Distributed Systems - Interprocess Communication

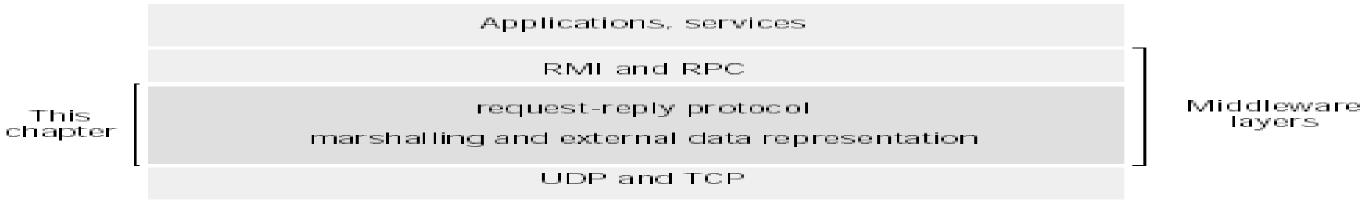
- **4. Topics**
- **4.1 Intro**
  4.2 API for Internet Protocols
- **4.3 External data representation**
- **4.4 Client-Server Communication**
- **4.5 Group communication**
- **4.6 Unix – An example**

Interprocess Communication – 4.1 Introduction

- Focus:
  - Characteristics of protocols for communication between processes to model distributed computing architecture
    - Effective means for communicating objects among processes at language level
  - Java API
    - Provides both datagram and stream communication primitives/interfaces – building blocks for communication protocols
  - Representation of objects
    - providing a common interface for object references
  - Protocol construction
    - Two communication patterns for distributed programming: C-S using RMI/RPC and Group communication using ‘broadcasting’
  - Unix RPC

Interprocess Communication – 4.1 Introduction

Figure 4.1 Middleware layers



- **In Chapter 3**, we covered Internet transport (TCP/UDP) and network (IP) protocols – without emphasizing how they are used at programming level
- **In Chapter 5**, we cover RMI facilities for accessing remote objects' methods AND the use of RPC for accessing the procedures in a remote server
- **Chapter 4** is on how TCP and UDP are used in a program to effect communication via socket (e.g., Java sockets) – the Middle Layers – for object request/reply invocation and parameter marshalling/representation, including specialized protocols that avoid redundant messaging (e.g., using piggybacked ACKs)

Interprocess Communication – 4.2 API for Internet

- Characteristics of IPC – message passing using send/receive facilities for sync and addressing in distributed programs
- Use of sockets as API for UDP and TCP implementation – much more specification can be found at *java.net*
- Synchronous
  - Queues at remote sites are established for message placement by clients (sender). The local process (at remote site) dequeues the message on arrival
  - If synchronous, both the sender and receiver must 'rendezvous' on each message, i.e., both *send* and *receive* invocations are *blocking-until*
- Asynchronous communication
  - *Send* from client is non-blocking and proceeds in parallel with local operations
  - *Receive* could be <u>non-blocking</u> (requiring a background buffer for when message finally arrives, with notification – using interrupts or polling) AND if <u>blocking</u>, perhaps, remote process needs the message, then the process must wait on it
  - Having both sync/async is advantageous, e.g., one thread of a process can do blocked-receive while other thread of same process perform non-block receive or are active – simplifies synchronization. In general non-blocking-receive is simple but complex to implement due to messages arriving out-of-order in the background buffer

Interprocess Communication – 4.2 API for Internet

- Message destinations
  - Typically: send(IP, port#, buffer) – a many-to-one (many senders to a single receiving port), except multicast, which is many-to-group.
  - Possibility: receiving process can have many ports for different message types
  - Server processes usually publish their service-ports for clients
  - Clients can use static IP to access service-ports on servers (limiting, sometimes), but could use location-independent IP by
    - using name server or binder to bind names to servers at run-time – for relocation
    - Mapping location-independent identifiers onto lower-level address to deliver/send messages – supporting service migration and relocation
  - IPC can also use 'processes' in lieu of 'ports' for services but ports are flexible and also (a better) support for multicast or delivery to groups of destinations

Interprocess Communication – 4.2 API for Internet

- Reliability
  - Validity: transmission is reliable if packets are delivered despite some drops/losses, and unreliable even if there is a single drop/loss
  - Integrity: message must be delivered uncorrupted and no duplicates

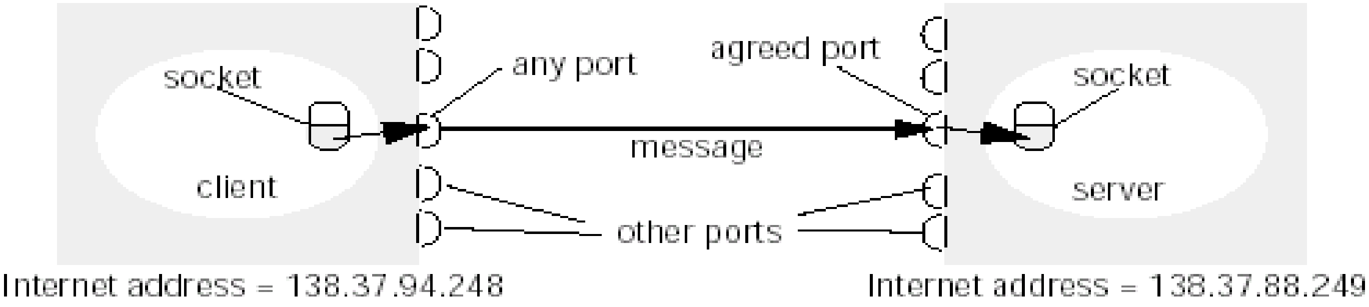
- Ordering
  - Message packets, even if sent out-of-order, must be reordered and delivered otherwise it is a failure of protocol

Interprocess Communication – 4.2 API for Internet

- Sockets
  - Provide an abstraction of endpoints for both TCP and UDP communication
  - Sockets are bound to ports on given computers (via the computer’s IP address)
  - Each computer has 2^{16} possible ports available to local processes for receiving messages
  - Each process can designate multiple ports for different message types (but such designated ports can’t be shared with other processes on the same computer – unless using IP multicast)
  - Many processes in the same computer can deliver to the same port (many-to-one), however
  - Sockets are typed/associated with either TCP or UDP

Interprocess Communication – 4.2 API for Internet

Figure 4.2 Sockets and ports


socket
client
any port
agreed port
socket
server
message
other ports
Internet address = 138.37.94.248
Internet address = 138.37.88.249

Interprocess Communication – 4.2 API for Internet

- Java API for IPs

  - For either TCP or UDP, Java provides an InetAddress class, which contains a method: getByName(DNS) for obtaining IP addresses, irrespectively of the number of address bits (32 bits for IPv4 or 128 bits for IPv6) by simply passing the DNS hostname. For example, a user Java code invokes:

    InetAddress aComputer = InetAddress.getByName(“nsfcopire.spsu.edu”);

  - The class encapsulates the details of representing the IP address

Interprocess Communication – 4.2 API for Internet

- UDP Datagram communication
  - Steps:
    - Client finds an available port for UPD connection
    - Client binds the port to local IP (obtained from InetAddress.getByName(DNS))
    - Server finds a designated port, publicizes it to clients, and binds it to local IP
    - Sever process issues a receive methods and gets the IP and port # of sender (client) along with the message
  - Issues
    - **Message size** – set to 8KByte for most, general protocol support 216 bytes, possible truncation if receiver buffer is smaller than message size
    - **Blocking** – send is non-blocking and op returns if message gets pass the UDP and IP layers; receive is blocking (with discard if no socket is bound or no *thread* is waiting at destination port)
    - **Timeouts** – reasonably large time interval set on receiver sockets to avoid indefinite blocking
    - **Receive from any** – no specification of sources (senders), typically many-to-one, but one-to-one is possible by a designated send-receive socket (know by both C/S)

Interprocess Communication – 4.2 API for Internet

- UDP Failure Models:
  - Due to Omission of send or receive (either checksum error or no buffer space at source or destination)
  - Due to out-of-order delivery
  - UDP lacks built in checks, but failure can be modeled by implementing an ACK mechanism

Interprocess Communication – 4.2 API for Internet

- Use of UDP – Client/Sender code

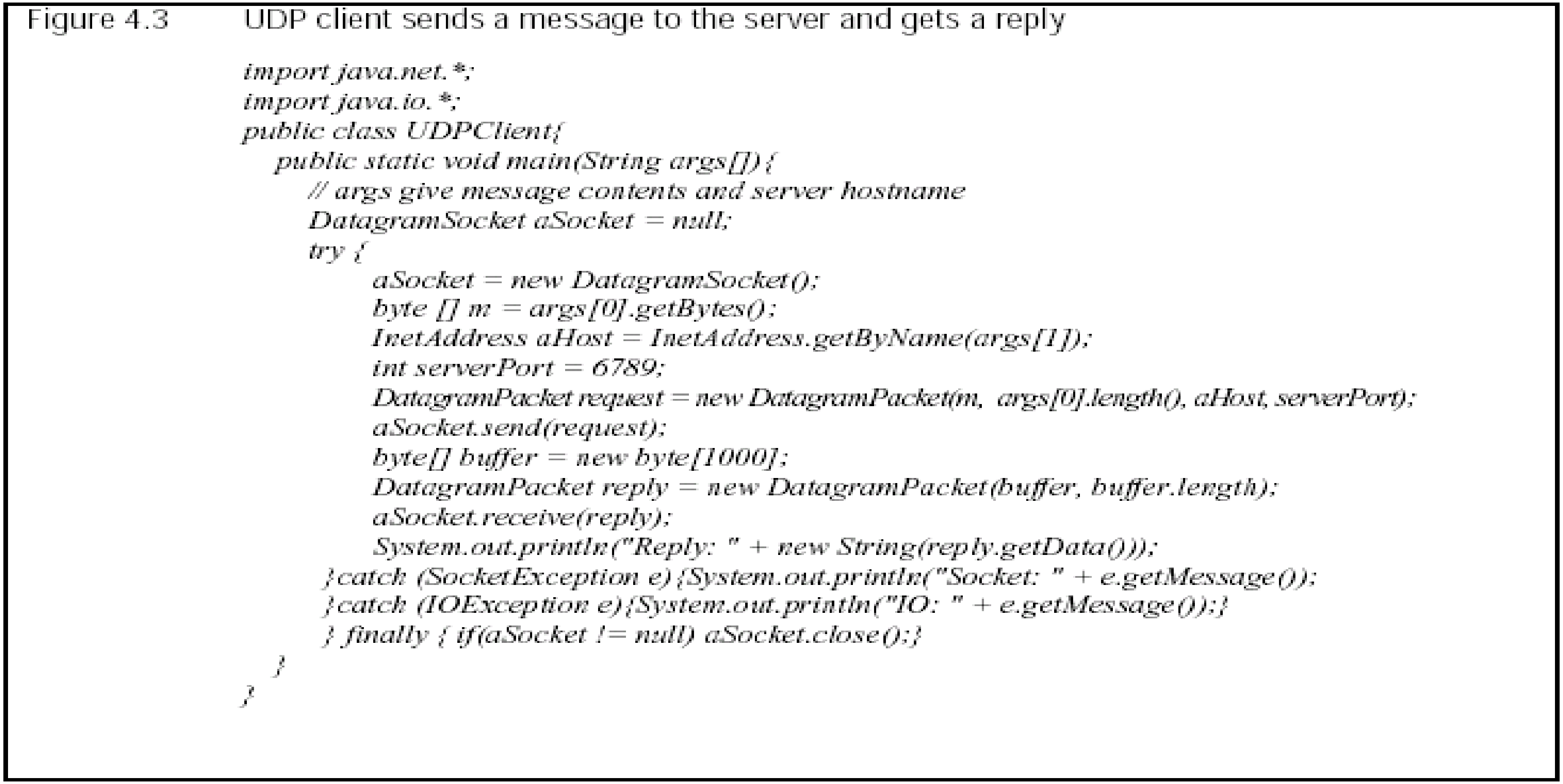


Figure 4.3 UDP client sends a message to the server and gets a reply

```
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,  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();}
    }
}
```

Interprocess Communication – 4.2 API for Internet

- Use of UDP – Server/Receiver code

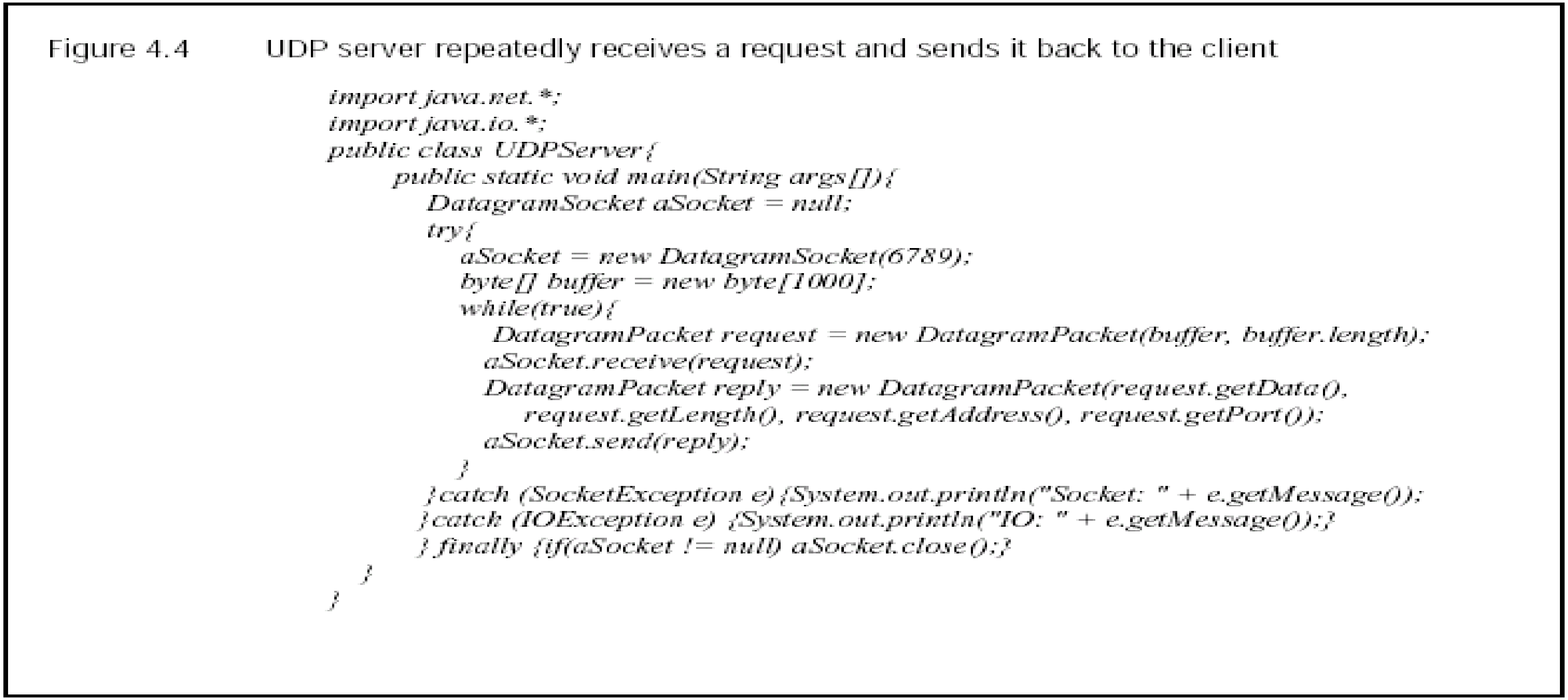


Figure 4.4 UDP server repeatedly receives a request and sends it back to the client

```
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();}
        }
}
```

Interprocess Communication – 4.2 API for Internet

- TCP Stream Communication
  - Grounded in the ‘piping’ architecture of Unix systems using BSD Unix sockets for streaming bytes
  - Characteristics:
    - Message sizes – user application has option to set IP packet size, small or large
    - Lost messages – Sliding window protocol with ACKs and retransmission is used
    - Flow control – Blocking or throttling is used
    - Message duplication and ordering – Seq #s with discard of dups & reordering
    - Message destinations – a connection is established first, using *connection-accept* methods for rendezvous, and no IP addresses in packets. [Each connection socket is bidirectional – using two streams: output/write and input/read]. A client *close*s a socket to sign off, and last stream of bytes are sent to receiver with ‘broken-pipe’ or empty-queue indicator

Interprocess Communication – 4.2 API for Internet

- TCP Stream Communication
  - Other Issues
    - Matching of data items – both client/sender and server/receiver must agree on data types and order in the stream
    - Blocking – data is streamed and kept in server queue: empty server queue causes a block AND full server queue causes a blocking of sender
    - Threads – used by servers (in the background) to service clients, allowing asynchronous blocking. [Systems without threads, e.g., Unix, use *select*]
  - Failure Model
    - Integrity: uses <u>checksums</u> for detection/rejection of corrupt data and <u>seq #s</u> for rejecting duplicates
    - Validity: uses timeout with retransmission techniques (takes care of packet losses or drops)
    - Pathological: excessive drops/timeouts signal broken sockets and TCP throws in the towel (no one knows if pending packets were exchanged) – unreliable
  - Uses – TCP sockets used for such services as: HTTP, FTP, Telnet, SMTP

Interprocess Communication – 4.2 API for Internet

- Use of TCP – Client/Sender code

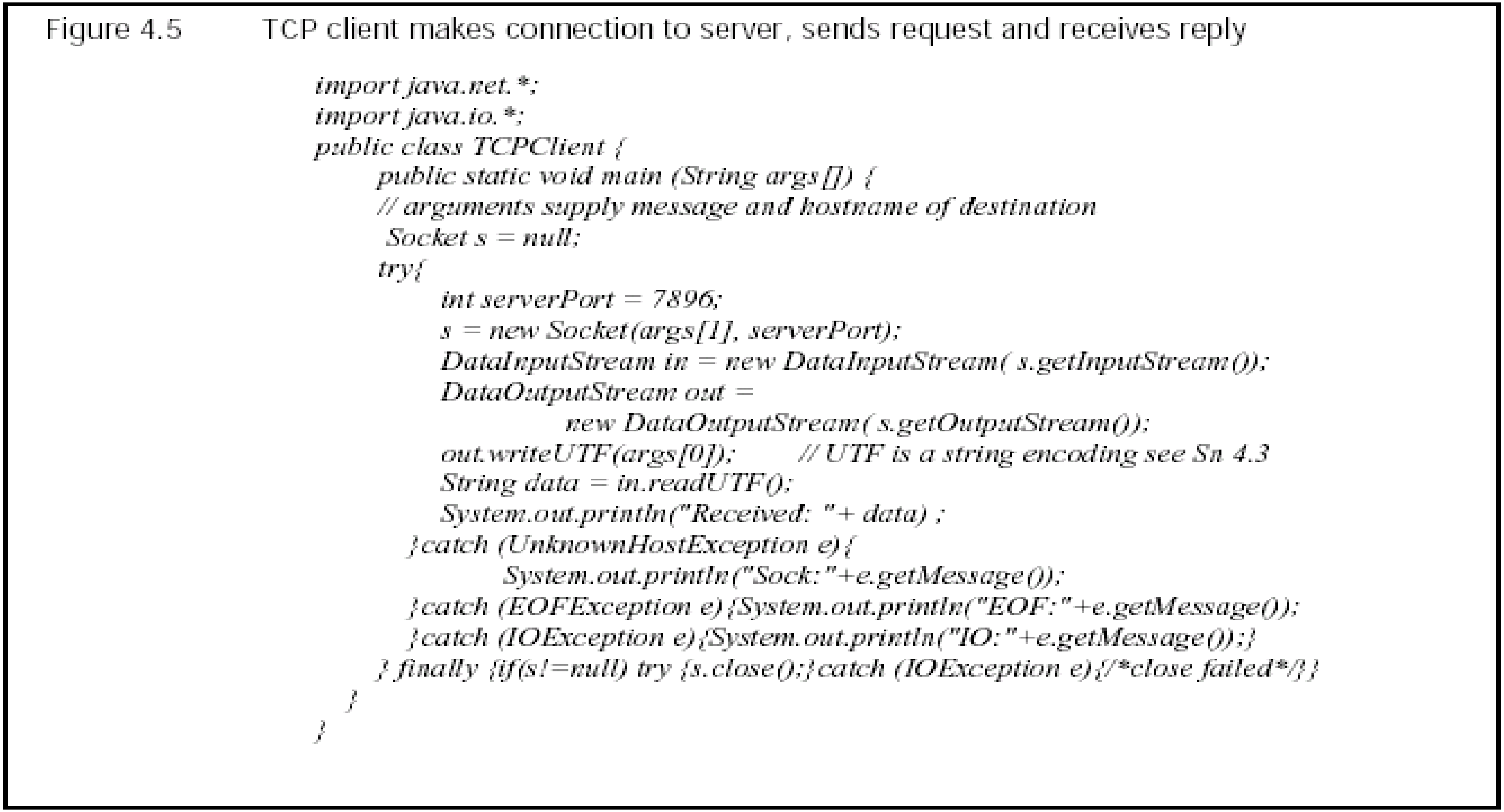


Figure 4.5 TCP client makes connection to server, sends request and receives reply

```
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){/*close failed*/}}
    }
}
```

Interprocess Communication – 4.2 API for Internet

- Use of TCP – Server/Receiver code

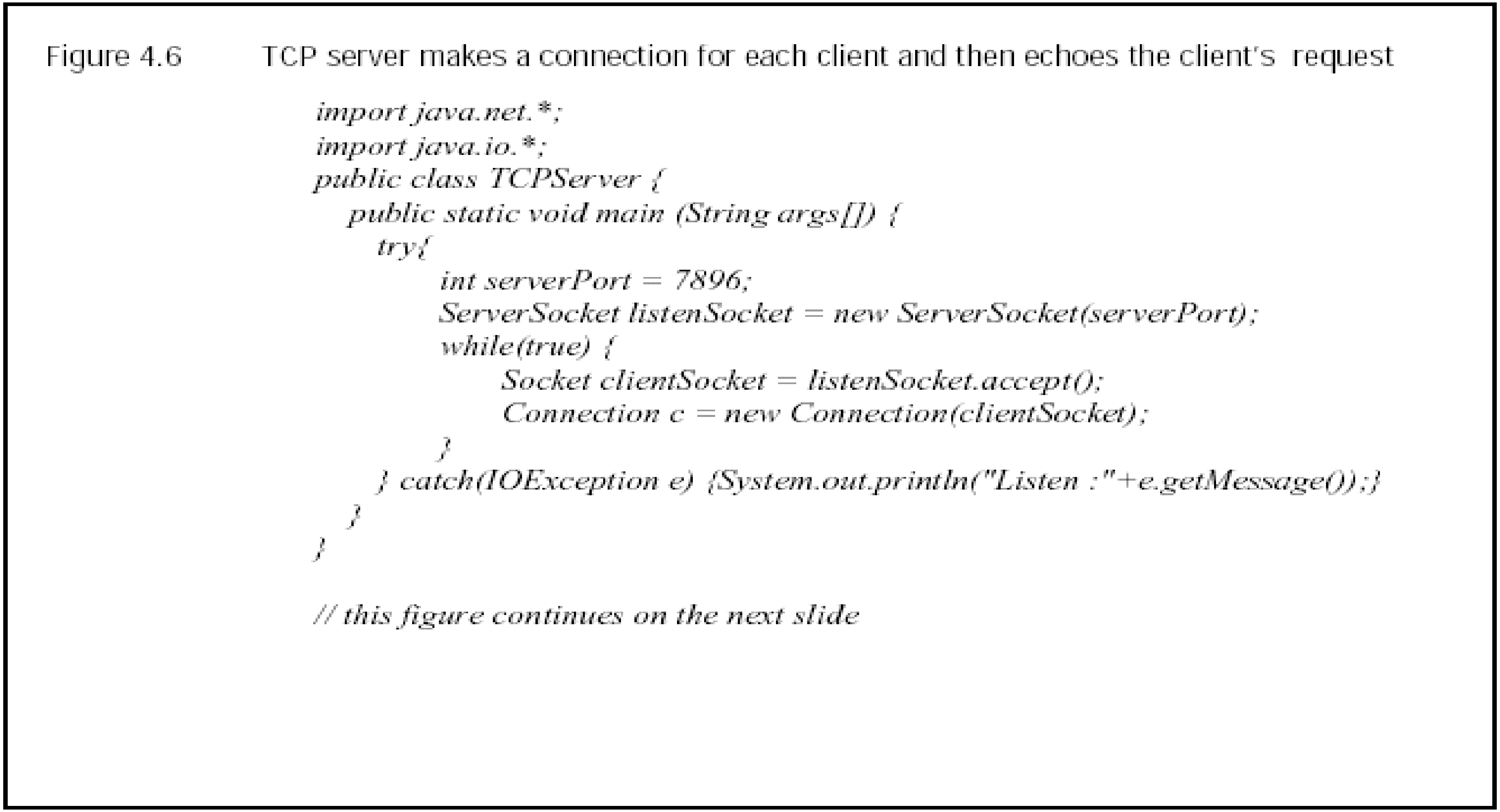


Figure 4.6 TCP server makes a connection for each client and then echoes the client's request

```
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
```

Interprocess Communication – 4.2 API for Internet

Use of TCP – Server/Receiver code (cont'd)

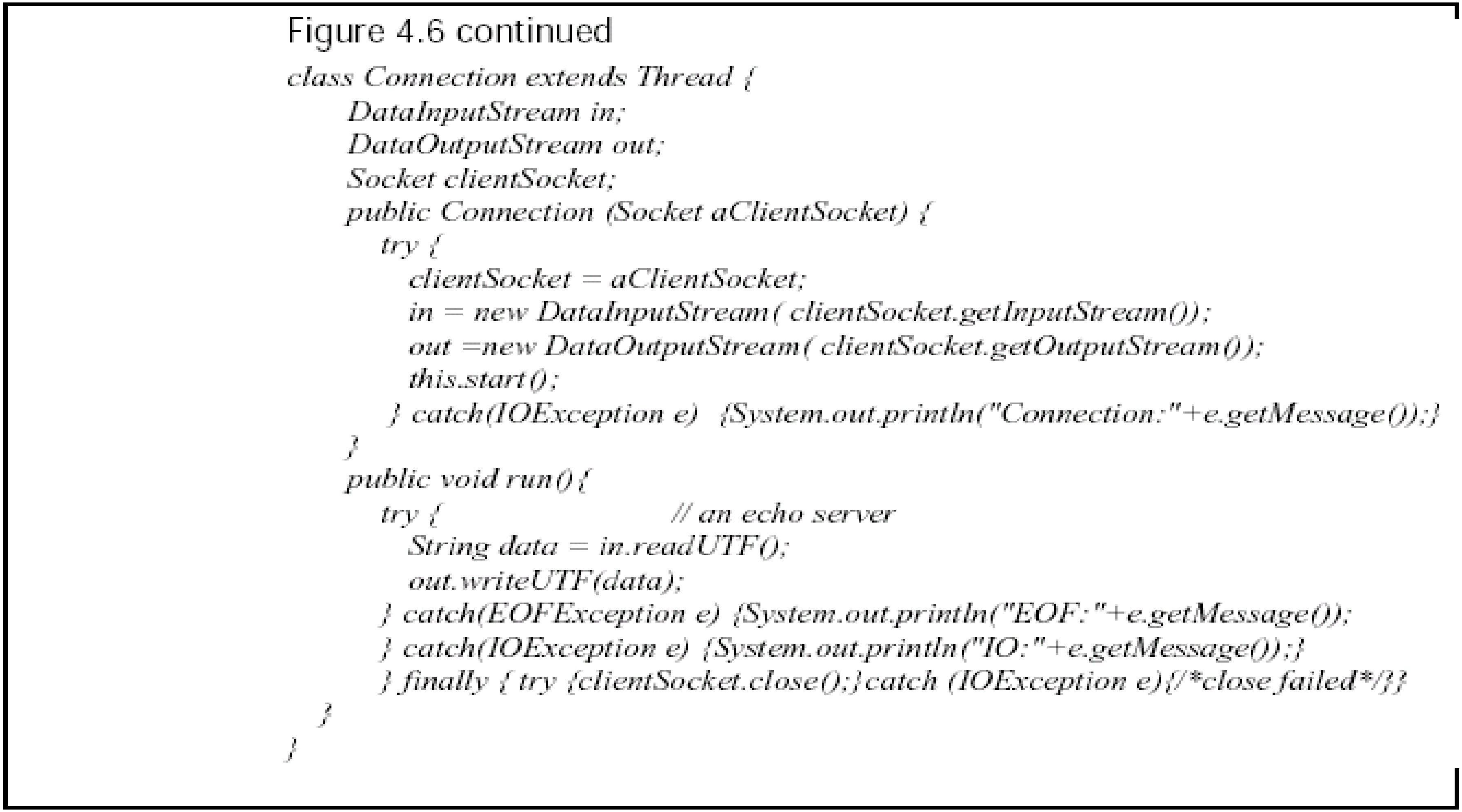


Figure 4.6 continued

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

Interprocess Communication – 4.3 External data representation

- Issues
  - At language-level data (for comm) are stored in data structures
  - At TCP/UDP-level data are communicated as ‘messages’ or streams of bytes – hence, conversion/flattening is needed
  - Problem? Different machines have different primitive data reps, e.g., big-endian and little-endian order of integers, float-type, char codes
  - Marshalling (before trans) and unmarshalling (restored to original on arrival)
  - Either both machines agree on a format type (included in parameter list) or an *intermediate* external standard (external data rep) is used, e.g., CORBA Common Data Rep (CDR)/IDL for many languages; Java object serialization for Java code only, Sun XDR standard for Sun NFSs

Interprocess Communication – 4.3 External data representation

- This masks the differences due to different computer hardware.
- CORBA CDR
  - only defined in CORBA 2.0 in 1998, before that, each implementation of CORBA had an external data representation, but they could not generally work with one another. That is:
    - the heterogeneity of hardware was masked
    - but not the heterogeneity due to different programmers (until CORBA 2)
  - CORBA CDR represents simple and constructed data types (sequence, string, array, struct, enum and union)
    - note that it does not deal with objects (*only Java does: objects and tree of objects*)
  - it requires an IDL specification of data to be serialised
- Java object serialisation
  - represents both objects and primitive data values
  - it uses reflection to serialise and deserialise objects– it does not need an IDL specification of the objects. (Reflection: inquiring about class properties, e.g., names, types of methods and variables, of objects]

Interprocess Communication – 4.3 External data representation

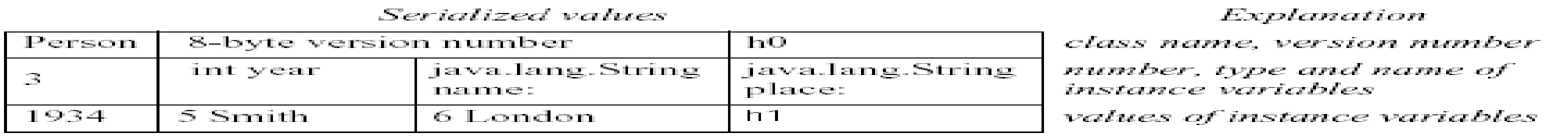
- **Example of Java serialized message**

```
public class Person implements Serializable {
        private String name;
        private String place;
        private int year;
        public Person(String aName, String aPlace, int aYear) {
                name = aName;
                place = aPlace;
                year = aYear;
        }
        // followed by methods for accessing the instance variables
}
```

  - Consider the following object:

    Person p = new Person("Smith", "London", 1934);

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

CORBA IDL example

CORBA has a struct

```
struct Person {
            string name;
            string place;
            long year;
} ;
interface PersonList {
            readonly attribute string listname;
            void addPerson(in Person p) ;
            void getPerson(in string name, out Person
p);
            long number();
};
```

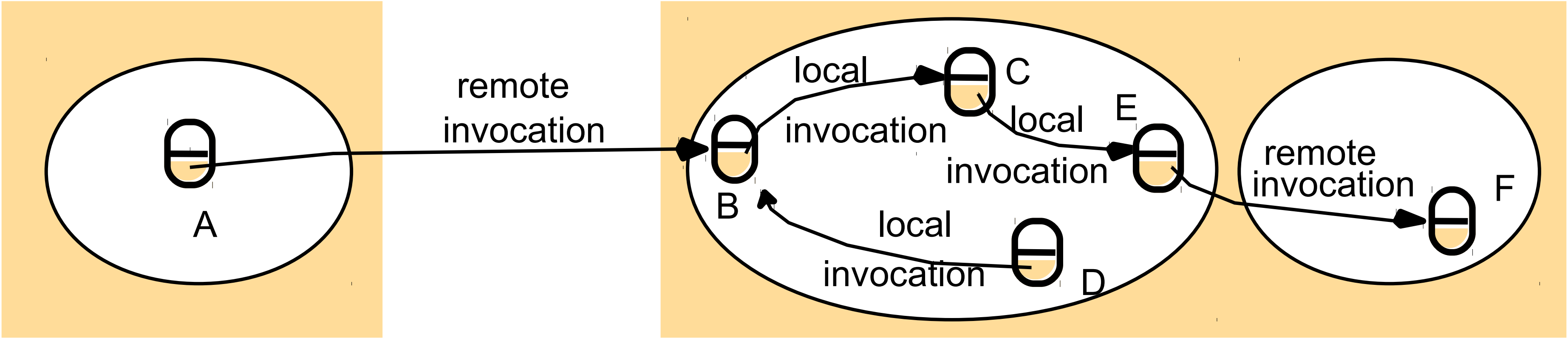
remote interface

remote interface defines methods for RMI

parameters are *in*, *out* or *inout*

- Remote interface:
  - specifies the methods of an object available for remote invocation
  - an interface definition language (or IDL) is used to specify remote interfaces. E.g. the above in CORBA IDL.
  - Java RMI would have a class for *Person*, but CORBA has a *struct*

Interprocess Communication – 4.3 External data representation



- each process contains objects, some of which can receive remote invocations, others only local invocations
- those that can receive remote invocations are called *remote objects*
- objects need to know the *remote object reference* of an object in another process in order to invoke its methods. How do they get it?
- the *remote interface* specifies which methods can be invoked remotely
- Remote object references are passed as arguments and compared to ensure uniqueness over time and space in Distributed Computing system

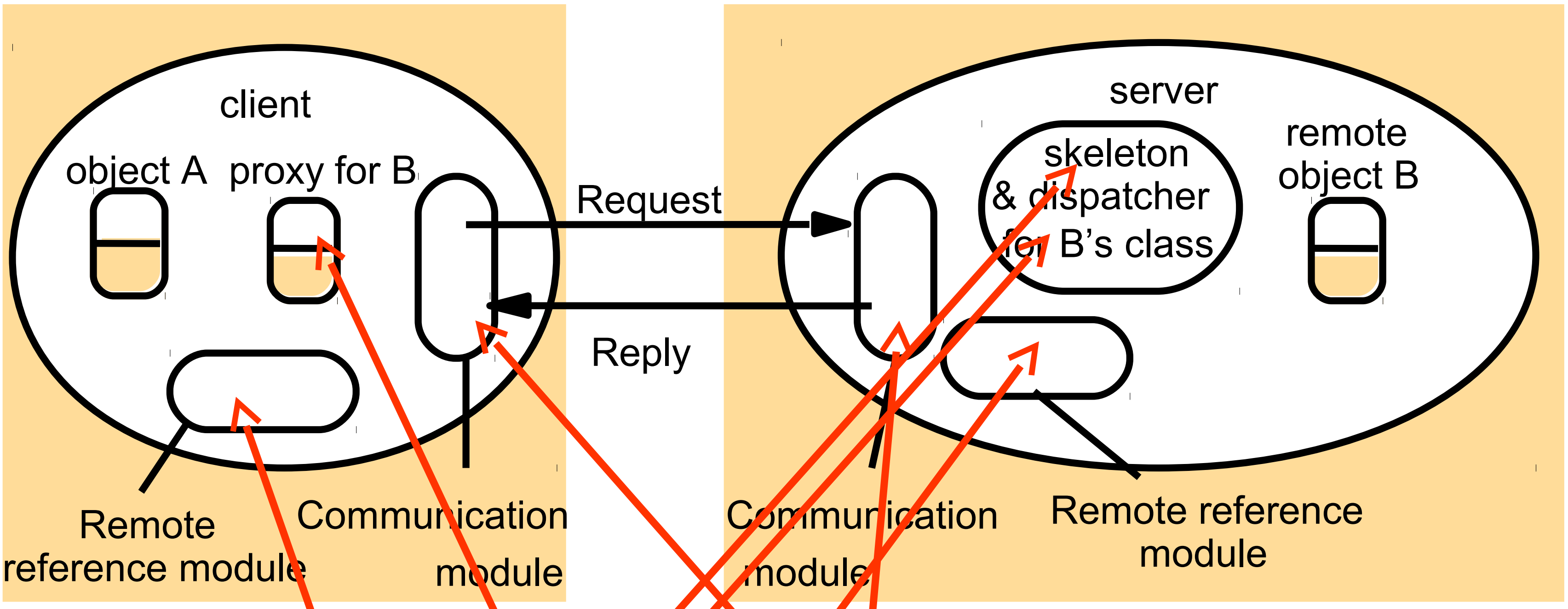
Representation of a remote object reference

Figure 4.10

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

- a remote object reference must be unique in the distributed system and over time. It should not be reused after the object is deleted. Why not?
- the first two fields locate the object unless migration or re-activation in a new process can happen
- the fourth field identifies the object within the process
- its interface tells the receiver what methods it has (e.g. class *Method*)
- a remote object reference is created by a remote reference module when a reference is passed as argument or result to another process
  - it will be stored in the corresponding proxy
  - it will be passed in request messages to identify the remote object whose method is to be invoked

The architecture of remote method invocation



Proxy - makes RMI transparent to … s
remote interface. Marshals reques…
results. Forwards request.

carries out Request-reply protocol

translates between local and remote object references and creates remote object references. Uses remote object table

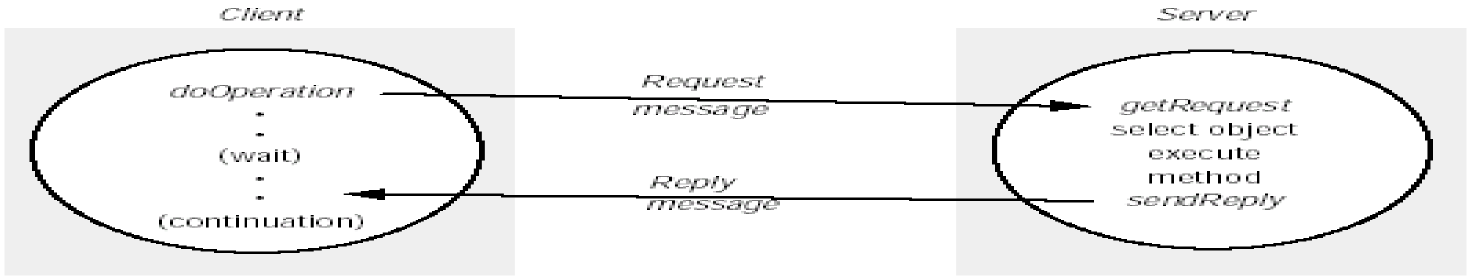
…terface.
…nvokes

RMI software - between application level objects and communication and remote reference modules

Interprocess Communication – 4.4 Client-Server Communication

- Modes:
  - Request-reply: client process blocks until and ACK is received from server (Synchronous)
  - Use send/receive operations in Java API for UDP (or TCP streams – typically with much overhead for the ‘guarantees’)
  - Protocol over UDP, e.g., piggybacked ACKs,

Figure 4.11 Request-reply communication



Interprocess Communication – 4.4 Client-Server Communication

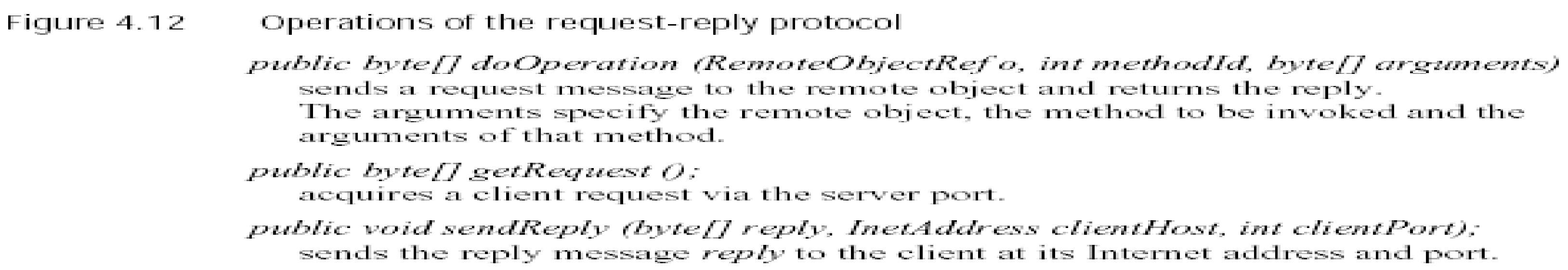


Figure 4.12 Operations of the request-reply protocol

public byte[] doOperation (RemoteObjectRef o, int methodId, byte[] arguments)
sends a request message to the remote object and returns the reply.
The arguments specify the remote object, the method to be invoked and the arguments of that method.

public byte[] getRequest ();
acquires a client request via the server port.

public void sendReply (byte[] reply, InetAddress clientHost, int clientPort);
sends the reply message *reply* to the client at its Internet address and port.

Request-Reply Protocol

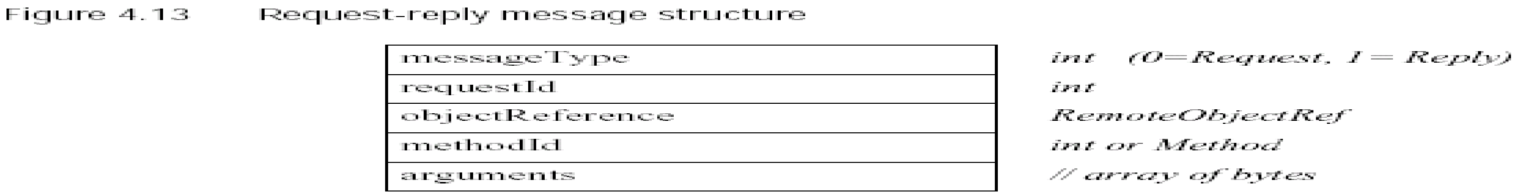


Figure 4.13 Request-reply message structure

messageType	*int (0=Request, 1= Reply)*
requestId	*int*
objectReference	*RemoteObjectRef*
methodId	*int or Method*
arguments	*// array of bytes*

MessageIDs: *requestID* + *IP.portnumber* // IP.portnumber from packet if UDP

Interprocess Communication – 4.4 Client-Server Communication

- Failure Model of Request-Reply Protocol

  - For doOperation, getRequest, sendReply over UDP, the possible problems include:
    - Omission failure (link failures, drops/losses, missed/corrupt addresses)
    - Out-of-order delivery
    - Node/process down

  - Solved by
    - Timeouts with retrans or retries until reply is received/confirmed
    - Discards of repeated requests (by server process)
    - On lost reply messages, servers repeats idempotent operations
    - Maintain history (reqid, message, client-id) or buffer replies and retrans – memory intensive

Interprocess Communication – 4.4 Client-Server Communication

- Failure handling –
  - RPC exchange protocols (for handling failures)
    - Request (R) Protocol – for remote procedures with return results or need for ACK
    - Request-Reply Protocol – converse of R protocol, with reply message as piggybacked ACK
    - Request-Reply-ACK-reply (RRA) – request and ACK reply message (with id of specific range of requests being ACKed – Go-back/Selective Repeat protocols) – avoids keeping histories

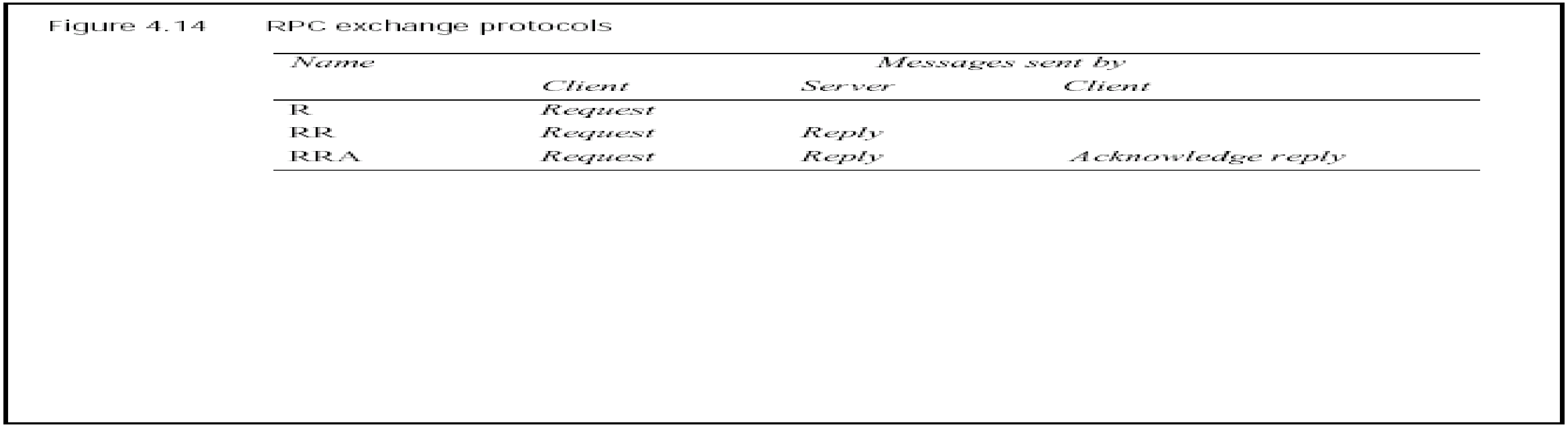


Figure 4.14 RPC exchange protocols

Name	*Messages sent by*		
	Client	*Server*	*Client*
R	*Request*		
RR	*Request*	*Reply*	
RRA	*Request*	*Reply*	*Acknowledge reply*

Interprocess Communication – 4.5 Group Communication

- Limitations of peer-to-peer (point-to-point)
- Support for (concurrent) multiple services from a single clients requires a different communication paradigm

- Multicast – message from a single client to group of server processes (useful for distributed processing)
- Characteristics
  - Fault tolerance based on replicated services (redundancy of same service)
  - Finding discovery servers (with registry of new service interfaces)
  - Better performance via data replication (updates are multicast for currency)
  - Propagation of event notification (inform affected servers of new events, e.g., new routing tables, new servers or clients)

Interprocess Communication – 4.5 Group Communication

- IP Multicast
  - Built on top of the IP layer (using IP addresses) [ports for TCP and UDP]
  - Thus, multicast packets are sent to computers using an IP-addressing scheme
  - Client/sender is unaware of individual computers in a group
  - A multicast-group address is specified using class D Internet address
  - Membership of a group is dynamic
  - Multicasting is through UDP protocol only (at programming level) via multicast addresses and ordinary port numbers
  - A receiver joins a group by adding its socket to the group
  - At IP level, adding process's socket #s to a group makes the computer a member of the group – allowing all such sockets to receive incoming packets at specified port #s.
  - How? Via local multicast capability Or via multicast routers @ Internet level
  - Membership? Permanent (designated addresses) or temporary

Interprocess Communication – 4.5 Group Communication

IP Multicast – Example @ Application level

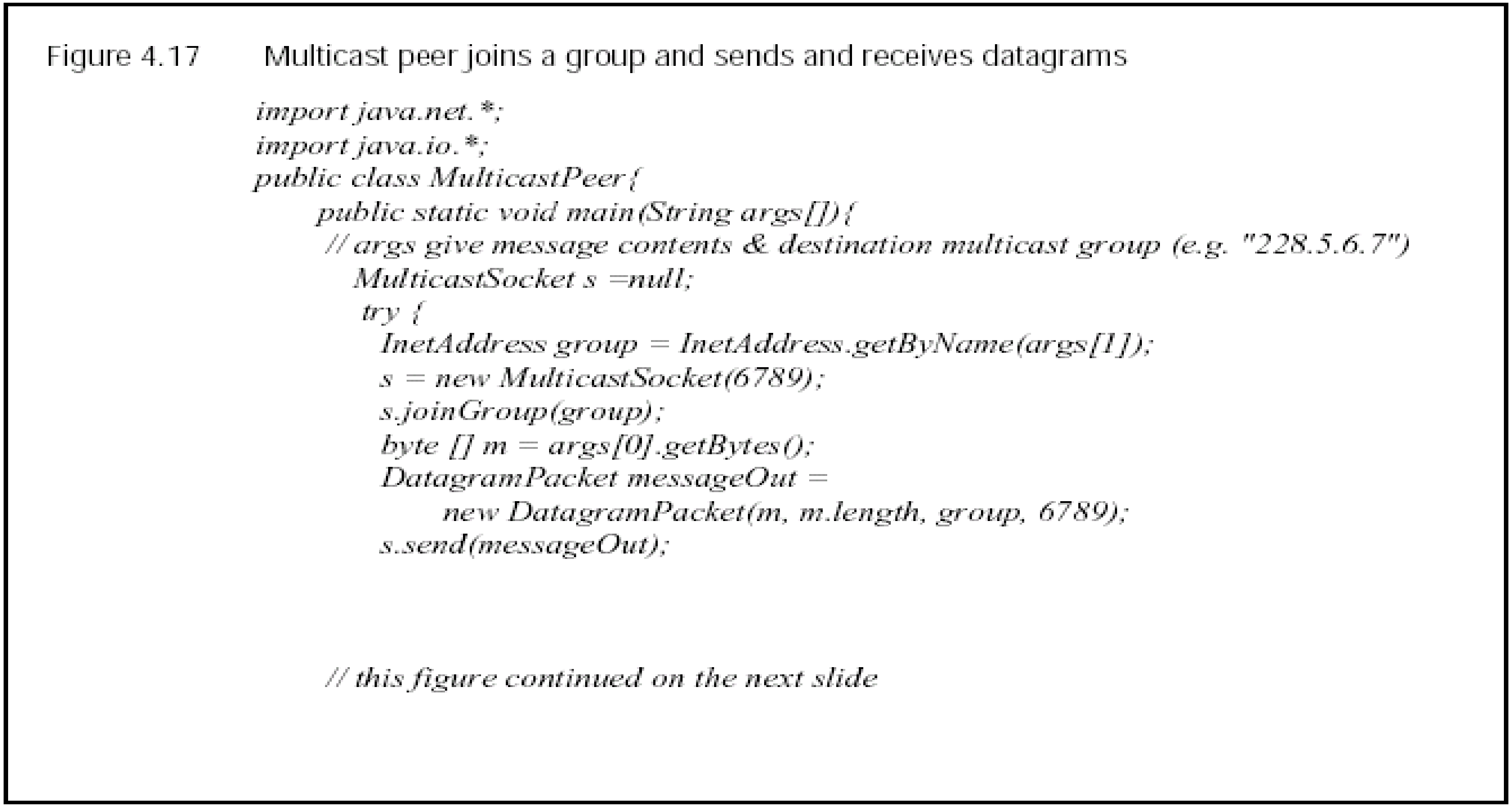


Figure 4.17 Multicast peer joins a group and sends and receives datagrams

```
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
```

Interprocess Communication – 4.5 Group Communication

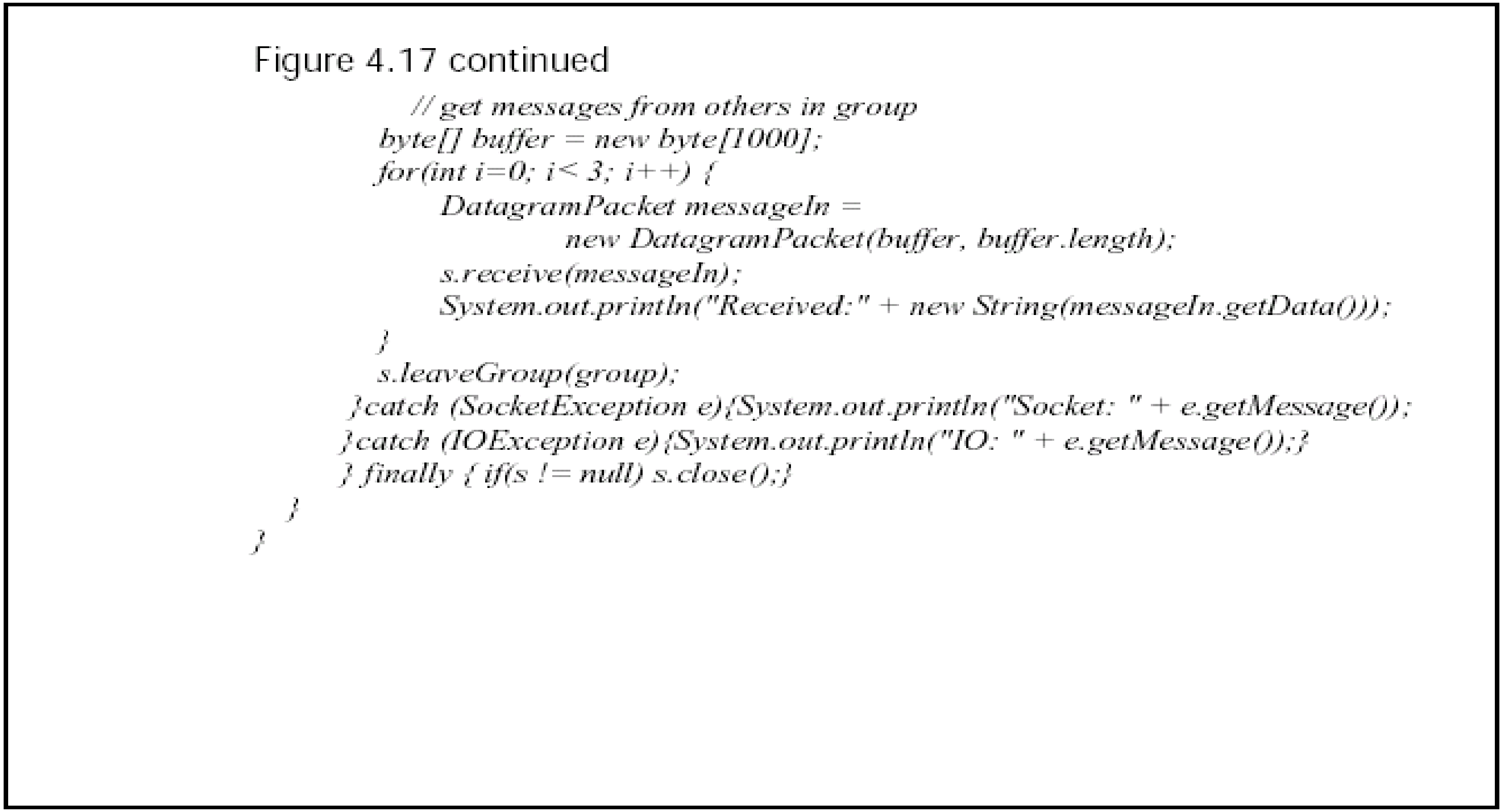
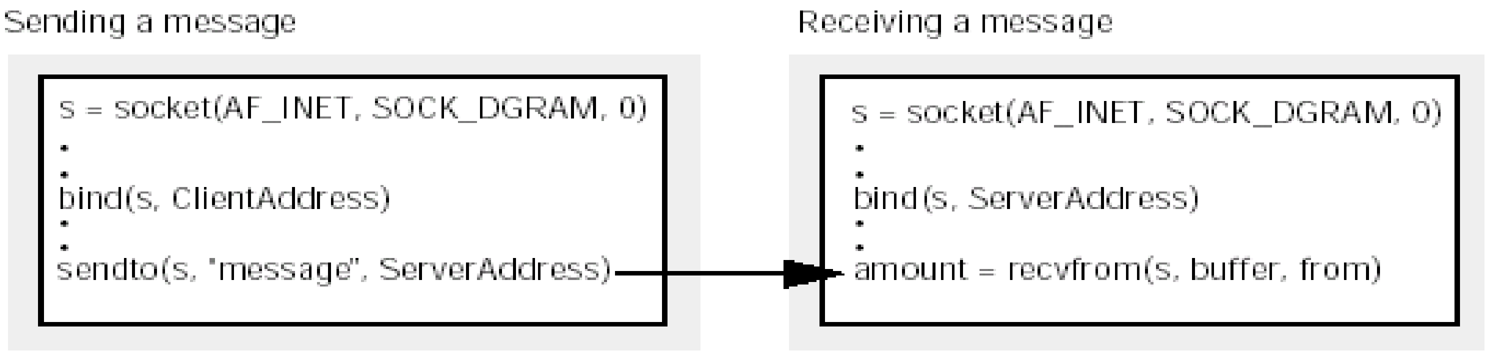


Figure 4.17 continued

```
            // 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();}
    }
}
```

Interprocess Communication – 4.6 UNIX Example

Figure 4.18 Sockets used for datagrams



Sending a message

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

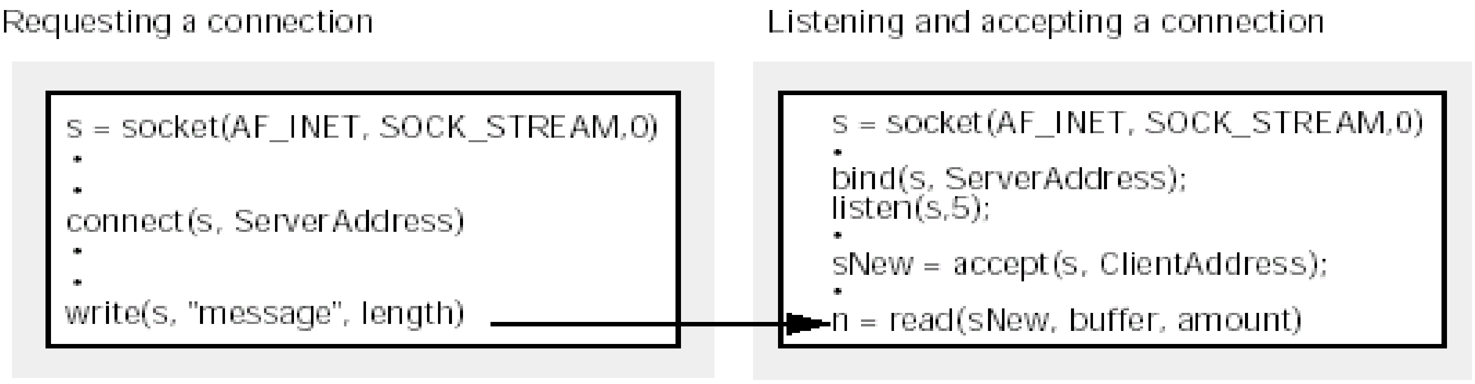
Receiving a message

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

ServerAddress and *ClientAddress* are socket addresses

Interprocess Communication – 4.6 UNIX Example

Figure 4.19 Sockets used for streams



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