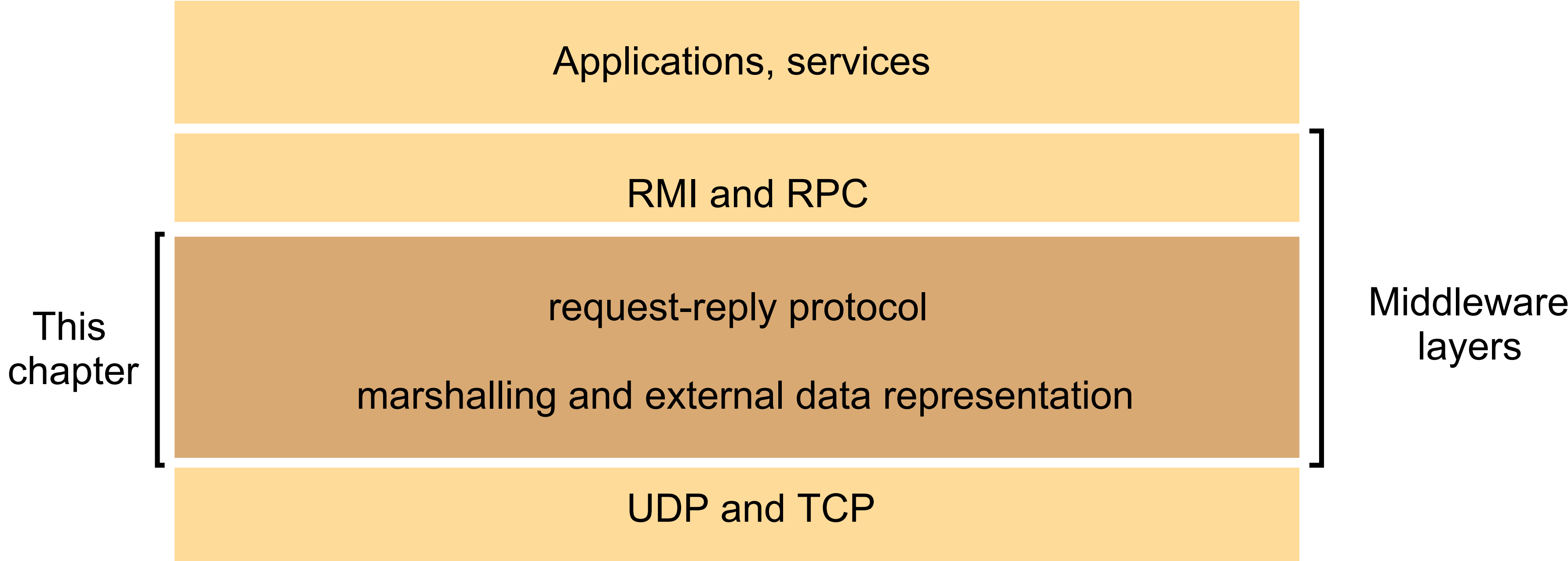
Slides for Chapter 4: Interprocess Communication

From **Coulouris, Dollimore and Kindberg**

Distributed Systems: Concepts and Design

Edition 4, © Addison-Wesley 2005

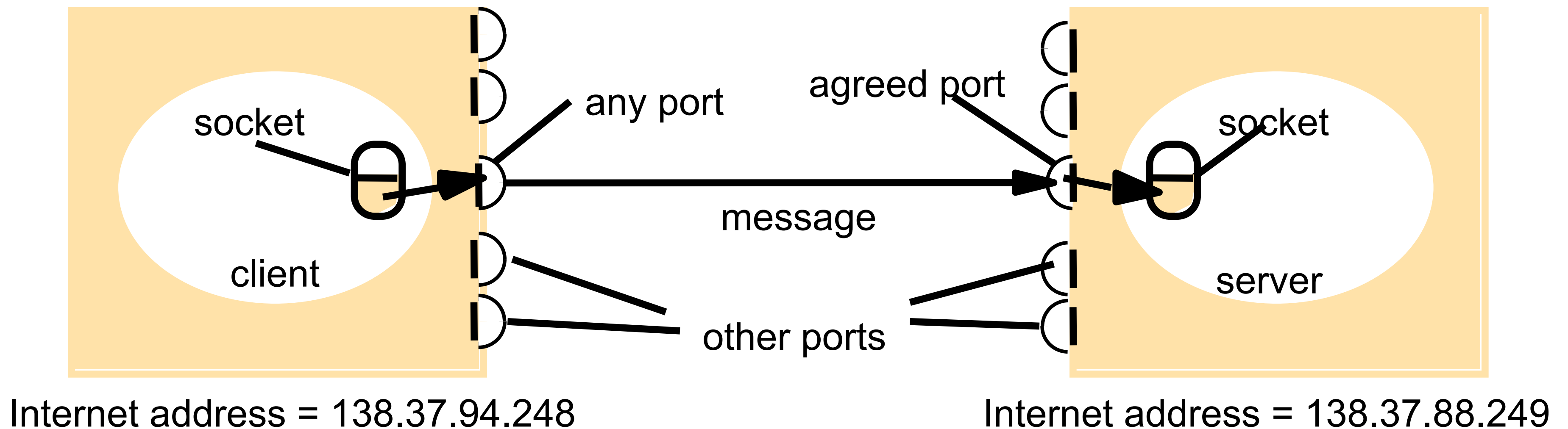
Figure 4.1
Middleware layers



API for InterProcess Communications

- synchronous and asynchronous communication
  - blocking send: waits until the corresponding receive is issued
  - non-blocking send: sends and moves on
  - blocking receive: waits until the msg is received
  - non-blocking receive: if the msg is not here, moves on
  - synchronous: blocking send and receive
  - asynchronous: non-blocking send and blocking or non-blocking receive
- Message Destination
  - IP address + port: one receiver, many senders
  - Location transparency
    - name server or binder: translate service to location
    - OS (e.g. Mach): provides location-independent identifier mapping to lower-lever addresses
  - send directly to processes (e.g. V System)
  - multicast to a group of processes (e.g. Chorous)
- Reliability
- Ordering

- programming abstraction for UDP/TCP
- originated from BSD UNIX



API for Internet Protocols (3): UDP Datagram

- message size: up to 2^{16}, usually restrict to 8K
- blocking: non-blocking send, blocking receive
- timeouts: timeout on blocking receive
- receive from any: doesn't specify sender origin (possible to specify a particular host for send and receive)
- failure model:
  - omission failures: can be dropped
  - ordering: can be out of order
- use of UDP:
  - DNS
  - less overhead: no state information, extra messages, latency due to start up

Sockets used for datagrams (in C)

Sending a message

```
s = socket(AF_INET, SOCK_DGRAM, 0)
•
•
bind(s, ClientAddress)
•
•
sendto(s, "message", ServerAddress)
```

Receiving a message

```
s = socket(AF_INET, SOCK_DGRAM, 0)
•
•
bind(s, ServerAddress)
•
•
amount = recvfrom(s, buffer, from)
```

ServerAddress and *ClientAddress* are socket addresses

Figure 4.3
Java 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,  args[0].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 Java 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();}
  }
}
```

API for Internet Protocols (6): TCP stream

- message size: unlimited
- lost messages: sequence #, ack, retransmit after timeout of no ack
- flow control: sender can be slowed down or blocked by the receiver
- message duplication and ordering: sequence #
- message destination: establish a connection, one sender-one receiver, high overhead for short communication
- matching of data items: two processes need to agree on format and order (protocol)
- blocking: non-blocking send, blocking receive (send might be blocked due to flow control)
- concurrency: one receiver, multiple senders, one thread for each connection
- failure model
  - checksum to detect and reject corrupt packets
  - sequence # to deal with lost and out-of-order packets
  - connection broken if ack not received when timeout
    - could be traffic, could be lost ack, could be failed process..
    - can't tell if previous messages were received
- use of TCP: http, ftp, telnet, smtp

Sockets used for streams (in C)

Requesting a connection

```
s = socket(AF_INET, SOCK_STREAM,0)
•
•
connect(s, ServerAddress)
•
•
write(s, "message", length)
```

Listening and accepting a connection

```
s = socket(AF_INET, SOCK_STREAM,0)
•
bind(s, ServerAddress);
listen(s,5);
•
sNew = accept(s, ClientAddress);
•
n = read(sNew, buffer, amount)
```

ServerAddress and *ClientAddress* are socket addresses

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*/}}
        }
}
```

Case study: Message Passing Interface

- MPI: standard API for message passing
- introduced in 1994 especially for HPC
- designed to be simple, practical, efficient and portable

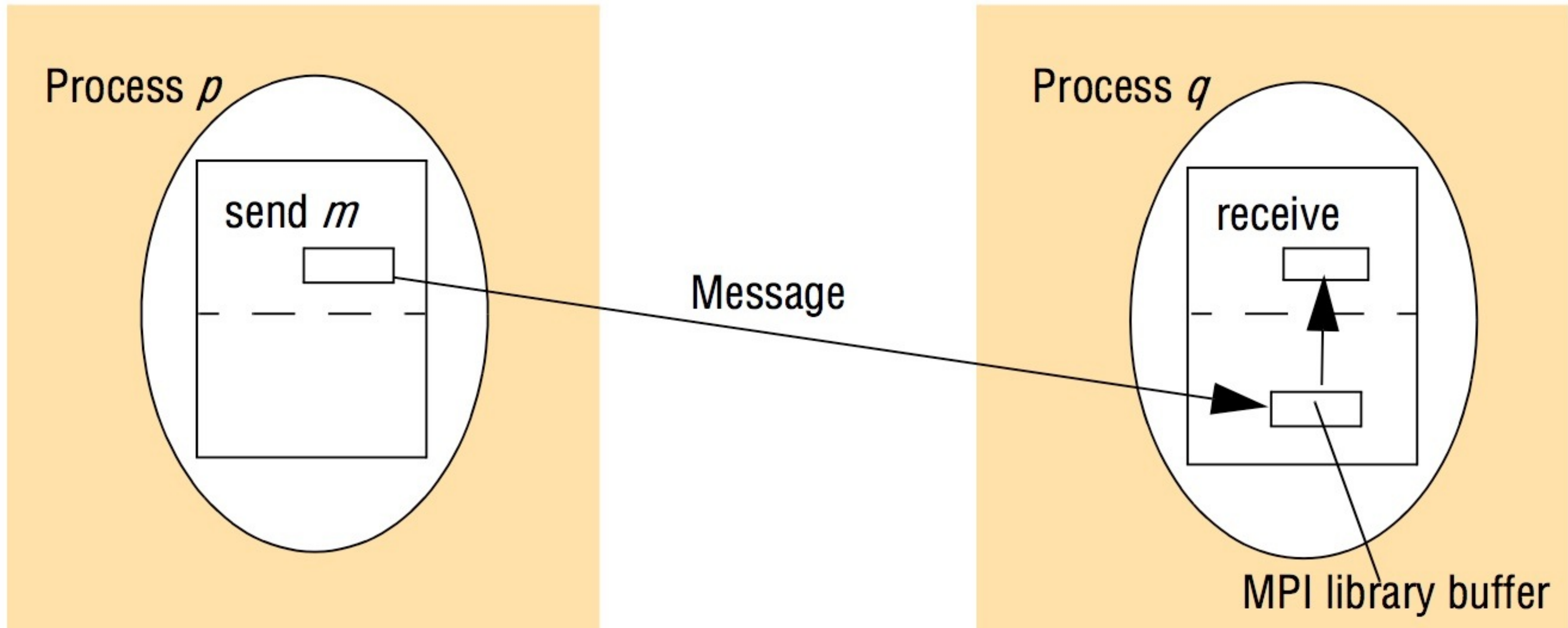


Figure 4.18
Selected send operations in MPI

- Despite simplicity, a lot of choice and control is provided to the programmer by several different calls
- Ported to many languages

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),

External Data Representation

- different ways to represent int, float, char… (internally)
- byte ordering for integers
  - big-endian: most significant byte first
  - small-endian: least significant byte first
- We need a standard external data representation
  - **marshal** before sending, **unmarshal** before receiving
- send in sender's format and indicates what format, receivers translate if necessary
- **External data representation**
  - SUN's External data representation (XDR)
  - CORBA's Common Data Representation (CDR)
  - Java's object serialization
  - JavaScript Object Notation (JSON)
  - PER/ASN.1
  - ASCII (XML+XSD/XSLT, HTTP)

External Data Representation: Java serialization

- serialization and de-serialization are automatic in arguments and return values of Remote Method Interface (RMI)
- flattened to be transmitted or stored on the disk
  - write class information, types and names of instance variables
  - new classes, recursively write class information, types, names...
  - each class has a handle, for subsequent references
  - values are in Universal Transfer Format (UTF)

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*
1934	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: XML

- Extensible markup language (XML)
  - User-defined tags (vs. HTML has a fixed set of tags)
- different applications agree on a different set of tags
- E.g. SOAP for web services, tags are published in WSDL
- Tags are in plain text (not binary format)—not space efficient
- Binary data needs to be converted (base64) before embedding, (or can be attached as MIME)

```
<person id="123456789">
        <name>Smith</name>
        <place>London</place>
        <year>1934</year>
        <!-- a comment -->
</person >
```

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

- Name clashes within an application
- **Namespaces**: a set of names for a collection of element types and attributes
- xmlns: xml namespace
  pers: name of the name space (used as a prefix) http://www.cdk4.net/person :location of schema

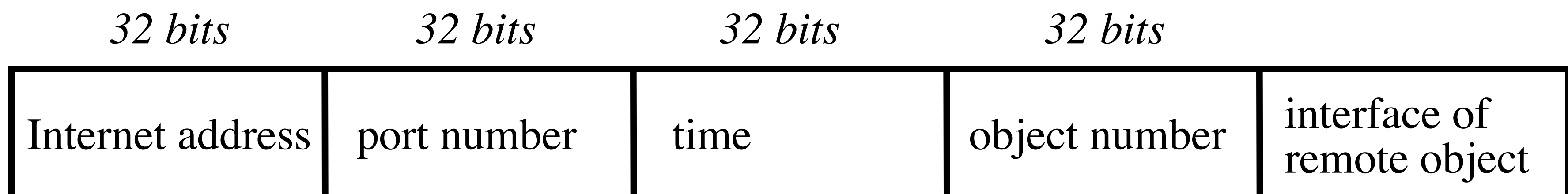
```
<person pers:id="123456789" xmlns:pers = "http://www.cdk4.net/person">
 <pers:name> Smith </pers:name>
 <pers:place> London </pers:place >
 <pers:year> 1934 </pers:year>
</person>
```

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

External Data Representation: Remote object reference

- call methods on a remote object (CORBA, Java)
- unique reference in the distributed system
- Reference = IP address + port + process creation time + local object # in a process + interface
- Port + process creation time -> unique process
- Address can be derived from the reference
- Objects usually don't move
  - is there a problem if the remote object moves?
- name of interface: what interface is available?



Example IDL

```
Hello.idl

module HelloApp
{
   interface Hello
   {
   string sayHello();
   oneway void shutdown();
   };
};
```

Example IDL

```
HelloClient.java

import HelloApp.*;
import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.*;
import org.omg.CORBA.*;

public class HelloClient
{
  static Hello helloImpl;

  public static void main(String args[])
    {
      try{
        // create and initialize the ORB
        ORB orb = ORB.init(args, null);

        // get the root naming context
        org.omg.CORBA.Object objRef =
            orb.resolve_initial_references("NameService");
        // Use NamingContextExt instead of NamingContext. This is
        // part of the Interoperable naming Service.
        NamingContextExt ncRef = NamingContextExtHelper.narrow(objRef);

        // resolve the Object Reference in Naming
        String name = "Hello";
        helloImpl = HelloHelper.narrow(ncRef.resolve_str(name));

        System.out.println("Obtained a handle on server object: " + helloImpl);
        System.out.println(helloImpl.sayHello());
        helloImpl.shutdown();

        } catch (Exception e) {
          System.out.println("ERROR : " + e) ;
          e.printStackTrace(System.out);
          }
    }

}
```

Group communication: Multicast

- multicast: useful for:
  - fault tolerance based on replicated services
    - requests multicast to servers, some may fail, the client will be served
  - discovering services
    - multicast to find out who has the services
  - better performance through replicated data
    - multicast updates
  - event notification
    - new items arrived, advertising services

Group communication: IP Multicast

- class D addresses, first four bits are 1110 in IPv4
- uses UDP
- Join a group via socket binding to the multicast address
- messages arriving on a host deliver them to all local sockets in the group
- multicast routers: route messages to out-going links that have members (implementing protocols like PIM - not widespread)
- multicast address allocation
  - permanent (e.g. 224.0.1.1 is for NTP)
  - temporary:
    - no central registry by IP (one addr might have different groups)
    - use (time to live) TTL to limit the # of hops, hence distance
    - tools like sd (session directory) can help manage multicast addresses and find new ones dynamically

- UDP-level reliability: missing, out-of-order...
- Effects on
  - fault tolerance based on replicated services: ordering of the requests might be important, servers can be inconsistent with one another
  - discovering services: not too problematic
  - better performance through replicated data: loss and out-of-order updates could yield inconsistent data, sometimes this may be tolerable
  - event notification: not too problematic

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 4.20 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();}
  }
}
```

Networks virtualization: Overlay networks

- Specific applications needs specific routing protocol and addressing spaces
  - peer-to-peer file sharing, multicast, Skype...
- It would be impractical to modify the TCP/IP suite
- Solution: **network virtualization (overlays)**
  - construction of (many) virtual networks over Internet
  - each virtual (or *overlay*) network is dedicated to a specific application and coexists with others
  - each virtual network is optimized for the specific application, with proper address space, routing protocols…
  - traffic in the overlay network is actually encapsulated into the underlaying network

Overlay network

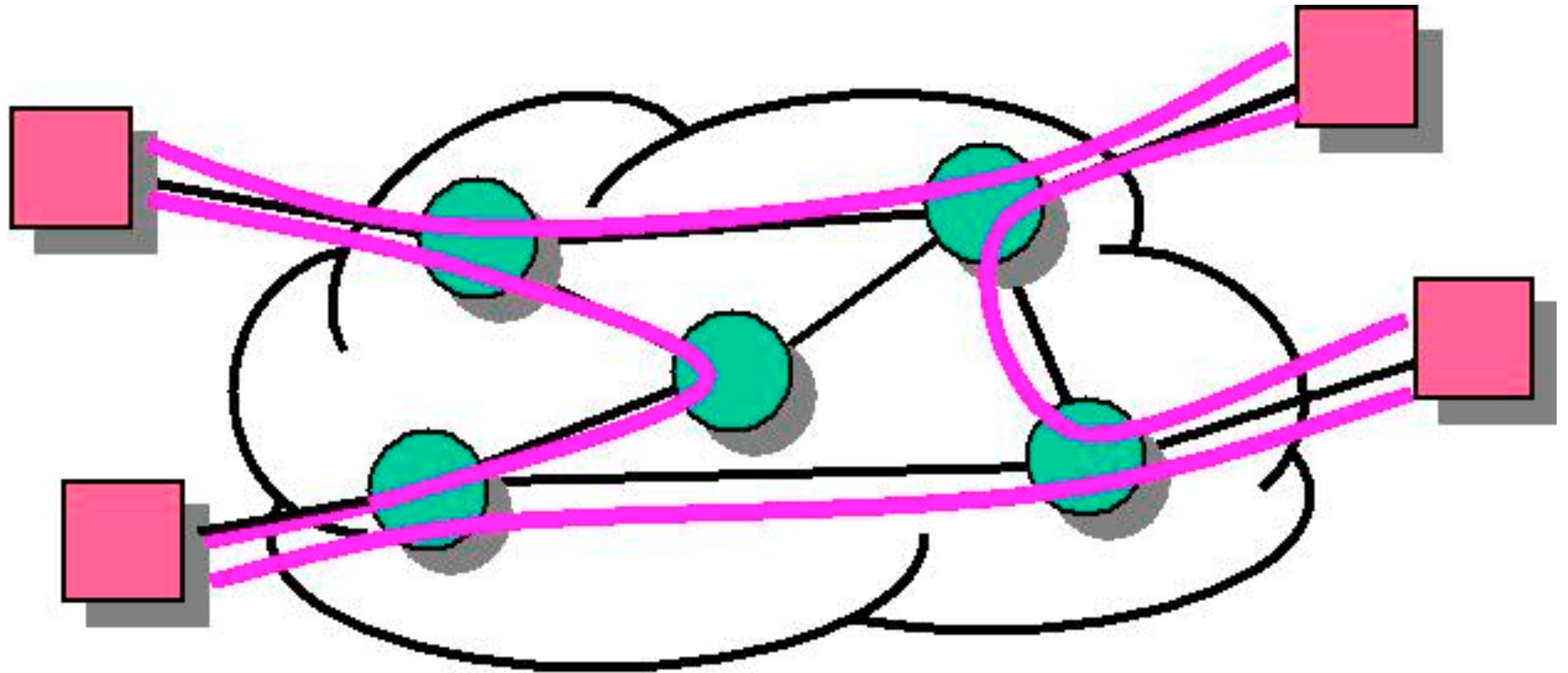


Figure 1.1. An overlay network in the Internet is a virtual network that is implemented on top of a network of routers and links. Each logical link of the virtual network consists of a complete end-to-end unicast route. Nodes in the overlay network can be hosts, routers, servers, or applications.

- traffic in the overlay network is encapsulated into the underlaying IP network
- which is encapsulated in the underlying physical network
- the overlay network sees the IP network as “physical”
- the overlay network can implement its routing protocols, algorithms, QoS, policies, etc

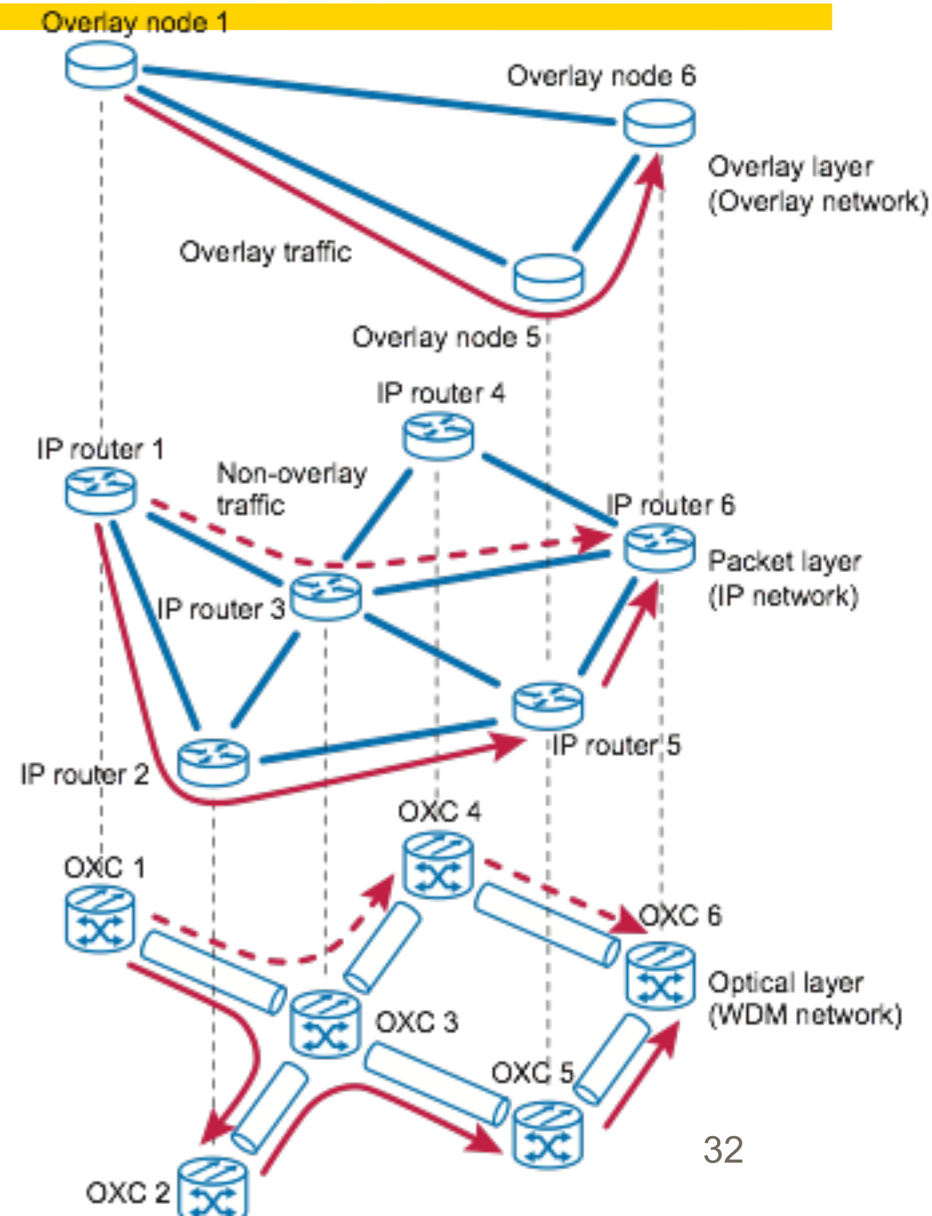


Figure 4.15 Types of overlay

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.

Case study: Skype

- Skype is a p2p application offering VoIP and other.
- Developed by KaZaA in 2003, then stand-alone
- Widely deployed
- Design undisclosed; only reverse engineering
- Architecture: p2p with hosts and super-nodes
- In the original architecture, a host is automatically promoted supernode if it satisfies some constraints on bandwidth, connection, uptime…
- User connection: users are authenticated via a well-known login server, then connect to a close supernode (hosts keep a list of supernodes)
- Supernodes are used for look up of users (search or call). ~8 supernodes are contacted, in 3-6 seconds
- Voice connection: when required user is found, Skype establishes between the two parties
  - a TCP connection for control and signalling
  - a UDP or TCP connection for audio/video

Figure 4.16
Skype overlay architecture