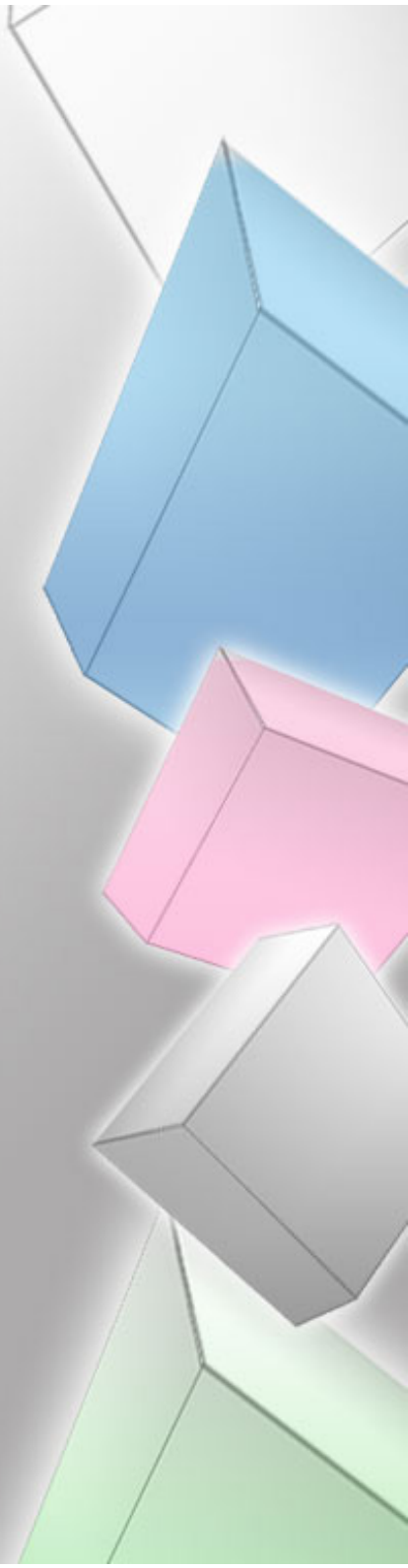
Advanced Topics in Operating Systems

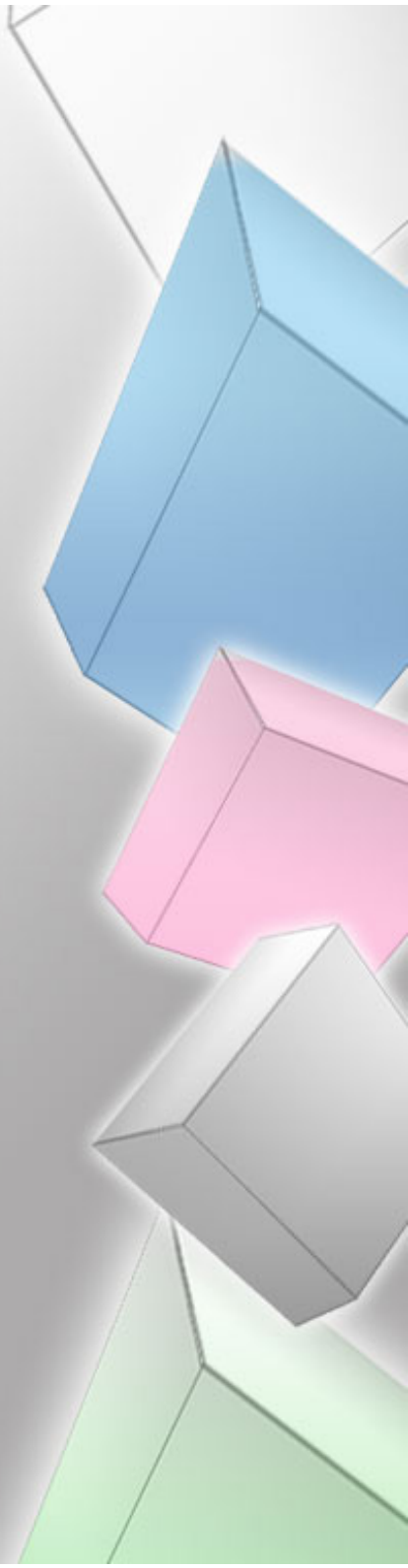
MSc in Computer Science
UNYT-UoG

Dr. Marenglen Biba
18-19-20 December 2009



Lesson 4

01: Introduction
02: Architectures
03: Processes
04: Communication
05: Naming
06: Synchronization
07: Consistency & Replication
08: Fault Tolerance
09: Security
10: Distributed Object-Based Systems
11: Distributed File Systems
12: Distributed Web-Based Systems
13: Distributed Coordination-Based Systems

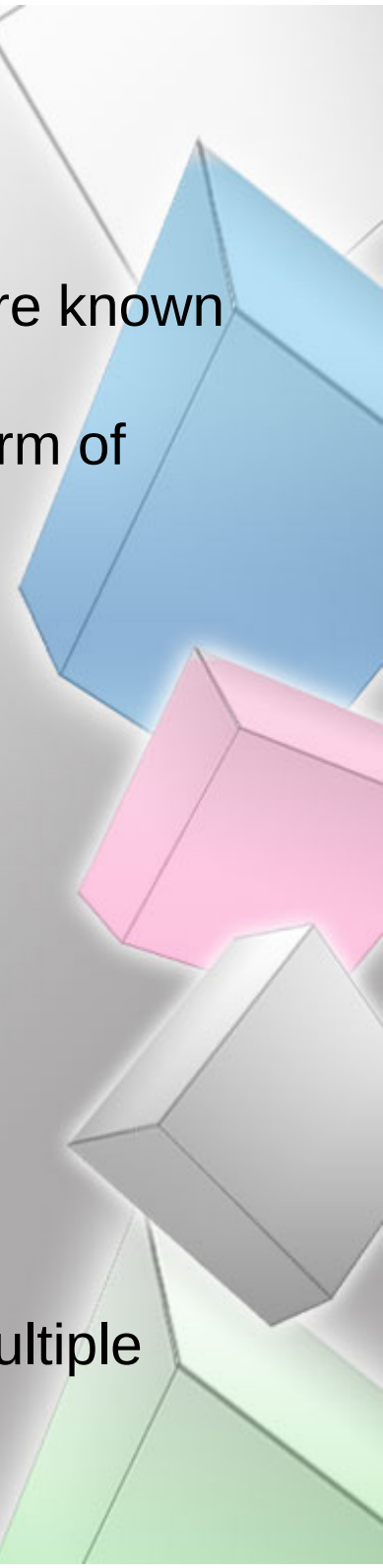


Interprocess communication

- Interprocess communication is at the heart of all distributed systems.
- Communication in distributed systems is always based on low-level message passing as offered by the underlying network.
- Expressing communication through message passing is harder than using primitives based on shared memory, as available for nondistributed platforms.
- Modem distributed systems often consist of thousands or even millions of processes scattered across a network with unreliable communication such as the Internet.
- Unless the primitive communication facilities of computer networks are replaced by something else, development of large-scale distributed applications is extremely difficult.

Protocols and Models

- The rules that communicating processes must adhere to are known as protocols,
  - We concentrate on structuring those protocols in the form of layers.
    - Low-level layers
    - Transport layer
    - Application layer
    - Middleware layer
- Widely-used models for communication:
  - Remote Procedure Call (RPC)
  - Message-Oriented Middleware (MOM)
  - Data streaming
- We also discuss the general problem of sending data to multiple receivers, called multicasting.



Absence of shared memory

- Due to the absence of shared memory, all communication in distributed systems is based on sending and receiving (low level) messages.
- When process *A* wants to communicate with process *B,* it first builds a message in its own address space.
- Then it executes a system call that causes the operating system to send the message over the network to *B.*
- *A* and *B* have to agree on the meaning of the bits being sent!!!
  - If *A* sends a brilliant new novel written in French and encoded in IBM's EBCDIC character code, and *B* expects the inventory of a supermarket written in English and encoded in ASCII, communication will be less than optimal. ☺

How do they agree?

- Processes must agree. How?

- Many different agreements are needed.
  - How many volts should be used to signal a 0--bit, and how many volts for a 1-bit?
  - How does the receiver know which is the last bit of the message?
  - How can it detect if a message has been damaged or lost, and what should it do if it finds out?
  - How long are numbers, strings, and other data items, and how are they represented?

- In short, agreements are needed at a variety of levels, varying from the low-level details of bit transmission to the high-level details of how information is to be expressed.

The ISO/OSI reference model

- To make it easier to deal with the numerous levels and issues involved in communication, the International Standards Organization (ISO) developed a reference model that clearly identifies the various levels involved, gives them standard names, and points out which level should do which job.

- This model is called the Open Systems Interconnection Reference Model usually abbreviated as ISO OSI or sometimes just the OSI model.

- It should be emphasized that the protocols that were developed as part of the OSI model were never widely used and are essentially dead now.

- However, the underlying model itself has proved to be quite useful for understanding computer networks.

Open Systems

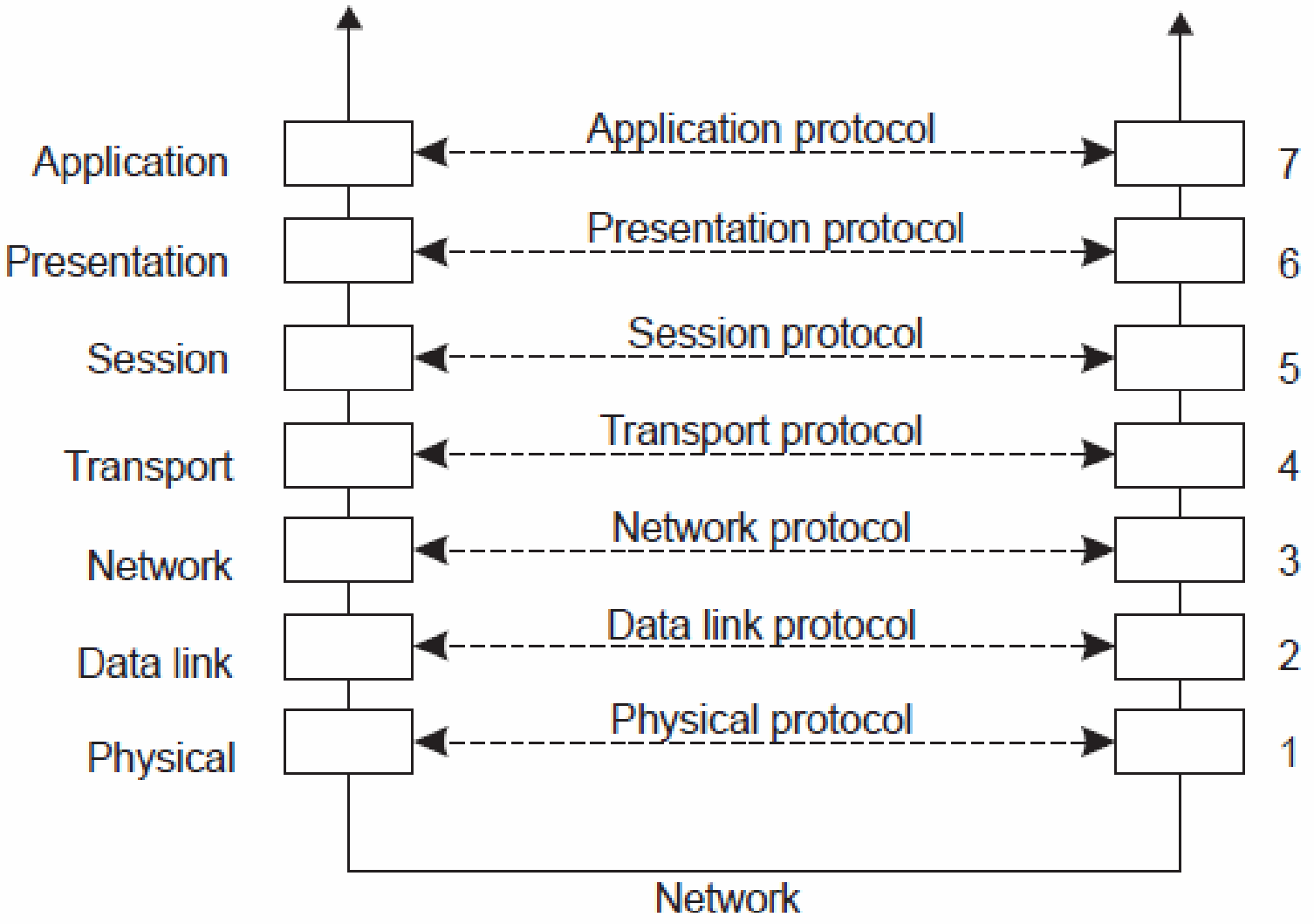
- The OSI model is designed to allow open systems to communicate.
- An open system is one that is prepared to communicate with any other open system by using standard rules that govern the format, contents, and meaning of the messages sent and received.
  - These rules are formalized in what are called protocols.
- To allow a group of computers to communicate over a network, they must all agree on the protocols to be used.

Connection oriented and connectionless protocols

- With connection oriented protocols, before exchanging data the sender and receiver:
  - First explicitly establish a connection
  - Possibly negotiate the protocol they will use.
  - When they are done, they must release (terminate) the connection.
- The telephone is a connection-oriented communication system.

- With connectionless protocols, no setup in advance is needed.
  - The sender just transmits the first message when it is ready.
  - Dropping a letter in a mailbox is an example of connectionless communication.
- With computers, both connection-oriented and connectionless communication are common.

The OSI model



Building messages

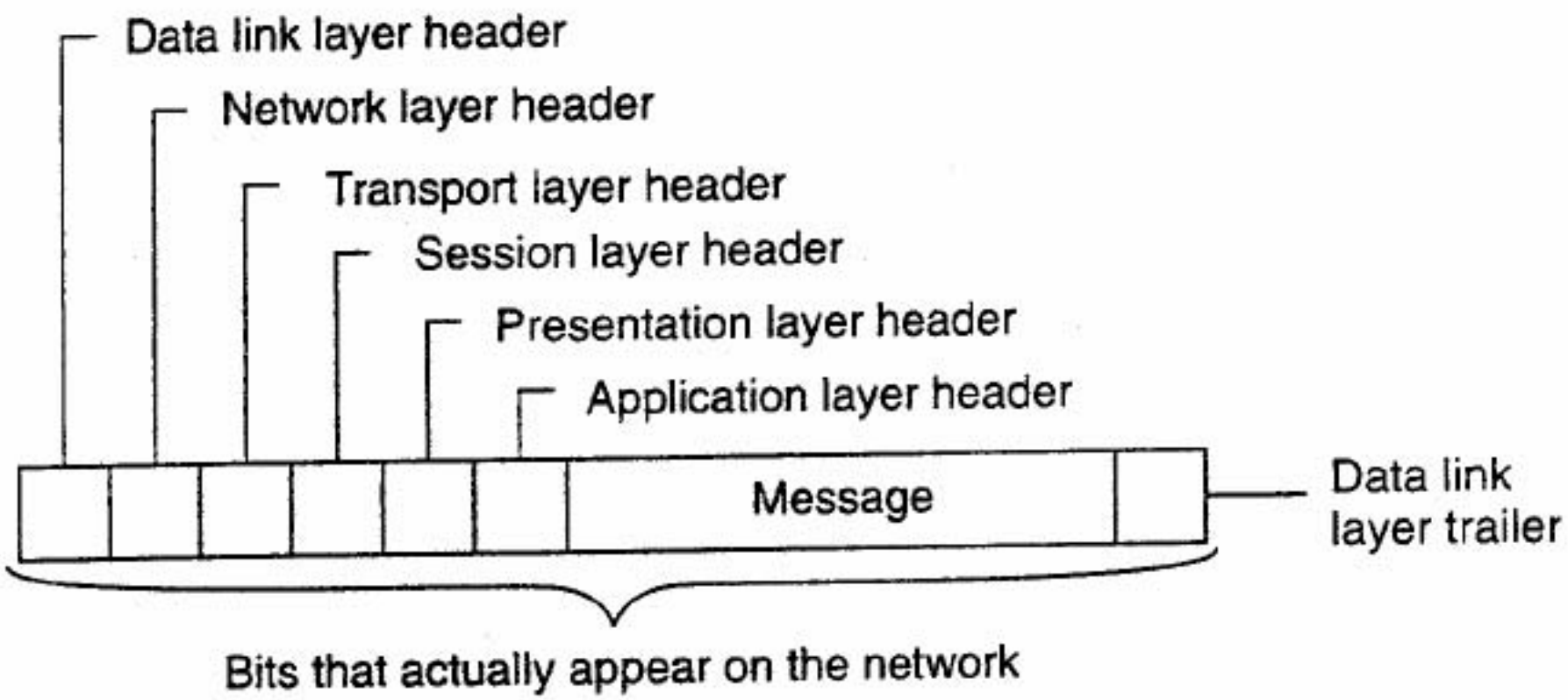


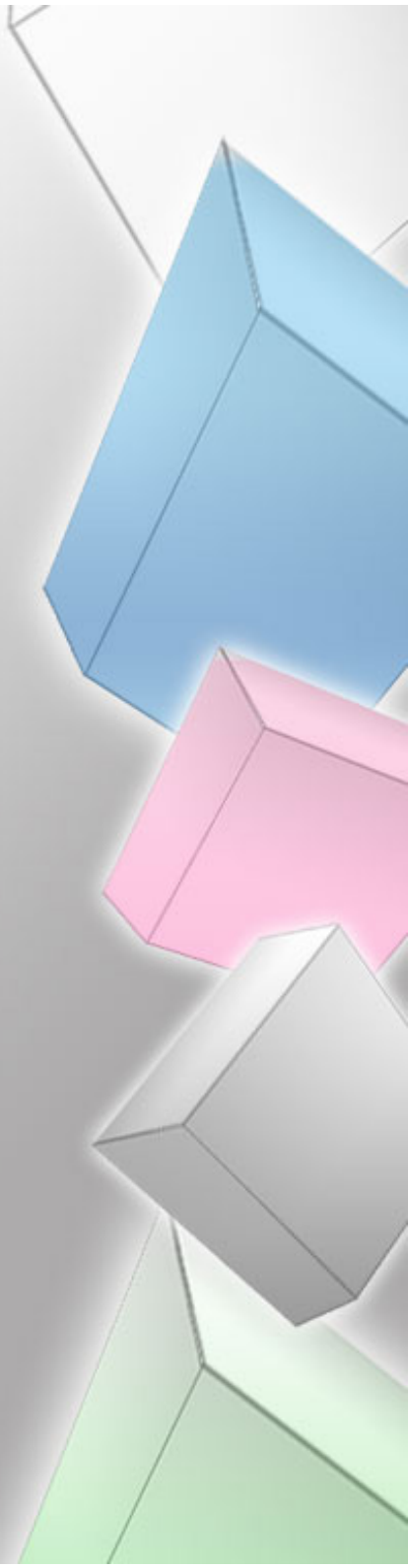
Figure 4-2. A typical message as it appears on the network.

Protocol Stack

- The collection of protocols used in a particular system is called a protocol suite or protocol stack.
- It is important to distinguish a *reference model* from its actual *protocols.*
- The OSI protocols were never popular.
- In contrast, protocols developed for the Internet, such as TCP and IP, are mostly used.

Low-level layers

- Physical layer
- Data link layer
- Network layer



Physical layer

- Contains the specification and implementation of bits, and their transmission between sender and receiver.

- It is concerned with transmitting the 0s and 1s.
  - How many volts to use for 0 and 1, how many bits per second can be sent, and whether transmission can take place in both directions simultaneously.
  - In addition, the size and shape of the network connector (plug), as well as the number of pins and meaning of each are of concern here.

- It deals with standardizing the electrical, mechanical, and signaling interfaces so that when one machine sends a 0 bit it is actually received as a 0 bit and not a 1 bit.

- Many physical layer standards have been developed (for different media), for example, the RS-232-C standard for serial communication lines.

Data link layer

- Prescribes the transmission of a series of bits into a frame to allow for error and flow control.
  - It puts a special bit pattern on the start and end of each frame to mark them, as well as computing a checksum by adding up all the bytes in the frame in a certain way.
  - The data link layer appends the checksum to the frame.

- When the frame arrives, the receiver recomputes the checksum from the data and compares the result to the checksum following the frame.

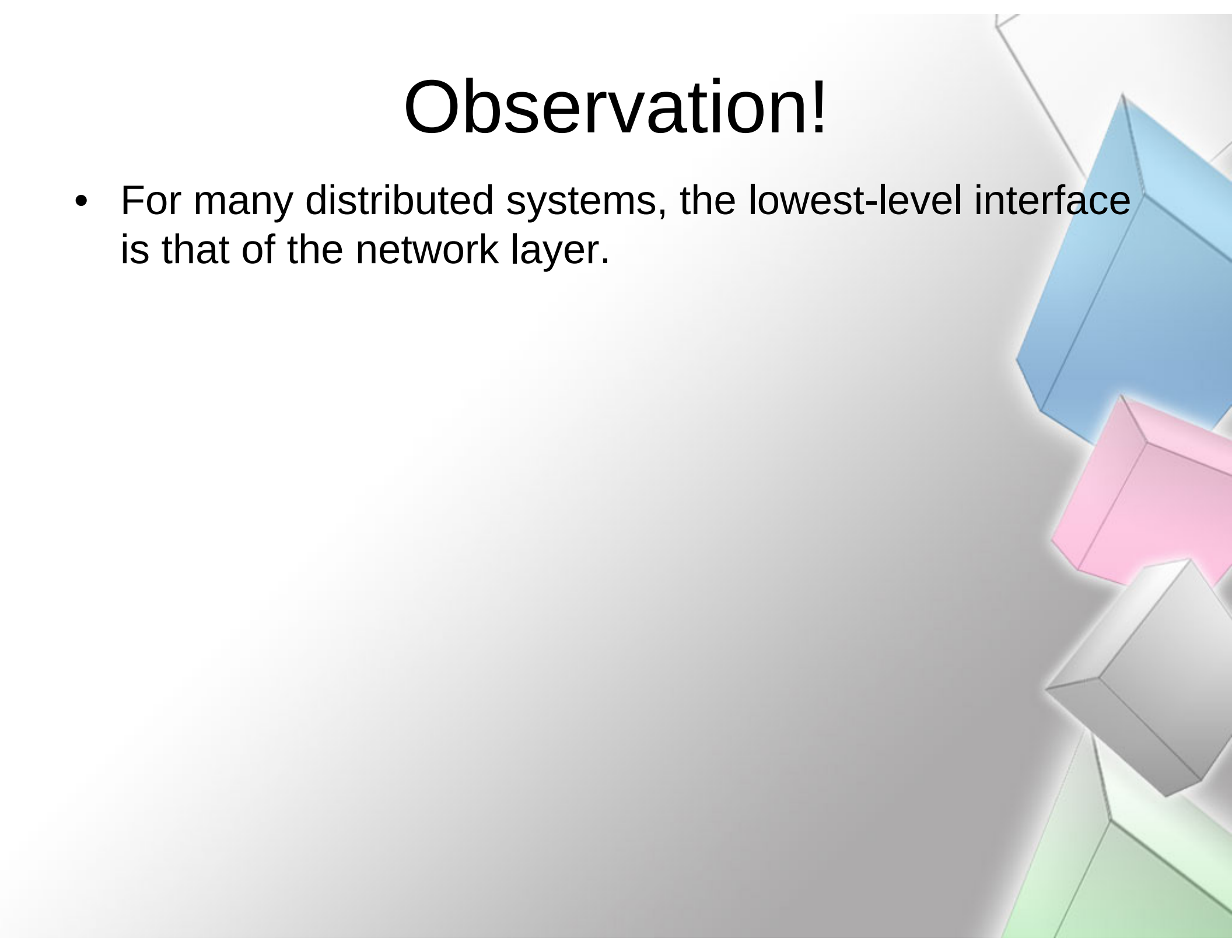
- If the two agree, the frame is considered correct and is accepted.

Network layer

- Describes how packets in a network of computers are to be routed.
- At present, the most widely used network protocol is the connectionless IP (Internet Protocol), which is part of the Internet protocol suite.
- An IP packet (the technical term for a message in the network layer) can be sent without any setup.
- Each IP packet is routed to its destination independent of all others.

Observation!

- For many distributed systems, the lowest-level interface is that of the network layer.



Transport Layer

- The transport layer provides the actual communication facilities for most distributed systems.

- The transport layer forms the last part of what could be called a basic network protocol stack, in the sense that it implements all those services that are not provided at the interface of the network layer, but which are reasonably needed to build network applications.

- In other words, the transport layer turns the underlying network into something that an application developer can use.

Transport Protocol

- Packets can be lost on the way from the sender to the receiver.
  - Although some applications can handle their own error recovery, others prefer a reliable connection.

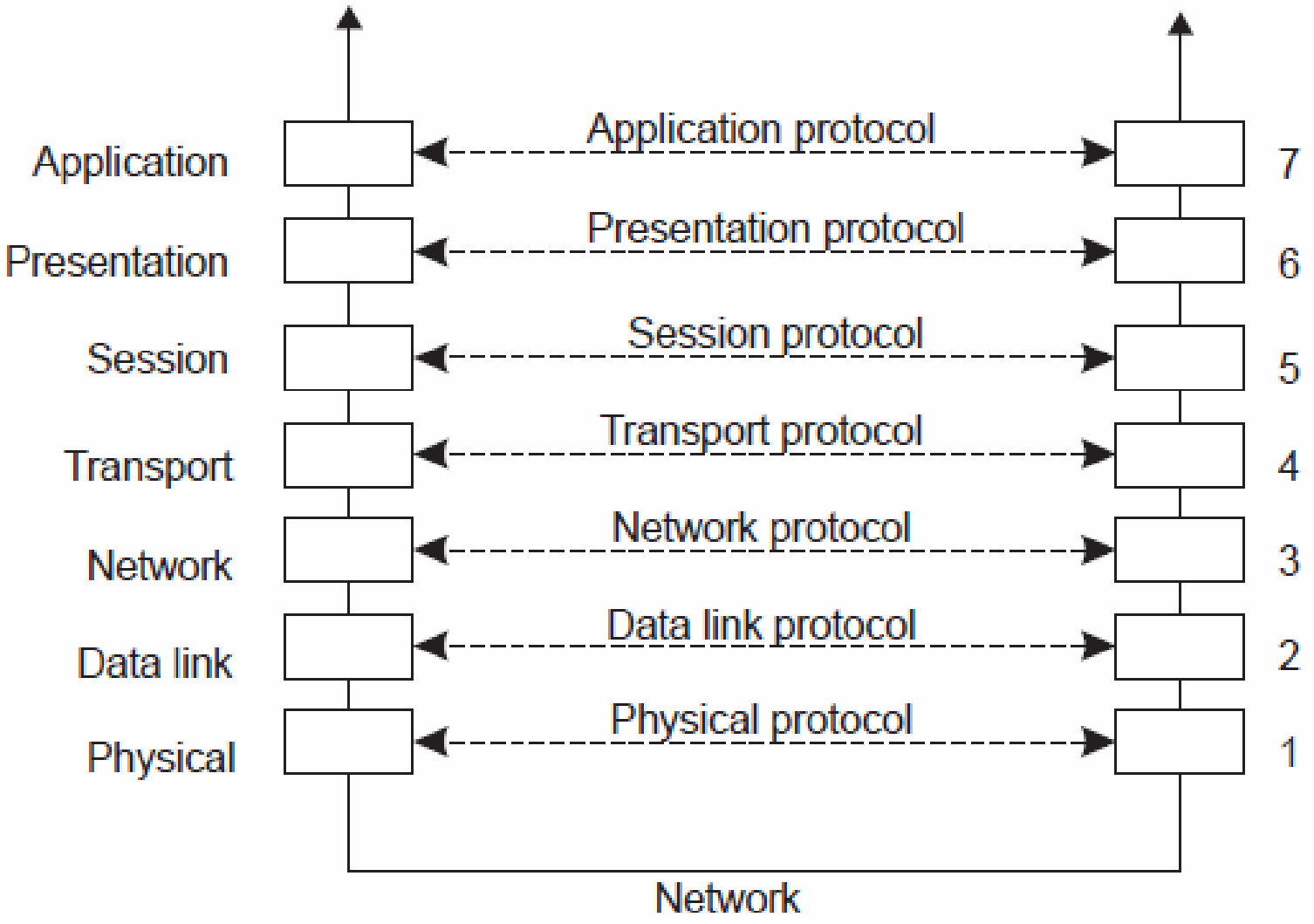
- The job of the transport layer is to provide a reliable connection.
  - The idea is that the application layer should be able to deliver a message to the transport layer with the expectation that it will be delivered without loss.
  - Upon receiving a message from the application layer, the transport layer breaks it into pieces small enough for transmission, assigns each one a sequence number, and then sends them all.

- The discussion in the transport layer header concerns:
  - which packets have been sent,
  - which have been received,
  - how many more the receiver has room to accept,
  - which should be retransmitted,
  - and similar topics.

TCP/IP

- The Internet transport protocol is called TCP (Transmission Control Protocol) and is described in detail in Comer (2006).
- The combination TCPI/IP is now used as a de facto standard for network communication.
- The Internet protocol suite also supports a connectionless transport protocol called UDP (Universal Datagram Protocol), which is essentially just IP with some minor additions.
- User programs that do not need a connection-oriented protocol normally use UDP.

The OSI model



Higher Level Protocols

- Above the transport layer, OSI distinguished three additional layers.
  - In practice, only the application layer is ever used.
- In fact, in the Internet protocol suite, everything above the transport layer is grouped together.
- In the face of middleware systems, we shall see in this section that neither the OSI nor the Internet approach is really appropriate.

Session Layer

- The session layer is essentially an enhanced version of the transport layer.
  - It provides dialog control, to keep track of which party is currently talking, and it provides synchronization facilities.
  - The latter are useful to allow users to insert checkpoints into long transfers, so that in the event of a crash, it is necessary to go back only to the last checkpoint, rather than all the way back to the beginning.
- In practice, few applications are interested in the session layer and it is rarely supported.
  - Session layer is not even present in the Internet protocol suite.
- However, in the context of developing middleware solutions, the concept of a session and its related protocols has turned out to be quite relevant, notably when defining higher-level communication protocols.

Presentation Layer

- Unlike the lower layers, which are concerned with getting the bits from the sender to the receiver reliably and efficiently, the presentation layer is concerned with the meaning of the bits.
  - Most messages do not consist of random bit strings, but more structured information such as people's names, addresses, amounts of money, and so on.

- In the presentation layer it is possible to define records containing fields like these and then have the sender notify the receiver that a message contains a particular record in a certain format.
  - This makes it easier for machines with different internal representations to communicate with each other.

Problems of OSI

- What is missing in OSI is a clear distinction between applications, application-specific protocols, and general-purpose protocols.
  - Internet File Transfer Protocol (FTP) defines a protocol for transferring files between a client and server machine.
  - The protocol should not be confused with the *ftp* program, which is an end-user application for transferring files and which also (not entirely by coincidence) happens to implement the Internet FTP.

- A typical application-specific protocol is the HyperText Transfer Protocol (HTTP), designed to remotely manage the transfer of Web pages. Implemented by Web browsers and Web servers.
  - However, HTTP is now also used by systems that are not intrinsically tied to the Web. For example, Java's object-invocation mechanism uses HTTP to request the invocation of remote objects that are protected by a firewall (Sun Microsystems, 2004b).

- There are also many general-purpose protocols that are useful to many applications, but which cannot be qualified as transport protocols. In many cases, such protocols fall into the category of middleware protocols

Middleware layer

- Middleware is invented to provide common services and protocols that can be used by many different applications.
- Middleware is an application that logically lives (mostly) in the application layer, but which contains many general-purpose protocols that warrant their own layers, independent of other, more specific applications.
- A distinction can be made between high-level communication protocols and protocols for establishing various middleware services.

Middleware services 1

- There are numerous protocols to support a variety of middleware services.
- For example, there are various ways to establish authentication, that is, provide proof of a claimed identity.
  - Authentication protocols are not closely tied to any specific application, but instead, can be integrated into a middleware system as a general service.
  - Authorization protocols by which authenticated users and processes are granted access only to those resources for which they have authorization.

- Many protocols tend to have a general application-independent nature

Middleware services 2

- Commit Protocols
  - Commit protocols establish that in a group of processes either all processes carry out a particular operation, or that the operation is not carried out at all. This phenomenon is also referred to as atomicity and is widely applied in transactions.
- Distributed locking protocol
  - By which a resource can be protected against simultaneous access by a collection of processes that are distributed across multiple machines.
- These are examples of protocols that can be used to implement a general middleware service, but which, at the same time, are highly independent of any specific application.

Middleware communication services

- Middleware communication protocols support high-level communication services.
- For example, there are protocols that allow a process to call a procedure or invoke an object on a remote machine in a highly transparent way.
- Likewise, there are high-level communication services:
  - for setting and synchronizing streams
  - for transferring real-time data, such as needed for multimedia applications.
- Finally some middleware systems offer reliable multicast services that scale to thousands of receivers spread across a wide area network.

Plugging the Middleware

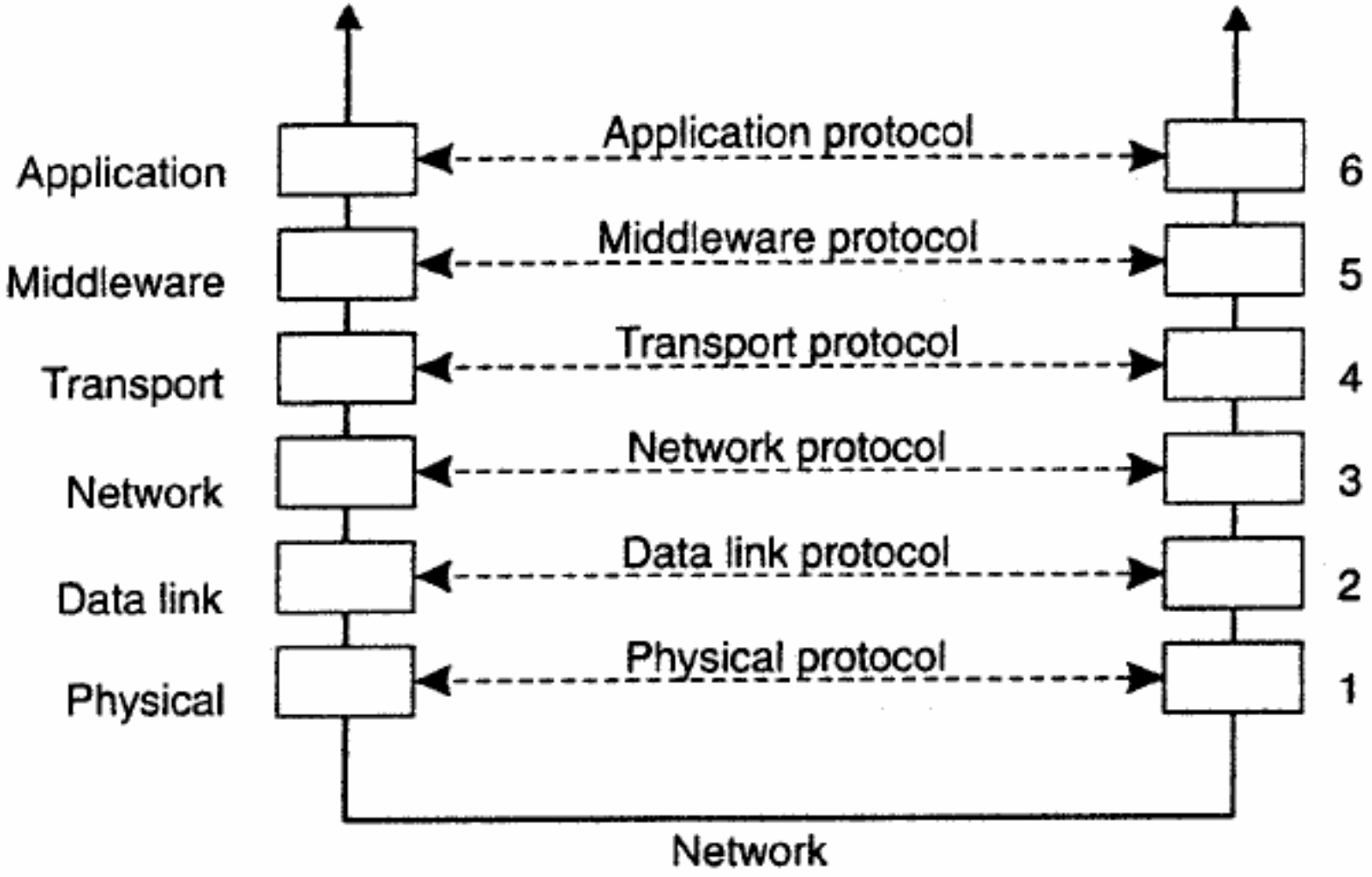


Figure 4-3. An adapted reference model for networked communication.

Types of communication

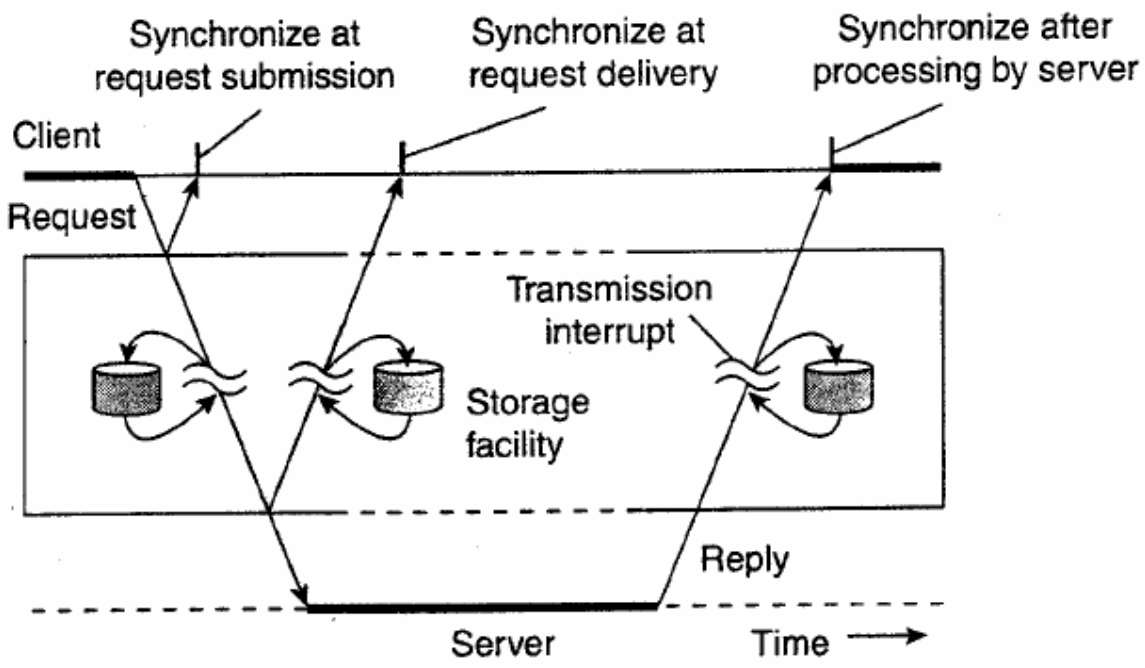


Figure 4-4. Viewing middleware as an intermediate (distributed) service in application-level communication.

Let's view the middleware as an additional service in client/server computing. For example an electronic mail system. In principle, the core of the mail delivery system can be seen as middleware communication service. Each host runs a user agent allowing users to compose, send, and receive e-mail. A sending user agent passes such mail to the mail delivery system, expecting it, in turn, to eventually deliver the mail to the intended recipient. Likewise, the user agent at the receiver's side connects to the mail delivery system to see whether any mail has come in.

Persistent and 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.
  - An electronic mail system is a typical example in which communication is persistent.
  - In this case, the middleware will store the message at one or several of the storage facilities.
  - 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.
  - More precisely, 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.

Asynchronous or synchronous

- In asynchronous communication the 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.

Observations! ☺/☹

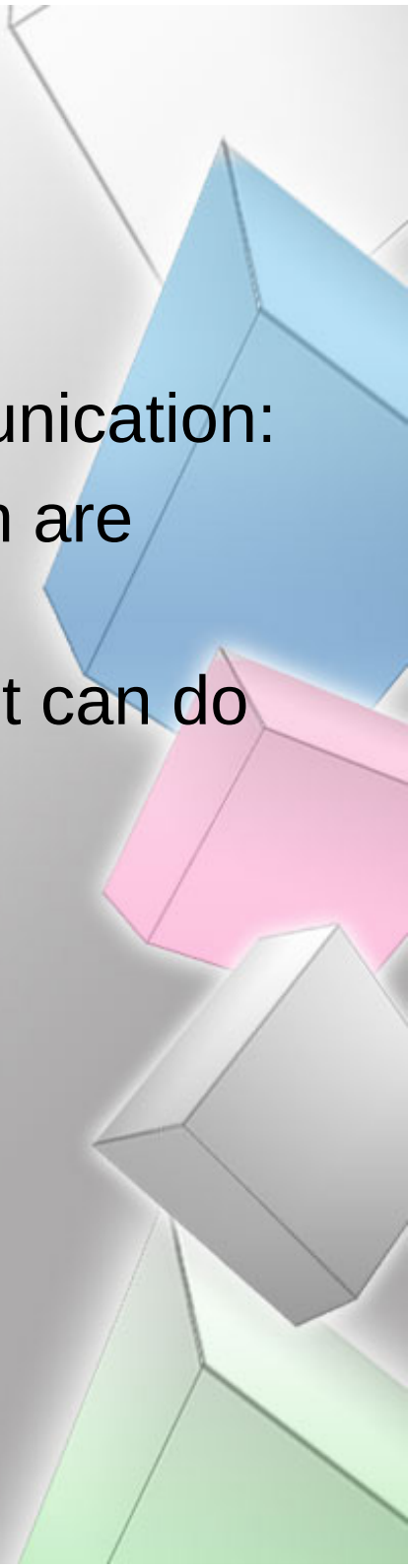
- Client/Server computing is generally based on a model of transient synchronous communication:
  - Client and server have to be active at time of communication.
  - Client issues request and blocks until it receives reply
  - Server essentially waits only for incoming requests, and subsequently processes them
- Drawbacks of synchronous communication
  - Client cannot do any other work while waiting for reply
  - Failures have to be handled immediately: the client is waiting

Observations! ☺/☹

Message-oriented middleware

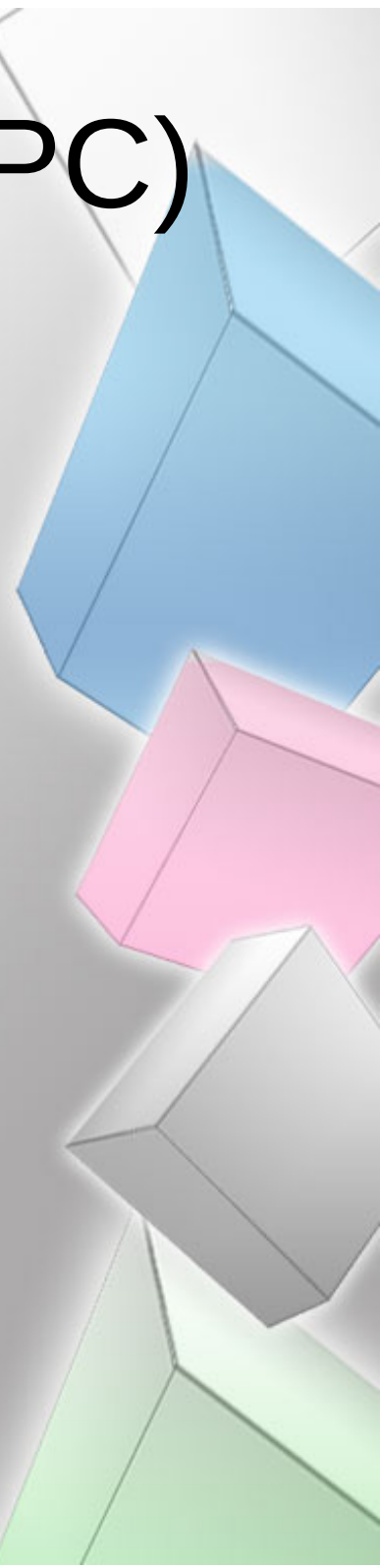
Aims at high-level persistent asynchronous communication:

- Processes send each other messages, which are queued
- Sender need not wait for immediate reply, but can do other things
- Middleware often ensures fault tolerance



Remote Procedure Call (RPC)

- Basic RPC operation
- Parameter passing
- Variations



The RPC basic idea

- All application developers are familiar with simple procedure model
  - Procedures operate in isolation (black box)
  - There is no fundamental reason not to execute procedures on separate machine ☺
- A paper by Birrell and Nelson (1984) introduced a completely different way of handling communication
  - Allow programs to call procedures located on other machines. 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: let's start playing

- While the basic idea of RPC sounds simple and elegant, subtle problems exist.
  - To start with, because the calling and called procedures run on different machines, they execute in different address spaces, which causes complications.
  - Parameters and results also have to be passed, which can be complicated, especially if the machines are not identical.
  - Finally, either or both machines can crash and each of the possible failures causes different problems.
- Still, most of these can be dealt with, and RPC is a widely-used technique that underlies many distributed systems.

Recollect Procedure Calls in C

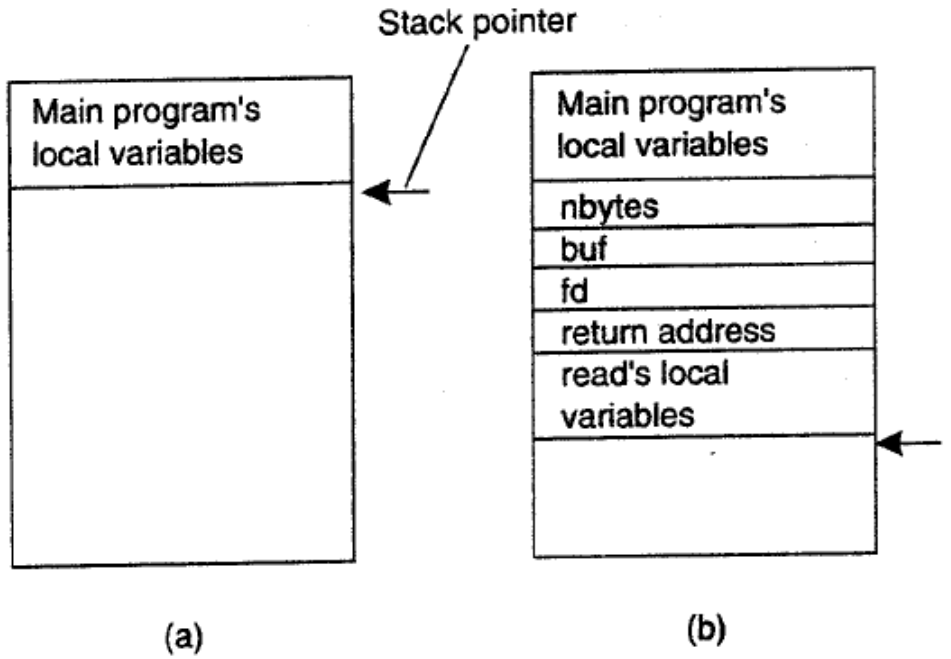


Figure 4-5. (a) Parameter passing in a local procedure call: the stack before the call to read. (b) The stack while the called procedure is active.

In C, parameters can be call-by-value or call-by-reference.
A value parameter is simply copied to the stack,
A reference parameter in C is a pointer to a variable (i.e., the address of the variable), rather than the value of the variable.
The difference between call-by-value and call-by-reference is quite important for RPC

RPC and Transparency

- The idea behind RPC is to make a remote procedure call look as much as possible like a local one.
- In other words, we want RPC to be transparent - the calling procedure should not be aware that the called procedure is executing on a different machine or vice versa.

Example:

- Suppose that a program needs to read some data from a file.
  - The programmer puts a call to read in the code to get the data.
  - In a traditional (single-processor) system, the read routine is a short procedure, which is generally implemented by calling an equivalent read system call.
  - In other words, the read procedure is a kind of interface between the user code and the local operating system.

Client-Server with RPC

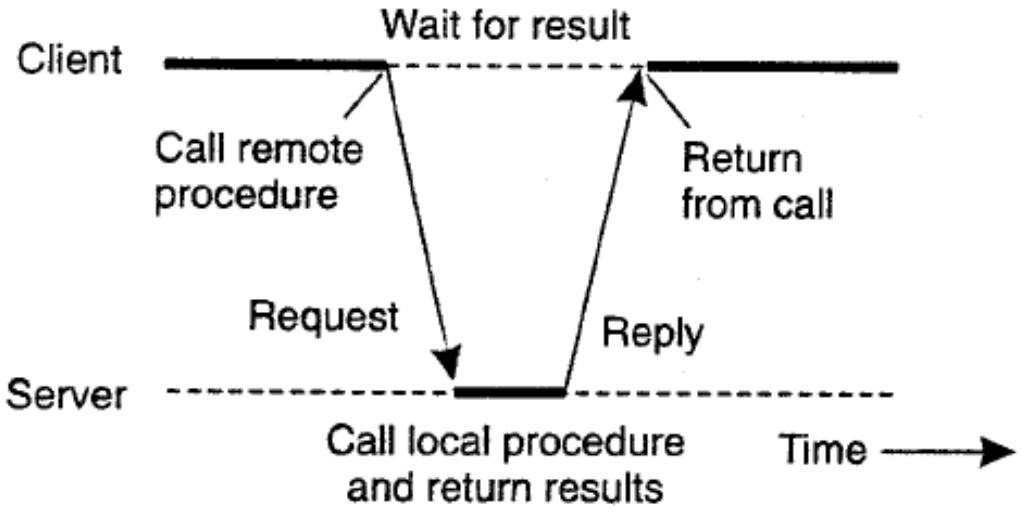
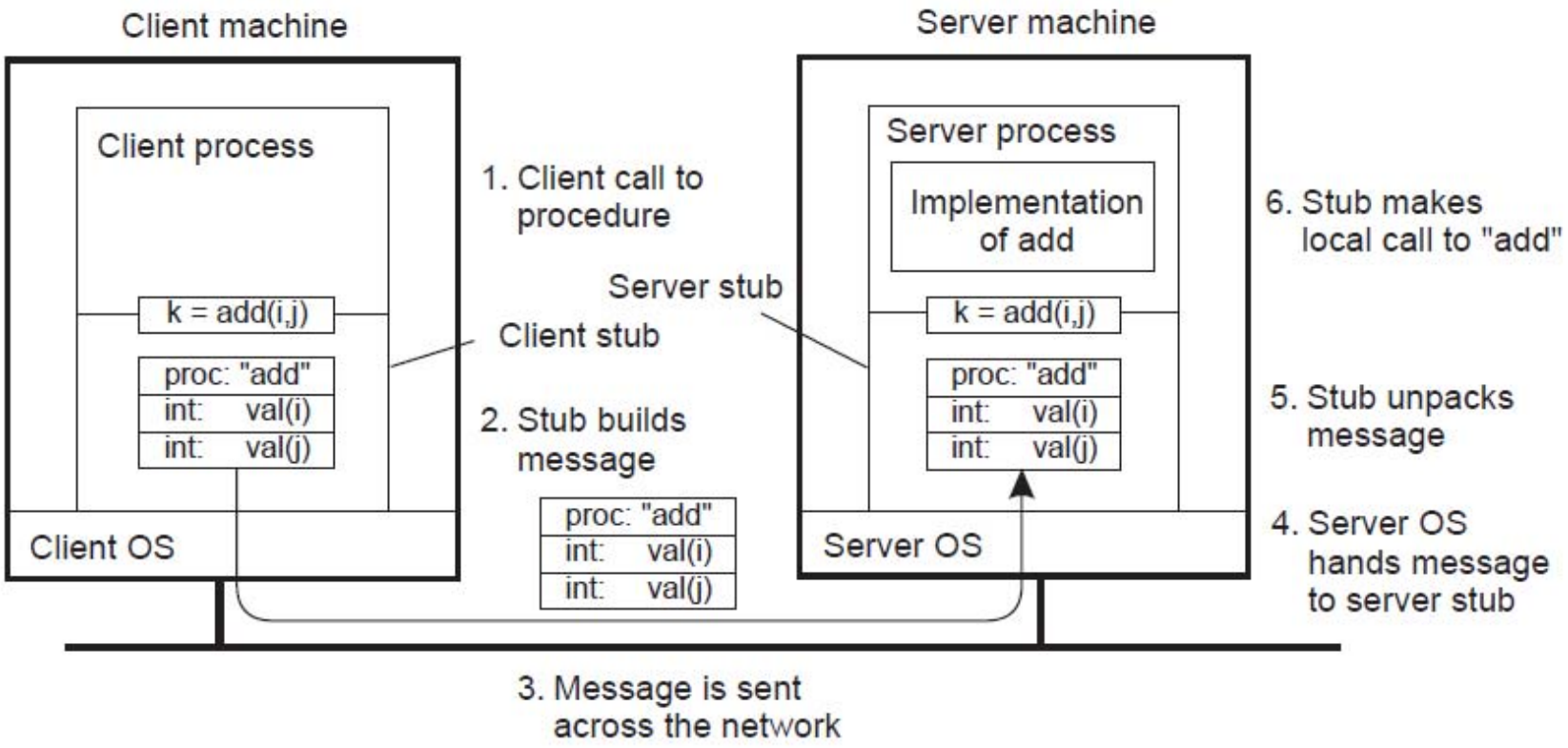


Figure 4-6. Principle of RPC between a client and server program.

When read is actually a remote procedure (e.g., one that will run on the file server's machine), a different version of read, called a client stub, is put into the library.

When the message arrives at the server, the server's operating system passes it up to a server stub.
A server stub is the server-side equivalent of a client stub: it is a piece of code that transforms requests coming in over the network into local procedure calls.

RPC Step-by-Step



1. Client procedure calls client stub.
2. Stub builds message; calls local OS.
3. OS sends message to remote OS.
4. Remote OS gives message to stub.
5. Stub unpacks parameters and calls server.
6. Server returns result to stub.
7. Stub builds message; calls OS.
8. OS sends message to client's OS.
9. Client's OS gives message to stub.
10. Client stub unpacks result and returns to the client.

RPC: Parameter Passing

- The function of the client stub is to take its parameters, pack them into a message, and send them to the server stub.
- Packing parameters into a message is called **parameter marshaling**.
- Problems:
  - Client and server machines may have different data representations (for example byte ordering)
  - Wrapping a parameter means transforming a value into a sequence of bytes
  - Client and server have to agree on the same encoding:
    - How are basic data values represented (integers, floats, characters)
    - How are complex data values represented (arrays, unions)
  - Client and server need to properly interpret messages, transforming them into machine-dependent representations.

Example for parameter problems

- 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, IBM 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 of RPC the server will interpret the character incorrectly.

Passing Reference Parameters

- A pointer is meaningful only within the address space of the process in which it is being used.
  - Getting back to the read example discussed earlier, 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.
- One solution is just to forbid pointers and reference parameters in general.
  - However, these are so important that this solution is highly undesirable.
- One strategy then becomes apparent: copy the array into the message and send it to the server.

Interface Definition Language

- Once the RPC protocol has been fully defined, the client and server stubs need to be implemented.
  - Fortunately, stubs for the same protocol but different procedures normally differ only in their interface to the applications.
- An interface consists of a collection of procedures that can be called by a client, and which are implemented by a server.
- An interface is usually available in the same programming language as the one in which the client or server is written (although this is strictly speaking, not necessary).
- To simplify matters, interfaces are often specified by means of an Interface Definition Language (IDL).
- An interface specified in such an IDL is then subsequently compiled into a client stub and a server stub, along with the appropriate compile-time or run-time interfaces.

Asynchronous RPCs

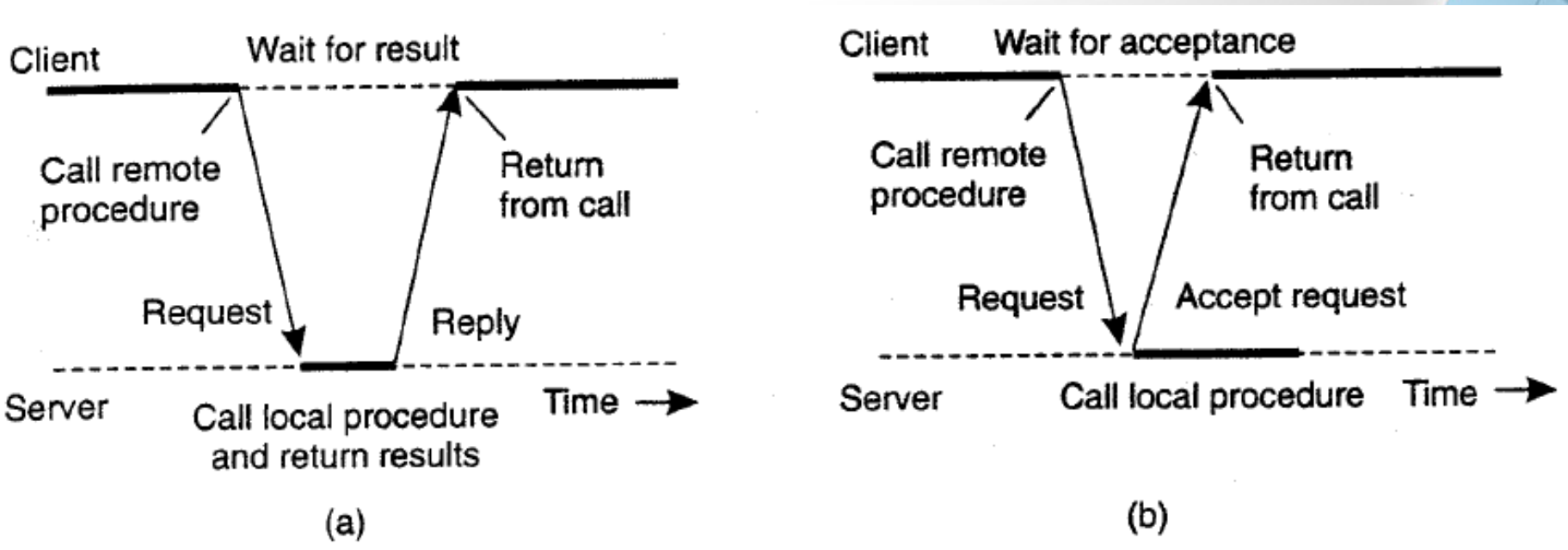


Figure 4-10. (a) The interaction between client and server in a traditional RPC. (b) The interaction using asynchronous RPC.

Try to get rid of the strict request-reply behavior. Let the client continue without waiting for an answer from the server.

Deferred synchronous RPCs

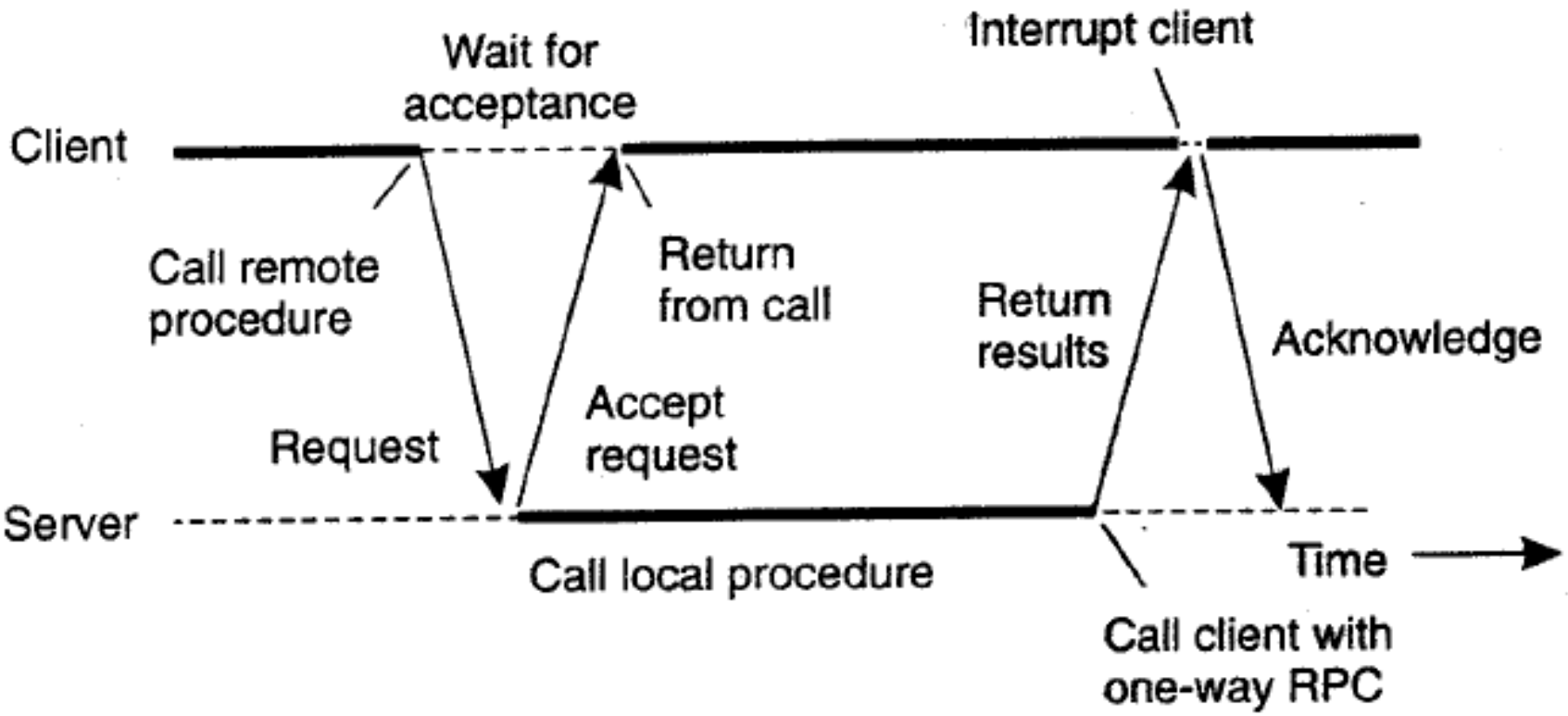


Figure 4-11. A client and server interacting through two asynchronous RPCs.

DCE RPC

- Remote procedure calls have been widely adopted as the basis of middleware and distributed systems in general.
- One specific RPC system is the Distributed Computing Environment (DCE), which was developed by the Open Software Foundation (OSF), now called The Open Group.
- DCE RPC is not as popular as some other RPC systems, notably Sun RPC.
- However, DCE RPC is nevertheless representative of other RPC systems, and its specifications have been adopted in Microsoft's base system for distributed computing, DCOM.

DCE: a true middleware

- DCE is a true middleware system in that it is designed to execute as a layer of abstraction between existing (network) operating systems and distributed applications.
- Initially designed for UNIX, it has now been ported to all major operating systems including VMS and Windows variants, as well as desktop operating systems.
- The idea is that the customer can take a collection of existing machines, add the DCE software, and then be able to run distributed applications, all without disturbing existing (nondistributed) applications.
- Although most of the DCE package runs in user space, in some configurations a piece (part of the distributed file system) must be added to the kernel.
- The Open Group itself only sells source code, which vendors integrate into their systems

DCE: client-server

- The programming model underlying all of DCE is the client-server model, which was extensively discussed in the previous lesson.
- User processes act as clients to access remote services provided by server processes.
- Some of these services are part of DCE itself, but others belong to the applications and are written by the applications programmers.
- All communication between clients and servers takes place by means of RPCs.
- There are a number of services that form part of DCE itself.

DCE: services

- There are a number of services that form part of DCE itself.
- The distributed file service is a worldwide file system that provides a transparent way of accessing any file in the system in the same way. It can either be built on top of the hosts' native file systems or used instead of them.
- The directory service is used to keep track of the location of all resources in the system. These resources include machines, printers, servers, data, and much more, and they may be distributed geographically over the entire world.
  - The directory service allows a process to ask for a resource and not have to be concerned about where it is, unless the process cares.
- The security service allows resources of all kinds to be protected, so access can be restricted to authorized persons.
- The distributed time service is a service that attempts to keep clocks on the different machines globally synchronized.
  - As we shall see, having some notion of global time makes it much easier to ensure consistency in a distributed system.

DCE-RPC

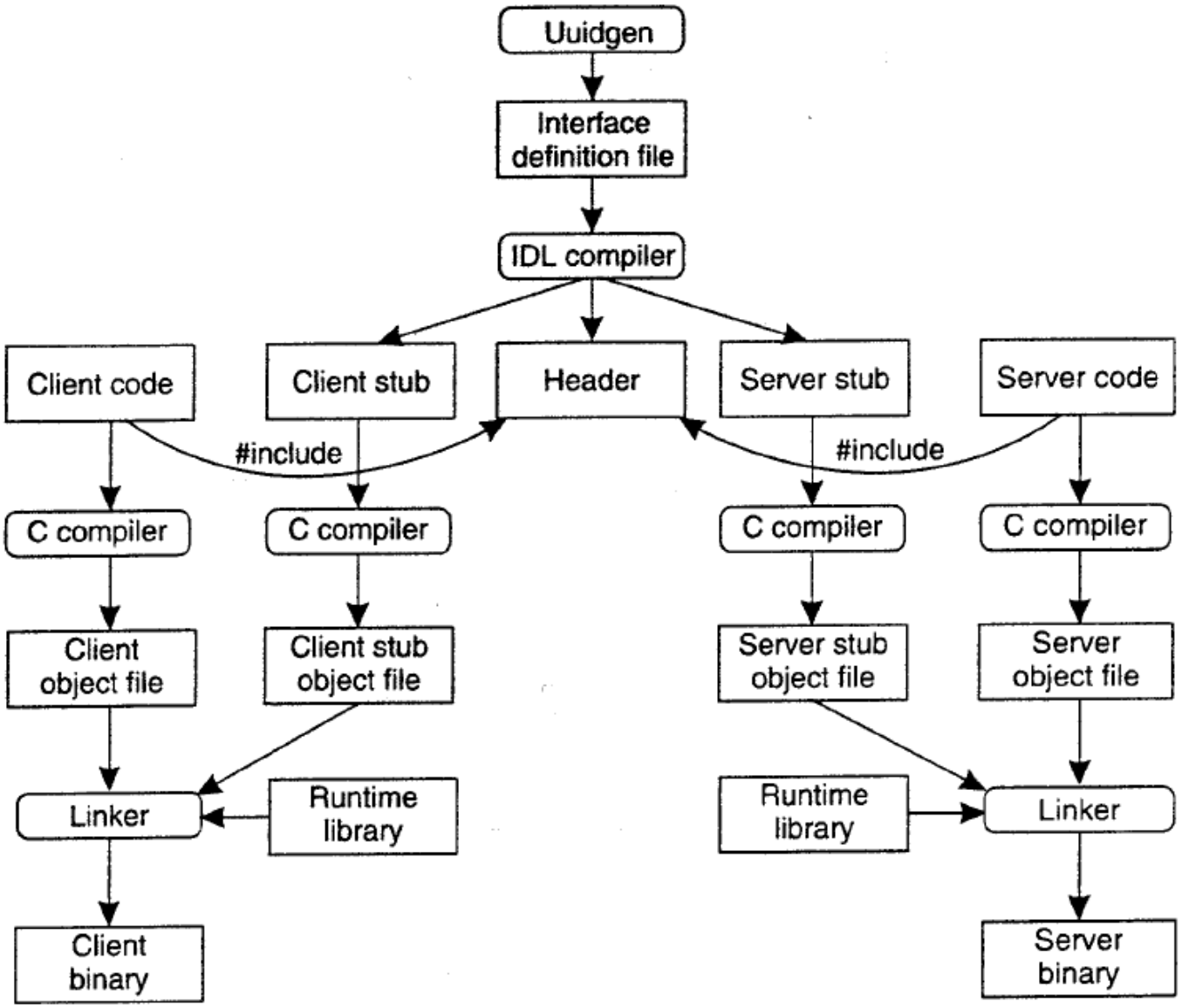


Figure 4-12. The steps in writing a client and a server in DeE RPC.

Client-to-server binding (DCE)

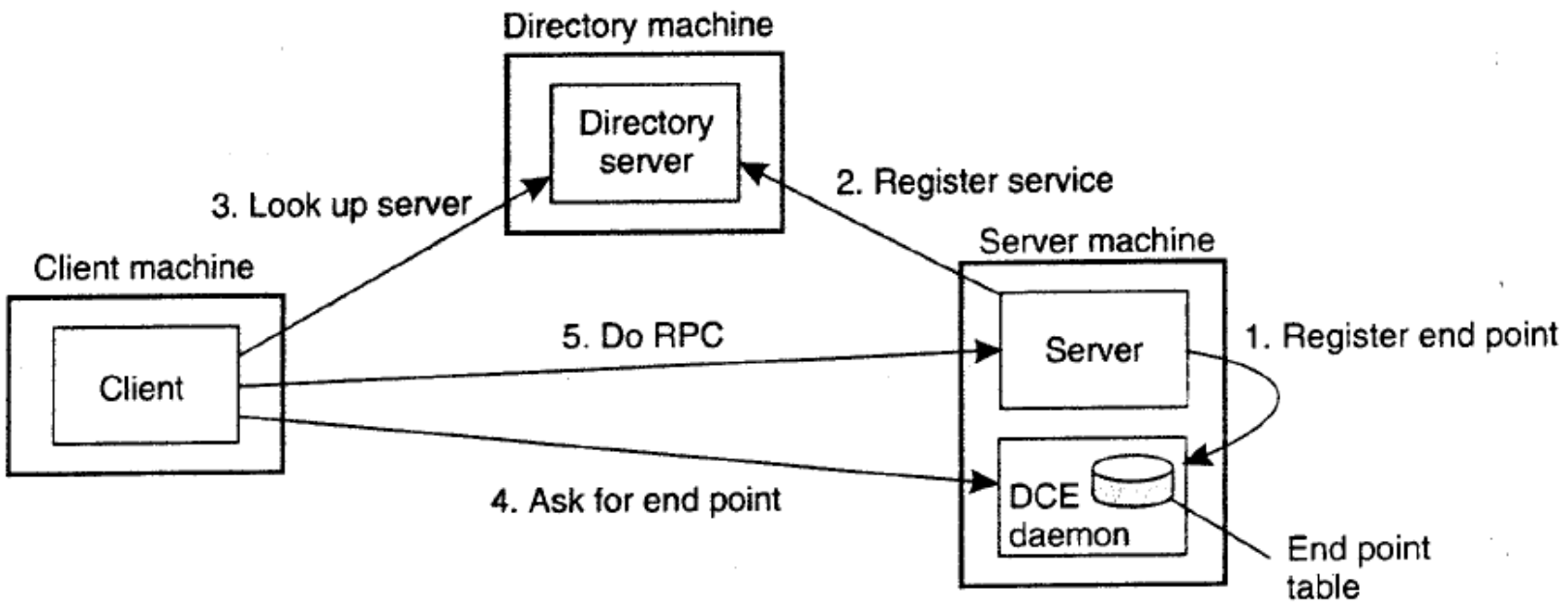


Figure 4-13. Client-to-server binding in DCE.

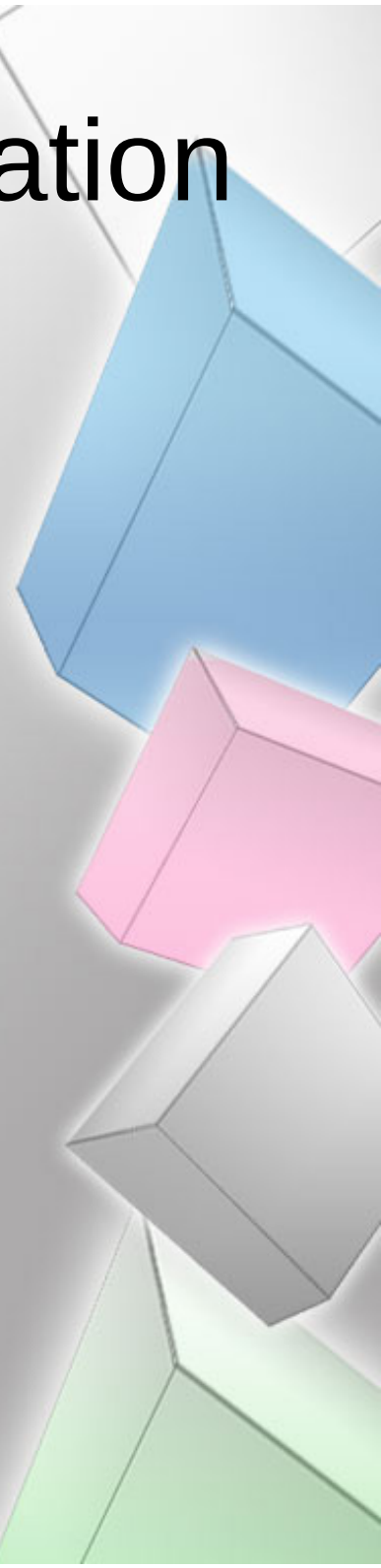
To communicate with a server, the client needs to know an end point, on the server's machine to which it can send messages.

An end point (also port) is used by the server's operating system to distinguish incoming messages for different processes. In DCE, a table of *(server, end point)pairs* is maintained on each server machine by a process called the DCE daemon.

Before it becomes available for incoming requests, the server must ask the operating system for an end point. It then registers this end point with the DCE daemon.

Message-Oriented Communication

- Transient Messaging
- Message-Queuing System
- Message Brokers
- Example: IBM Websphere



RPC: not always solves

- Remote procedure calls and remote object invocations contribute to hiding communication in distributed systems, that is, they enhance access transparency.

- Unfortunately, neither mechanism is always appropriate.
  - When it cannot be assumed that the receiving side is executing at the time a request is issued, alternative communication services are needed.
  - The inherent synchronous nature of RPCs, by which a client is blocked until its request has been processed, sometimes needs to be replaced by something else.

Transient messaging: Berkeley sockets

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

Socket Primitives for TCP/IP

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 TCPIIP.

Transient messaging: sockets

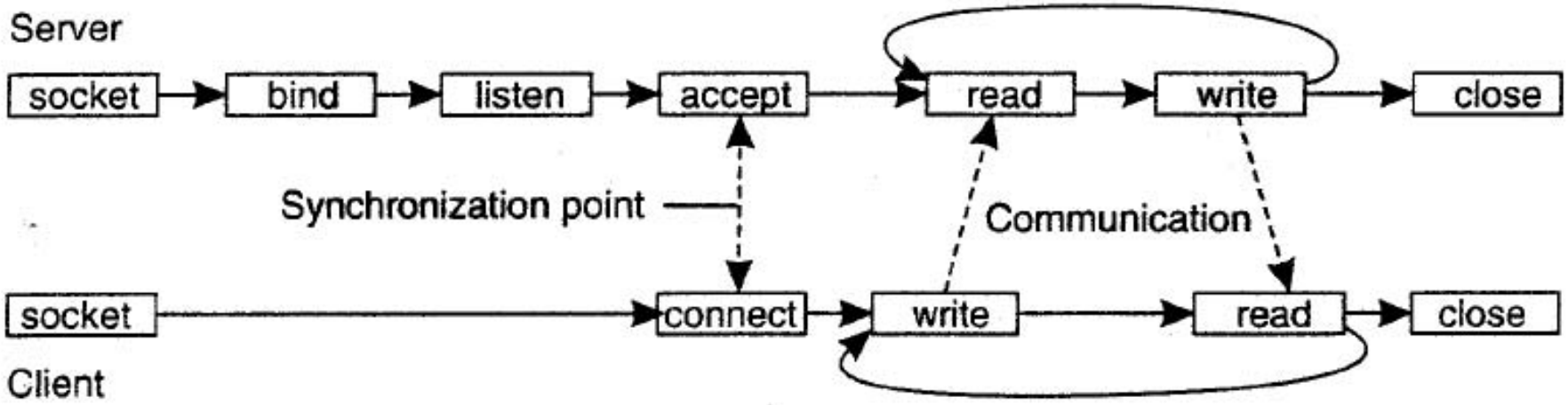


Figure 4-15. Connection-oriented communication pattern using sockets.

Sockects: simply insufficient

- Sockets were deemed insufficient for two reasons.
- First, they were at the wrong level of abstraction by supporting only simple send and receive primitives.
- Second, sockets had been designed to communicate across networks using general-purpose protocol stacks such as TCP/IP.
  - They were not considered suitable for the proprietary protocols developed for high-speed interconnection networks, 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.

MPI: Message Passing Interface

- The result of developing particular solutions was that most interconnection networks and high-performance multicomputers were shipped with proprietary communication libraries.
- These libraries offered a wealth of high-level and generally efficient communication primitives.
  - Of course, all libraries were mutually incompatible, so that application developers now had a portability problem.
- 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.

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 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

- Asynchronous persistent communication through support of middleware-level queues.
  - Queues correspond to buffers at communication servers.
- Message-queuing systems
  - An important aspect of message-queuing systems is that a 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.
- These semantics permit communication loosely-coupled in time.
  - There is thus no need for the receiver to be executing when a message is being sent to its queue.
  - Likewise, there is no need for the sender to be executing at the moment its message is picked up by the receiver.

Who is running where and when

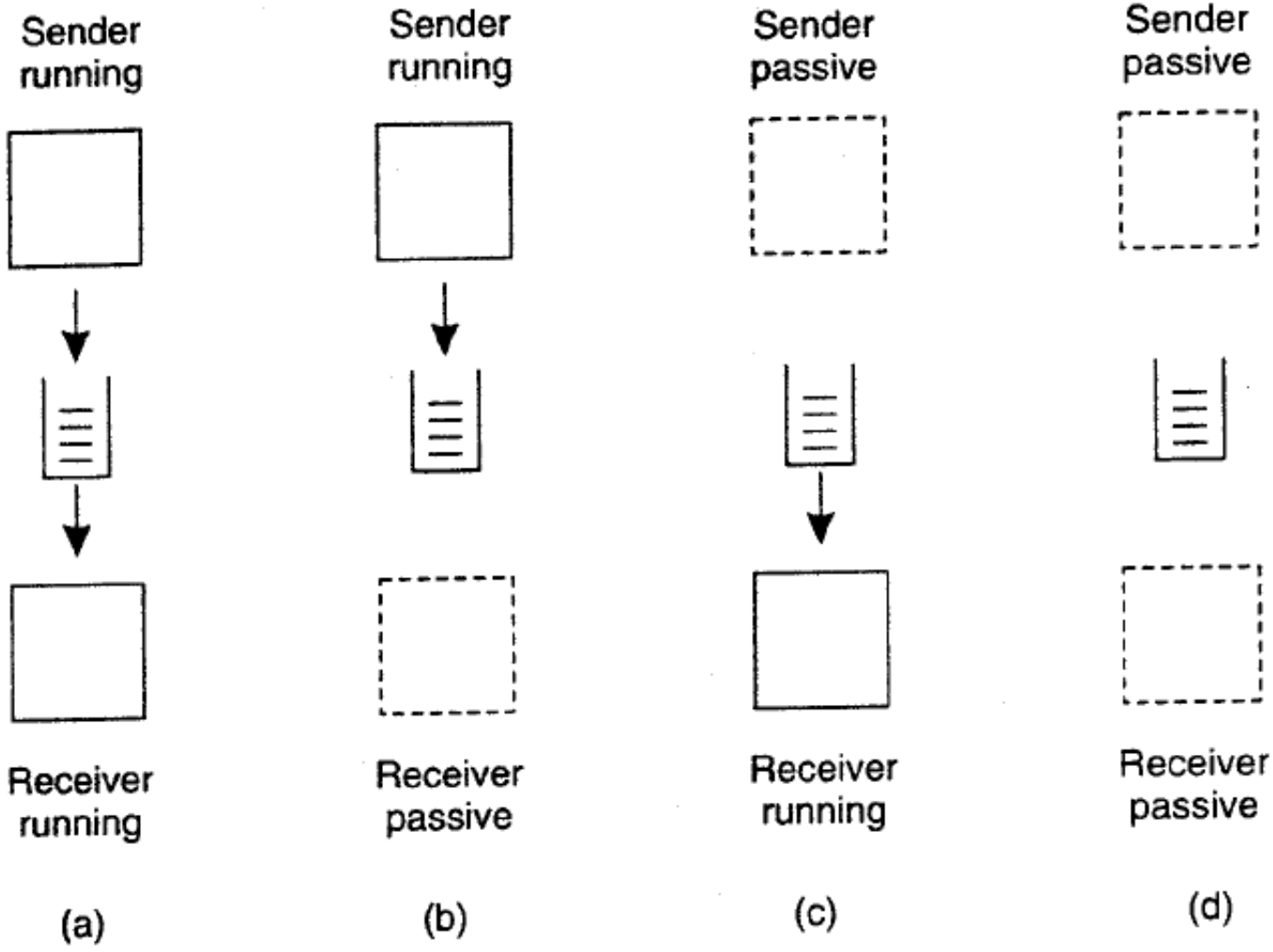


Figure 4-17. Four combinations for loosely-coupled communications using queues.

Basic interface

- 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 such as:

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.

Architecture of a Message-Queuing System

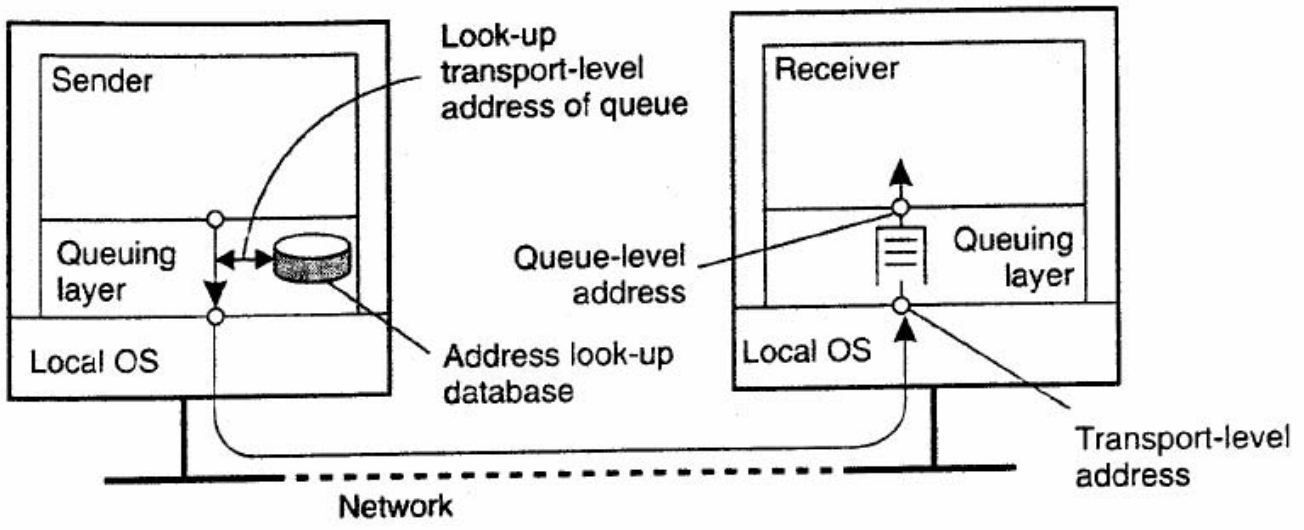


Figure 4-19. The relationship between queue-level addressing and network-level addressing.

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.

Such a mapping is analogous to the Domain Name System (DNS) for e-mail in the Internet. For example, when sending a message to the logical *mail* address *xxx@yy.zz.vv,* 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.

Message broker

Observation

- Message queuing systems assume a common messaging protocol: all applications agree on message format (i.e., structure and data representation)

Message broker

Centralized component that takes care of application heterogeneity in an MQ system:

- Transforms incoming messages to target format
- Very often acts as an application gateway
- May provide subject-based routing capabilities => Enterprise Application Integration

Message Broker

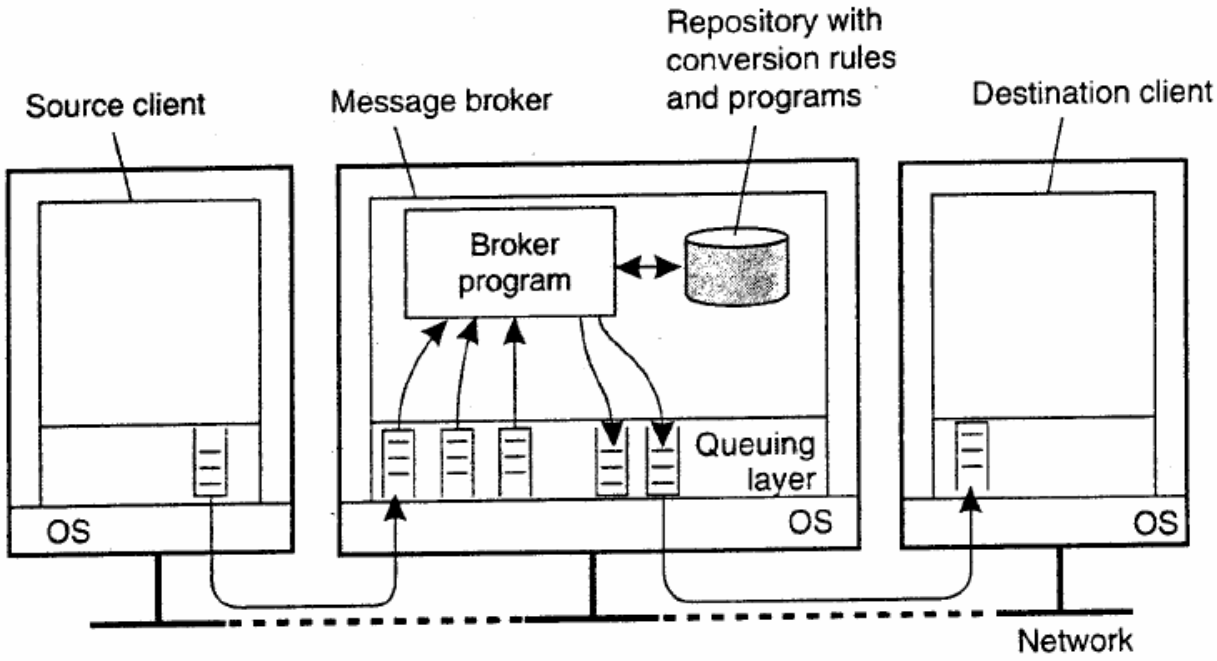


Figure 4-21. The general organization of a message broker in a message-queuing system.

Message Broker in Enterprise Information Systems

- 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.

E-Mail systems

- E-mail systems are generally implemented through a collection of mail servers that store and forward messages on behalf of the users on hosts directly connected to the server.
  - Routing is generally left out, as e-mail systems can make direct use of the underlying transport services.
  - For example, in the mail protocol for the Internet, SMTP (Postel, 1982), a message is transferred by setting up a direct TCP connection to the destination mail server.
- What makes e-mail systems special compared to message-queuing systems is that they are primarily aimed at providing direct support for end users.
  - In addition, e-mail systems may have very specific requirements such as automatic message filtering, support for advanced messaging databases (e.g., to easily retrieve previously stored messages), and so on.
- General message-queuing systems are not aimed at supporting only end users.
  - They are set up to enable persistent communication between processes, regardless of whether a process is running a user application, handling access to a database or performing computations.

IBM Websphere

Message transfer

- Messages are transferred between queues
- Message transfer between queues at different processes, requires a channel
- At each endpoint of channel is a message channel agent
- Message channel agents are responsible for:
  - Setting up channels using lower-level network communication
  - facilities (e.g., TCP/IP)
  - (Un)wrapping messages from/in transport-level packets
  - Sending/receiving packets

Websphere

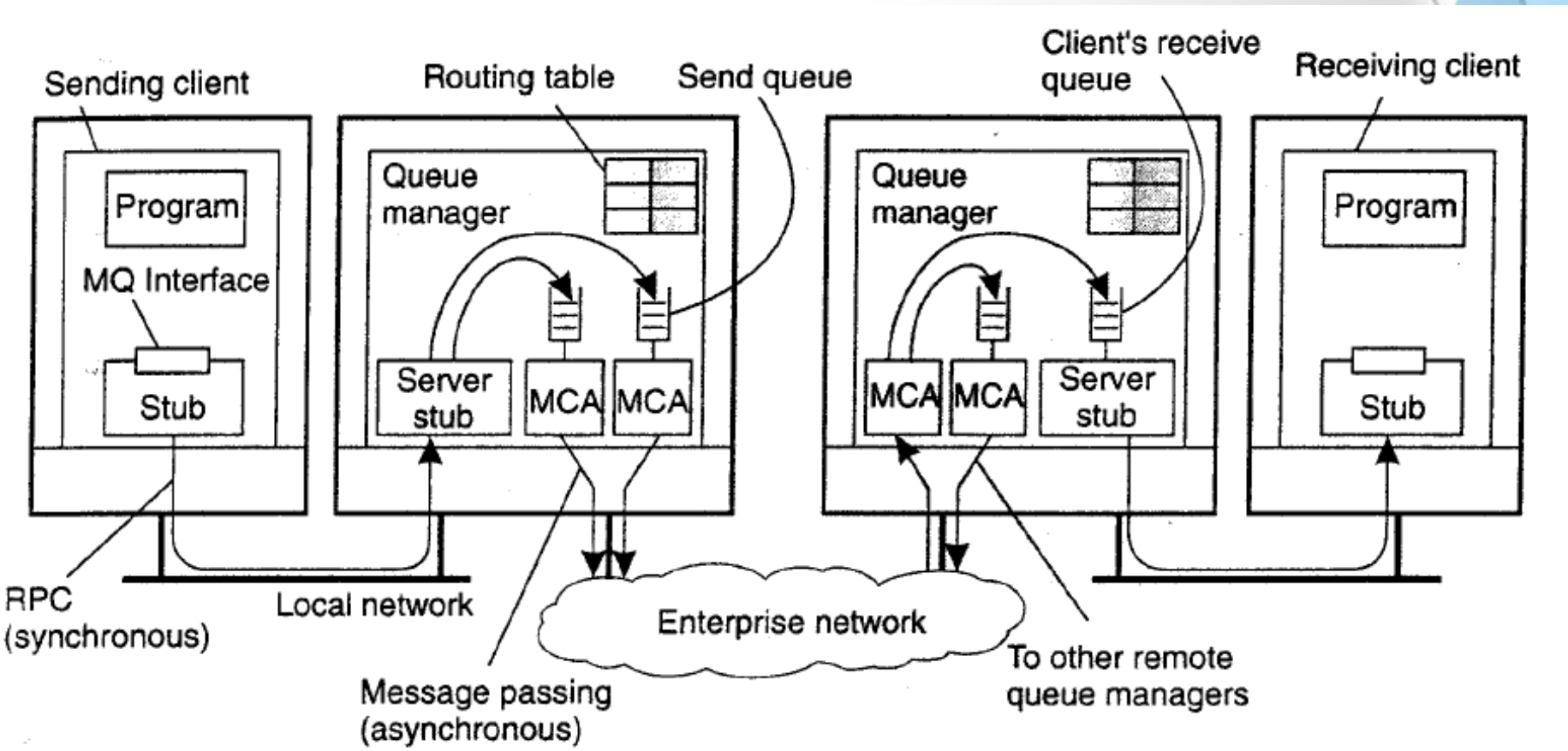


Figure 4-22. General organization of IBM's message-queuing system.

Websphere

Routing

- By using logical names, in combination with name resolution to local queues, it is possible to put a message in a remote queue

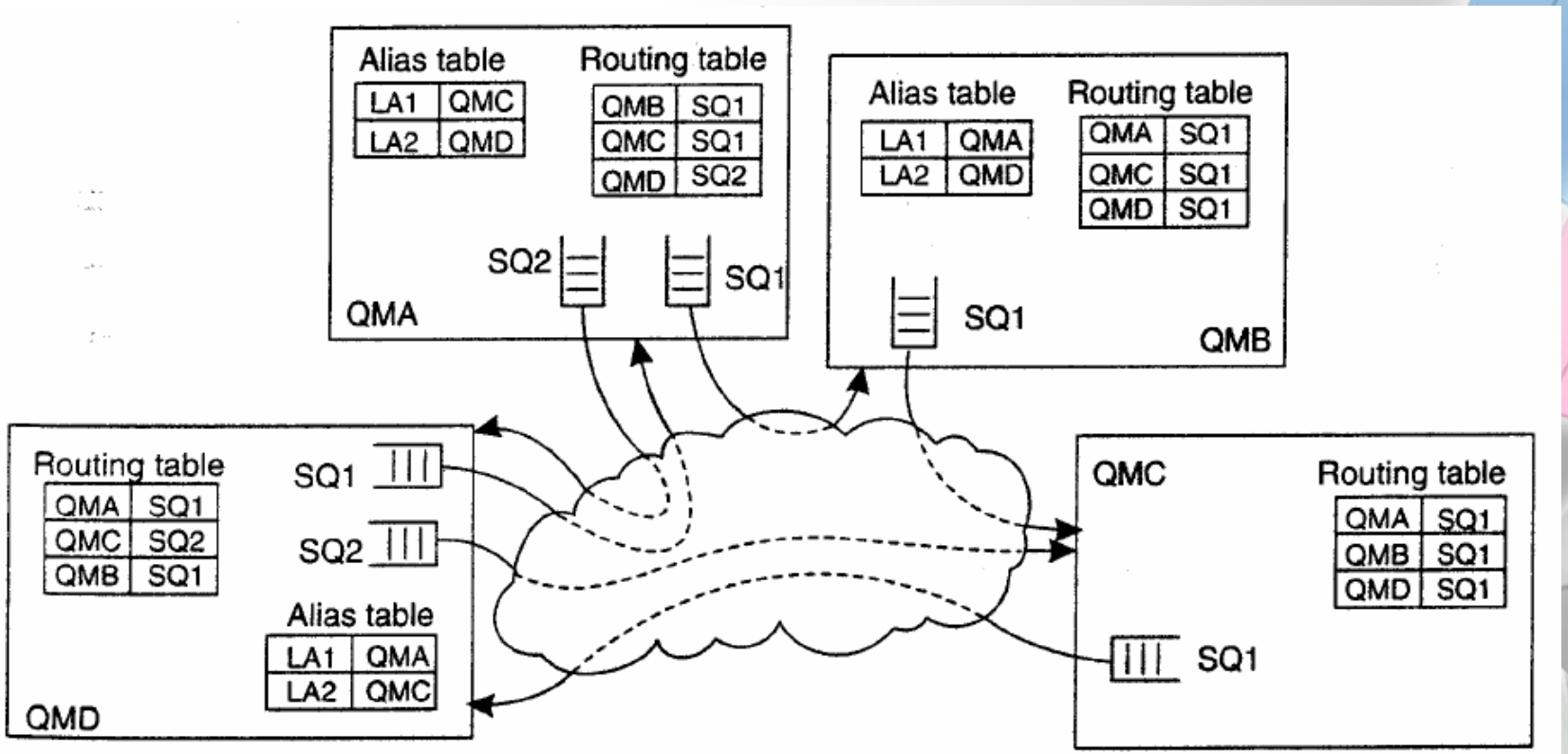
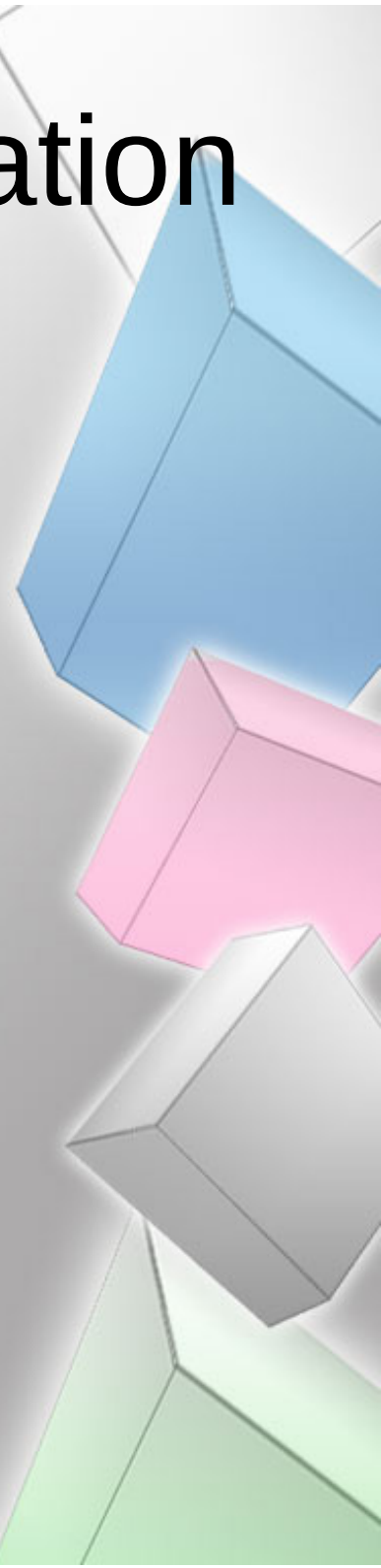


Figure 4-24. The general organization of an MQ queuing network using routing tables and aliases.

Stream-oriented communication

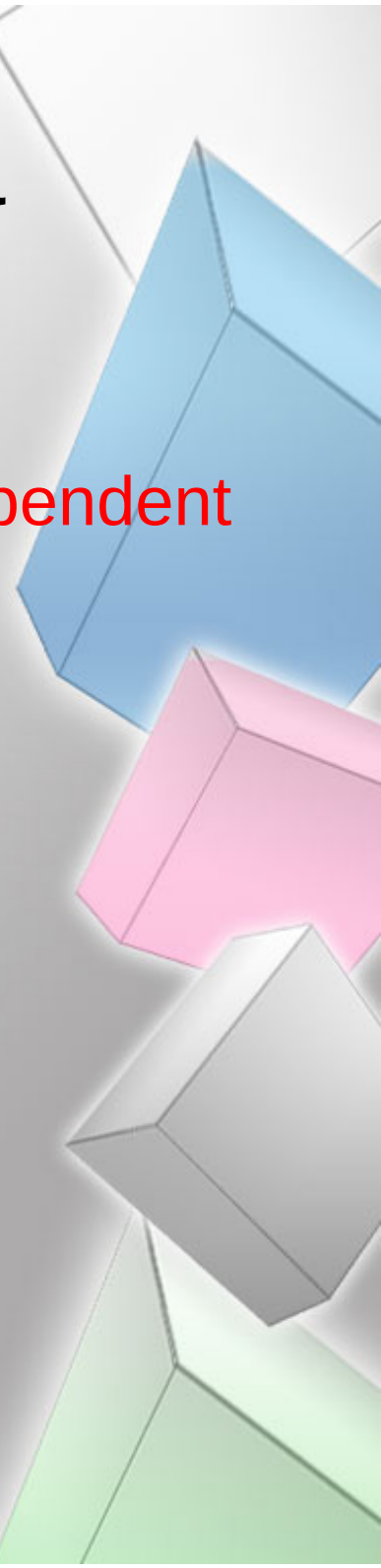
- Support for continuous media
- Streams in distributed systems
- Stream management



Continuous flow of data

Observation

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



Continuous media

Transmission modes

Different timing guarantees with respect to data transfer:

- Asynchronous: no restrictions with respect to when data is to be delivered
- Synchronous: define a maximum end-to-end delay for individual data packets
- Isochronous: define a maximum and minimum end-to-end delay

Stream

Definition

A (continuous) data stream is a connection-oriented communication facility that supports isochronous data transmission.

Some common stream characteristics

- Streams are unidirectional
- There is generally a single source, and one or more sinks
- Often, either the sink and/or source is a wrapper around hardware (e.g., camera, CD device, TV monitor)
- Simple stream: a single flow of data, e.g., audio or video
- Complex stream: multiple data flows, e.g., stereo audio or combination audio/video

Streaming

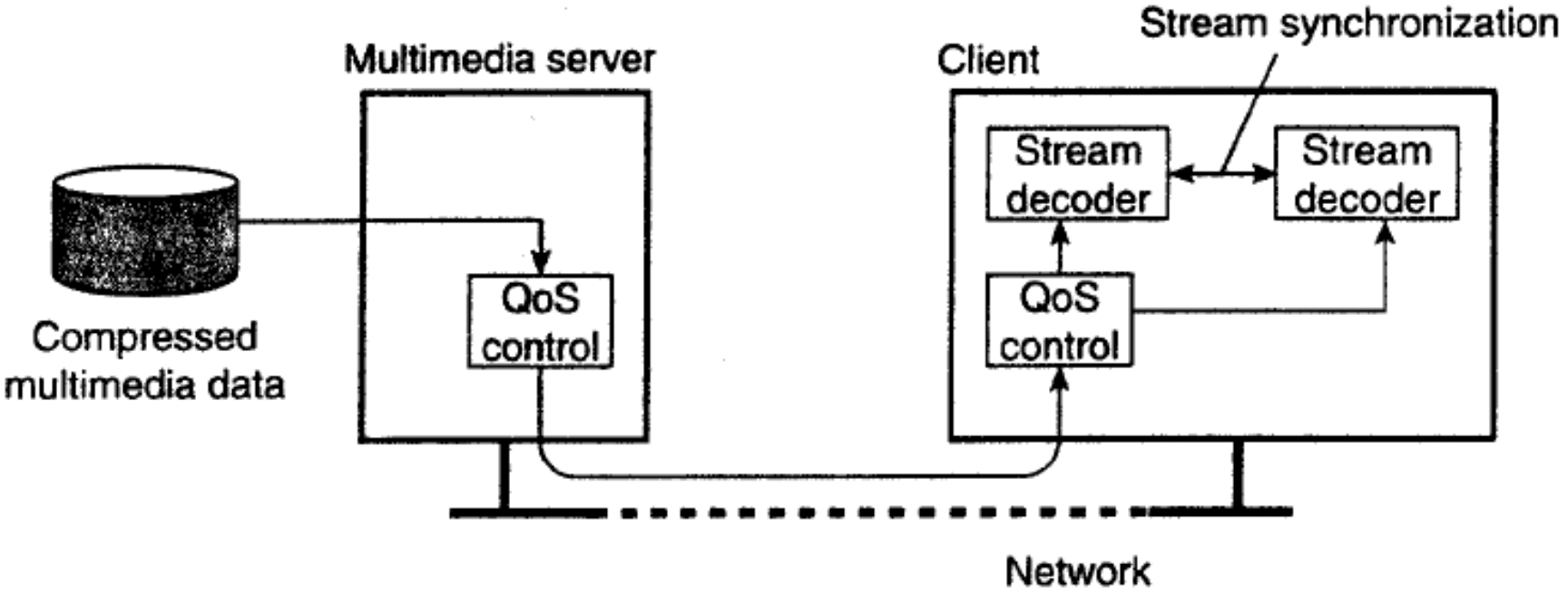


Figure 4-26. A general architecture for streaming stored multimedia data over a network.

Streams and QoS

Essence

- Streams are all about timely delivery of data. How do you specify this Quality of Service (QoS)? Basics:
  - The required bit rate at which data should be transported.
  - The maximum delay until a session has been set up (i.e., when an application can start sending data).
  - The maximum end-to-end delay (i.e., how long it will take until a data unit makes it to a recipient).
  - The maximum delay variance, or jitter.
  - The maximum round-trip delay.

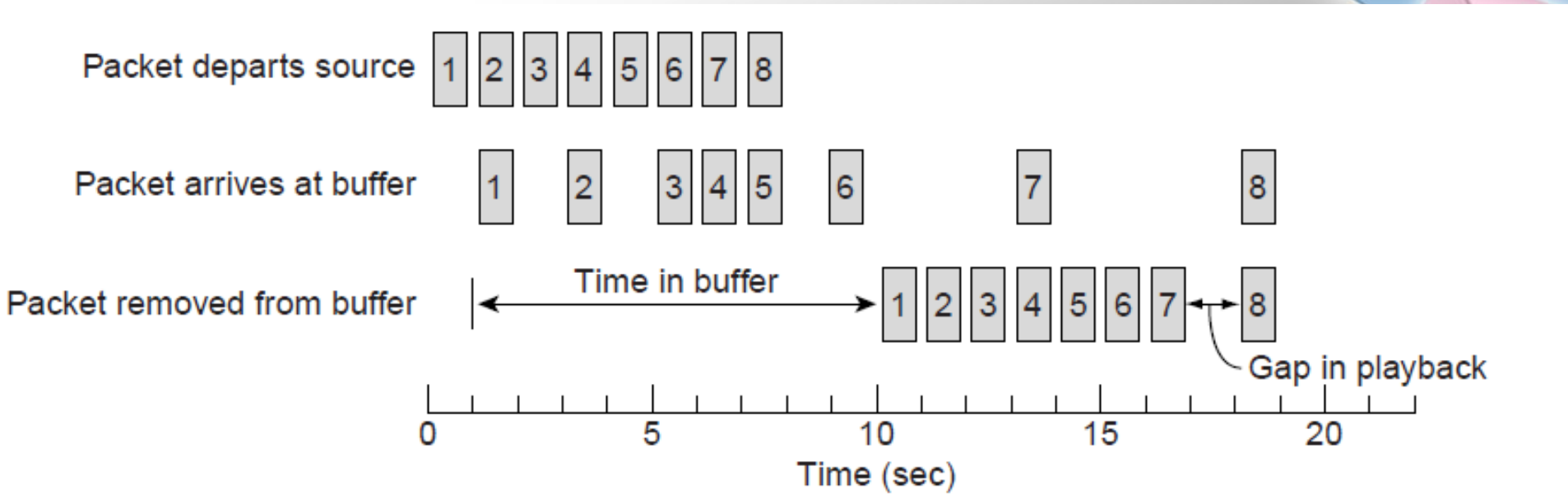
Enforcing QoS

Observation

- There are various network-level tools, such as differentiated services by which certain packets can be prioritized.

Also

- Use buffers to reduce jitter:



Enforcing QoS

Problem

- How to reduce the effects of packet loss (when multiple samples are in a single packet)?

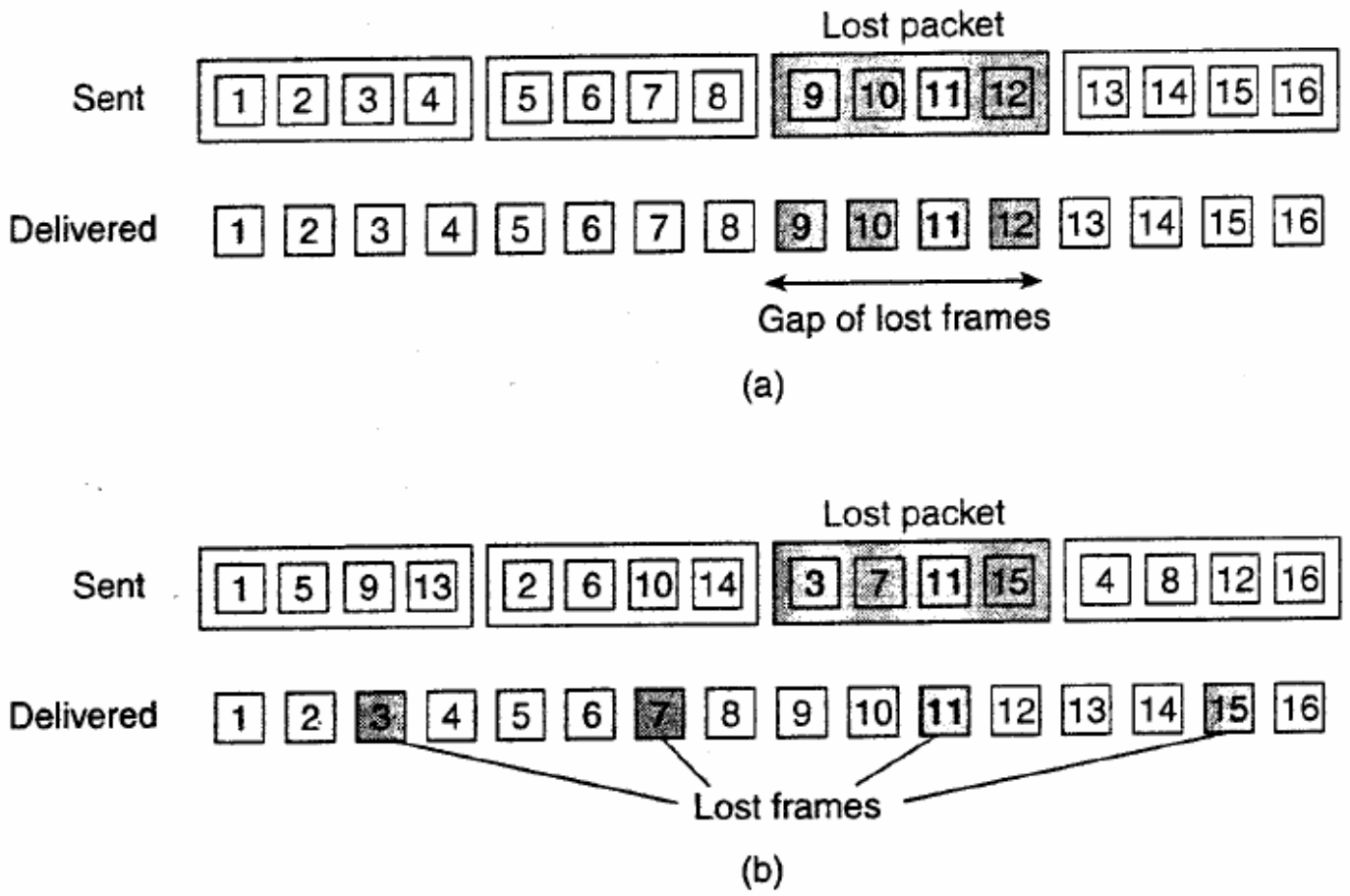


Figure 4-28. The effect of packet loss in (a) noninterleaved transmission and (b) interleaved transmission.

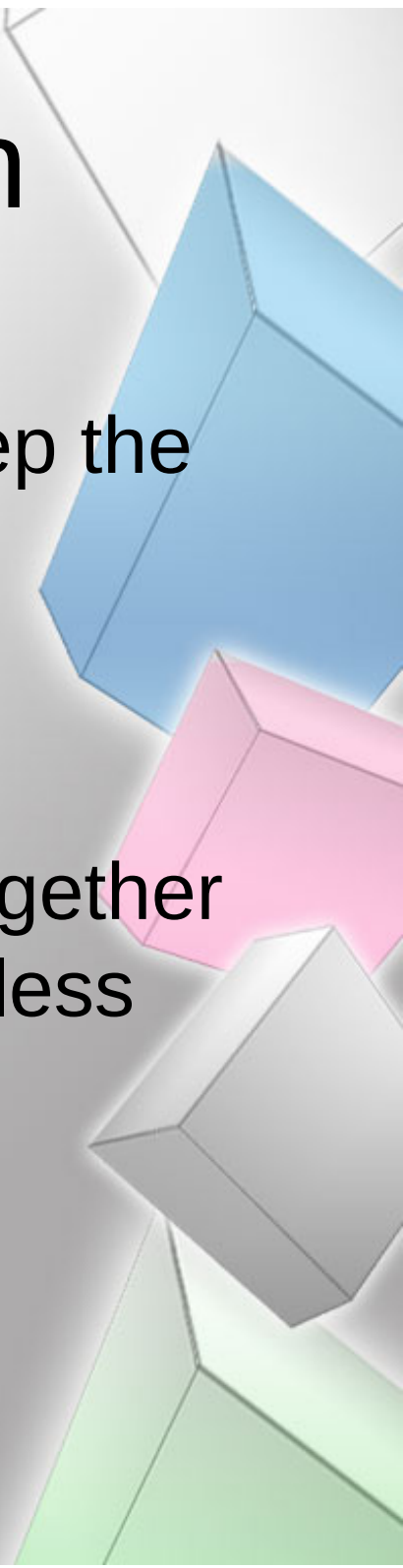
Stream synchronization

Problem

- Given a complex stream, how do you keep the different substreams in synch?

Example

- Think of playing out two channels, that together form stereo sound. Difference should be less than 20–30 msec!



Stream synchronization: level data units

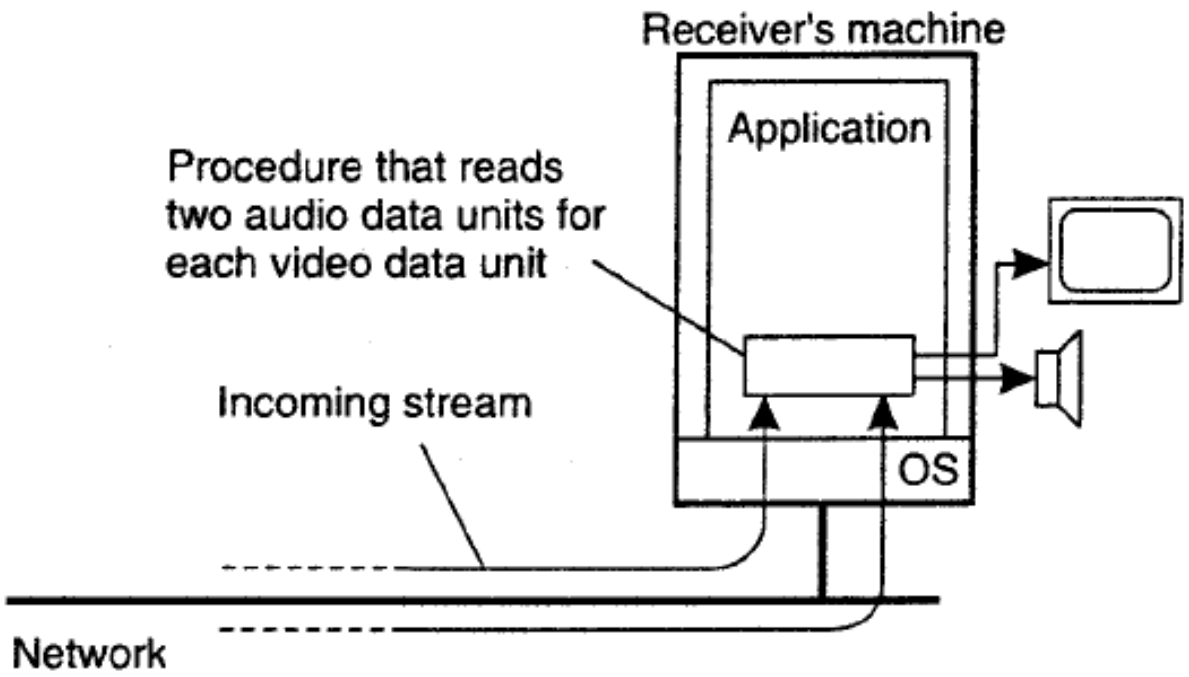


Figure 4-29. The principle of explicit synchronization on the level data units.

Stream synchronization: high-level interface

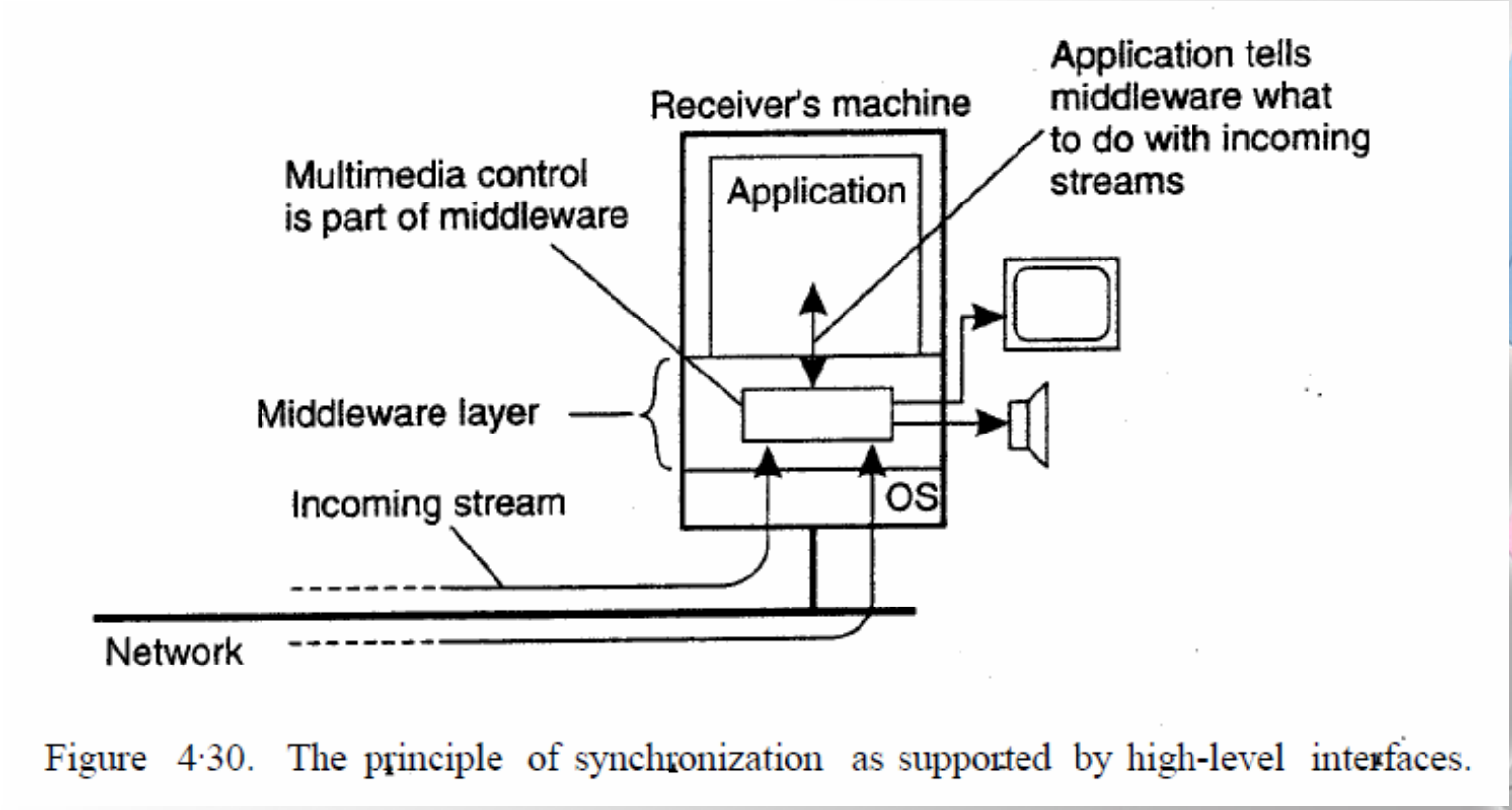


Figure 4·30. The principle of synchronization as supported by high-level interfaces.

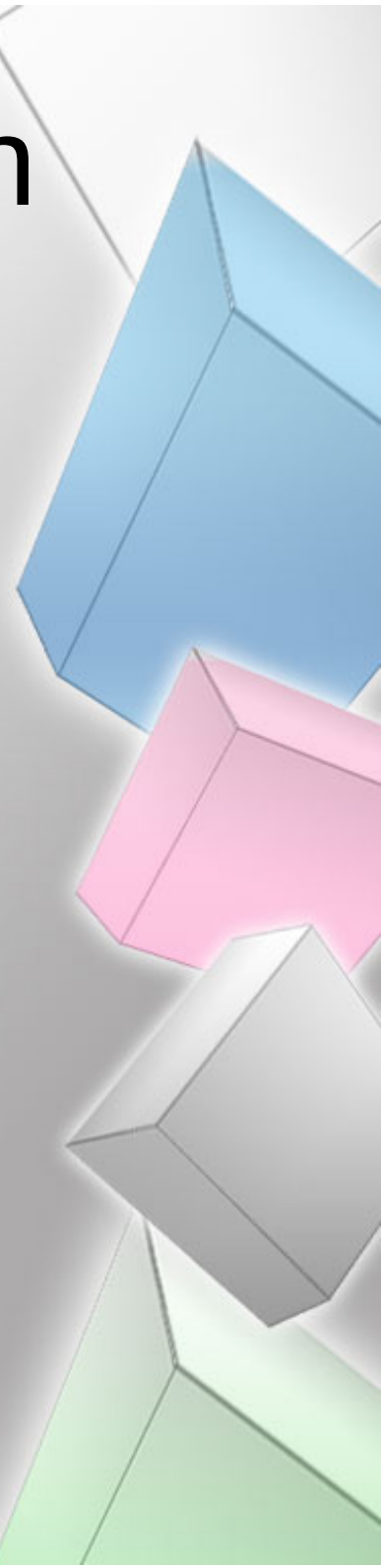
Multiplex all substreams into a single stream, and demultiplex at the receiver. Synchronization is handled at multiplexing/demultiplexing point (MPEG).

Multicasting

- An important topic in communication in distributed systems is the support for sending data to multiple receivers, also known as multicast communication.
- For many years, this topic has belonged to the domain of network protocols, where numerous proposals for network-level and transport-level solutions have been implemented and evaluated.
  - A major issue in all solutions was setting up the communication paths for information dissemination.
- With the advent of peer-to-peer technology, and notably structured overlay management, it became easier to set up communication paths.
- As peer-to-peer solutions are typically deployed at the application layer, various application-level multicasting techniques have been introduced

Multicast communication

- Application-level multicasting
- Gossip-based data dissemination



Application-level multicasting

- The basic idea in application-level multicasting is that nodes organize into an overlay network, which is then used to disseminate information to its members.

- An important observation is that network routers are not involved in group membership.

- As a consequence, the connections between nodes in the overlay network may cross several physical links, and as such, routing messages within the overlay may not be optimal in comparison to what could have been achieved by network-level routing.

Organization of the Overlay Network

- First, nodes may organize themselves directly into a tree, meaning that there is a unique (overlay) path between every pair of nodes.

- An alternative approach is that nodes organize into a mesh network in which every node will have multiple neighbors and, in general, there exist multiple paths between every pair of nodes.

- The main difference between the two is that the latter generally provides higher robustness: if a connection breaks (because a node fails), there will still be an opportunity to disseminate information without having to immediately reorganize the entire overlay network.

Gossip-Based Data Dissemination

- An increasingly important technique for disseminating information is to rely on *epidemic behavior.*

- Observing how diseases spread among people, researchers have since long investigated whether simple techniques could be developed for spreading information in very large-scale distributed systems.

- The main goal of these epidemic protocols is to rapidly propagate information among a large collection of nodes using only local information. In other words, there is no central component by which information dissemination is coordinated.

End of Lesson 4

- Readings
  - Distributed Systems, Chapter 4.