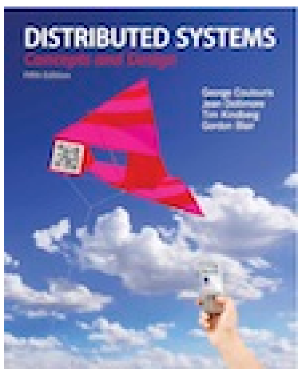
Slides for Chapter 4: Interprocess Communication



From Coulouris, Dollimore, Kindberg and Blair

Distributed Systems:
Concepts and Design

Edition 5, © Addison-Wesley 2012

Figure 4.1
Middleware layers

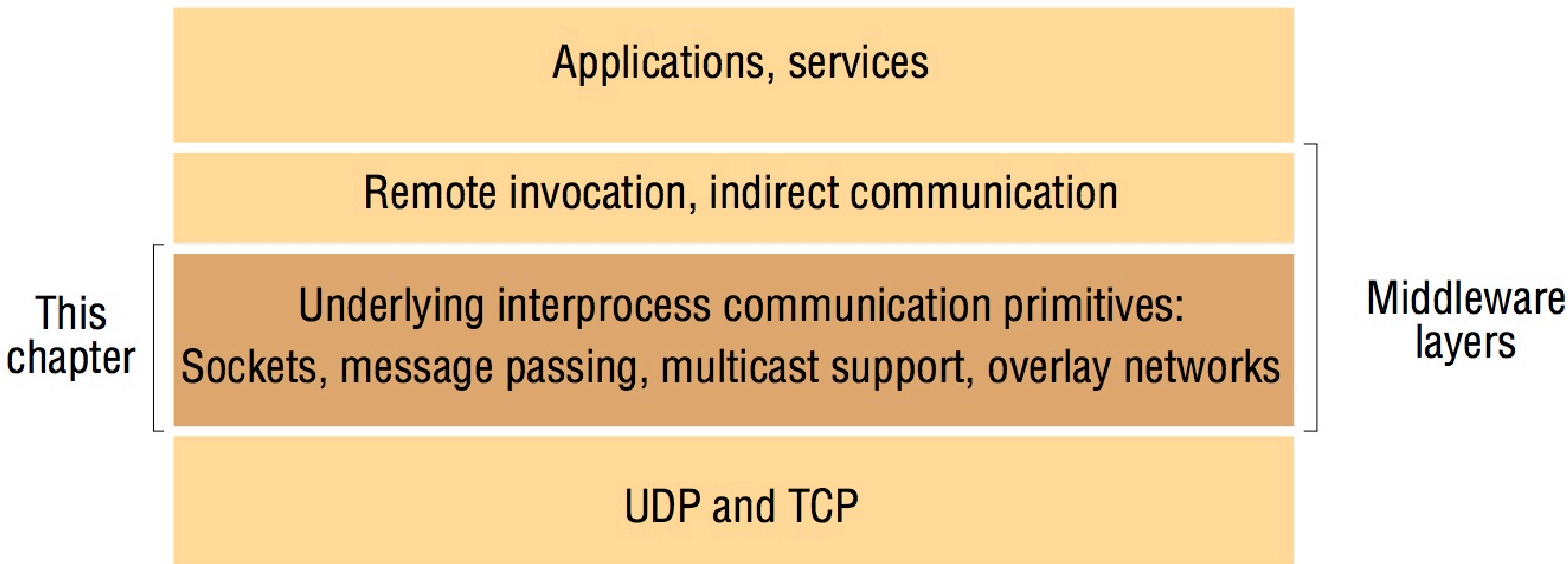


Figure 4.2
Sockets and ports

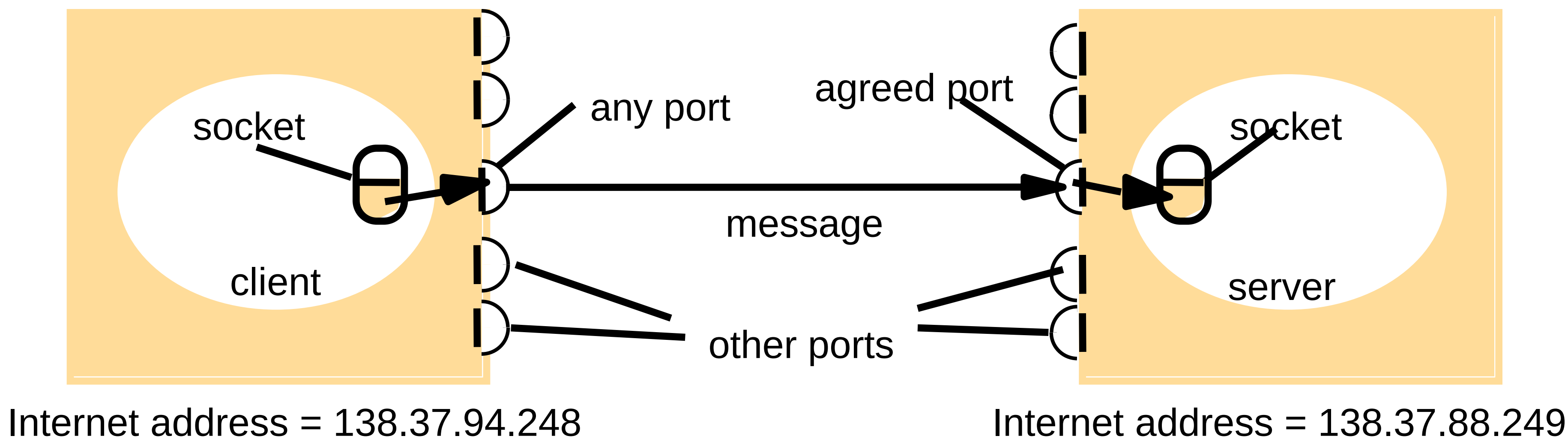


Figure 4.3
UDP client sends a message to the server and gets a reply

```
import java.net.*;
import java.io.*;
public class UDPClient{
    public static void main(String args[]){
		// args give message contents and server hostname
		DatagramSocket aSocket = null;
		  try {
			aSocket = new DatagramSocket();
			byte [] m = args[0].getBytes();
			InetAddress aHost = InetAddress.getByName(args[1]);
			int serverPort = 6789;
			DatagramPacket request = new DatagramPacket(m,  m.length(), aHost, serverPort);
			aSocket.send(request);
			byte[] buffer = new byte[1000];
			DatagramPacket reply = new DatagramPacket(buffer, buffer.length);
			aSocket.receive(reply);
			System.out.println("Reply: " + new String(reply.getData()));
		  }catch (SocketException e){System.out.println("Socket: " + e.getMessage());
		  }catch (IOException e){System.out.println("IO: " + e.getMessage());}
		}finally {if(aSocket != null) aSocket.close();}
   }
}
```

Figure 4.4
UDP server repeatedly receives a request and sends it back to the client

```
import java.net.*;
import java.io.*;
public class UDPServer{
		public static void main(String args[]){
		DatagramSocket aSocket = null;
		   try{
				aSocket = new DatagramSocket(6789);
				byte[] buffer = new byte[1000];
				while(true){
				   DatagramPacket request = new DatagramPacket(buffer, buffer.length);
				  aSocket.receive(request);
				  DatagramPacket reply = new DatagramPacket(request.getData(),
					request.getLength(), request.getAddress(), request.getPort());
				  aSocket.send(reply);
				}
		   }catch (SocketException e){System.out.println("Socket: " + e.getMessage());
		  }catch (IOException e) {System.out.println("IO: " + e.getMessage());}
		}finally {if(aSocket != null) aSocket.close();}
   }
}
```

Figure 4.5
TCP client makes connection to server, sends request and receives reply

```
import java.net.*;
import java.io.*;
public class TCPClient {
        public static void main (String args[]) {
        // arguments supply message and hostname of destination
        Socket s = null;
          try{
                int serverPort = 7896;
                s = new Socket(args[1], serverPort);
                DataInputStream in = new DataInputStream( s.getInputStream());
                DataOutputStream out =
                        new DataOutputStream( s.getOutputStream());
                out.writeUTF(args[0]);          // UTF is a string encoding see Sn 4.3
                String data = in.readUTF();
                System.out.println("Received: "+ data) ;
          }catch (UnknownHostException e){
                        System.out.println("Sock:"+e.getMessage());
          }catch (EOFException e){System.out.println("EOF:"+e.getMessage());
          }catch (IOException e){System.out.println("IO:"+e.getMessage());}
        }finally {if(s!=null) try {s.close();}catch (IOException e){System.out.println("close:"+e.getMessage());}}
        }
}
```

Figure 4.6
TCP server makes a connection for each client and then echoes the client’s request

```
import java.net.*;
import java.io.*;
public class TCPServer {
   public static void main (String args[]) {
		try{
			int serverPort = 7896;
			ServerSocket listenSocket = new ServerSocket(serverPort);
			while(true) {
				Socket clientSocket = listenSocket.accept();
				Connection c = new Connection(clientSocket);
			}
		} catch(IOException e) {System.out.println("Listen :"+e.getMessage());}
   }
}

// this figure continues on the next slide
```

Figure 4.6 continued

```
class Connection extends Thread {
        DataInputStream in;
        DataOutputStream out;
        Socket clientSocket;
        public Connection (Socket aClientSocket) {
          try {
                clientSocket = aClientSocket;
                in = new DataInputStream( clientSocket.getInputStream());
                out =new DataOutputStream( clientSocket.getOutputStream());
                this.start();
           } catch(IOException e)  {System.out.println("Connection:"+e.getMessage());}
        }
        public void run(){
          try {                                   // an echo server
                String data = in.readUTF();
                out.writeUTF(data);
          } catch(EOFException e) {System.out.println("EOF:"+e.getMessage());
          } catch(IOException e) {System.out.println("IO:"+e.getMessage());}
          } finally{ try {clientSocket.close();}catch (IOException e){/*close failed*/}}
        }
}
```

Figure 4.7
CORBA CDR for constructed types

Type	*Representation*
sequence	length (unsigned long) followed by elements in order
string	length (unsigned long) followed by characters in order (can also can have wide characters)
array	array elements in order (no length specified because it is fixed)
struct	in the order of declaration of the components
enumerated	unsigned long (the values are specified by the order declared)
union	type tag followed by the selected member

Figure 4.8
CORBA CDR message

index in sequence of bytes	← 4 bytes →	*notes on representation*
0–3	`5`	*length of string*
4–7	`"Smit"`	*'Smith'*
8–11	`"h___"`	
12–15	`6`	*length of string*
16–19	`"Lond"`	*'London'*
20-23	`"on__"`	
24–27	`1984`	*unsigned long*

The flattened form represents a *Person* struct with value: {'Smith', 'London', 1984}

Figure 4.9
Indication of Java serialized form

Serialized values				*Explanation*
Person	8-byte version number		h0	*class name, version number*
3	int year	java.lang.String name:	java.lang.String place:	*number, type and name of instance variables*
1984	5 Smith	6 London	h1	*values of instance variables*

The true serialized form contains additional type markers; h0 and h1 are handles

Figure 4.10 XML definition of the Person structure

```
<person id="123456789">
        <name>Smith</name>
        <place>London</place>
        <year>1984</year>
        <!-- a comment -->
</person >
```

Figure 4.11 Illustration of the use of a namespace in the *Person* structure

```
<person pers:id="123456789" xmlns:pers = "http://www.cdk5.net/person">
        <pers:name> Smith </pers:name>
        <pers:place> London </pers:place >
        <pers:year> 1984 </pers:year>
</person>
```

Figure 4.12 An XML schema for the *Person* structure

```
<xsd:schema  xmlns:xsd = URL of XML schema definitions >
        <xsd:element name= "person" type ="personType" />
                <xsd:complexType name="personType">
                        <xsd:sequence>
                                <xsd:element name = "name"  type="xs:string
                                <xsd:element name = "place"  type="xs:string
                                <xsd:element name = "year"  type="xs:positive
                        </xsd:sequence>
                        <xsd:attribute name= "id"   type = "xs:positiveInteger
                </xsd:complexType>
</xsd:schema>
```

Figure 4.13
Representation of a remote object reference

32 bits	*32 bits*	*32 bits*	*32 bits*	
Internet address	port number	time	object number	interface of remote object

Figure 4.14
Multicast peer joins a group and sends and receives datagrams

```
import java.net.*;
import java.io.*;
public class MulticastPeer{
        public static void main(String args[]){
         // args give message contents & destination multicast group (e.g. "228.5.6.7")
        MulticastSocket s =null;
         try {
                InetAddress group = InetAddress.getByName(args[1]);
                s = new MulticastSocket(6789);
                s.joinGroup(group);
                byte [] m = args[0].getBytes();
                DatagramPacket messageOut =
                        new DatagramPacket(m, m.length, group, 6789);
                s.send(messageOut);


         // this figure continued on the next slide
```

Figure 4.14 continued

```
// get messages from others in group
        byte[] buffer = new byte[1000];
        for(int i=0; i< 3; i++) {
           DatagramPacket messageIn =
                   new DatagramPacket(buffer, buffer.length);
           s.receive(messageIn);
           System.out.println("Received:" + new String(messageIn.getData()));
        }
        s.leaveGroup(group);
 }catch (SocketException e){System.out.println("Socket: " + e.getMessage());
}catch (IOException e){System.out.println("IO: " + e.getMessage());}
}finally {if(s != null) s.close();}
    }
}
```

Figure 4.15
Types of overlay

Motivation	*Type*	*Description*
Tailored for application needs	Distributed hash tables	One of the most prominent classes of overlay network, offering a service that manages a mapping from keys to values across a potentially large number of nodes in a completely decentralized manner (similar to a standard hash table but in a networked environment).
	Peer-to-peer file sharing	Overlay structures that focus on constructing tailored addressing and routing mechanisms to support the cooperative discovery and use (for example, download) of files.
	Content distribution networks	Overlays that subsume a range of replication, caching and placement strategies to provide improved performance in terms of content delivery to web users; used for web acceleration and to offer the required real-time performance for video streaming [www.kontiki.com].

table continues on the next slide

Figure 4.15 (continued)
Types of overlay

Tailored for network style	Wireless ad hoc networks	Network overlays that provide customized routing protocols for wireless ad hoc networks, including proactive schemes that effectively construct a routing topology on top of the underlying nodes and reactive schemes that establish routes on demand typically supported by flooding.
	Disruption-tolerant networks	Overlays designed to operate in hostile environments that suffer significant node or link failure and potentially high delays.
Offering additional features	Multicast	One of the earliest uses of overlay networks in the Internet, providing access to multicast services where multicast routers are not available; builds on the work by Van Jacobsen, Deering and Casner with their implementation of the MBone (or Multicast Backbone) [mbone].
	Resilience	Overlay networks that seek an order of magnitude improvement in robustness and availability of Internet paths [nms.csail.mit.edu].
	Security	Overlay networks that offer enhanced security over the underling IP network, including virtual private networks, for example, as discussed in Section 3.4.8.

Figure 4.16
Skype overlay architecture

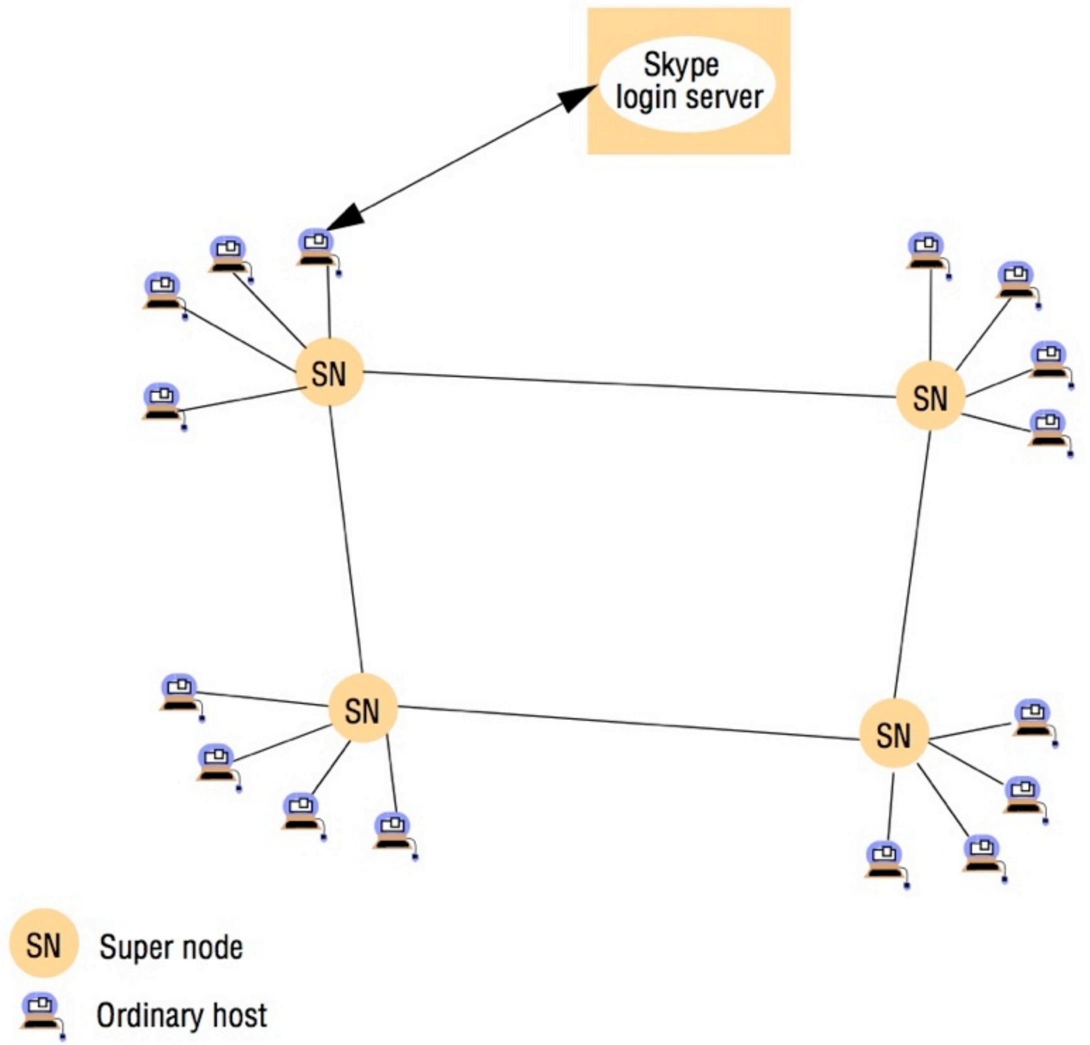


Figure 4.17
An overview of point-to-point communication in MPI

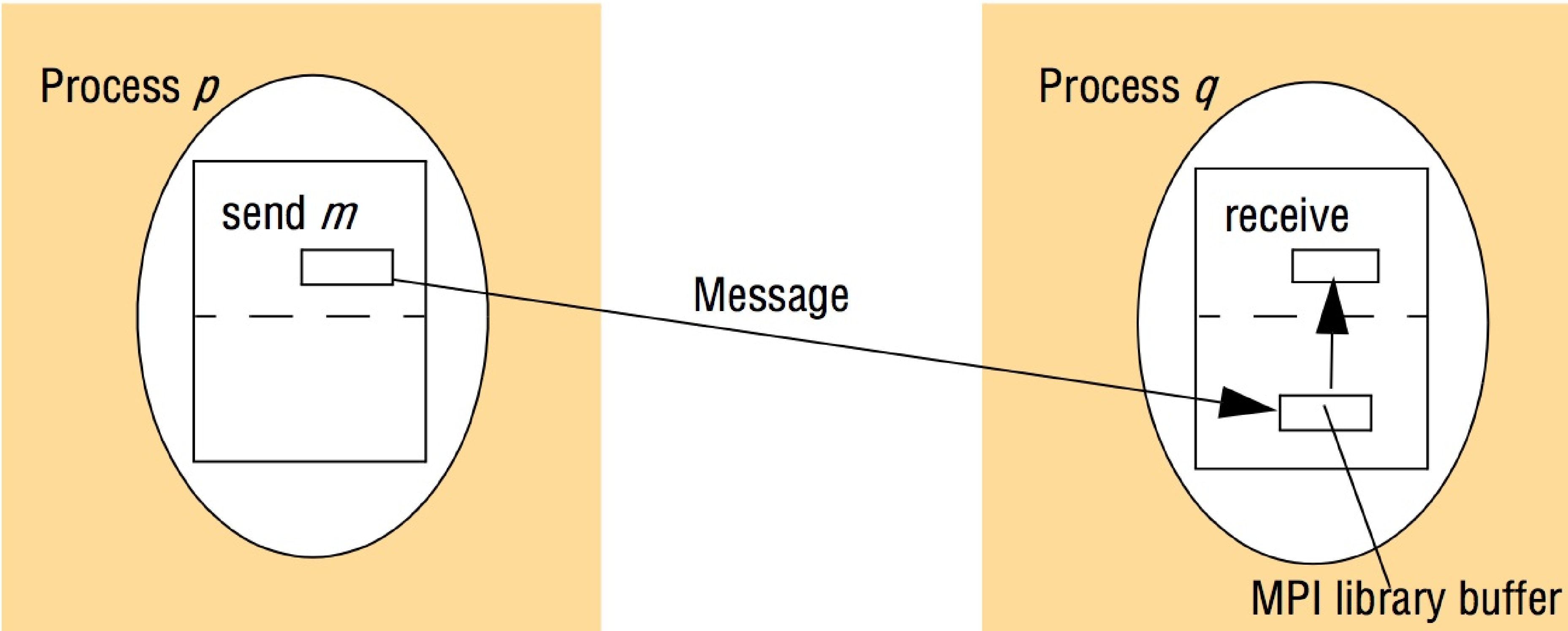


Figure 4.18
Selected send operations in MPI

Send operations	*Blocking*	*Non-blocking*
Generic	*MPI_Send*: the sender blocks until it is safe to return – that is, until the message is in transit or delivered and the sender's application buffer can therefore be reused.	*MPI_Isend*: the call returns immediately and the programmer is given a communication request handle, which can then be used to check the progress of the call via *MPI_Wait* or *MPI_Test*.
Synchronous	*MPI_Ssend*: the sender and receiver synchronize and the call only returns when the message has been delivered at the receiving end.	*MPI_Issend*: as with *MPI_Isend*, but with *MPI_Wait* and *MPI_Test* indicating whether the message has been delivered at the receive end.
Buffered	*MPI_Bsend*: the sender explicitly allocates an MPI buffer library (using a separate *MPI_Buffer_attach* call) and the call returns when the data is successfully copied into this buffer.	*MPI_Ibsend*: as with *MPI_Isend* but with *MPI_Wait* and *MPI_Test* indicating whether the message has been copied into the sender's MPI buffer and hence is in transit.
Ready	*MPI_Rsend*: the call returns when the sender's application buffer can be reused (as with *MPI_Send*), but the programmer is also indicating to the library that the receiver is ready to receive the message, resulting in potential optimization of the underlying implementation.	*MPI_Irsend*: the effect is as with *MPI_Isend*, but as with *MPI_Rsend*, the programmer is indicating to the underlying implementation that the receiver is guaranteed to be ready to receive (resulting in the same optimizations),