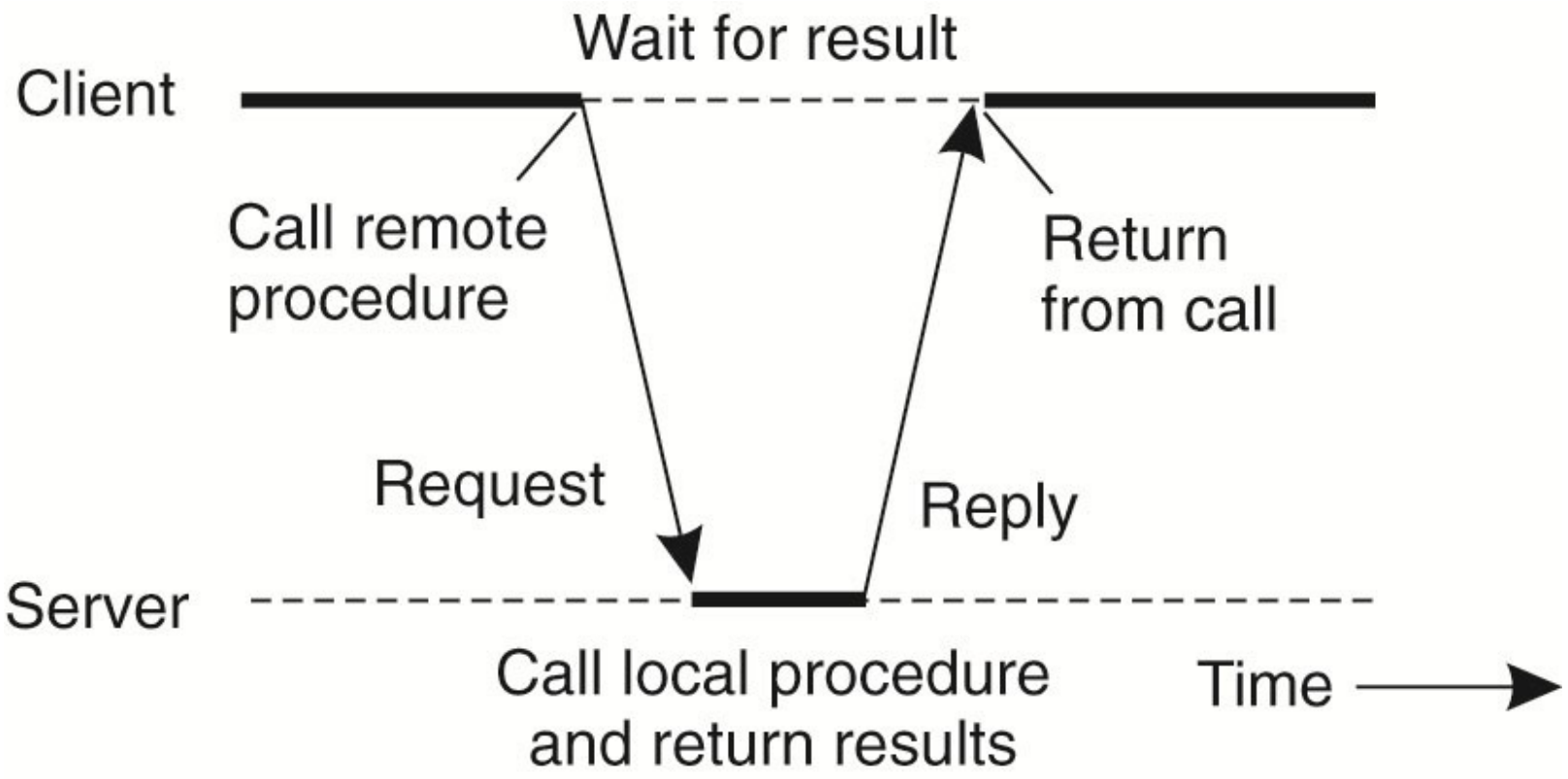


Figure 4-6. Principle of RPC between a client and server program.

Remote Procedure Calls (1)

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 OS sends the message to the remote OS.
4. The remote OS gives the message to the server stub.
5. The server stub unpacks the parameters and calls the server.

Continued …

Remote Procedure Calls (2)

A remote procedure call occurs in the following steps (continued):

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

Passing Value Parameters (1)

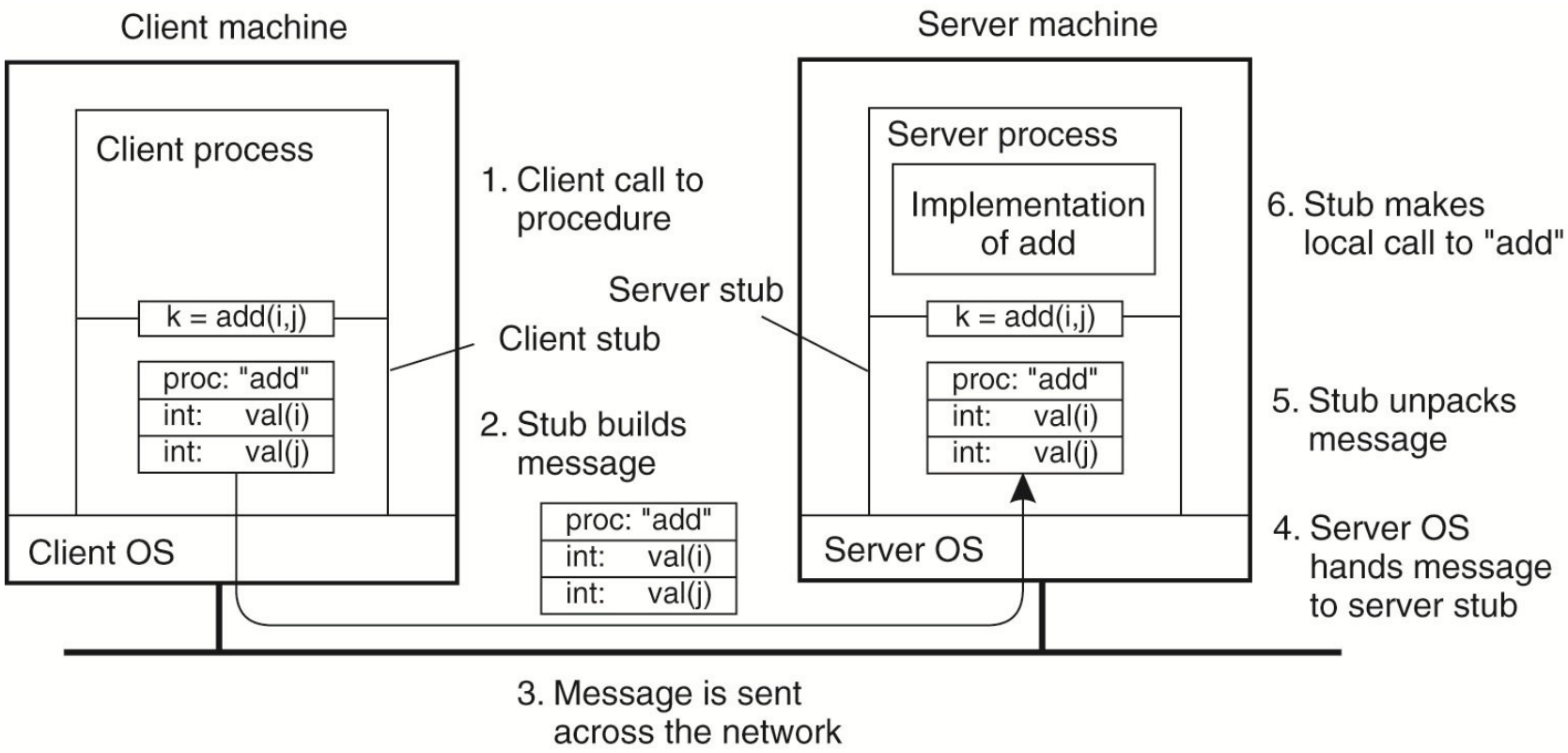
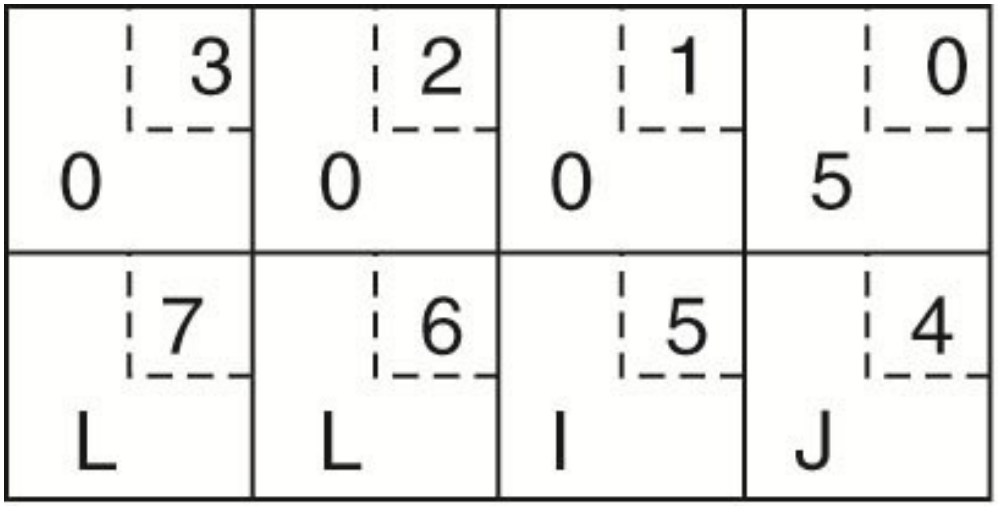


Figure 4-7. The steps involved in a doing a remote computation through RPC.

Passing Value Parameters (2)



(a)

Figure 4-8. (a) The original message on the Pentium.

Passing Value Parameters (3)

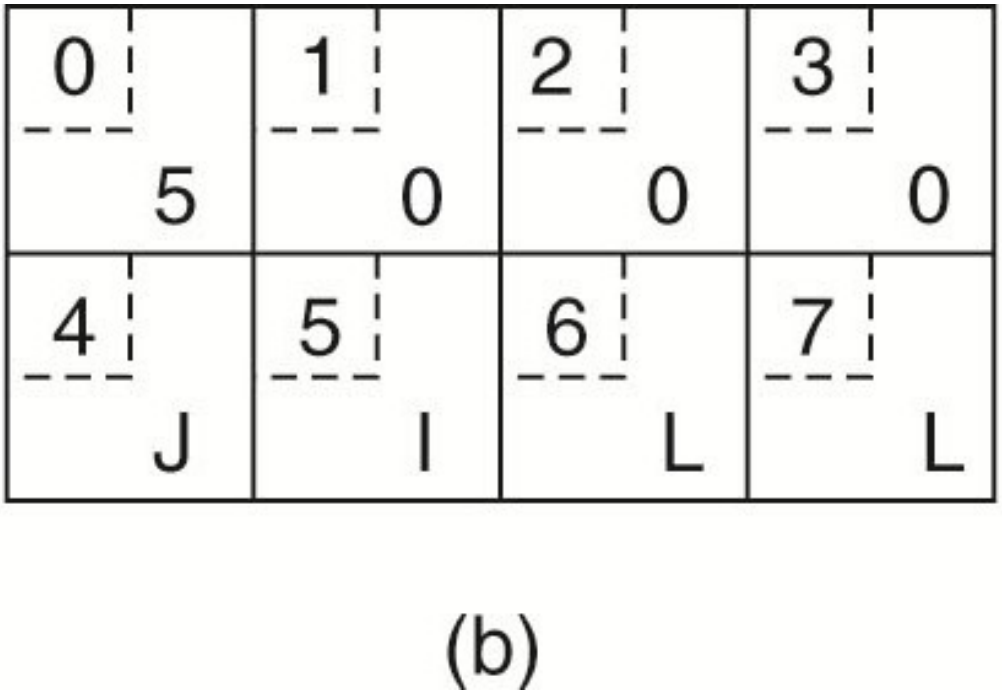
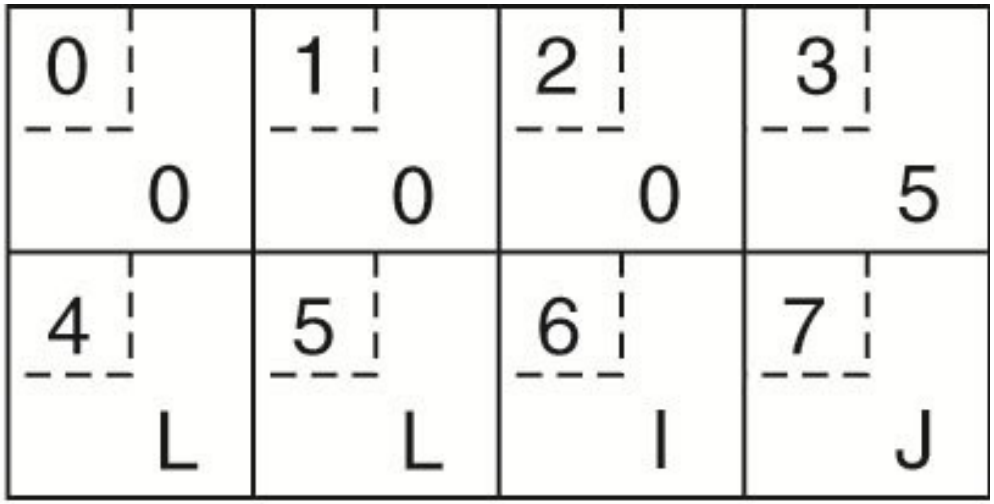


Figure 4-8. (a) The original message on the Pentium.

Passing Value Parameters (4)



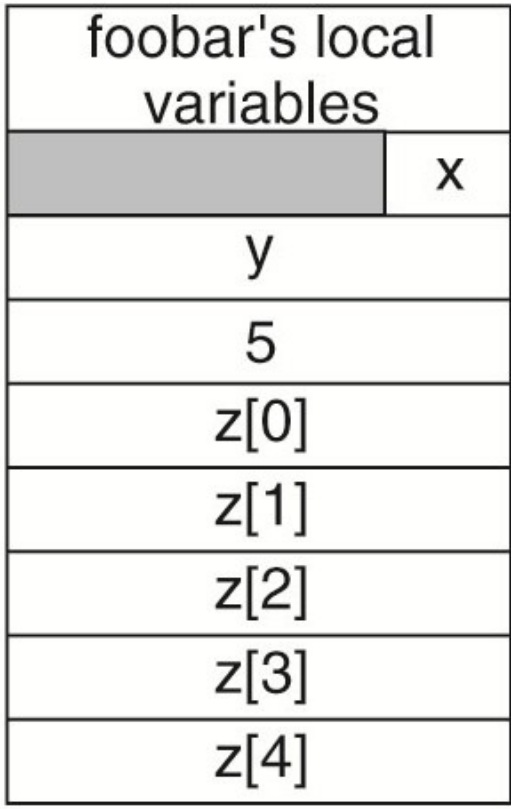
(c)

Figure 4-8. (c) The message after being inverted. The little numbers in boxes indicate the address of each byte.

Parameter Specification and Stub Generation

```
foobar( char x; float y; int z[5] )
{
   ....
}
```

(a)



(b)

Figure 4-9. (a) A procedure. (b) The corresponding message.

Asynchronous RPC (1)

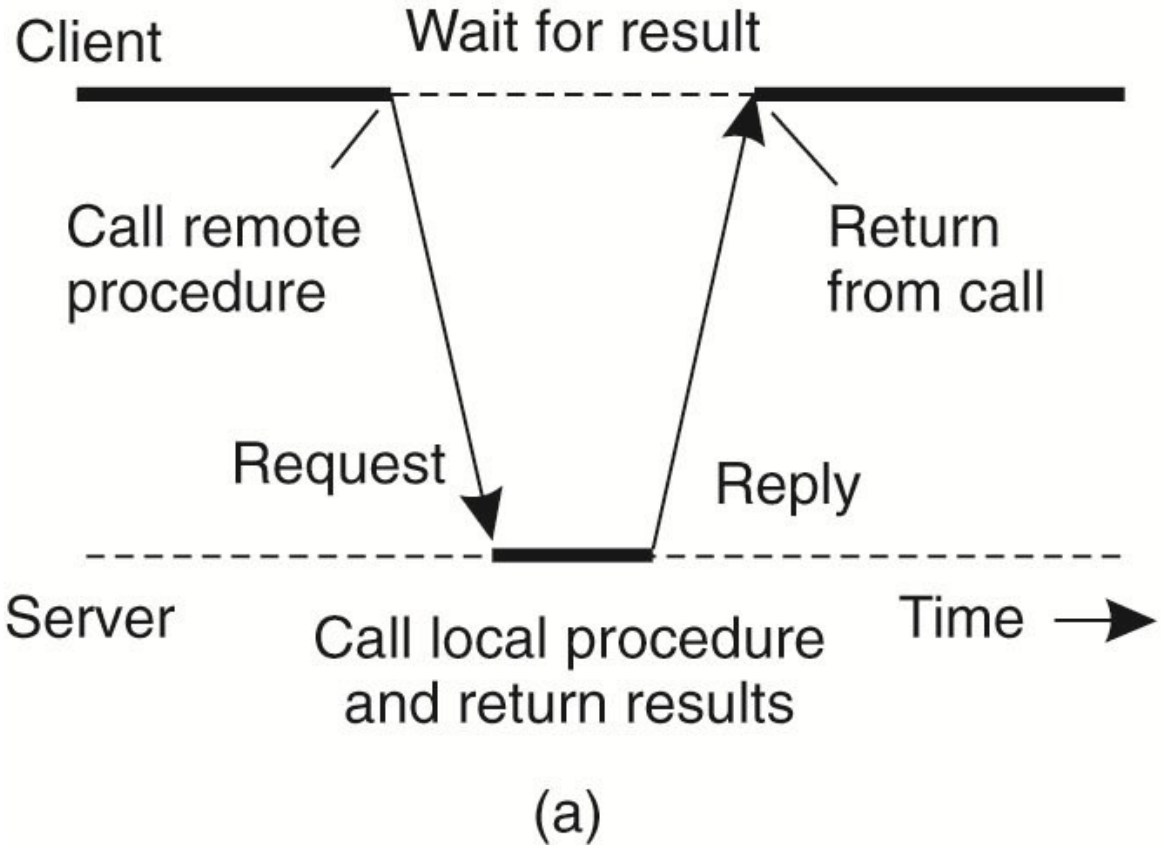


Figure 4-10. (a) The interaction between client and server in a traditional RPC.

Asynchronous RPC (2)

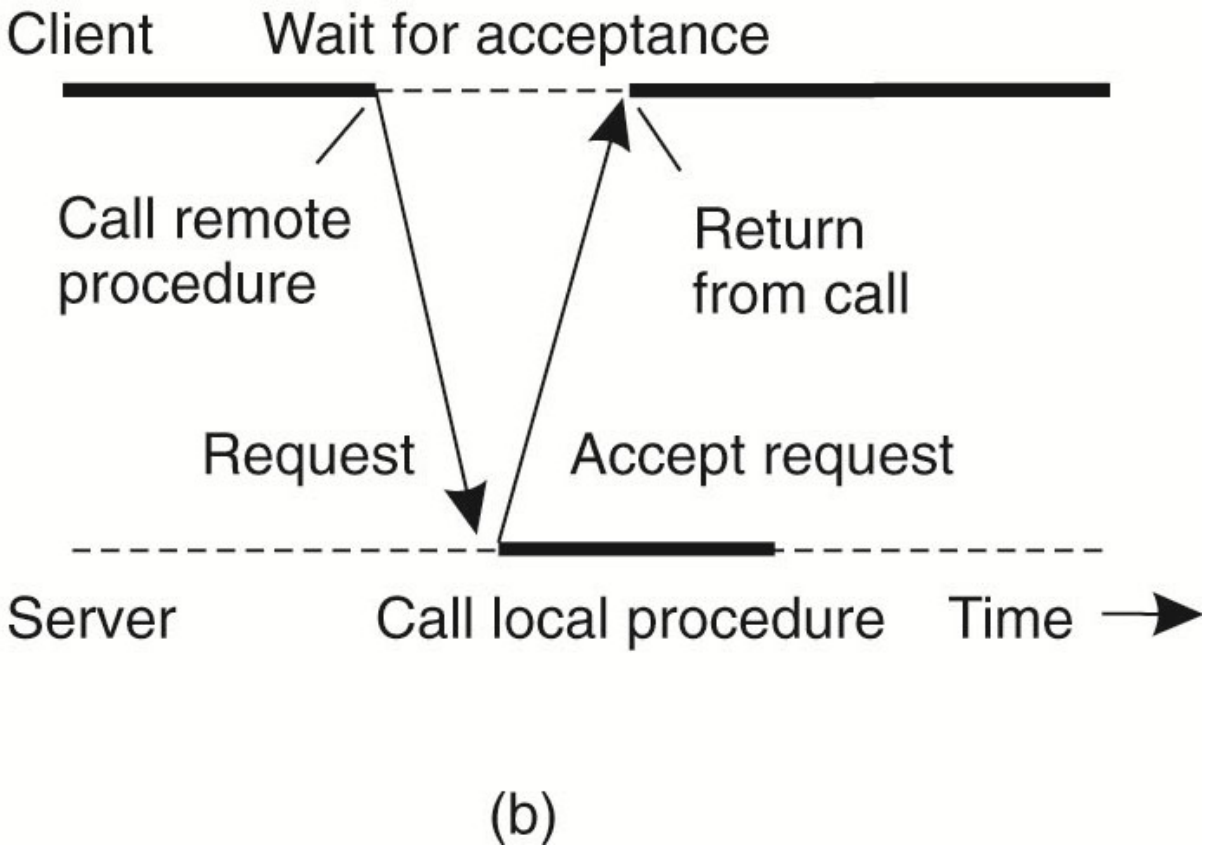


Figure 4-10. (b) The interaction using asynchronous RPC.

Asynchronous RPC (3)

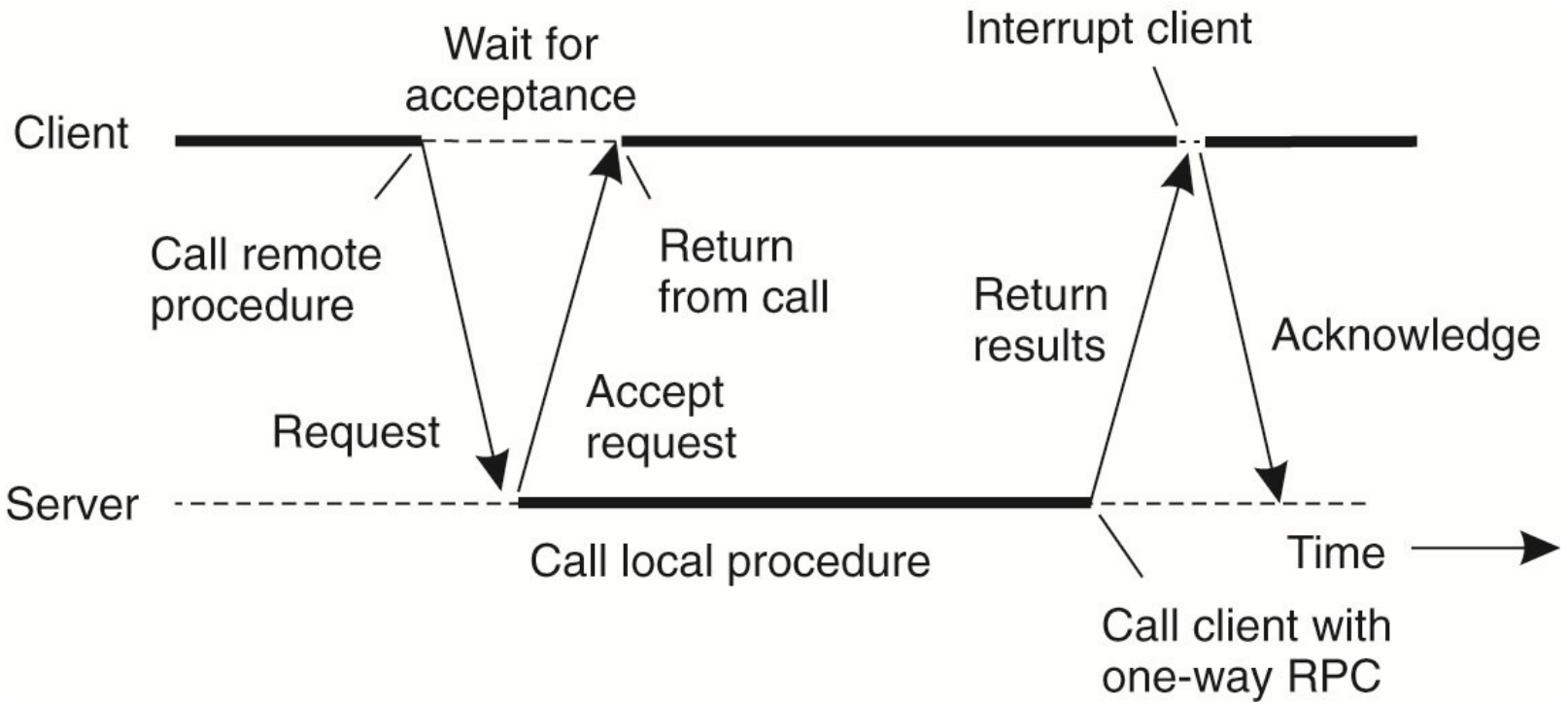


Figure 4-11. A client and server interacting through two asynchronous RPCs.

Writing a Client and a Server (1)

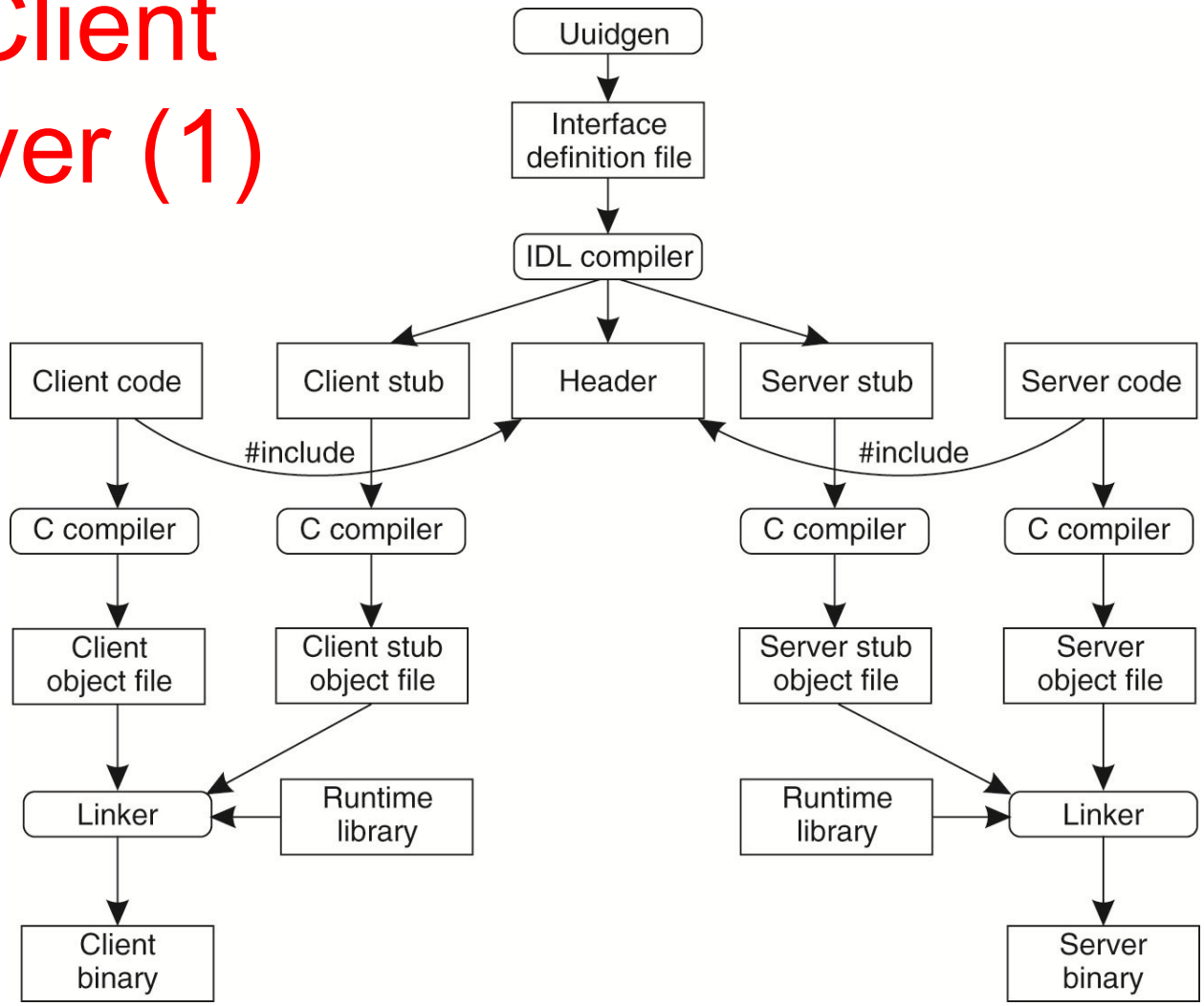


Figure 4-12. The steps in writing a client and a server in DCE RPC.

Writing a Client and a Server (2)

Three files output by the IDL compiler:

- A header file (e.g., interface.h, in C terms).
- The client stub.
- The server stub.

Binding a Client to a Server (1)

- Registration of a server makes it possible for a client to locate the server and bind to it.

- Server location is done in two steps:
  1. Locate the server's machine.
  2. Locate the server on that machine.

Binding a Client to a Server (2)

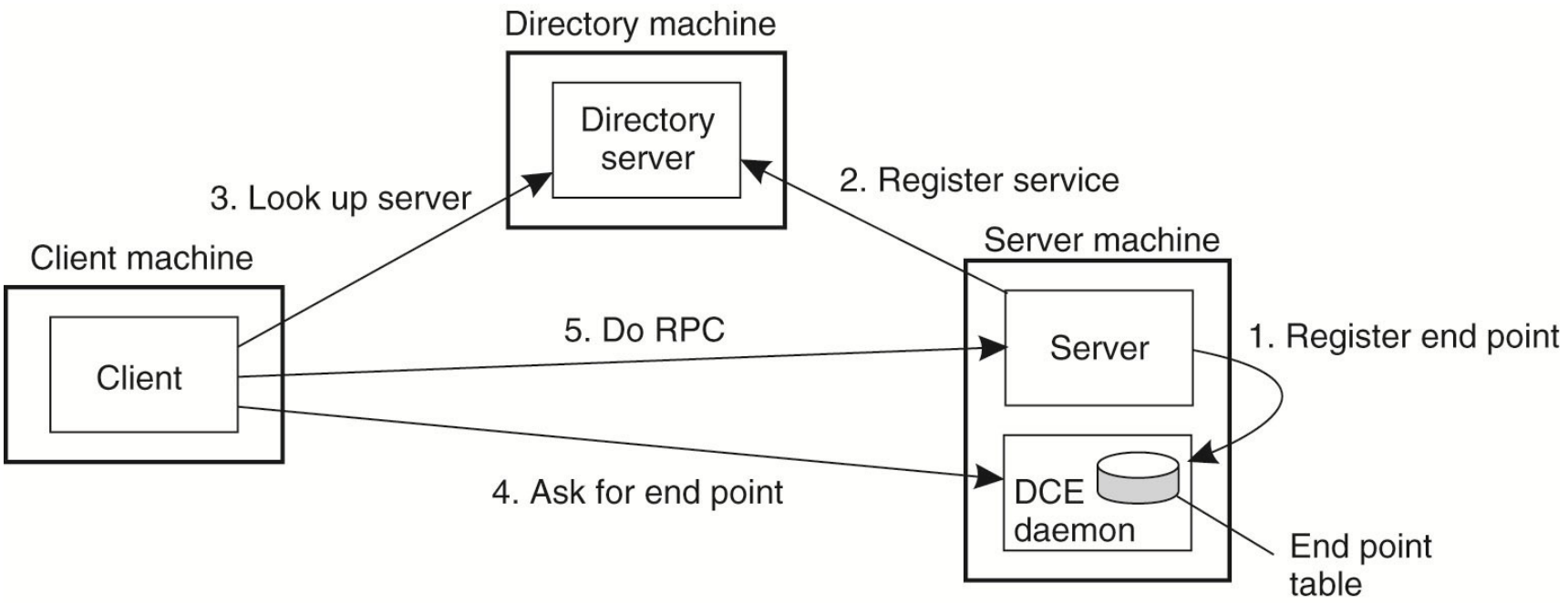


Figure 4-13. Client-to-server binding in DCE.

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

Figure 4-14. The socket primitives for TCP/IP.

The Message-Passing Interface

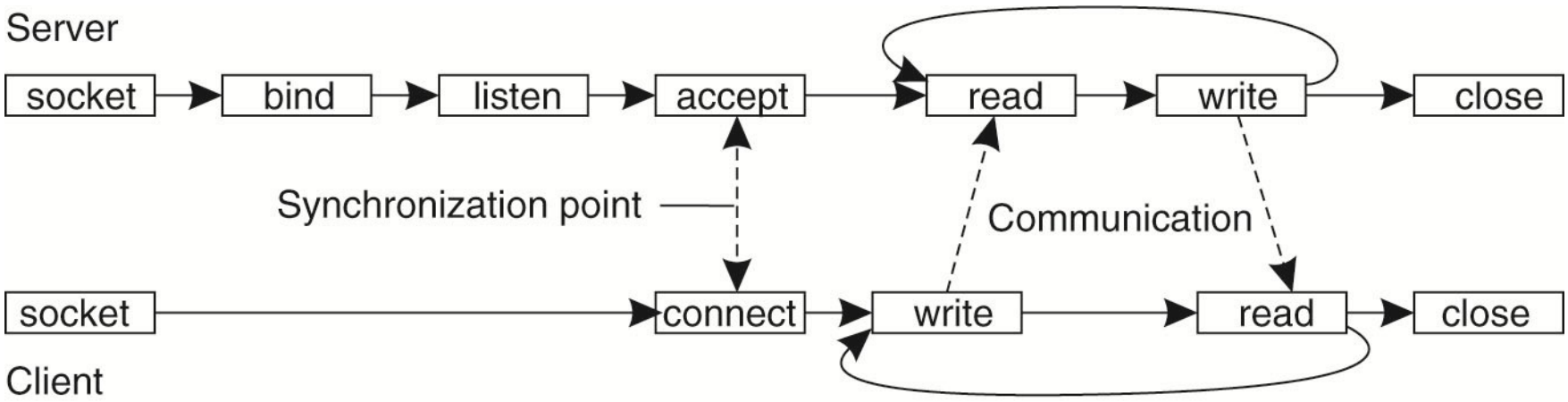


Figure 4-15. Connection-oriented communication pattern using sockets.

The Message-Passing Interface

The need to be hardware and platform independent eventually led to the definition of a standard for message passing, simply called the Message-Passing Interface or MPI. MPI is designed for parallel applications and as such is tailored to transient communication. It makes direct use of the underlying network. Also, it assumes that serious failures such as process crashes or network partitions are fatal and do not require automatic recovery.

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

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

Message-Oriented Persistent Communication

message-oriented middle ware services, generally known as message-queuing systems, or just Message-Oriented Middleware (MOM). Message-queuing systems provide extensive support for persistent asynchronous communication.

The essence of these systems is that they offer intermediate-term storage capacity for messages, without requiring either the sender or receiver to be active during message transmission. An important difference with Berkeley sockets and MPI is that message-queuing systems are typically

targeted to support message transfers that are allowed to take minutes instead of seconds or milliseconds.

Message Queue Models

- Message-oriented middle ware services, generally known as message-queuing systems, or just Message-Oriented Middleware (MOM). Message-queuing systems provide extensive support for persistent asynchronous communication.
- The essence of these systems is that they offer intermediate storage capacity for messages, without requiring either the sender or receiver to be active during message transmission.
- An important difference with Berkeley sockets and MPI is that message-queuing systems are typically targeted to support message transfers that are allowed to take minutes instead of seconds or milliseconds.
- A sender is generally given only the guarantees that its message will eventually be inserted in the recipient's queue.

Message-Queuing Model

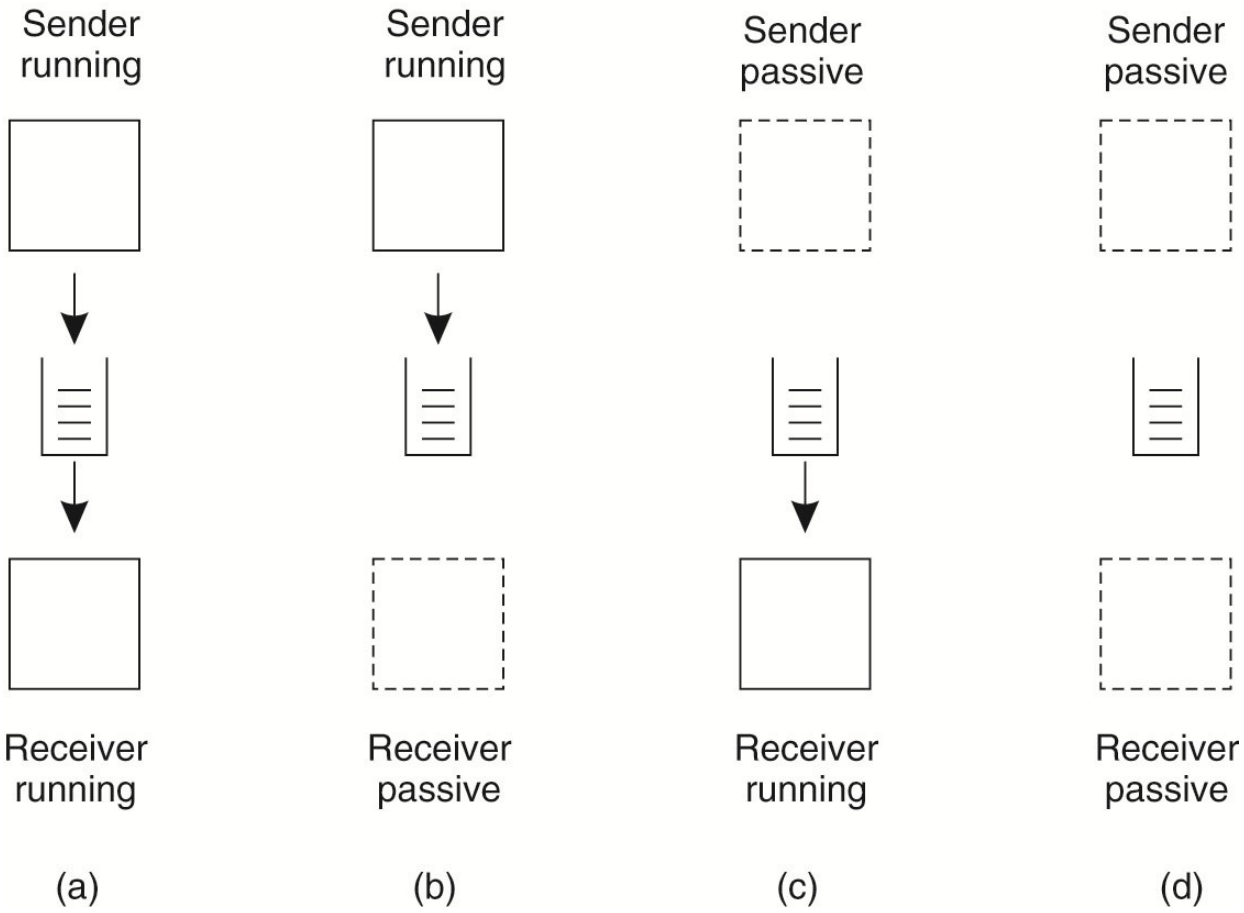


Figure 4-17. Four combinations for loosely-coupled communications using queues.

Message-Queuing Model

Messages can, in principle, contain any data. The only important aspect from the perspective of middleware is that messages are properly addressed. In practice, addressing is done by providing a system wide unique name of the destination queue. In some cases, message size may be limited, although it is also possible that the underlying system takes care of fragmenting and assembling large messages in a way that is completely transparent to applications. An effect of this approach is that the basic interface offered to applications can be extremely simple.

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

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

Message-Queue Model
Callback

Finally, most queuing systems also allow a process to install a handler as a callback function, which is automatically invoked whenever a message is put into the queue. Callbacks can also be used to automatically start a process that will fetch messages from the queue if no process is currently executing. This approach is often implemented by means of a daemon on the receiver's side that continuously monitors the queue for incoming messages and handles accordingly.

General Architecture of a Message-Queuing System

One of the first **restrictions** that we make is that messages can be put only into queues that are *local to the sender, that is, queues on the same machine*, or no worse than on a machine nearby such as on the same LAN that can be efficiently reached through an RPC. Such a queue is called the source queue.

Likewise, messages can be read only from local queues. However, a message put into a queue will contain the specification of a destination queue to which it should be transferred. **It is the responsibility of a message-queuing system to provide queues to senders and receivers and take care that messages are transferred from their source to their destination queue**.

General Architecture of a Message-Queuing System

It is important to realize that the collection of queues is distributed across multiple machines. Consequently, for a message-queuing system to transfer messages, it should maintain a mapping of queues to network locations.

In practice, this means that it should maintain a (possibly distributed) database of queue names to network locations, as shown in Fig. 4-19. (Note that such a mapping is completely analogous to the use of the Domain Name System (DNS) for e-mail in the Internet).

For example, when sending a message to the logical *mail address steen@cs.vu.nl, the mailing system will query DNS to find the network (i.e., IP)* address of the recipient's mail server to use for the actual message transfer.

General Architecture of a Message-Queuing System

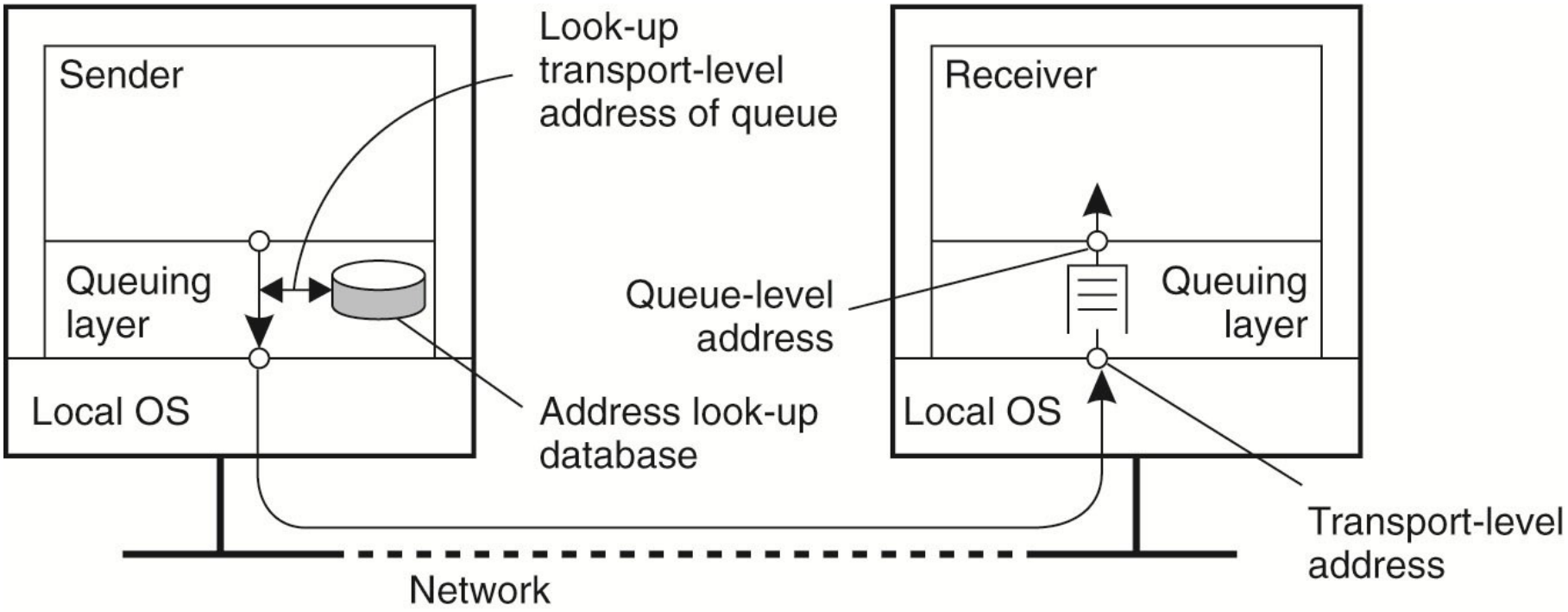


Figure 4-19. The relationship between queue-level addressing and network-level addressing.

General Architecture of a Message-Queuing System

Queues are managed by queue managers. Normally, a queue manager interacts directly with the application that is sending or receiving a message.

However, there are also special queue managers that operate as routers, or relays; they forward incoming messages to other queue managers. In this way, a message queuing system may gradually grow into a complete, application-level, overlay network, on top of an existing computer network.

General Architecture of a Message-Queuing System Queue-to-Location Mapping

In many message-queuing systems, there is no general naming service available that can dynamically maintain **queue-to-location mappings**.

Instead, the topology of the queuing network is **static**, and each queue manager needs a copy of the queue-to-location mapping. It is needless to say that in large-scale queuing systems this approach can easily lead to network-management problems.

One solution is to use a few **routers** that know about the network topology. When a sender *A puts a message for destination B in its local queue, that message* is first transferred to the nearest router, say *Rl, as shown in Fig. 4-20.*

At that point, the router knows what to do with the message and forwards it in the direction of *B. For example, Rl may derive from B's name that the message should be* forwarded to router *R2. In this way, only the routers need to be updated when* queues are added or removed. while every other queue manager has to know only where the nearest router is.

General Architecture of a Message-Queuing System

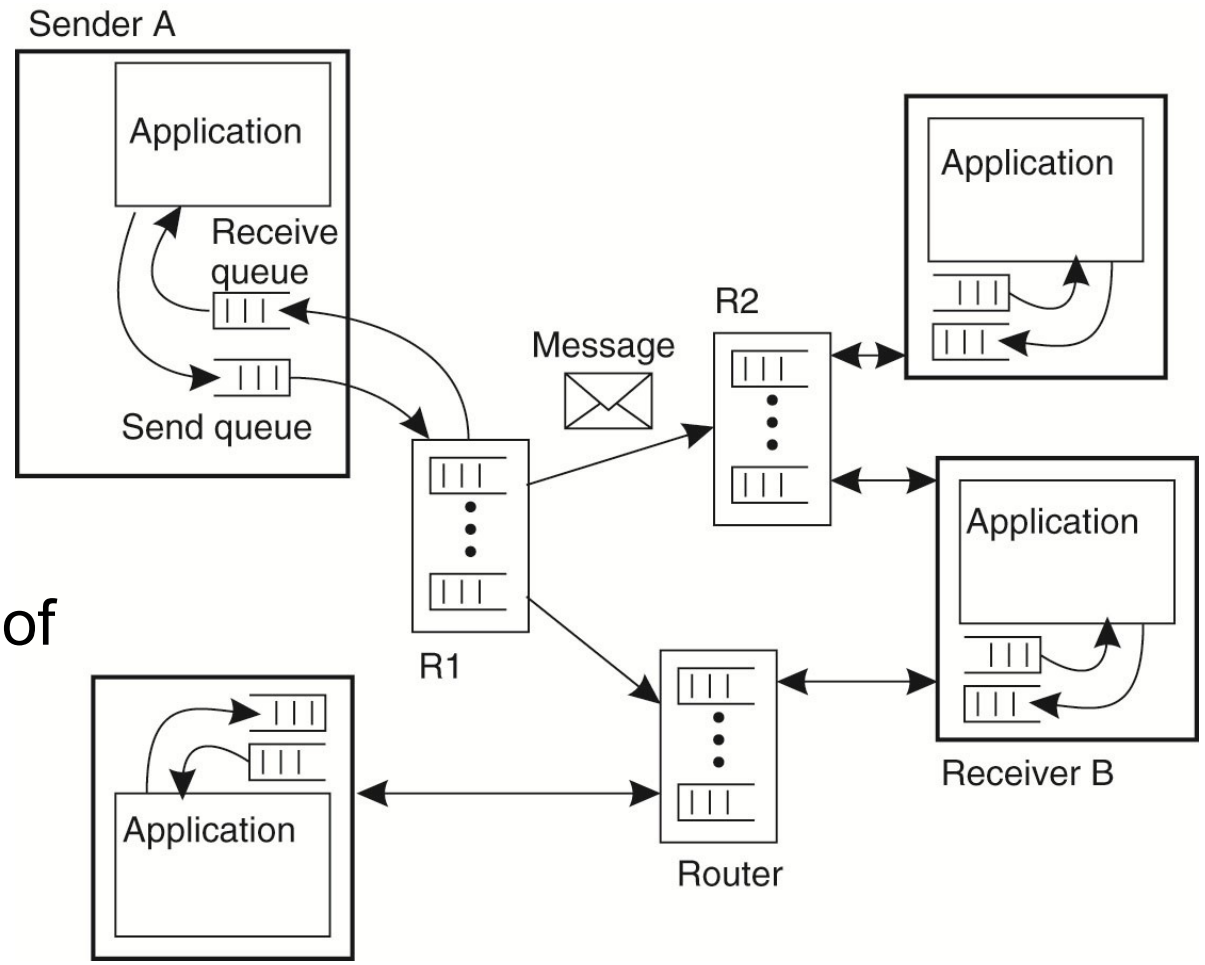


Figure 4-20. The general organization of a message-queuing system with routers.

Message Brokers

An important application area of message-queuing systems **is integrating existing and new applications into a single, coherent distributed information system**.

Integration requires that applications can understand the messages they receive. In practice, this requires the sender to have its outgoing messages in the same format as that of the receiver.

The problem with this approach is that each time an application is added to the system that requires a separate message format, each potential receiver will have to be adjusted in order to produce that format.

An alternative is to agree on a common message format, as is done with traditional network protocols. Unfortunately, this approach will generally not work for message-queuing systems.

Message Brokers

general approach is to learn to live with different formats, and try to provide the means to make conversions as simple as possible. In message-queuing systems, conversions are handled by special nodes in a queuing network, known as message brokers. 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. Note that to a message-queuing system, a message broker is just another application as shown in Fig. 4-21.

In other words, a message broker is generally not considered to be an integral part of the queuing system. A message broker can be as simple as a reformatter for messages. For example, assume an incoming message contains a table from a database,

Message Brokers

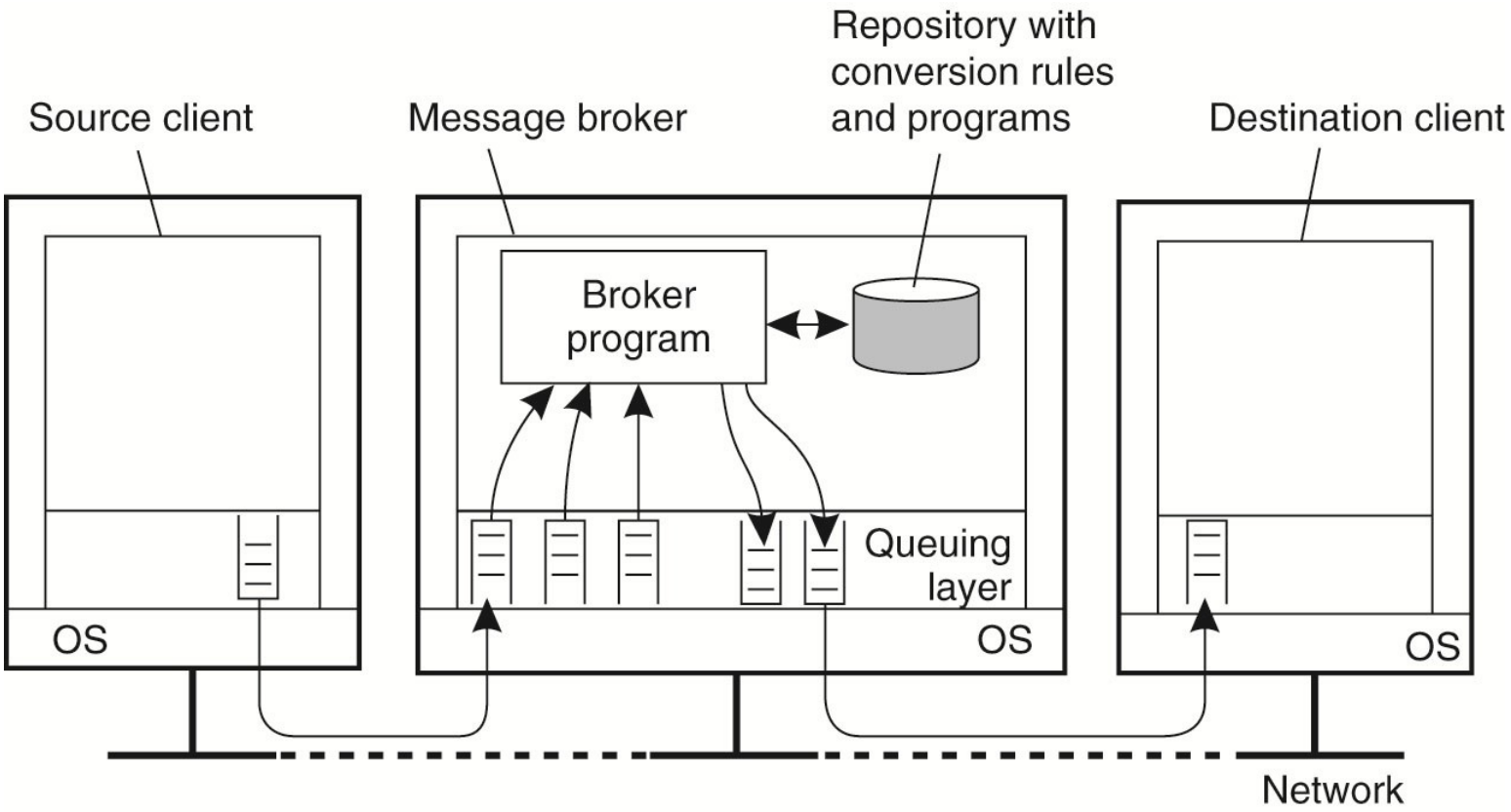


Figure 4-21. The general organization of a message broker in a message-queuing system.

Message Brokers

In a more advanced setting, **a message broker may act as an application-level gateway**, such as one that handles the conversion between two different database applications. In such cases, frequently it cannot be guaranteed that all information contained in the incoming message can actually be transformed into something appropriate for the outgoing message.

However, more common is the use of a message broker for advanced **enterprise application integration (EAI)**. In this case, rather than (only) converting messages, a broker is responsible for **matching applications** based on the messages that are being exchanged. In such a model, called **publish/subscribe**, applications send messages in the form of ***publishing***. *In particular,* they may publish a message on topic X, which is then sent to the broker. Applications that have stated their interest in messages on topic X, that is, who have ***subscribed*** *to those messages, will then receive these messages from the* broker. More advanced forms of mediation in Chap. 13.

IBM's WebSphere Message-Queuing System

- All queues are managed by queue managers.
- A queue manager is responsible for removing messages from its send queues, and forwarding those to other queue managers.
- A message has a maximum default size of 4 MB, but this can be increased up to 100 MB.
- A queue is normally restricted to 2 GB of data, but depending on the underlying operating system, this maximum can be easily set higher.
- A message channel is a unidirectional, reliable connection between a sending and a receiving queue manager, through which queued messages are transported. (For example, an Internet-based message channel using TCP).
- Each of the two ends of a message channel is managed by a message channel agent (MCA).
- A **sending MCA** checks send queues for a message, wrapping it into a transport-level packet, and sending it along the connection to its associated **receiving MCA**. A receiving MCA listens for an incoming packet, unwrapping it, and subsequently stores the unwrapped message into the appropriate queue.

IBM's WebSphere Message-Queuing System

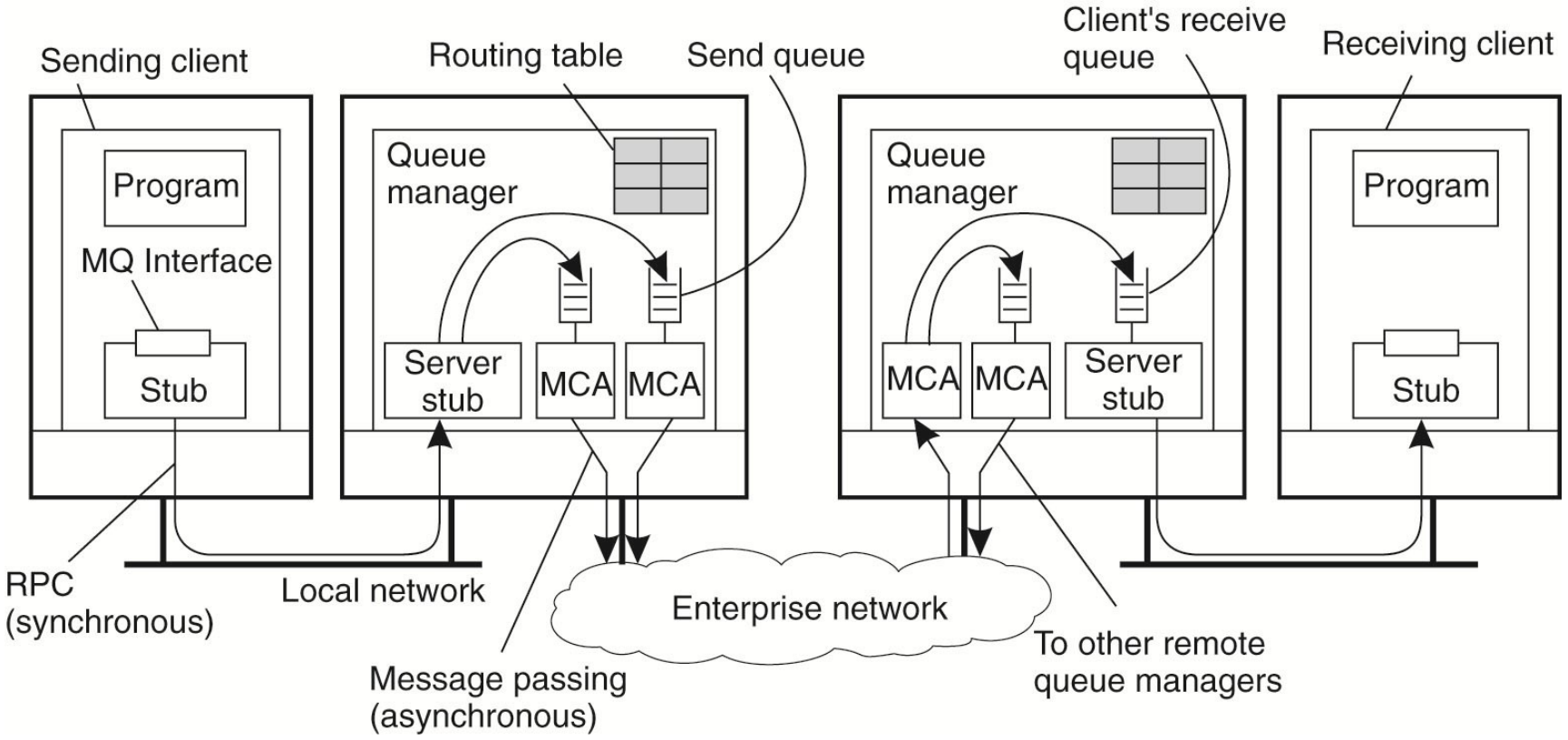


Figure 4-22. General organization of IBM's message-queuing system.

Channels

An important component of MQ is formed by the **message channels**. Each message channel has exactly one associated send queue from which it fetches the messages it should transfer to the other end.

To transfer along the channel both its sending and receiving MCA should be up and running. Apart from starting both MCAs manually, there are several alternative ways to **start a channel**:

- One alternative is to have an application directly start its end of a channel by activating the sending or receiving MCA. However, from a transparency point of view, this is not a very attractive alternative.
- A better approach to start a sending MCA is to configure the channel's send queue to set off a trigger when a message is first put into the queue. That trigger is associated with a handler to start the sending MCA so that it can remove messages from the send queue.
- Another alternative is to start an MCA over the network. In particular, if one side of a channel is already active, it can send a control message requesting that the other MCA to be started. Such a control message is sent to a daemon listening to a well-known address on the same machine as where the other MCA is to be started.

Note: **Channels are stopped** automatically after a specified time has expired during which no more messages were dropped into the send queue.

Channels

Each MCA has a set of associated attributes that determine the overall behavior of a channel. Some of the attributes are listed in Fig. 4-23. Attribute values of the sending and receiving MCA should be compatible and perhaps negotiated first before a channel can be set up. For example, both MCAs should obviously support the same transport protocol. An example of a nonnegotiable attribute is whether or not messages are to be delivered in the same order as they are put into the send queue.

Attribute	Description
Transport type	Determines the transport protocol to be used
FIFO delivery	Indicates that messages are to be delivered in the order they are sent
Message length	Maximum length of a single message
Setup retry count	Specifies maximum number of retries to start up the remote MCA
Delivery retries	Maximum times MCA will try to put received message into queue

Figure 4-23. Some attributes associated with message channel agents.

Message Transfer

To transfer a message from one queue manager to another (possibly remote) queue manager:

- it is necessary that each message carries its destination address, for which a transmission header is used.
- An address in MQ consists of two parts. The first part consists of the name of the queue manager to which the message is to be delivered. The second part is the name of the destination queue resorting under that manager to which the message is to be appended.
- it is also necessary to specify the route that a message should follow. Route specification is done by providing the name of the local send queue to which a message is to be appended.
- it is not necessary to provide the full route in a message. Recall that each message channel has exactly one send queue. By telling to which send queue a message is to be appended, we effectively specify to which queue manager a message is to be forwarded.

Message transfer

In most cases, routes are explicitly stored inside a queue manager in a routing table.

An entry in a routing table is a pair (destQM, sendQ), where destQM is the name of the destination queue manager, and sendQ is the name of the local send queue to which a message for that queue manager should be appended. (A routing table entry is called an alias in MQ.)

It is possible that a message needs to be transferred across multiple queue managers before reaching its destination. Whenever such an intermediate queue manager receives the message, it simply extracts the name of the destination queue manager from the message header, and does a routing-table look-up to find the local send queue to which the message should be appended.

Message Transfer (1)

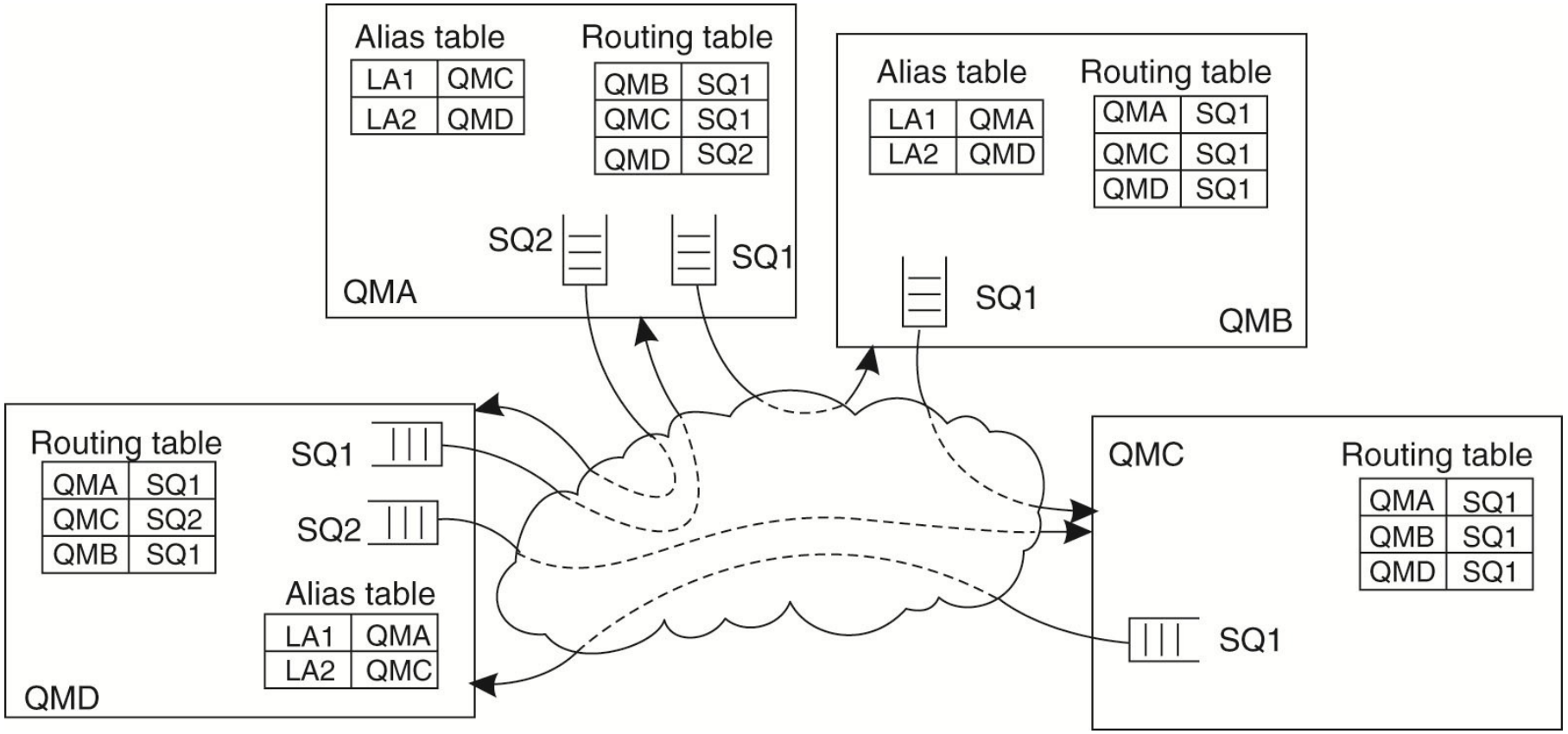


Figure 4-24. The general organization of an MQ queuing network using routing tables and aliases.

Message Transfer (2)

Primitive	Description
MQopen	Open a (possibly remote) queue
MQclose	Close a queue
MQput	Put a message into an opened queue
MQget	Get a message from a (local) queue

Figure 4-25. Primitives available in the message-queuing interface.

Data Stream

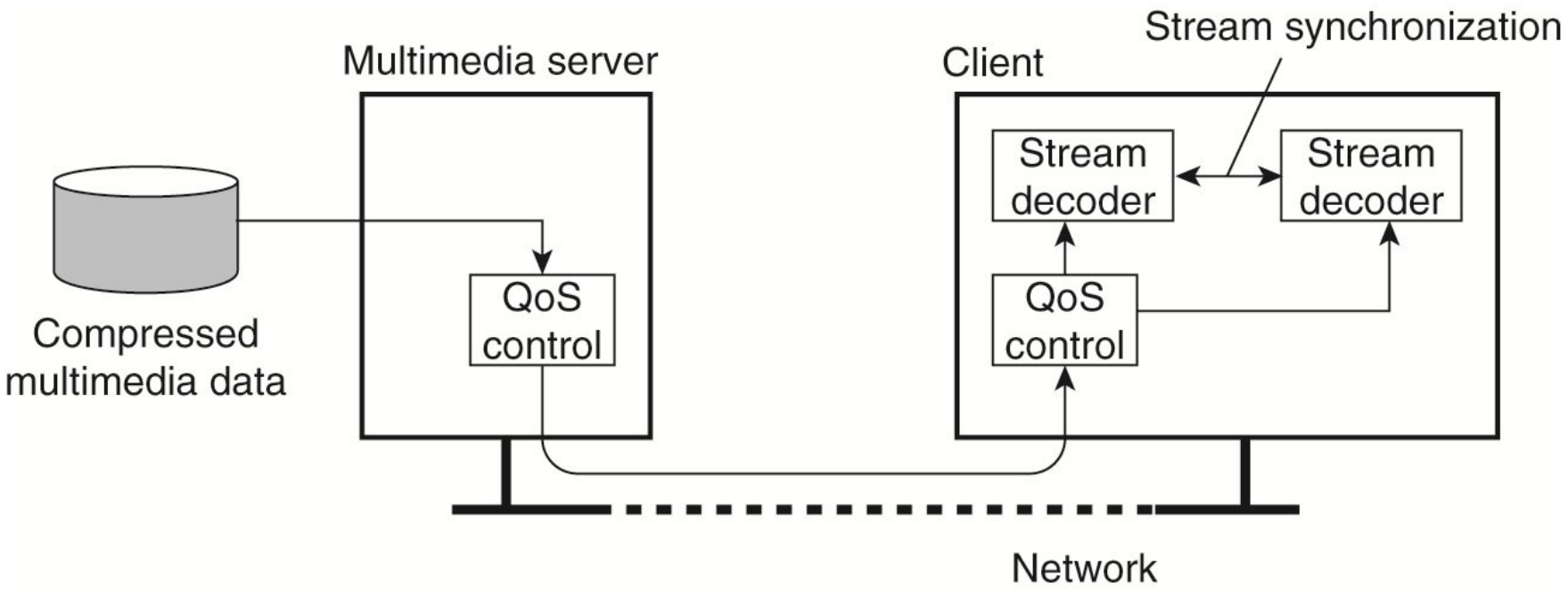


Figure 4-26. A general architecture for streaming stored multimedia data over a network.

Streams and Quality of Service

Properties for Quality of Service:

- The required bit rate at which data should be transported.
- The maximum delay until a session has been set up
- The maximum end-to-end delay .
- The maximum delay variance, or jitter.
- The maximum round-trip delay.

Enforcing QoS (1)

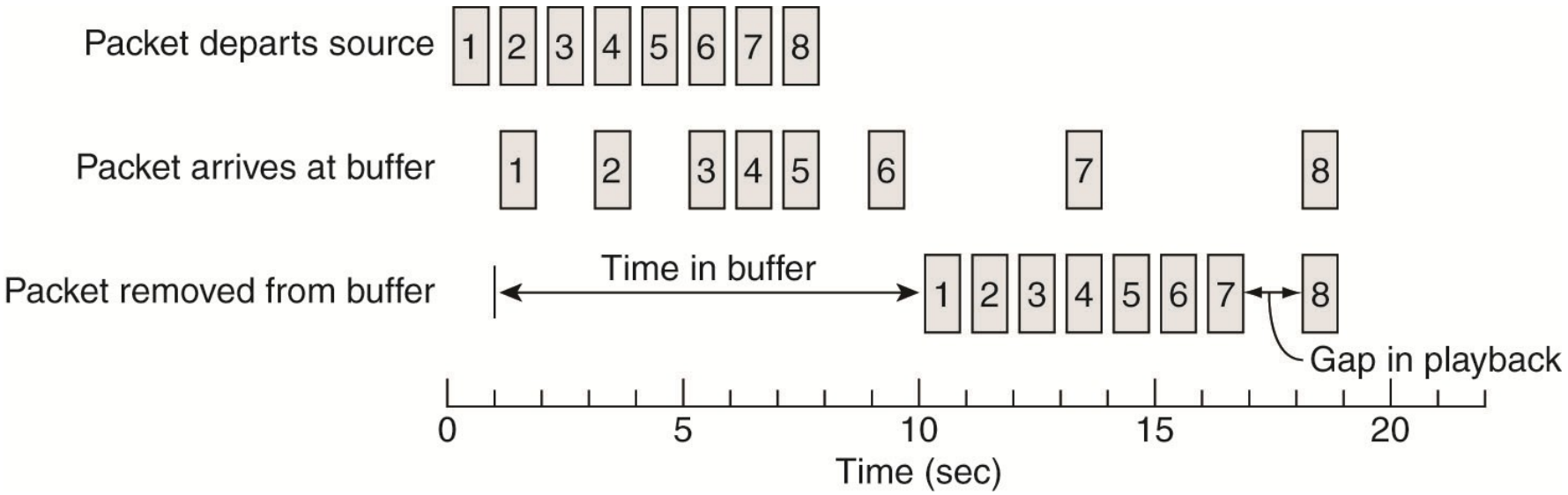


Figure 4-27. Using a buffer to reduce jitter.

Enforcing QoS (2)

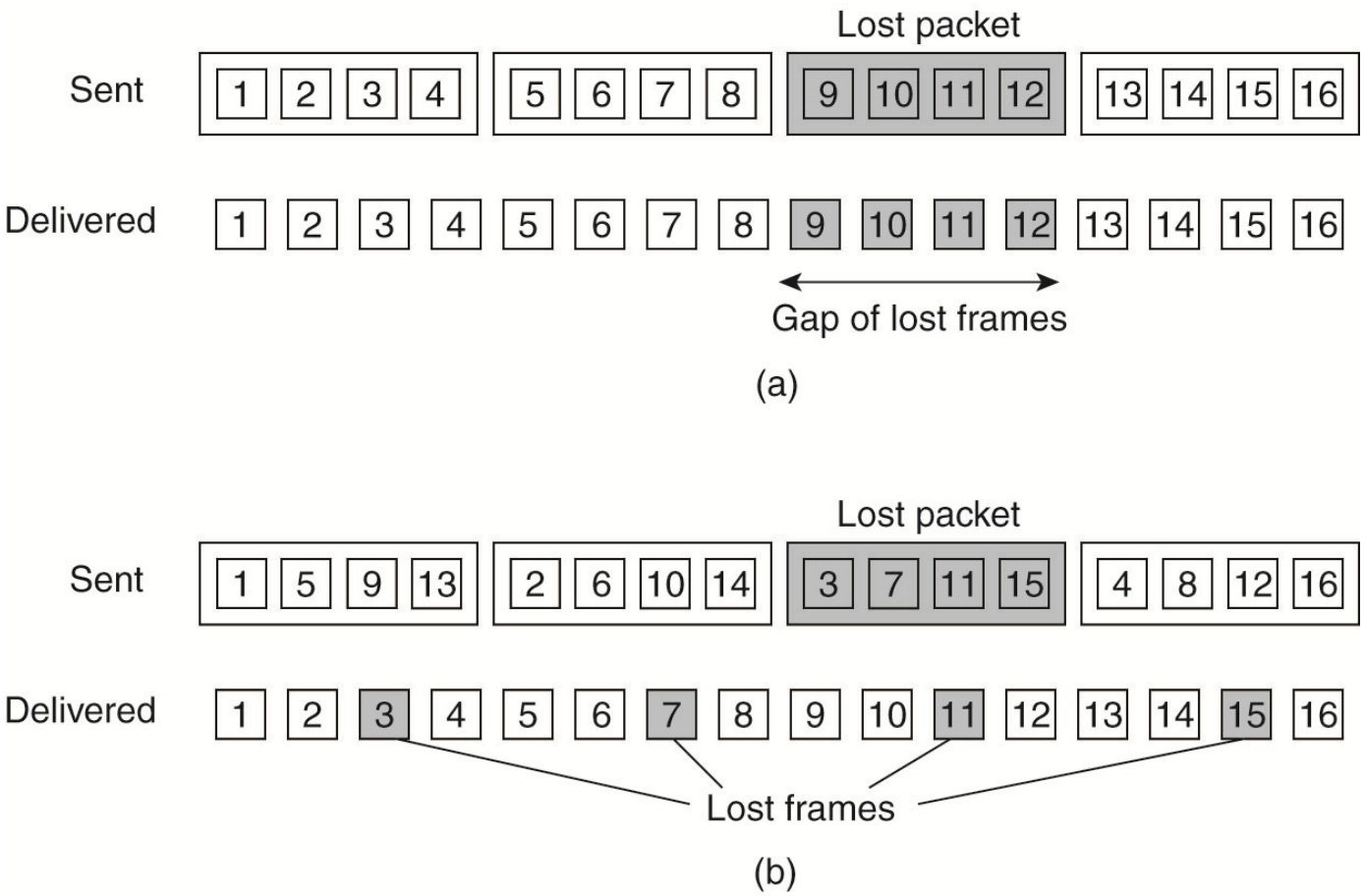


Figure 4-28. The effect of packet loss in (a) non interleaved transmission and (b) interleaved transmission.

Synchronization Mechanisms (1)

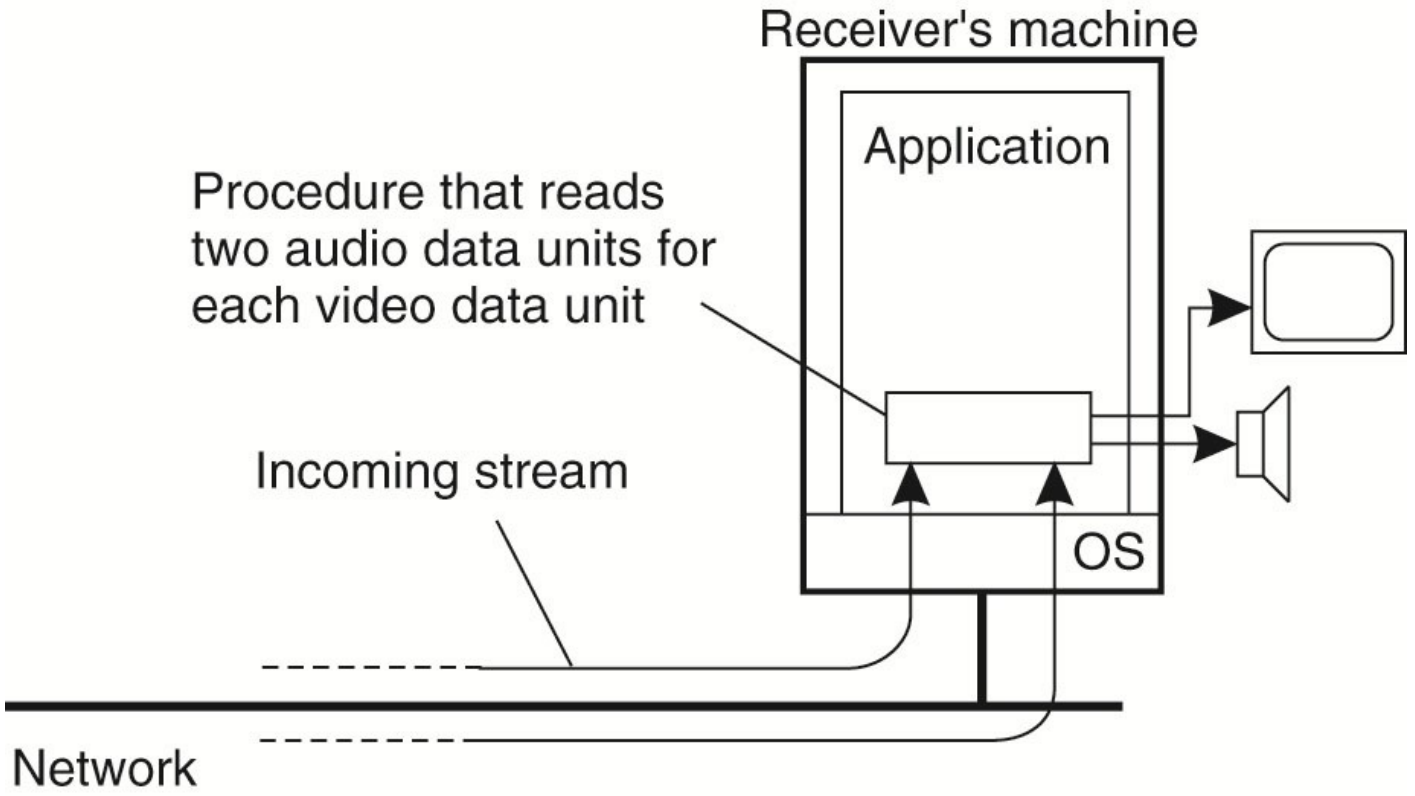


Figure 4-29. The principle of explicit synchronization on the level data units.

Synchronization Mechanisms (2)

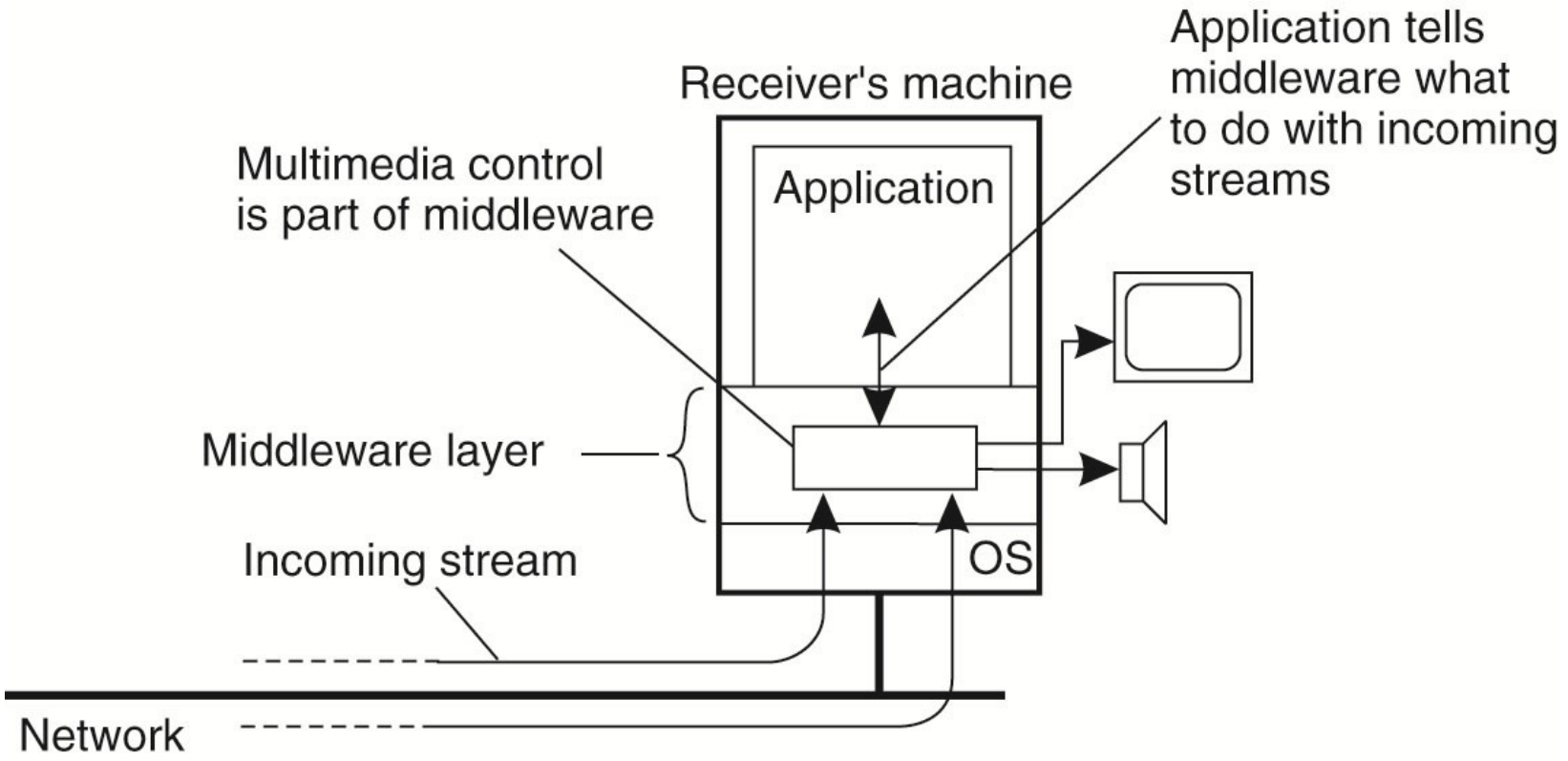


Figure 4-30. The principle of synchronization as supported by high-level interfaces.

Overlay Construction

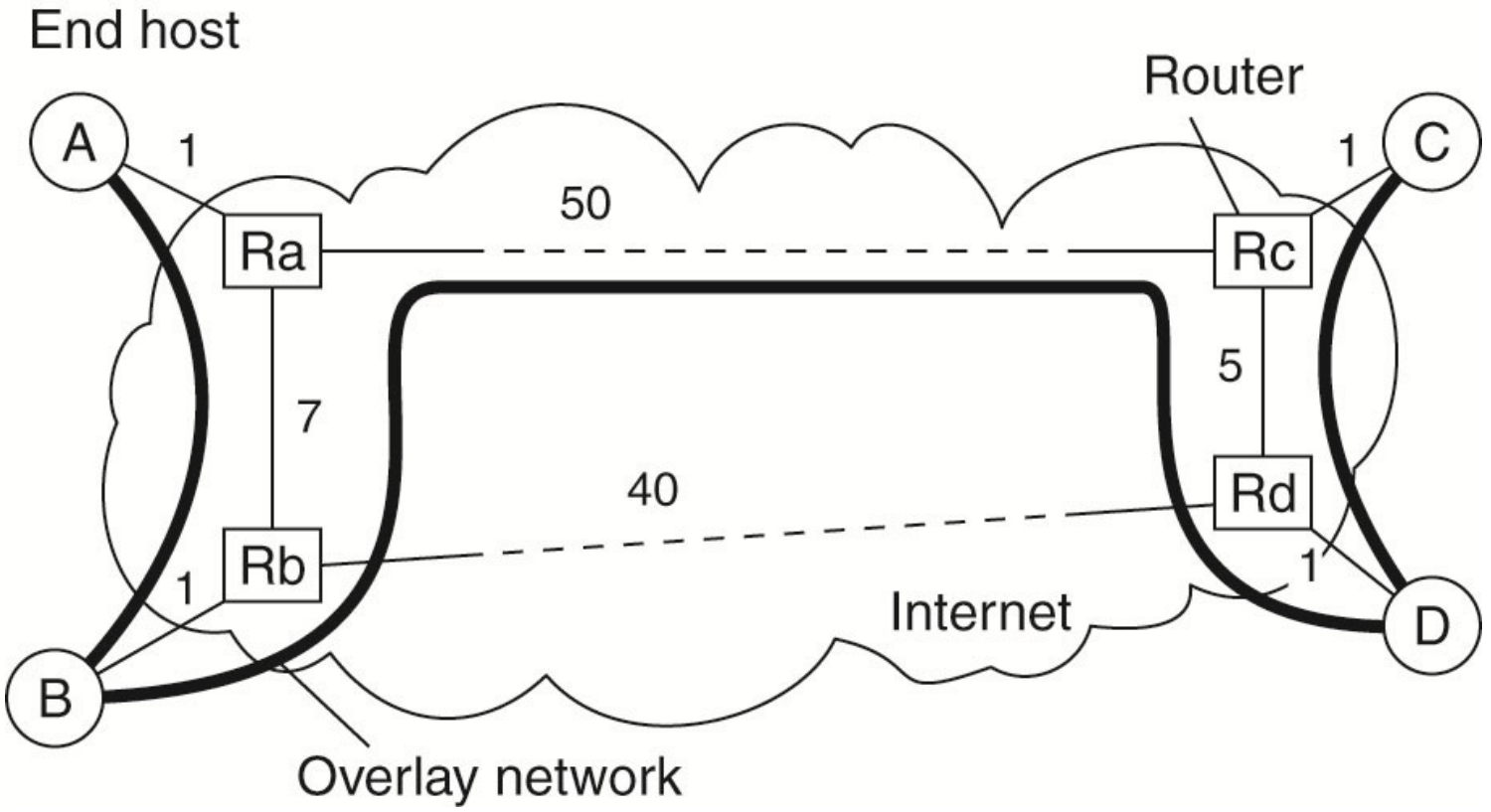


Figure 4-31. The relation between links in an overlay and actual network-level routes.

Information Dissemination Models (1)

- Anti-entropy propagation model
  - Node P picks another node Q at random
  - Subsequently exchanges updates with Q
- Approaches to exchanging updates
  - P only pushes its own updates to Q
  - P only pulls in new updates from Q
  - P and Q send updates to each other

Information Dissemination Models (2)

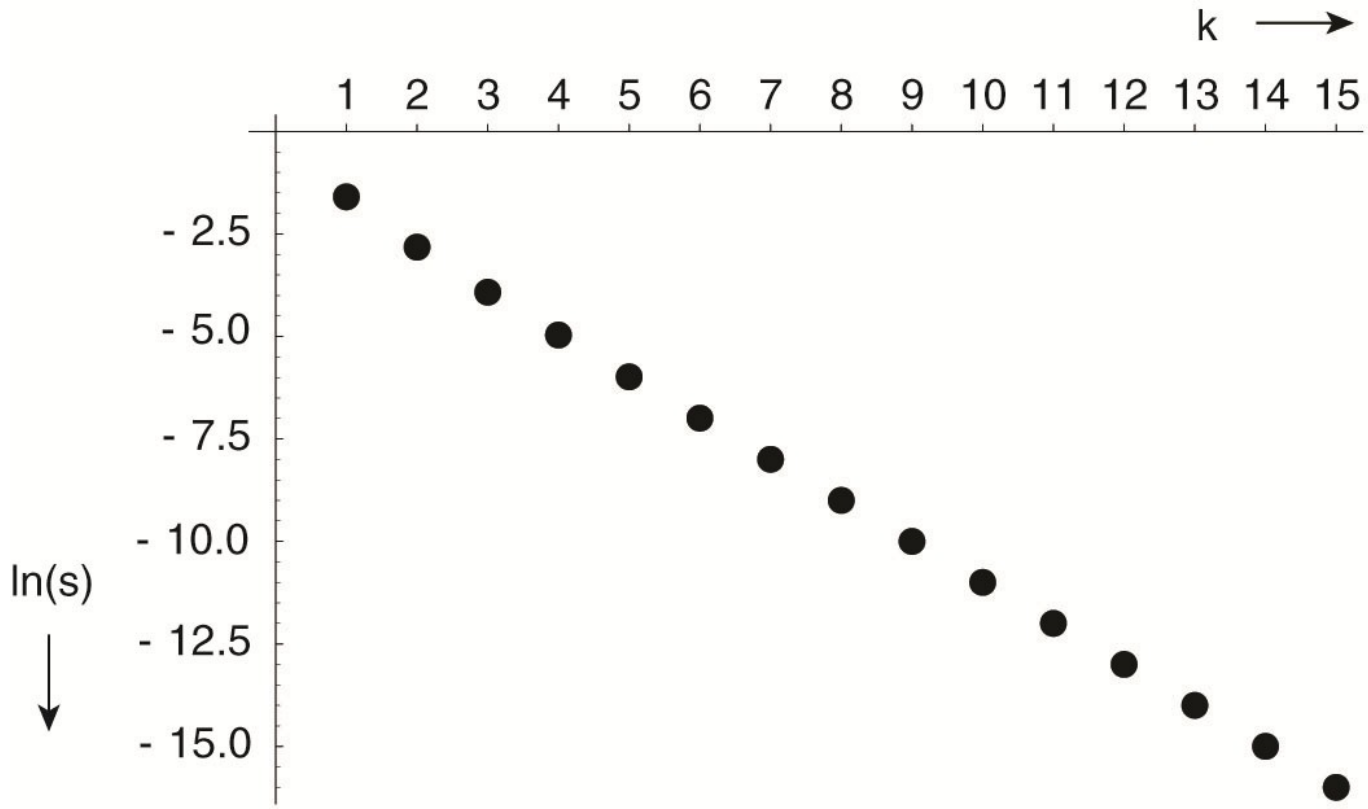


Figure 4-32. The relation between the fraction s of update-ignorant nodes and the parameter k in pure gossiping. The graph displays ln(s) as a function of k.