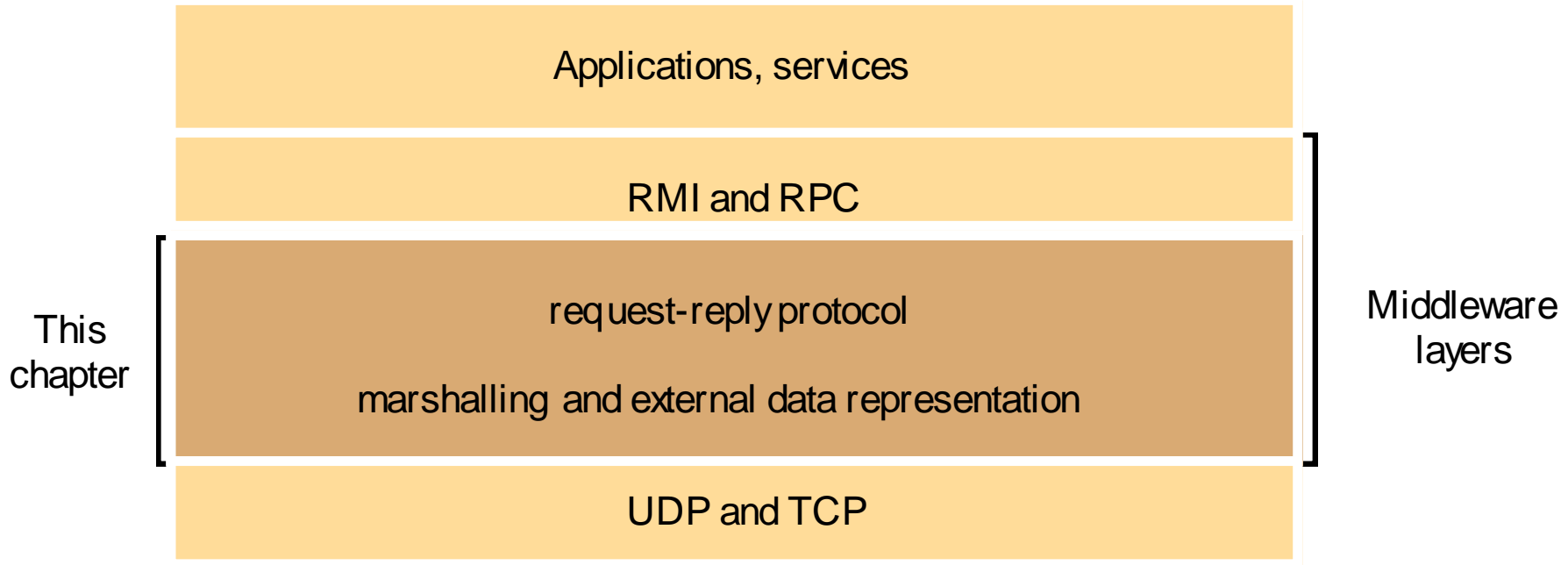


Interprocess Communication

Introduction



Introduction

- ❑ how the objects and data structures used in application programs can be translated into a form suitable for sending in messages over the network.
- ❑ design of suitable protocols to support client-server and group communication.

External data representation and marshalling

- Information stored in running programs is represented as data structures
- Information in messages consists of sequences of bytes.
- For communication, the data structures must be flattened before transmission and rebuilt on arrival.

External data representation . . .

To enable any two computers to exchange binary data values:

- The values are converted to an agreed external format before transmission and converted to the local form on receipt.
- The values are transmitted in the sender's format, together with an indication of the format used, and the recipient converts the values if necessary.

External data representation . . .

External data representation

- An agreed standard for the representation of data structures and primitive values.

External data representation . . .

Marshalling: the process of taking a collection of data items and assembling them into a form suitable for transmission in a message.

Unmarshalling: the process of disassembling them on arrival to produce an equivalent collection of data items at the destination.

External data representation . . .

Three alternative approaches to external data representation and marshalling:

- CORBA's common data representation.
- Java's object serialization.
- XML or Extensible Markup Language.

External data representation . . .

Marshalling & unmarshalling are intended to be carried out by middleware

Data types are marshalled into binary or textual form

Whether type information should be included?

CORBA's Common Data Representation (CDR)

- CDR can represent all of the data types that can be used as arguments and return values in remote invocations in CORBA.
- 15 primitive types - *short* (16-bit), *long* (32-bit), *unsigned short*, *unsigned long*, *float* (32-bit), *double* (64-bit), *char*, *boolean*, *octet*, and *any*;
- a range of composite types.

CORBA's Common Data Representation (CDR)

Primitive types:

- Defines a representation for both big-endian and little-endian orderings.
- Values are transmitted in the sender's ordering, which is specified in each message. The recipient translates it, if required.
- Each primitive value is placed at an index in the sequence of bytes according to its size.

CORBA CDR for constructed types

Constructed types: The primitive values that comprise each constructed type are added to a sequence of bytes in a particular order

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

CORBA CDR message

index in
sequence of bytes \longleftrightarrow 4 bytes \longrightarrow *notes*
on representation

0–3	5	<i>length of string</i>
4–7	"Smit"	<i>'Smith'</i>
8–11	"h____"	
12–15	6	<i>length of string</i>
16–19	"Lond"	<i>'London'</i>
20–23	"on____"	
24–27	1984	<i>unsigned long</i>

CORBA CDR message

Figure shows a message in CORBA CDR that contains the three fields of a *struct* whose respective types are *string*, *string* and *unsigned long*.

The figure shows the sequence of bytes with four bytes in each row.

The representation of each string consists of an *unsigned long* representing its length followed by the characters in the string.

Each *unsigned long*, which occupies four bytes, starts at an index that is a multiple of four.

Marshalling in CORBA

Marshalling operations can be generated automatically from the specification of the types of data items to be transmitted in a message.

The types of the data structures and the types of the basic data items are described in CORBA IDL.

CORBA IDL which provides a notation for describing the types of the arguments and results of RMI methods.

Marshalling in CORBA

Ex: CORBA IDL to describe the data structure in the message:

```
struct Person{  
    string name;  
    string place;  
    unsigned long year;  
};
```

The CORBA interface compiler generates appropriate marshalling and unmarshalling operations for the arguments and results of remote methods from the definitions of the types of their parameters and results.

Java object serialization

In Java RMI, both objects and primitive data values may be passed as arguments and results of method invocations.

Serialization

Deserialization

The class should implement the *Serializable* interface.

Ex: the Java class equivalent to the
Person struct.

Java object serialization

```
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  
}
```

Java object serialization

Assumed that the process that does the deserialization has no prior knowledge of the types of the objects in the serialized form. . . .

The information about the class consists of the name of the class and a version number. . . .

Java objects can contain references to other objects. . . .

References are serialized as *handles*. The handle is a reference to an object within the serialized form

Java object serialization

To serialize an object,

- its class information is written out, followed by the types and names of its instance variables.
- If the instance variables belong to new classes, then their class information must also be written out, followed by the types and names of their instance variables.
- This recursive procedure continues until the class information and types and names of the instance variables of all of the necessary classes have been written out.
- Each class is given a handle, and no class is written more than once to the stream of bytes.

Indication of Java serialized form

consider the serialization of the following object:

Person p = new Person("Smith", "London", 1984);

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

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

Remote object references

- It is an identifier for a remote object that is valid throughout a distributed system.
- It is passed in the invocation message to specify which object is to be invoked.

Remote object references

- It must be unique among all of the processes in the distributed system.
- constructed by concatenating the Internet address of its host computer and the port number of the process that created it with the time of its creation and a local object number.

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

Remote object references

The last field of the remote object reference contains some information about the interface of the remote object.

- Any process that receives a remote object reference as an argument or as the result of a remote invocation, will come to know about the methods offered by the remote object.

Remote object references

Simplest implementation of RMI – remote objects live only in the process that created them and survive only as long as that process continues to run.

Client-server communication

In the normal case, request-reply communication is

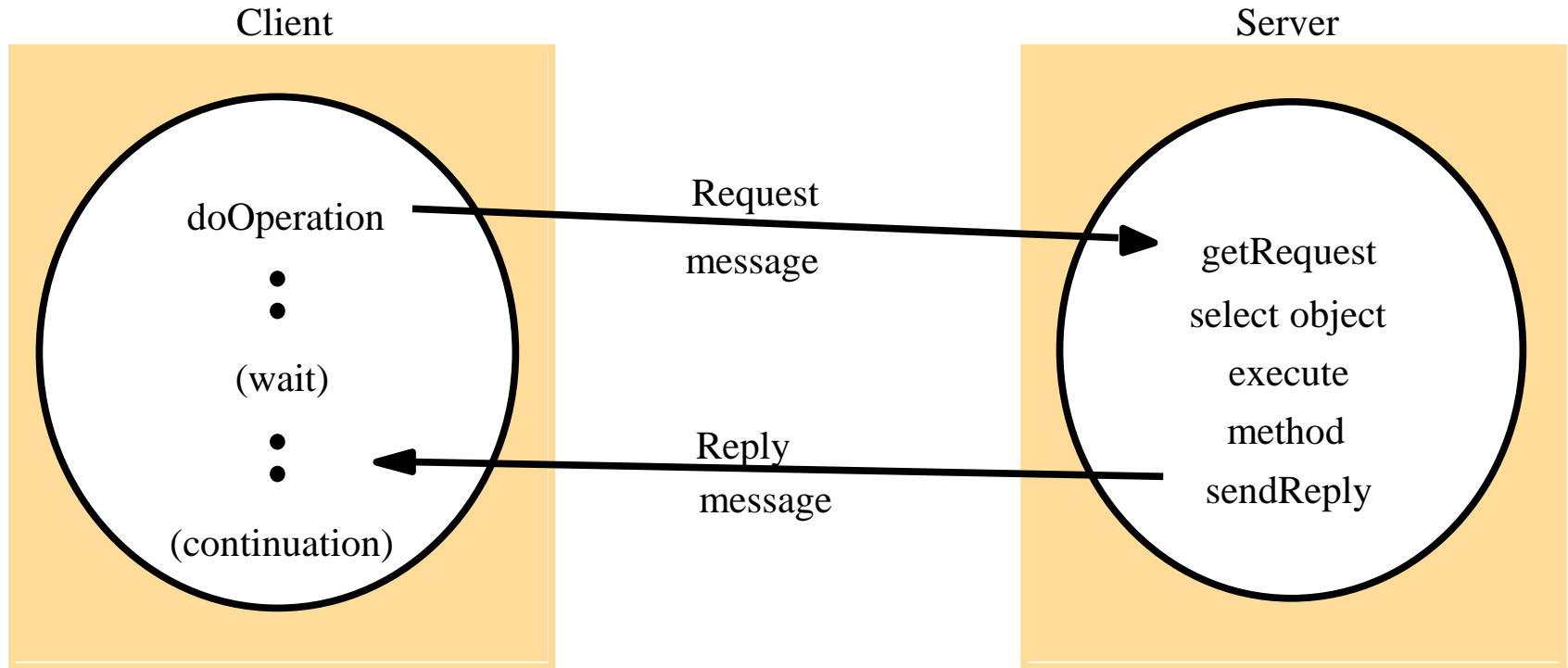
- synchronous
- reliable

If clients can afford to retrieve replies later

- asynchronous request-reply communication

A protocol built over datagrams avoids unnecessary overheads associated with the TCP stream protocol.

Request-reply communication



The request-reply protocol

- based on primitives:
doOperation, getRequest and *sendReply*
- tailored for supporting RMI.
- matches requests to replies.
- may be designed to provide certain delivery guarantees.

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 to the client at its Internet address and port.

The request-reply protocol

The *doOperation* method is

- used by clients to invoke remote operations. Its arguments specify the remote object and which method to invoke, together with additional information required by the method. Its result is an RMI reply.
- assumes that the client calling *doOperation* marshals the arguments into an array of bytes and unmarshals the results from the array of bytes that is returned.
- sends a request message to the server whose Internet address and port are specified in the remote object reference given as argument.

The request-reply protocol

The *doOperation* method:

- After sending the request message, *doOperation* invokes *receive* to get a reply message, from which it extracts the result and returns it to the caller.
- The caller of *doOperation* is blocked until the remote object in the server performs the requested operation and transmits a reply message to the client process.

The request-reply protocol

GetRequest is used by a server process to acquire service requests.

- When the server has invoked the method in the specified object it then uses *sendReply* to send the reply message to the client.
- When the reply message is received by the client the original *doOperation* is unblocked and execution of the client program continues.

Request-reply message structure

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

The request-reply protocol

The information to be transmitted in a request message or a reply message:

- The first field indicates whether the message is a *Request* or a *Reply* message.
- The second field *requestId* contains a message identifier.
- The third field is a remote object reference.
- The fourth field is an identifier for the method to be invoked.

Message identifiers

For providing additional properties such as reliable message delivery or request-reply communication

- each message must have a unique message identifier.

A message identifier consists of:

1. a *requestId*, which is taken from an increasing sequence of integers by the sending process;
2. an identifier for the sender process, for example its port and Internet address.

Failure model of the request-reply protocol

If *doOperation*, *getRequest* and *sendReply* are implemented over UDP datagrams, then they suffer from the same communication failures:

- omission failures;
- messages not guaranteed to be delivered in sender order;
- the protocol can suffer from the failure of processes.

Timeouts

- simplest option is to return immediately with an indication that the *doOperation* has failed.
- send the request message repeatedly until either it gets a reply or else it is reasonably sure that the delay is due to lack of response from the server rather than to lost messages.

Discarding duplicate request messages

- the server may receive the message more than once.
- the protocol is designed to recognize successive messages (from the same client) with the same request identifier and to filter out duplicates.

Lost reply messages

- If the server has already sent the reply when it receives a duplicate request it will need to execute the operation again to obtain the result.
- *Idempotent operation*: an operation that can be performed repeatedly with the same effect as if it had been performed exactly once.
- A server whose operations are all idempotent need not take special measures to avoid executing its operations more than once.

History

- refers to a structure that contains a record of (reply) messages that have been transmitted.
- an entry in a history contains a request identifier, a message and an identifier of the client to which it was sent.

A problem associated with the use of a history

- memory cost.

Failure model of the request-reply protocol

RPC exchange protocols: Protocols which produce differing behaviours in the presence of communication failures:

- the *request* (R) protocol;
- the *request-reply* (RR) protocol;
- the *request-reply-acknowledge reply* (RRA) protocol.

RPC exchange protocols

<i>Name</i>	<i>Messages sent by</i>		
	<i>Client</i>	<i>Server</i>	<i>Client</i>
R	<i>Request</i>		
RR	<i>Request</i>	<i>Reply</i>	
RRA	<i>Request</i>	<i>Reply</i>	<i>Acknowledge reply</i>

RPC exchange protocols

R protocol - a single *Request* message is sent by the client to the server.

- used when there is no value to be returned from the remote method
- the client requires no confirmation that the method has been executed.

RR protocol - useful for most client-server exchanges because it is based on the request-reply protocol.

- special acknowledgement messages are not required.

RPC exchange protocols

RRA protocol is based on the exchange of three messages: request, reply and acknowledge reply.

- The *acknowledge reply* message contains the *requestId* from the reply message being acknowledged.
- The arrival of a *requestId* in an acknowledgement message will be interpreted as acknowledging the receipt of all reply messages with lower *requestIds*.

Group communication

Multicast operation:

- an operation that sends a single message from one process to each of the members of a group of processes
- membership of the group is transparent to the sender.
- range of possibilities in the desired behaviour of a multicast.

Group communication

Multicast messages provide a useful infrastructure for constructing distributed systems with the following characteristics:

1. Fault tolerance based on replicated services
2. Discovering services in spontaneous networking
3. Better performance through replicated data
4. Propagation of event notifications

IP multicast – An implementation of group communication

- allows the sender to transmit a single IP packet to a set of computers that form a multicast group.
- sender is unaware of the identities of the individual recipients and of the size of the group.
- A *multicast group* is specified by a Class D Internet address.
- the membership of multicast groups is dynamic.

IP multicast – An implementation of group communication

- An application program performs multicasts by sending UDP datagrams with multicast addresses and ordinary port numbers.
- It can join a multicast group by making its socket join the group.

IP multicast – An implementation of group communication

Multicast routers:

- Internet multicasts make use of multicast routers.
- these forward single datagrams to routers on other networks with members.
- the sender can specify the *time to live*, or TTL.

IP multicast – An implementation of group communication

Multicast address allocation: Multicast addresses may be permanent or temporary.

Permanent groups:

- exist even when there are no members
- addresses are assigned by the Internet authority.

Temporary groups:

- must be created before use and cease to exist when all the members have left
- the remainder of the multicast addresses are available for use.

IP multicast – An implementation of group communication

Failure model for multicast datagrams:

- Datagrams multicast over IP multicast have the same failure characteristics as UDP datagrams (omission failures).
- can be called *unreliable* multicast.

Figure 4.20

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();}
    }
    }
```

```
public class MulticastPeer{  
    public static void main(String args[]){  
        // args give message contents & destination multicast group  
        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);  
        }  
    }  
}
```

```
byte[] buffer = new byte[1000];  
for(int i=0; i< 3; i++) { // get messages from others in group  
    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();}  
}
```


Reliability and ordering of multicast

IP multicast suffers from omission failures.

- multicast on a local area network
- datagram sent from one multicast router to another.

Ordering

- IP packets sent over an internetwork do not necessarily arrive in the order in which they were sent.

Some examples of the effects of reliability and ordering

1. *Fault tolerance based on replicated services:*

- this application requires that either all of the replicas or none of them should receive each request to perform an operation.

2. *Discovering services in spontaneous networking:*

- an occasional lost request is not an issue when discovering services.

Some examples of the effects of reliability and ordering

3. *Better performance through replicated data:*

- the effect of lost messages and inconsistent ordering depends on the method of replication and the importance of all replicas being totally up-to-date.

4. *Propagation of event notifications:*

- the particular application determines the qualities required of multicast.

Need for

- *reliable multicast*
- *totally ordered multicast.*

Case study: interprocess communication in UNIX

Message destinations are specified as *socket addresses*

- a socket address consists of an Internet address and a local port number.

The interprocess communication operations are based on the socket abstraction.

A process can create a socket by invoking the *socket* system call, whose arguments specify the communication domain, the type and sometimes a protocol.

- the protocol is usually selected by the system according to whether the communication is datagram or stream.

Case study: interprocess communication in UNIX

The socket call returns a descriptor by which the socket may be referenced in subsequent system calls.

- the socket lasts until it is *closed* or until every process with the descriptor exits.

Before a pair of processes can communicate, the recipient must *bind* its socket descriptor to a socket address.

- the sender must also bind its socket descriptor to a socket address if it requires a reply.


Datagram communication

- A socket pair is identified each time a communication is made.
- achieved by the sending process using its local socket descriptor and the socket address of the receiving socket.

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

Datagram communication

Both processes use the *socket* call to create a socket and get a descriptor for it.

- the first argument of *socket* specifies the communication domain as the Internet domain and the second argument indicates that datagram communication is required.
- the last argument to the *socket* call may be used to specify a particular protocol, but setting it to zero causes the system to select a suitable protocol - UDP in this case.

Datagram communication

Both processes then use the *bind* call to bind their sockets to socket addresses.

- the sending process binds its socket to a socket address referring to any available local port number.
- the receiving process binds its socket to a socket address that contains its server port and must be made known to the sender.

Datagram communication

The sending process uses the *sendto* call

- arguments specify the socket through which the message is to be sent, the message itself and the socket address of the destination.
- the *sendto* call hands the message to the underlying UDP and IP protocols and returns the actual number of bytes sent.

Datagram communication

The receiving process uses the *recvfrom* call

- arguments specify the local socket on which to receive a message and memory locations in which to store the message and the socket address of the sending socket.
- it collects the first message in the queue at the socket, or if the queue is empty it will wait until a message arrives.

Stream communication


Two processes must first establish a connection between their pair of sockets.

- for communication between clients and servers, clients request connections and a listening server accepts them
- once a pair of sockets has been connected, they may be used for transmitting data in both or either direction

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

Stream communication

The server or listening process first uses the *socket* operation to create a stream socket and the *bind* operation to bind its socket to the server's socket address

- the second argument to the *socket* system call is given as `SOCK_STREAM`
- if the third argument is left as zero, the TCP/IP protocol will be selected automatically.

It uses the *listen* operation to listen on its socket for client requests for connections.

- the second argument to the *listen* system call specifies the maximum number of requests for connections that can be queued at this socket.

Stream communication

The server uses the *accept* system call to accept a connection requested by a client and obtain a new socket for communication with that client.

- the original socket may still be used to accept further connections with other clients.

The client process uses the *socket* operation to create a stream socket and then uses the *connect* system call to request a connection via the socket address of the listening process.

- as the *connect* call automatically binds a socket name to the caller's socket, prior binding is unnecessary.

Stream communication

After a connection has been established, both processes may then use the *write* and *read* operations on their respective sockets to send and receive sequences of bytes via the connection

- the *write* operation hands the message to the underlying TCP/IP protocol and returns the actual number of characters sent
- the *read* operation receives some characters in its buffer and returns the number of characters received.