Chapter 4
Communication
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.
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.
Remote Procedure Call (RPC)

- **RPC** allows a program to transparently call procedures located on another machine. No message passing is visible to the programmer.
- A widely used technique that underlies many distributed systems.
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.
Parameter Passing in Procedure Calls

- **Call-by-value**: The variable is copied to the stack. The original value is left unchanged.
- **Call-by-reference**: The address of the variable is copied to the stack so the called procedure modifies the same copy as the procedure calling.
- **Call-by-copy/restore**: The variable is copied to the stack by the caller and then copied back after the call, overwriting the caller’s original value.

How is call-by-copy/restore different from call-by-reference?
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 (contd.):

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.
Parameter Passing

Passing parameters to RPCs can be tricky since it requires packing parameters into a message (*parameter marshaling*) to be interpreted correctly on another system (*unmarshaling*). For example:

- Machines may use different character codes.
- Different representation of integers and floating-point numbers
- Different byte addressing: *little-endian* versus *big-endian* format
- How to pass reference parameters (like pointers) since an address will be meaningless on another system? Can we use call-by-copy-restore?
- How to pack types that take less space than a word? For example: a short or a character.
- How to pack an array?
Passing Value Parameters (1)

Figure 4-7. The steps involved in doing a remote computation through RPC.
Passing Value Parameters (2)

Figure 4-8. (a) The original message on the a little-endian system
Passing Value Parameters (3)

Figure 4-8. (b) The message after receipt on a big-endian system
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)

<table>
<thead>
<tr>
<th>foobar's local variables</th>
<th>x</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>y</td>
</tr>
<tr>
<td></td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>z[0]</td>
</tr>
<tr>
<td></td>
<td>z[1]</td>
</tr>
<tr>
<td></td>
<td>z[2]</td>
</tr>
<tr>
<td></td>
<td>z[3]</td>
</tr>
<tr>
<td></td>
<td>z[4]</td>
</tr>
</tbody>
</table>

(b)

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

IDL (Interface definition Language) is used to define interfaces for RPC. IDL is then compiled into client and server stub along appropriate compile/run-time interfaces.
Asynchronous RPC (1)

- **Asynchronous RPC**: A client immediately continues once the server accepts the RPC request. The server executes the RPC request after the acknowledgment.

- **Deferred asynchronous RPC**: The client calls the server with a RPC request and the server immediately acknowledges it. Later the server does a *callback* to the client with the result.
Asynchronous RPC (2)

Figure 4-10. (a) The interaction between client and server in a traditional RPC.
Asynchronous RPC (3)

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

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

- **Distributed Computing Environment / Remote Procedure Calls (DCE/RPC)** was commissioned by the Open Software Foundation, an industry consortium now renamed as the Open Group. The DCE/RPC specifications were adopted in Microsoft’s base for distributed computing, DCOM.

- **Open Network Computing Remote Procedure Call (ONC RPC)** is a widely deployed remote procedure call system. ONC was originally developed by Sun Microsystems as part of their Network File System project, and is sometimes referred to as Sun ONC or Sun RPC.
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
Figure 4-13. Client-to-server binding in DCE.
Performing an RPC

How to deal with network and system errors?

- **At-most-once operation**: No call is carried out more than once even if the system crashes.
- **At-least-once operation**: The call is tried multiple times until it succeeds. It is possible for the call to happen multiple times.
  - This only works for **idempotent** operations: something that can be done multiple times without harm.
RPC style implementations

- **Java RMI** (Remote Method Invocation)
- **XML-RPC** and its successor **SOAP**: An RPC protocol that uses XML to encode its calls and HTTP as a transport mechanism
- **Microsoft .NET Remoting**: offers RPC/RMI facilities for distributed systems implemented on the Windows platform
- **CORBA** provides remote procedure invocation through an intermediate layer called the *object request broker*.
- **Facebook's Thrift** protocol and framework
- **Google Protocol Buffers**
Berkeley Sockets

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

Figure 4-14. The socket primitives for TCP/IP.
The Message-Passing Interface (1)

Figure 4-15. Connection-oriented communication pattern using sockets.
### The Message-Passing Interface (2)

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

Figure 4-16. Some of the most intuitive message-passing primitives of MPI.
Message-Oriented Persistent Communication

Message-Oriented Middleware (MOM) or message-queueing systems support persistent asynchronous communication. These systems provide intermediate-term storage capacity for messages without requiring either the sender or receiver to be active during the message transmission.
Figure 4-17. Four combinations for loosely-coupled communications using queues.
Message-Queuing Model (2)

<table>
<thead>
<tr>
<th>Primitive</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Put</td>
<td>Append a message to a specified queue</td>
</tr>
<tr>
<td>Get</td>
<td>Block until the specified queue is nonempty, and remove the first message</td>
</tr>
<tr>
<td>Poll</td>
<td>Check a specified queue for messages, and remove the first. Never block</td>
</tr>
<tr>
<td>Notify</td>
<td>Install a handler to be called when a message is put into the specified queue</td>
</tr>
</tbody>
</table>

Figure 4-18. Basic interface to a queue in a message-queuing system.
Figure 4-19. The relationship between queue-level addressing and network-level addressing.
General Architecture of a Message-Queuing System (2)

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

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

- Email systems are aimed at primarily supporting end users while message-queuing systems are usually not.
- Email systems need not provide guaranteed message delivery, message priorities, logging facilities, efficient multicasting, load balancing, fault tolerance and so on for general usage.
- Message queuing systems can be used to implement email, workflow, groupware and batch processing. But the most important usage is the integration of a collection of databases and applications into a federated system.
IBM’s WebSphere Message-Queuing System

Figure 4-22. General organization of IBM’s message-queueing system.
## Channels

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

**Figure 4-23. Some attributes associated with message channel agents.**
Figure 4-24. The general organization of an MQ queuing network using routing tables and aliases.
Message Transfer (2)

<table>
<thead>
<tr>
<th>Primitive</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MQopen</td>
<td>Open a (possibly remote) queue</td>
</tr>
<tr>
<td>MQclose</td>
<td>Close a queue</td>
</tr>
<tr>
<td>MQput</td>
<td>Put a message into an opened queue</td>
</tr>
<tr>
<td>MQget</td>
<td>Get a message from a (local) queue</td>
</tr>
</tbody>
</table>

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