Slides for Chapter 4: Interprocess Communication

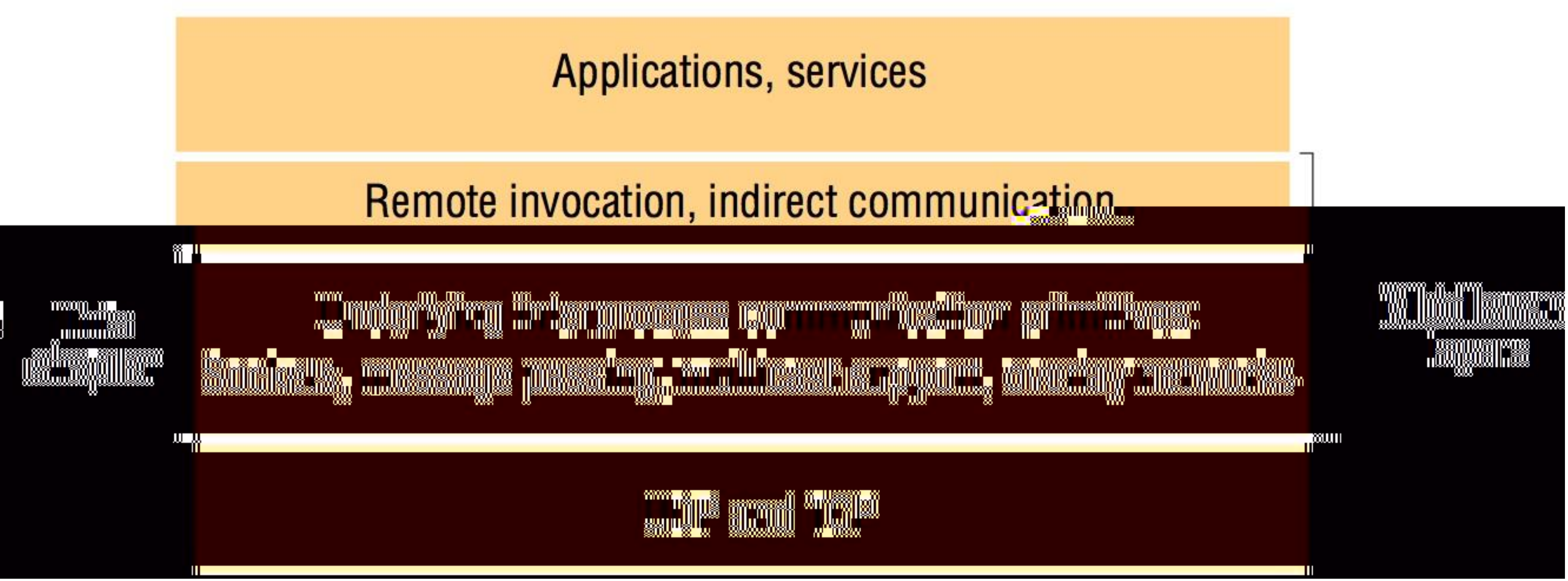
From **Coulouris, Dollimore, Kindberg and Blair**

Distributed Systems: Concepts and Design

Overview of Chapter

- Introduction to Interprocess Communication
- API for the Internet Protocols
- External data Representation and Marshalling
- Multicast Communication
- Overlay Networks
- Case Study: MPI

Figure 4.1
Middleware layers



Introduction

Underlying Interprocess Communication Primitives:

- API interfaces to TCP and UDP transport-level protocols
- UDP interface provides *message passing* abstraction
- Message consists of *datagrams* (or packets)
- Destination specified by a *socket* (indirect reference to a port)
- TCP interface provides *two-way stream* abstraction
- *Stream* of data items with no message boundaries
- Basis for *producer-consumer* communication
- Received data items are *queued* until consumed

Overview of Chapter

Introduction to Interprocess Communication

APIs for the Internet Protocols

External data Representation and Marshalling

Multicast Communication

Overlay Networks

Case Study: MPI

APIs for Internet Protocols

Characteristics of interprocess communication:

- Basic operations are *send* and *receive*
- Message is a *sequence of bytes* sent to a *destination*
- Process *at source* sends the message
- Process at *destination* receives the message
- A *queue* is associated with each message destination

APIs for Internet Protocols

Synchronous vs. asynchronous communication:

- In *synchronous communication* both *send* and *receive* are blocking operations
- Sending process is blocked until receive is issued
- Receiving process is blocked until message arrives
- In *asynchronous communication* sending process sends message and continues (send is non-blocking)
- Receive operation has two variants: blocking and non-blocking
- Blocking variant waits till message is received
- Non-blocking variant issues receive and continues - - gets notification that buffer for receiving message is filled by polling or interrupt
- Blocking receive is less complex can block a thread to wait for message while new thread continues the process

APIs for Internet Protocols

Message destinations:

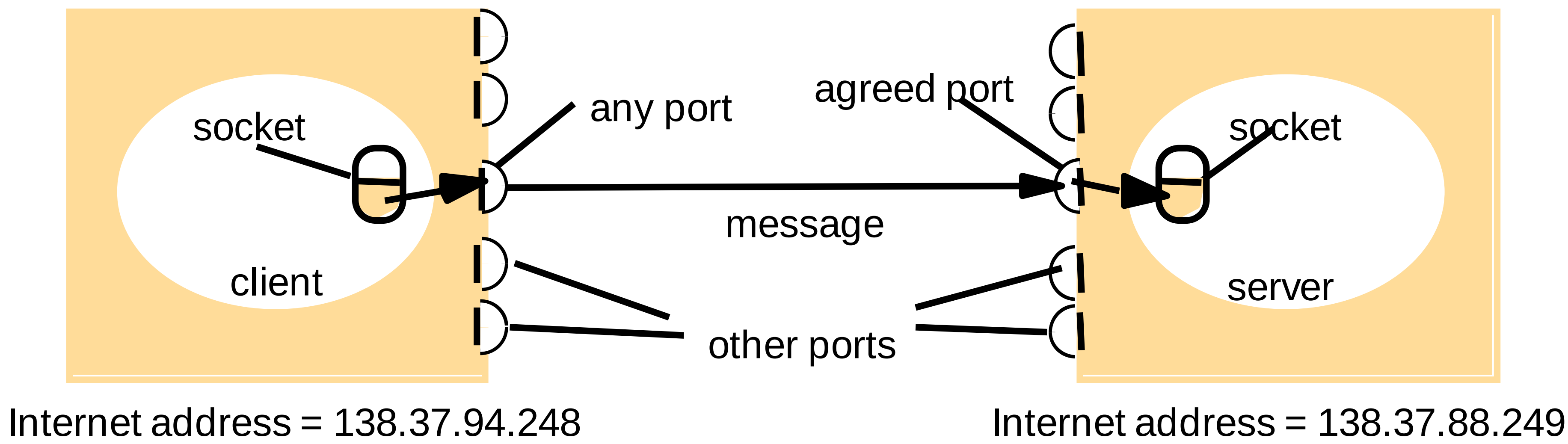
- Message sent to (Internet address, local port)
- Port typically associated with one receiving process
- Multiple sending processes can send to same remote port
- Remote process can receive messages at multiple ports
- Server processes publish their associated ports for use by clients
- To allow servers to have *location transparency* server can be accessed by name   name server matches name to server at runtime

APIs for Internet Protocols

Sockets:

- Message sent from one socket to another socket
- Socket specified by (Internet address, local port)
- Socket typically associated with one receiving process
- Socket can be used for both sending and receiving
- Large number of possible port numbers (e.g. 2^{16}) on each computer
- Each socket associated with either TCP or UDP

Figure 4.2
Sockets and ports



APIs for Internet Protocols

Java API for Internet addresses:

Java class *InetAddress*

Can get Internet address using host name (by access DNS server)

Ex: *InetAddress aHost = InetAddress.getByName*

Socket can be used for both sending and receiving

Large number of possible port numbers (e.g. 2^{16}) on each computer

Each socket associated with either TCP or UDP

APIs for Internet Protocols

UDP datagram communication:

In UDP sender and receiver must bind to sockets, some datagrams (packets) can be lost or out of order application must check on these errors

Receive method will return Internet address and port of sender

Message size limit datagrams over limit are truncated

Blocking sockets generally use non-blocking send and blocking receive if no server is bound to a receiving socket, messages are discarded

Timeouts can be set on sockets with blocking receive

Java provides two classes for UPD communication *DatagramPacket* and *DatagramSocket*

A *DatagramPacket* object holds a datagram (arrayOfBytes, messageLength, InternetAddress, portNumber)

APIs for Internet Protocols

UDP datagram communication (cont.):

A *DatagramSocket* object supports sending and receiving of datagrams

Has the following methods (operations):

send and *receive*

setSoTimeout

connect

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();}
   }
}
```

APIs for Internet Protocols

TCP stream communication:

Hides low-level networking characteristics such as:

Message sizes

Lost messages

Flow control using blocking if sender/receiver have different speeds

Message duplication and ordering

Message destinations overhead to establish connection between client and server once connection is established two-way communication via streams is possible

Client requests connection to server once connected, both are *peers*

Both sockets have *input* and *output* streams

Closing socket means no more writing to connection

APIs for Internet Protocols

Issues with stream communication:

- Matching of data items if processes do not cooperate correctly
- Blocking when queue buffers are full
- Threads can be used to avoid blocking of processes
- Failure issues: cannot distinguish between network and process failure, cannot tell when messages are received

TCP connections with fixed port numbers:

- HTTP (Hypertext Transfer Protocol)
- FTP (File Transfer Protocol)
- Telnet (terminal sessions to remote computer)
- SMTP (Simple mail Transfer Protocol)

APIs for Internet Protocols

Java API for TCP streams:

- Java provides two classes for TCP stream communication *ServerSocket* and *Socket*
- A *ServerSocket* object represents a server socket listening for *connect* requests from clients *accept* method creates a *Socket* object for communicating with a client
- A pair of *Socket* objects represents a stream connection for communication
- Methods for *Socket* objects include *getInputStream* method (returns *InputStream* object which can construct *DataInputStream*) and *getOutputStream* method (returns *OutputStream* object which can construct *DataOutputStream*)

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

```
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*/}}
        }
}
```

Overview of Chapter

Introduction to Interprocess Communication

API for the Internet Protocols

External data Representation and Marshalling

Multicast Communication

Overlay Networks

Case Study: MPI

External data representation and marshalling

Data must be *flattened* into a byte string for transmission

Marshalling is the process of converting data items to a suitable form for transmission from the source

Unmarshalling is the inverse process of assembling a transmitted message into its data items at the destination

Some standard representation tools exist:

CORBA common data representation (CDR)

Java object serialization

XML or JSON (Java Script Object Notation)

Assume middleware layer first two use binary representation, XML and JSON use textual representation

Google uses another technique protocol buffers

External data representation and marshalling

CORBA CDR:

- CORBA IDL (Interface definition Language) can describe object structure
- *Marshalling* and *unmarshalling* operations can be generated automatically from the IDL by CORBA middleware

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*

External data representation and marshalling

Java object serialization:

- Used with Java RMI (Remote Method Invocation)
- Assumes both client and server processes are written in Java
- References in an object are serialized as *handles*
- Classes used include *ObjectOutputStream* and *ObjectInputStream*
- Can do generic serialization/deserialization (marshalling/unmarshalling) uses *reflection* to discover object structure

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

External data representation and marshalling

XML (and JSON):

- Self-describing textual object representation
- *Tags* allow sender and receiver to agree on meaning/structure of transmitted objects
- Uses elements and attributes in a hierarchical (tree) structure
- *XML schemas* can provide a common set of tags

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:positiv
                        </xsd:sequence>
                        <xsd:attribute name= "id"   type = "xs:positiveInteger
                </xsd:complexType>
</xsd:schema>
```

External data representation and marshalling

Remote object references:

- Work with CORBA and Java that support a distributed object model
- Allows to refer to a remote object

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

Overview of Chapter

Introduction to Interprocess Communication

API for the Internet Protocols

External data Representation and Marshalling

Multicast Communication

Overlay Networks

Case Study: MPI

Multicast Communication

Multicast operation: a process sends a message to members of a group of processes

Can be used for:

- Fault tolerance based on replicated services
- Discovering services in spontaneous networking
- Better performance through replicated data
- Propagation of event notifications

Multicast Communication

IP multicast:

- Class D internet address can specify a *multicast group*
- Membership in group is dynamic sockets can be added to group
- Available only through UDP
- Groups can be permanent or temporary

Java API to *MulticastSocket:*

- Subclass of *DatagramSocket*
- Methods include *joinGroup* and *leaveGroup*

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();}
  }
}
```

Overview of Chapter

- Introduction to Interprocess Communication
- API for the Internet Protocols
- External data Representation and Marshalling
- Multicast Communication
- Overlay Networks
- Case Study: MPI

Overlay (virtual) networks

Constructing a virtual network over and existing network

For specific applications:

More efficient protocols tailored to the application

Provide additional features

Example: Skype

Figure 4.15
Types of overlay

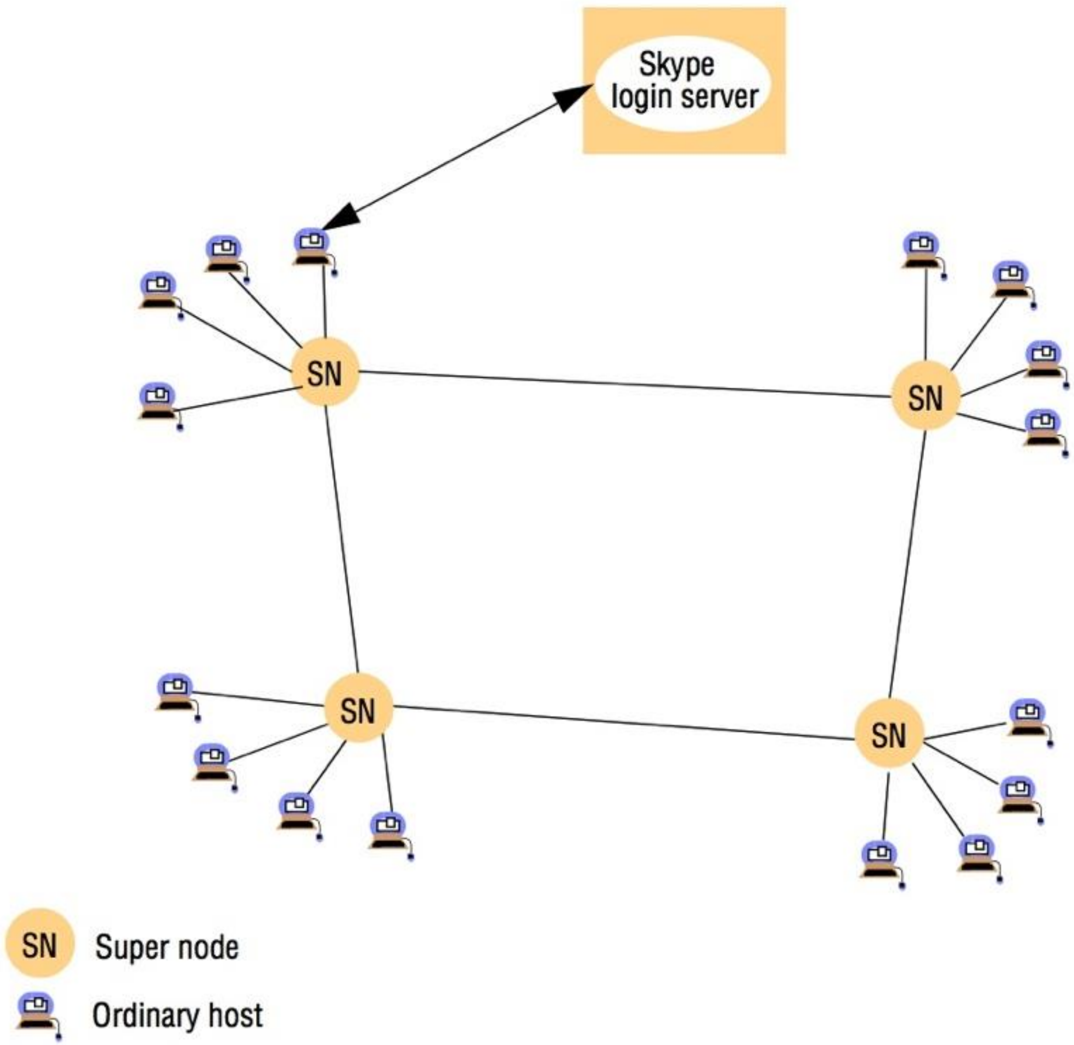
Motivation	*Type*	*Description*
Tailored for application needs	Distributed hash tables	One of the most prominent classes of overlay network, offering a service that manages a mapping from keys to values across a potentially large number of nodes in a completely decentralized manner (similar to a standard 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.
	sruption-tolerant tworks	Overlays designed to operate in hostile environments that suffer significant node or link failure and potentially high delays.
Offering additional features	ast	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].
	ence	Overlay networks that seek an order of magnitude improvement in robustness and availability of Internet paths [nms.csail.mit.edu].
	ty	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



Overview of Chapter

Introduction to Interprocess Communication

API for the Internet Protocols

External data Representation and Marshalling

Multicast Communication

Overlay Networks

Case Study: MPI (Message passing Interface)

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

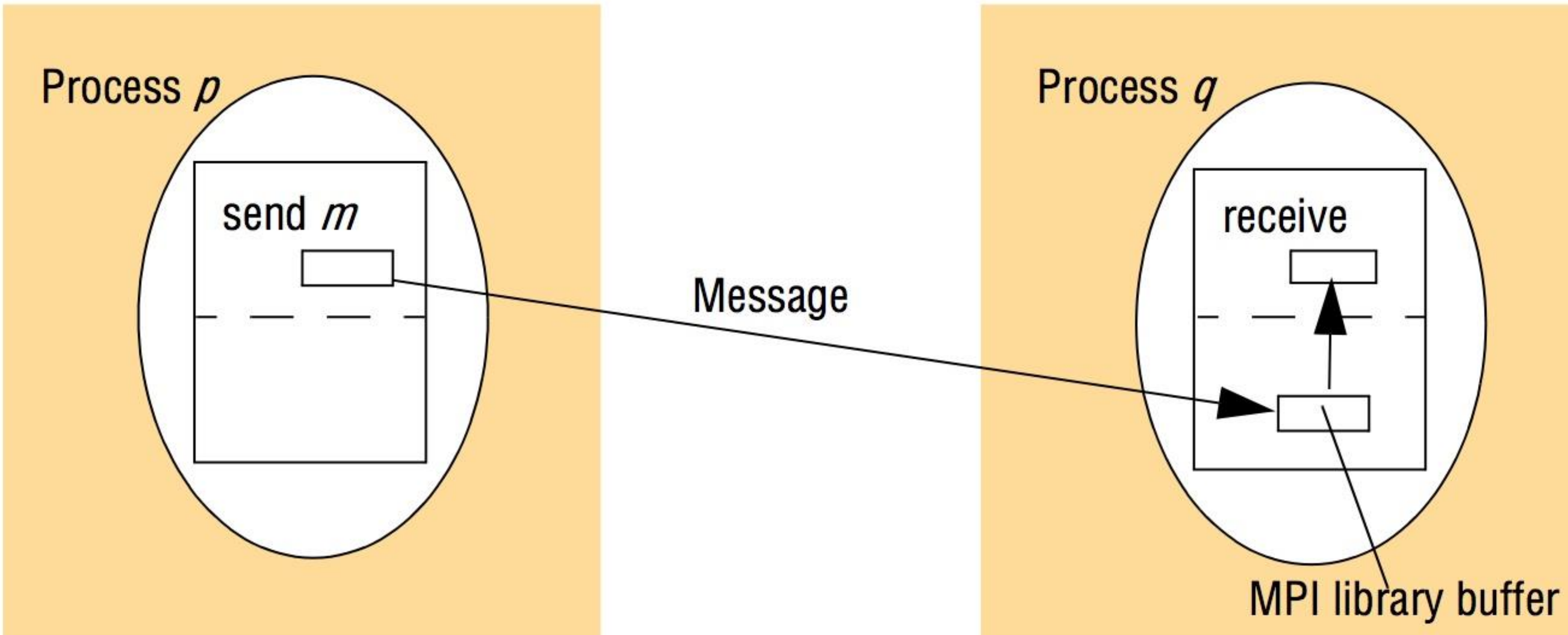


Figure 4.18
Selected send operations in MPI

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),