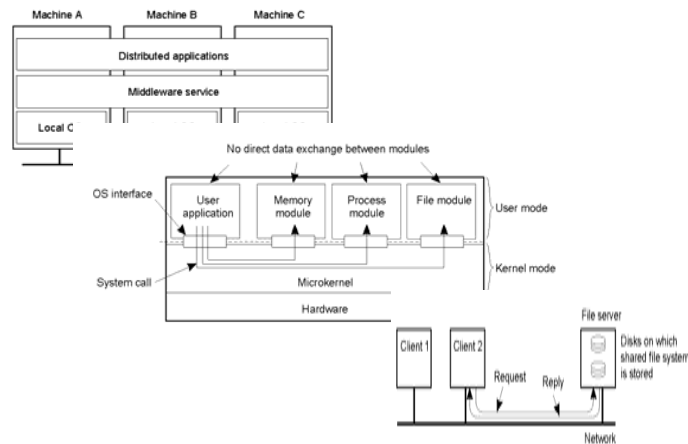




中山大學
SUN YAT-SEN UNIVERSITY

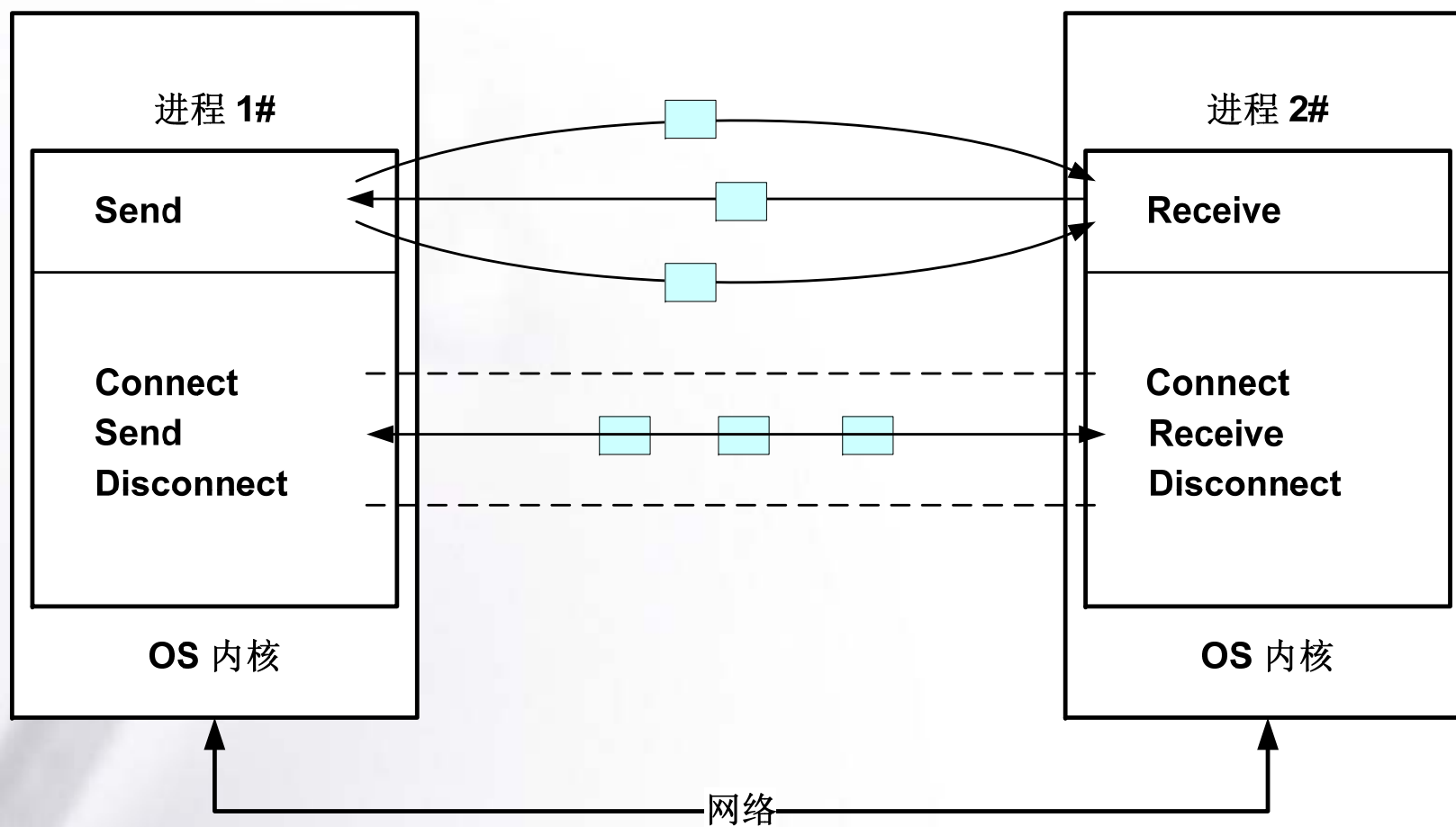


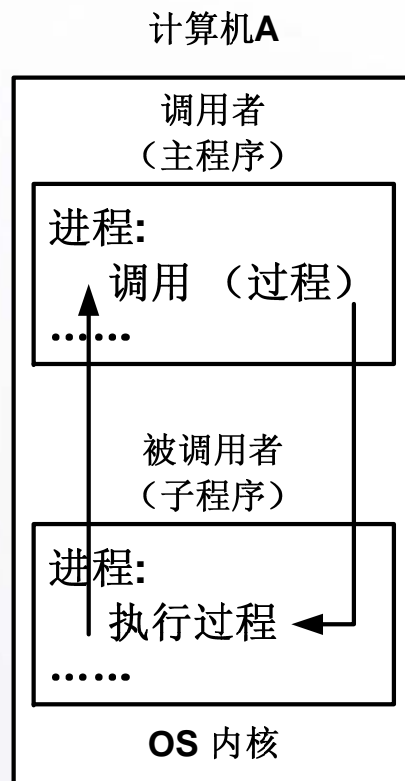
Distributed Computing Technology



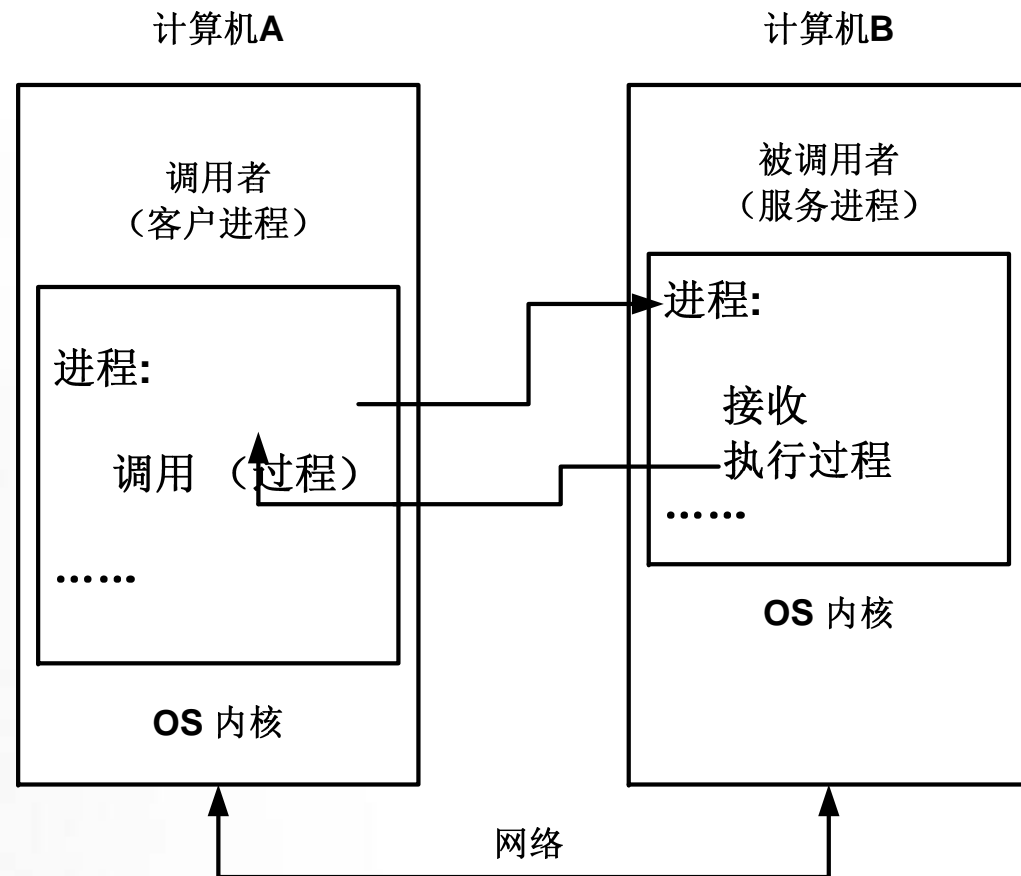
计算机A

计算机B





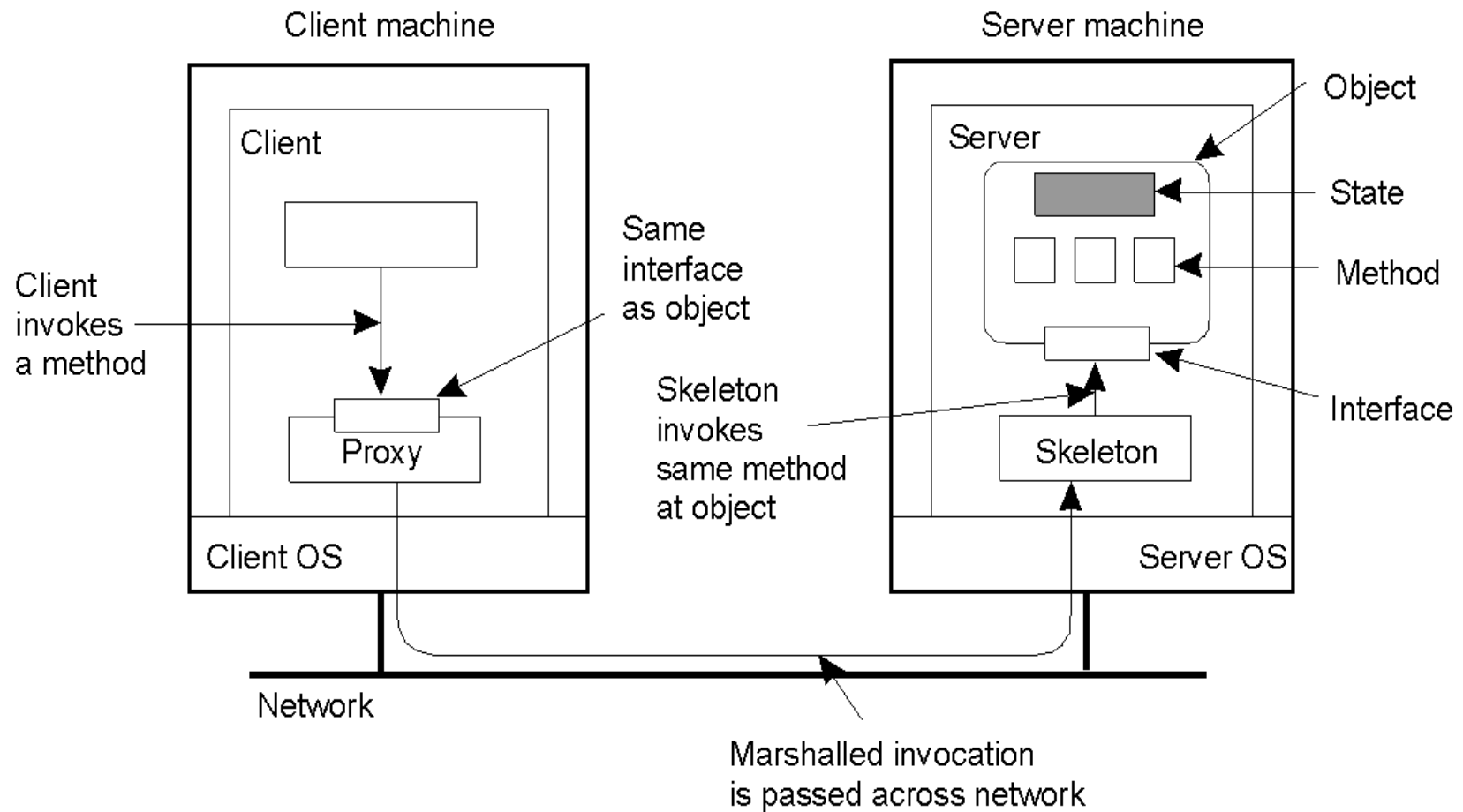
主程序调用子程序

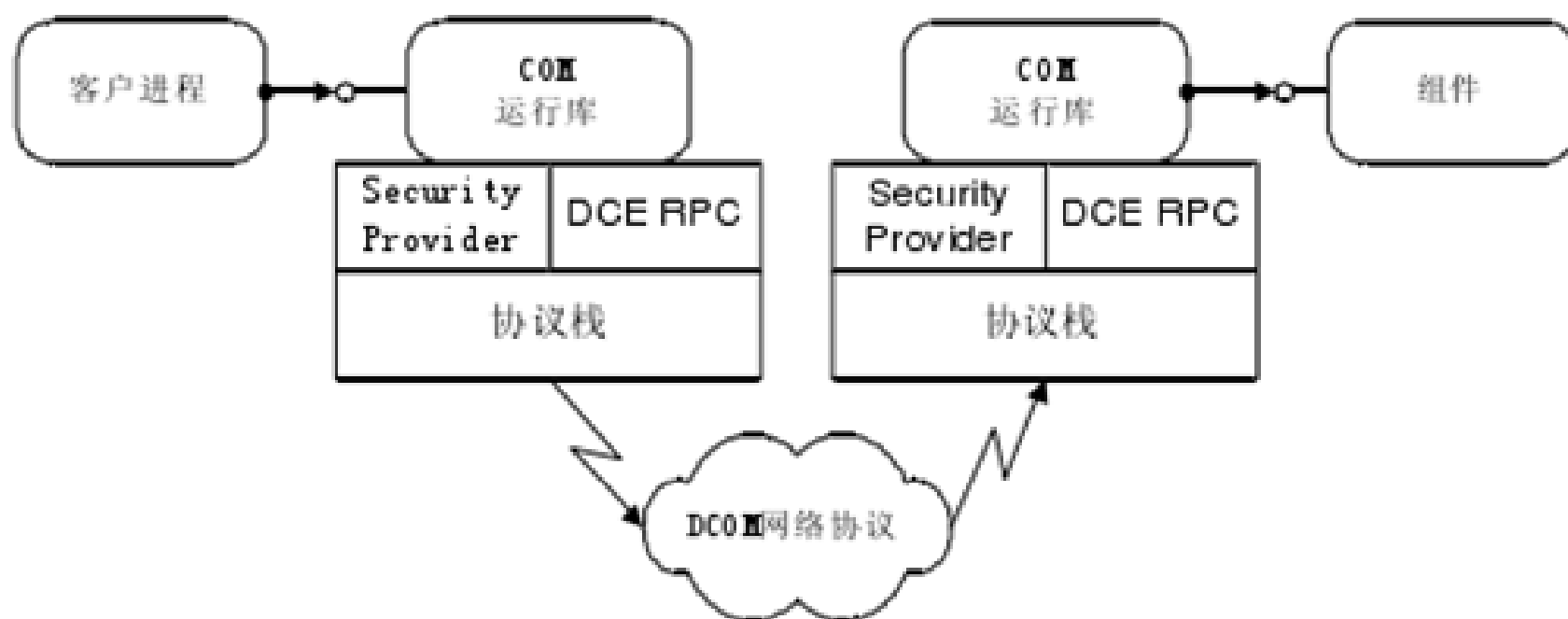


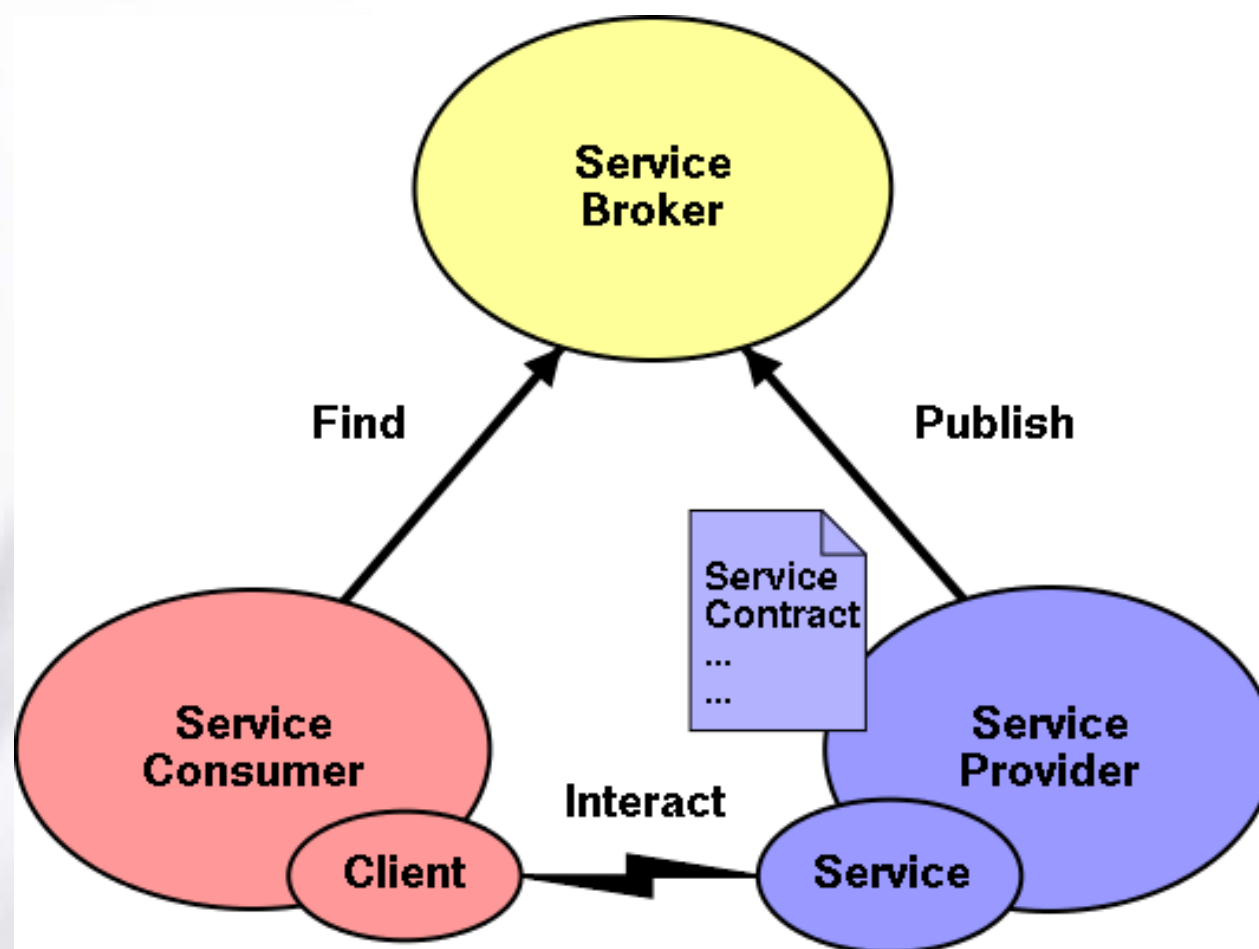
远程过程调用

Review

■ RMI(Remote Method Invoke)

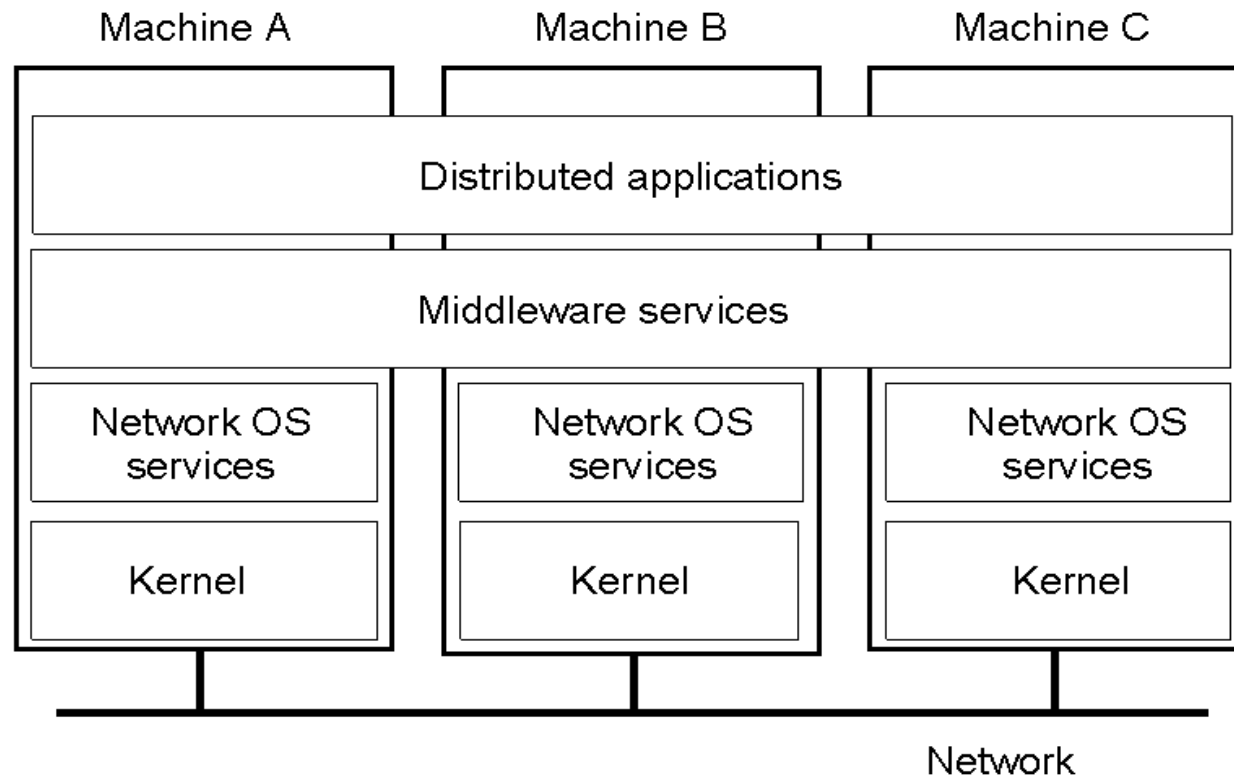






Review

分布式系统(中间件) Distributed Systems(Middleware)



- ➡ 1) 每个机器上的OS不需要知道别的机器
- ➡ 2) 不同机器上的OS可以是不同的
- ➡ 3) 服务是（透明地）分布在不同的计算机上

Review

■ Evolution of paradigms

- **Client-server: Socket API, remote method invocation**
- **Distributed objects**
- **Object broker: CORBA**
- **Network service: Jini**
- **Object space: JavaSpaces**
- **Mobile agents**
- **Message oriented middleware (MOM): Java Message Service**
- **Collaborative applications**

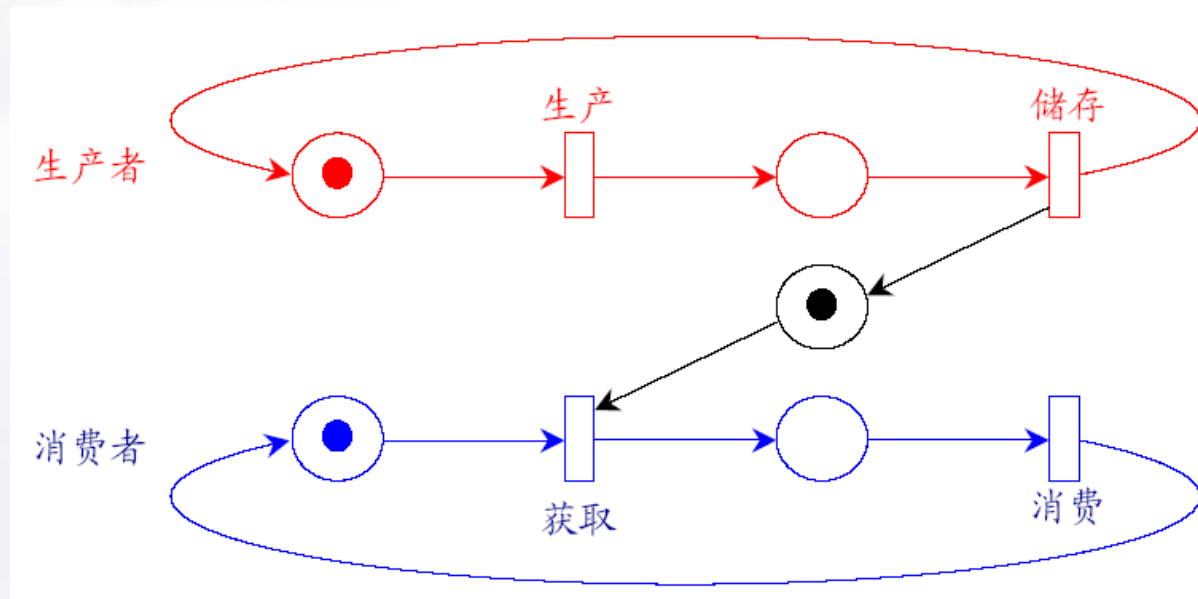
OMG组织的 **CORBA**
Sun的**EJB**;
Microsoft的**DCOM/COM+**

Topics

- **Socket API**
- **Client-Server Model**

Java语言:

- 每个线程都有自己的堆栈;
- 所有线程共享同一个堆.
- 线程间可利用共享的存储空间进行通信



程序运行时存储空间的管理

抽象背景:生产者和消费者共享一个缓冲区.

--- 当生产者快时,保证不会有数据遗漏消费.

--- 当消费者快时,保证不会有数据未生产就被消费.

此问题要求解决如下并发需求:

--- 互斥:任一时刻仅有一个生产者或消费者访问缓冲区.

--- 同步:缓冲区满时生产者等待,缓冲区空时消费者等待.

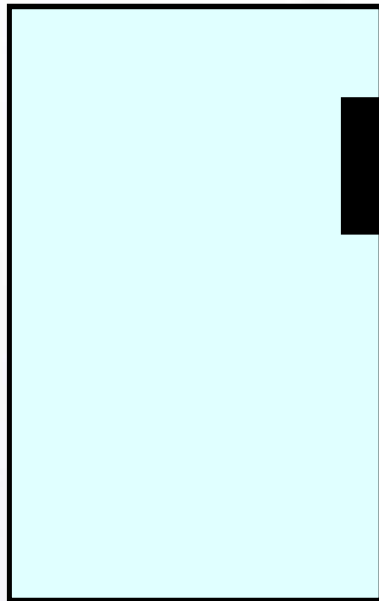
参考程序: *inprocess*

Introduction

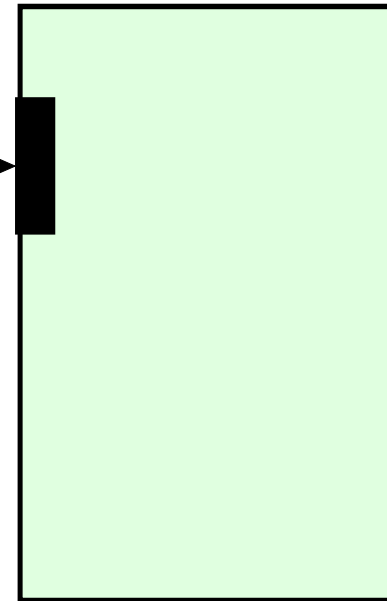
- ➔ The socket API is an Interprocessing Communication (IPC) programming interface originally provided as part of the Berkeley UNIX operating system.
- ➔ It has been ported to all modern operating systems, including Sun Solaris and Windows systems.
- ➔ It is a *de facto* standard for programming IPC, and is the basis of more sophisticated IPC interface such as remote procedure call and remote method invocation.

The conceptual model of the socket API

Process A



Process B



 a socket

The socket API

- ➔ A socket API provides a programming construct termed a socket. A process wishing to communicate with another process must create an instance, or instantiate, such a construct
- ➔ The two processes then issues operations provided by the API to send and receive data.

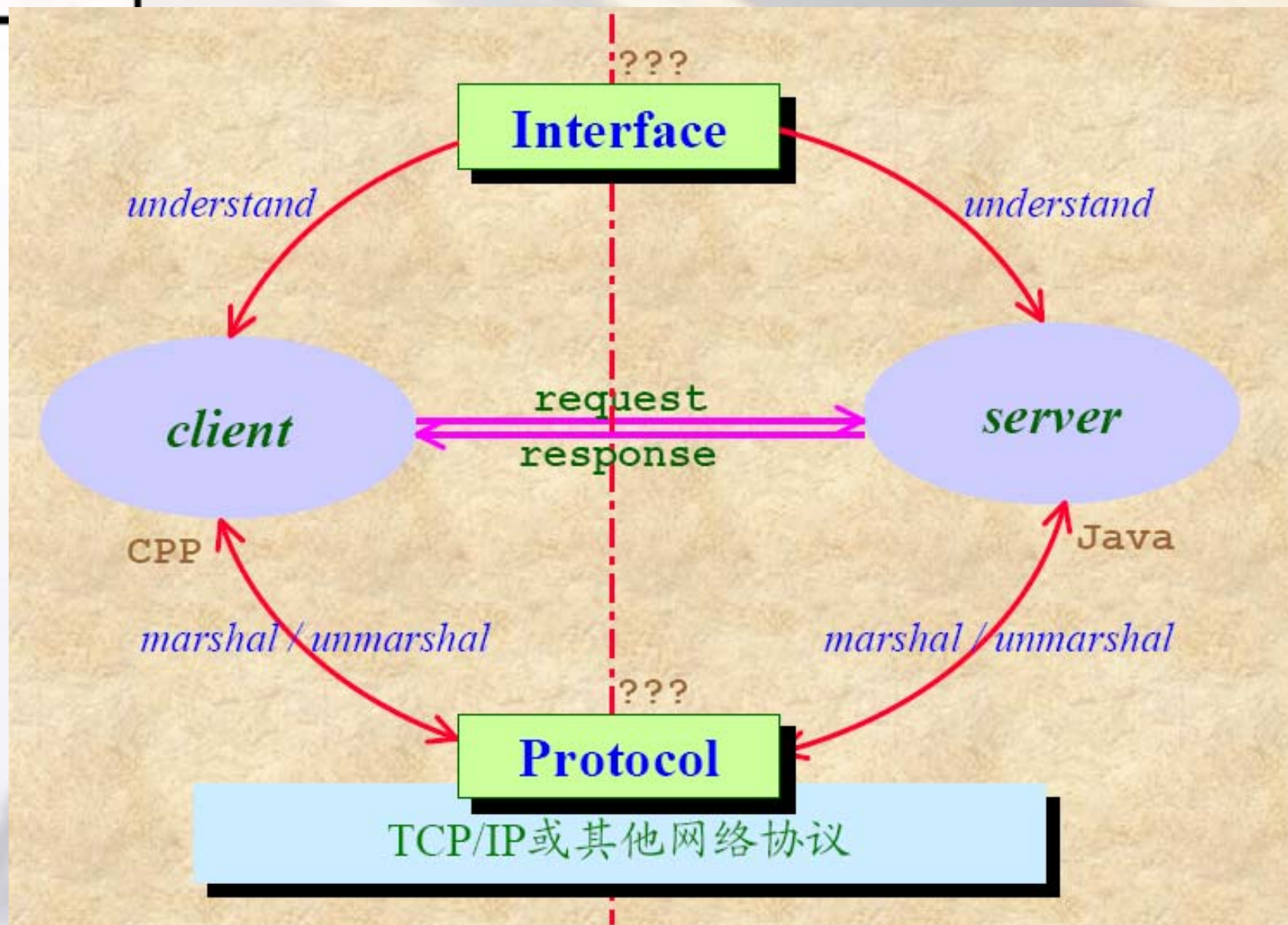
进程间(inter-process)通信

进程间通信(IPC)通常缺乏共享存储空间.

- 可借助分布式操作系统所支持的共享存储空间来实现;
- 但对Internet等异构平台?

消息传递(message Passing)是典型的通信模型

- 如何定义应用层接口?
- 如何约定传输层协议?



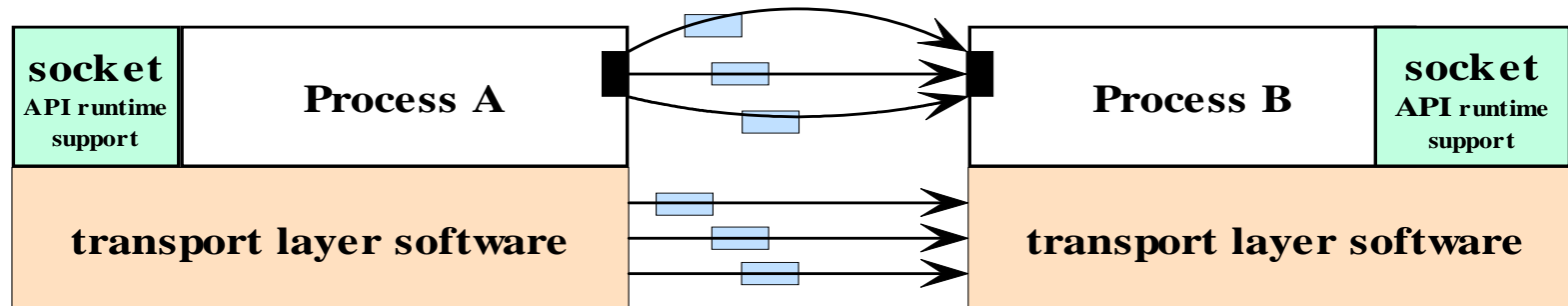
- ➔ A socket programming construct can make use of either the UDP or TCP protocol.
- ➔ Sockets that use UDP for transport are known as *datagram sockets*, while sockets that use TCP are termed *stream sockets*.
- ➔ Because of its relative simplicity, we will first look at datagram sockets, returning to stream sockets.

Datagram sockets can support both connectionless and connection-oriented communication at the application layer.



This is so because even though datagrams are sent or received without the notion of connections at the transport layer, the runtime support of the socket API can create and maintain logical connections for datagrams exchanged between two processes, as you will see in the next section.

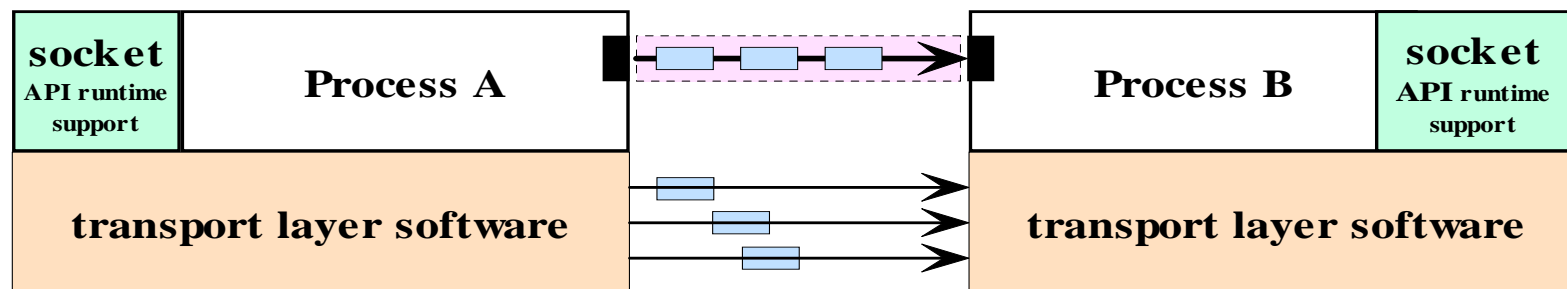
(The runtime support of an API is a set of software that is bound to the program during execution in support of the API.)

Connection-oriented & connectionless datagram socket



connectionless datagram socket

 a datagram
 a logical connection created and maintained by the runtime support of the datagram socket API



connection-oriented datagram socket

The Java Datagram Socket API

In Java, two classes are provided for the datagram socket API:

the *DatagramSocket* class for the sockets.

the *DatagramPacket* class for the datagram exchanged.

A process wishing to send or receive data using this API must instantiate a *DatagramSocket* object, or a socket in short. Each socket is said to be *bound* to a UDP port of the machine local to the process

The Java Datagram Socket API

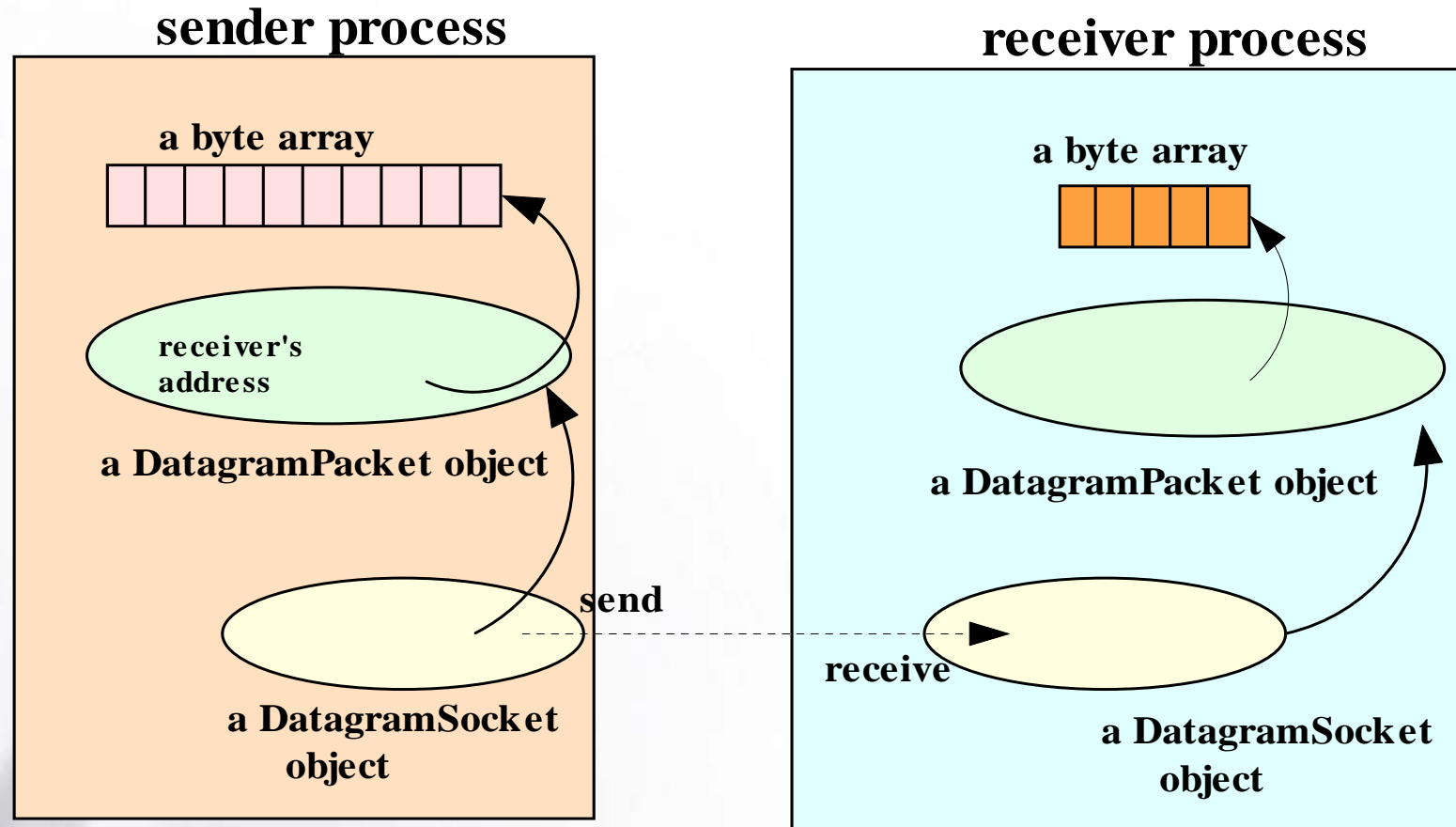
To send a datagram to another process, a process:

- ➔ creates an object that represents the datagram itself. This object can be created by instantiating a *DatagramPacket* object which carries
 - (i) the payload data as a reference to a byte array, and
 - (ii) the destination address (the host ID and port number to which the receiver's socket is bound).
- ➔ issues a call to a *send* method in the *DatagramSocket* object, specifying a reference to the *DatagramPacket* object as an argument

The Java Datagram Socket API

- In the receiving process, a *DatagramSocket* object must also be instantiated and bound to a local port, the port number must agree with that specified in the datagram packet of the sender.
- To receive datagrams sent to the socket, the process creates a *datagramPacket* object which references a byte array and calls a *receive* method in its *DatagramSocket* object, specifying as argument a reference to the *DatagramPacket* object.

The Data Structures in the sender and receiver programs



object reference

data flow

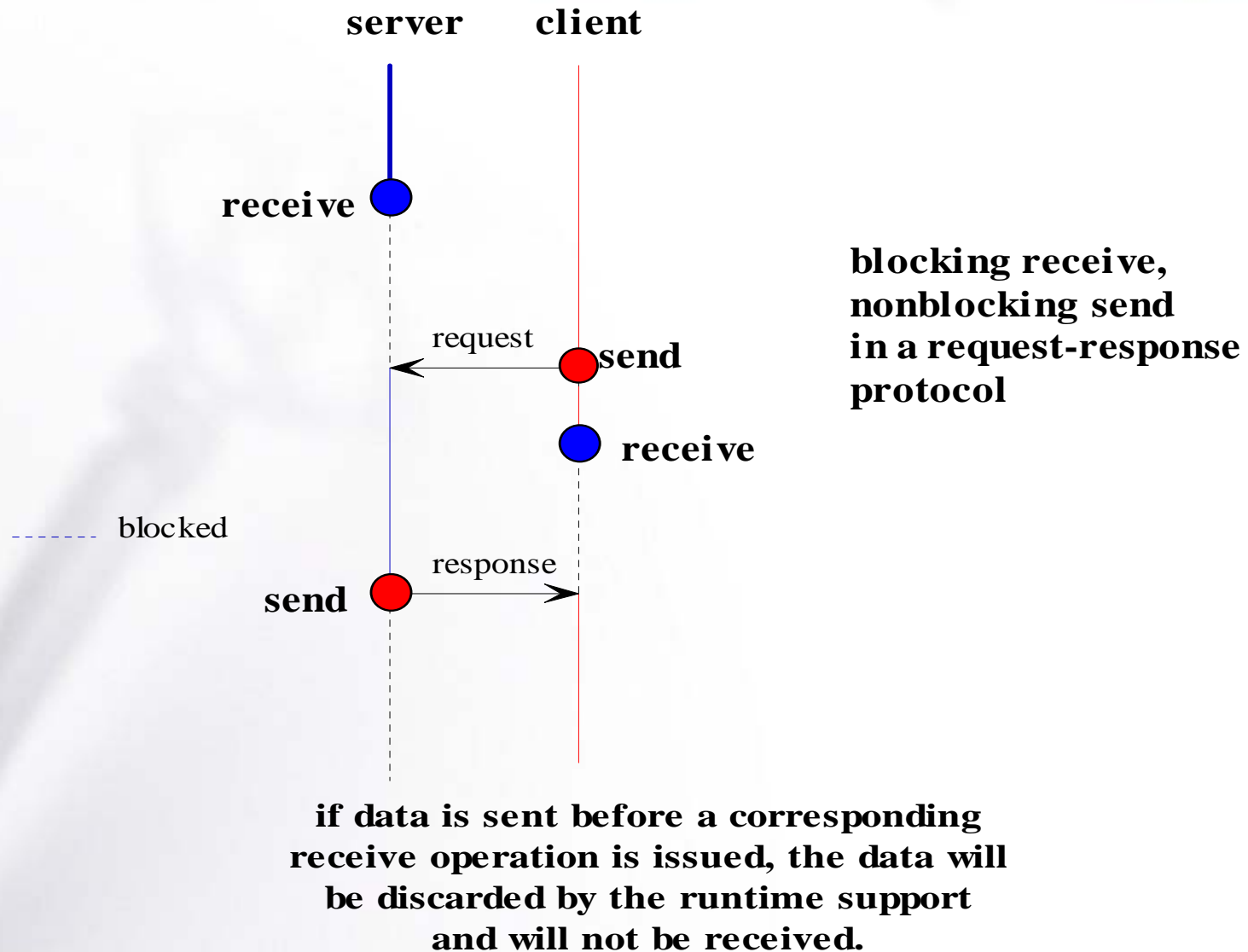
The program flow in the sender and receiver programs

sender program

create a datagram socket and bind it to any local port;
place data in a byte array;
create a datagram packet, specifying the data array and the receiver's address;
invoke the send method of the socket with a reference to the datagram packet;

receiver program

create a datagram socket and bind it to a specific local port;
create a byte array for receiving the data;
create a datagram packet, specifying the data array;
invoke the receive method of the socket with a reference to the datagram packet;



Setting timeout

To avoid indefinite blocking, a timeout can be set with a socket object:

```
void setSoTimeout(int timeout)
```

Set a timeout for the blocking receive from this socket, in milliseconds.

Once set, the timeout will be in effect for all blocking operations.

Key Methods and Constructors

Method/Constructor	Description
<u>DatagramPacket</u> (byte[] buf, int length)	Construct a datagram packet for receiving packets of length <i>length</i> ; data received will be stored in the byte array reference by <i>buf</i> .
<u>DatagramPacket</u> (byte[] buf, int length, <u>InetAddress</u> address, int port)	Construct a datagram packet for sending packets of length <i>length</i> to the socket bound to the specified port number on the specified host ; data received will be stored in the byte array reference by <i>buf</i> .
<u>DatagramSocket</u> ()	Construct a datagram socket and binds it to any available port on the local host machine; this constructor can be used for a process that sends data and does not need to receive data.
<u>DatagramSocket</u> (int port)	Construct a datagram socket and binds it to the specified port on the local host machine; the port number can then be specified in a datagram packet sent by a sender.
void close()	Close this datagramSocket object
void <u>receive</u> (<u>DatagramPacket</u> p)	Receive a datagram packet using this socket.
void send (<u>DatagramPacket</u> p)	Send a datagram packet using this socket.
void <u>setSoTimeout</u> (int timeout)	Set a timeout for the blocking receive from this socket, in milliseconds.

The coding

```
//Excerpt from a receiver program
DatagramSocket ds = new DatagramSocket(2345);
DatagramPacket dp =
    new DatagramPacket(buffer, MAXLEN);
ds.receive(dp);
len = dp.getLength( );
System.out.println(len + " bytes received.\n");
String s = new String(dp.getData( ), 0, len);
System.out.println(dp.getAddress( ) + " at port "
    + dp.getPort( ) + " says " + s);
```

```
// Excerpt from the sending process
InetAddress receiverHost=
    InetAddress.getByName("localhost");
DatagramSocket theSocket = new DatagramSocket( );
String message = "Hello world!";
byte[ ] data = message.getBytes( );
data = theLine.getBytes( );
DatagramPacket thePacket
    = new DatagramPacket(data, data.length,
        receiverHost, 2345);
theSocket.send(theOutput);
```

Connectionless sockets

With connectionless sockets, it is possible for multiple processes to simultaneously send datagrams to the same socket established by a receiving process, in which case the order of the arrival of these messages will be unpredictable, in accordance with the UDP protocol

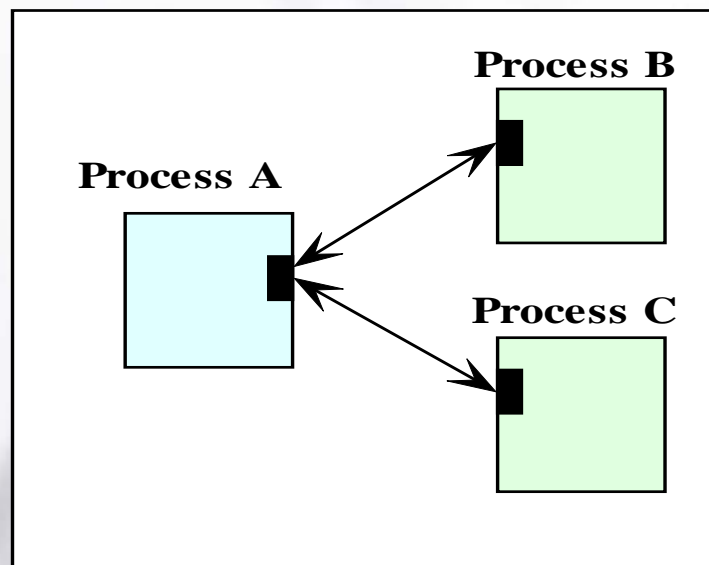


Figure 3a

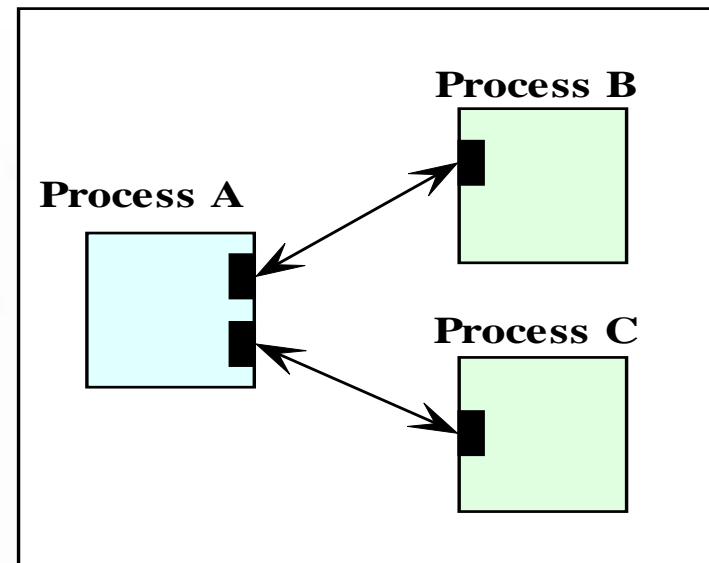


Figure 3b

■
a connectionless
datagram socket

Code samples

- ➔ **Example1Sender.java, ExampleReceiver.java**
- ➔ **MyDatagramSocket.java,
Example2SenderReceiver.java ,
Example2ReceiverSender.java**

Connection-oriented datagram socket API

It is uncommon to employ datagram sockets for connection-oriented communication; the connection provided by this API is rudimentary and typically insufficient for applications that require a true connection. Stream-mode sockets, which will be introduced later, are more typical and more appropriate for connection-oriented communication.

Methods calls for connection-oriented datagram socket

Method/Constructor	Description
public void connect (InetAddress address, int port)	Create a logical connection between this socket and a socket at the remote address and port.
public void disconnect ()	Cancel the current connection, if any, from this socket.

A connection is made for a socket with a remote socket.

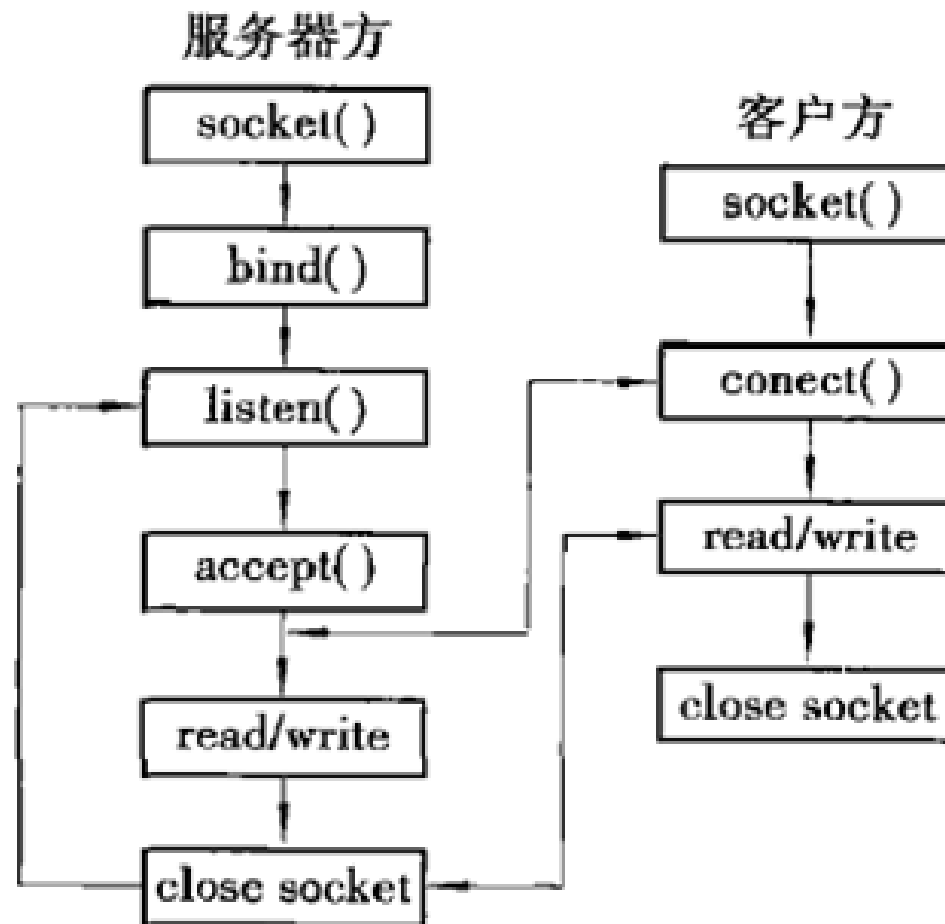
Once a socket is connected, it can only exchange data with the remote socket.

If a datagram specifying another address is sent using the socket, an `IllegalArgumentException` will occur.

If a datagram from another socket is sent to this socket, The data will be ignored.

The connection is unilateral, that is, it is enforced only on one side. The socket on the other side is free to send and receive data to and from other sockets, unless it too commits to a connection to the other socket.

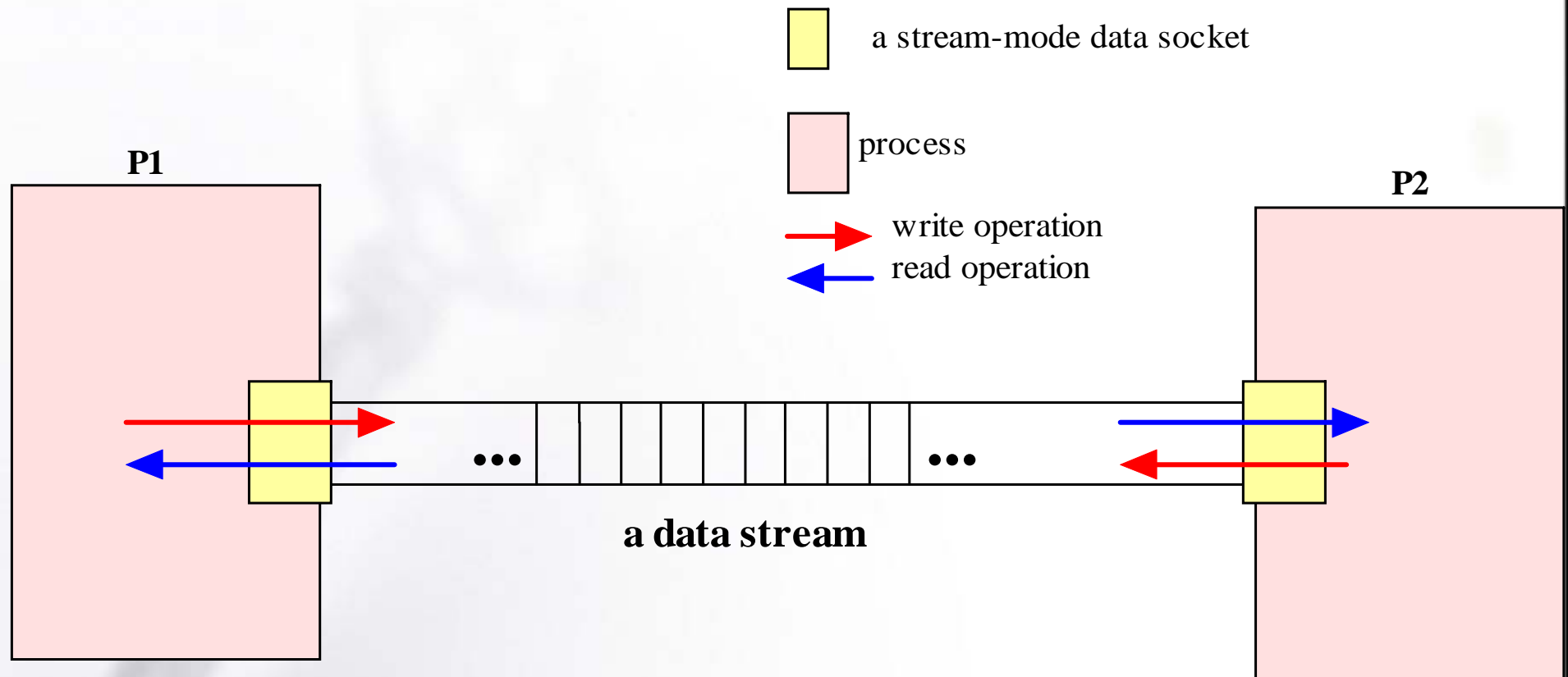
See Example3Sender, Example3Receiver.



The Stream-mode Socket API

- ➔ The datagram socket API supports the exchange of discrete units of data (that is, datagrams).
- ➔ the stream socket API provides a model of data transfer based on the stream-mode I/O of the Unix operating systems.
- ➔ By definition, a stream-mode socket supports connection-oriented communication only.

Stream-mode Socket API (connection-oriented socket API)



- ➔ **A stream-mode socket is established for data exchange between two specific processes.**
- ➔ **Data stream is written to the socket at one end, and read from the other end.**
- ➔ **A stream socket cannot be used to communicate with more than one process.**

In Java, the stream-mode socket API is provided with two classes:

Server socket: for accepting connections; we will call an object of this class a connection socket.

Socket: for data exchange; we will call an object of this class a data socket.

Stream-mode Socket API program flow

connection listener (server)

create a connection socket
and listen for connection
requests;
accept a connection;
creates a data socket for reading from
or writing to the socket stream;
get an input stream for reading
to the socket;
read from the stream;
get an output stream for writing
to the socket;
write to the stream;
close the data socket;
close the connection socket.

connection requester (server)

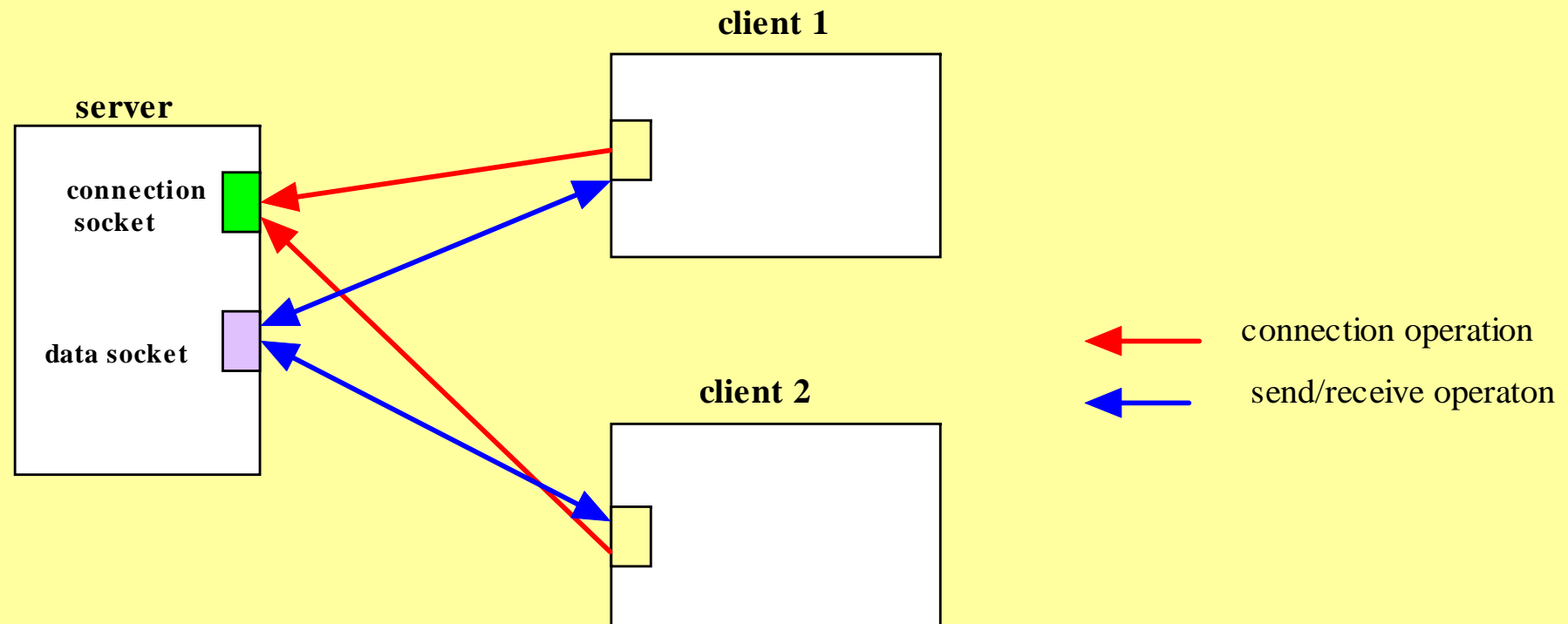
create a data socket
and request for a connection;

get an output stream for writing
to the socket;
write to the stream;

get an input stream for reading
to the socket;
read from the stream;
close the data socket.

The server (the connection listener)

A server uses two sockets: one for accepting connections, another for send/receive



Key methods in the ServerSocket class

Method/constructor	Description
<code>ServerSocket(int port)</code>	Creates a server socket on a specified port.
<code>Socket accept()</code> throws <code>IOException</code>	Listens for a connection to be made to this socket and accepts it. The method blocks until a connection is made.
<code>public void close()</code> throws <code>IOException</code>	Closes this socket.
<code>void setSoTimeout(int timeout)</code> throws <code>SocketException</code>	Set a timeout period (in milliseconds) so that a call to <code>accept()</code> for this socket will block for only this amount of time. If the timeout expires, a <code>java.io.InterruptedIOException</code> is raised

Note: Accept is a blocking operation.

Key methods in the Socket class

Method/constructor	Description
<code>Socket(InetAddress address, int port)</code>	Creates a stream socket and connects it to the specified port number at the specified IP address
<code>void close()</code> throws <code>IOException</code>	Closes this socket.
<code>InputStream getInputStream()</code> throws <code>IOException</code>	Returns an input stream so that data may be read from this socket.
<code>OutputStream getOutputStream()</code> throws <code>IOException</code>	Returns an output stream so that data may be written to this socket.
<code>void setSoTimeout(int timeout)</code> throws <code>SocketException</code>	Set a timeout period for blocking so that a <code>read()</code> call on the <code>InputStream</code> associated with this <code>Socket</code> will block for only this amount of time. If the timeout expires, a <code>java.io.InterruptedIOException</code> is raised

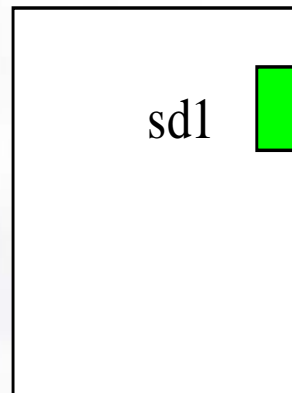
A read operation on the `InputStream` is blocking.

A write operation is nonblocking.

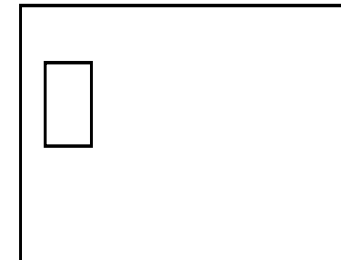
Connection-oriented socket API-3

1. Server establishes a socket sd1 with local address, then listens for incoming connection on sd1

server

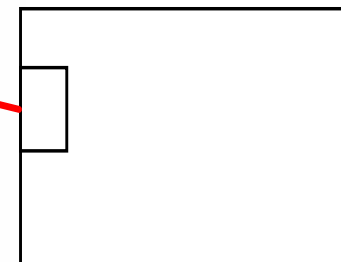
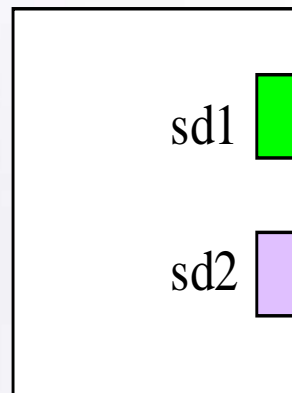


client



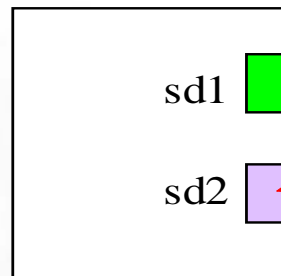
Client establishes a socket with remote (server's) address.

2. Server accepts the connection request and creates a new socket sd2 as a result.



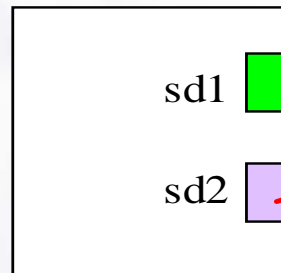
Connection-oriented socket API-3

3. Server issues receive operation using sd2.

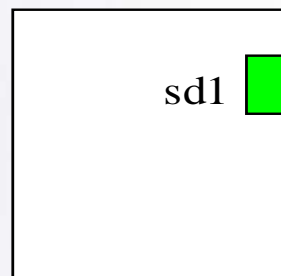


Client issues send operation.

4. Server sends response using sd2.



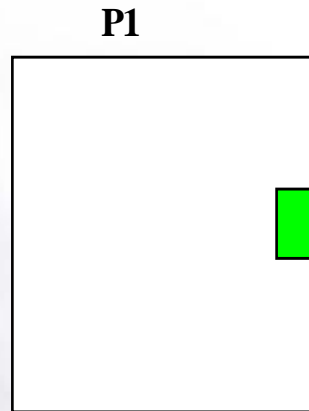
5. When the protocol has completed, server closes sd2; sd1 is used to accept the next connection



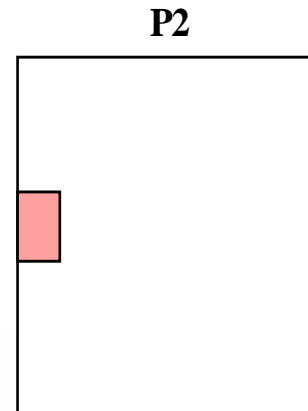
Client closes its socket when the protocol has completed

Connectionless socket API

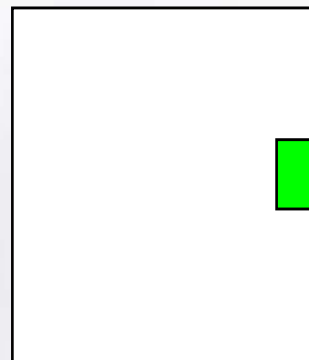
**P1 establishes
a local socket**



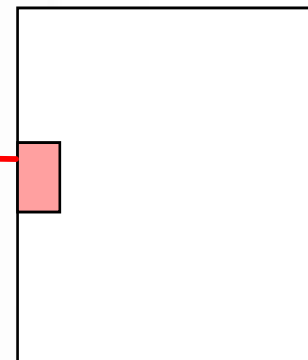
**P2 establishes
a local socket**



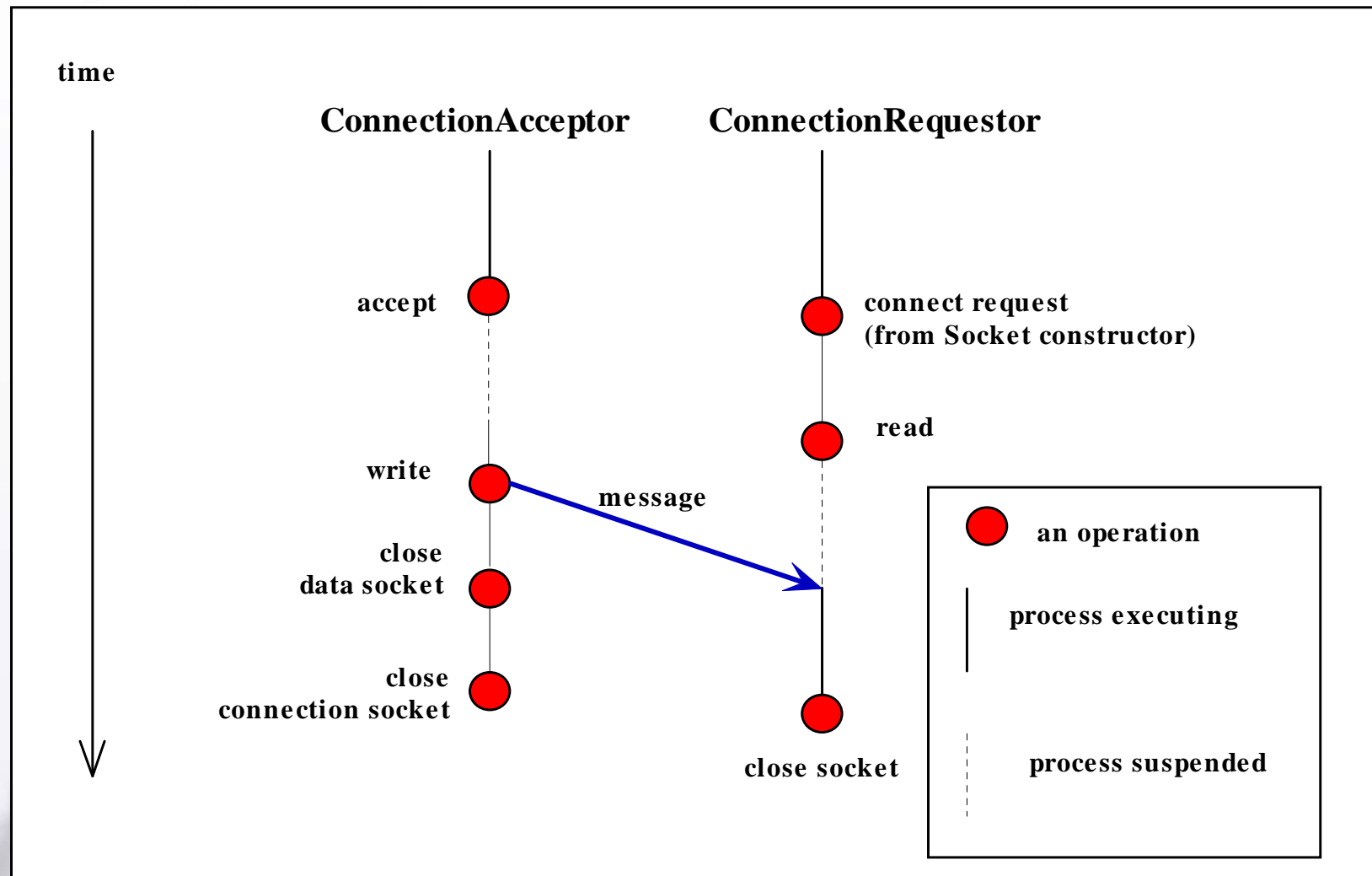
**P1 issues a
receive operation
to receive the
datagram.**



**P2 sends a datagram
addressed to P1**



Example 4 Event Diagram



Example4

Example4ConnectionAcceptor

```
try {  
    int portNo;  
    String message;  
    // instantiates a socket for accepting connection  
    ServerSocket connectionSocket = new ServerSocket(portNo);  
    Socket dataSocket = connectionSocket.accept();  
  
    // get a output stream for writing to the data socket  
    OutputStream outputStream = dataSocket.getOutputStream();  
    // create a PrintWriter object for character-mode output  
    PrintWriter socketOutput =  
        new PrintWriter(new OutputStreamWriter(outputStream));  
    // write a message into the data stream  
    socketOutput.println(message);  
    //The ensuing flush method call is necessary for the data to  
    // be written to the socket data stream before the  
    // socket is closed.  
    socketOutput.flush();  
    dataSocket.close();  
    connectionSocket.close();  
} // end try  
catch (Exception ex) {  
    System.out.println(ex);  
}
```

Example4ConnectionReceiver

```
try {  
    InetAddress acceptorHost =  
        InetAddress.getByName(args[0]);  
    int acceptorPort = Integer.parseInt(args[1]);  
    // instantiates a data socket  
    Socket mySocket = new Socket(acceptorHost, acceptorPort);  
    // get an input stream for reading from the data socket  
    InputStream inStream = mySocket.getInputStream();  
    // create a BufferedReader object for text-line input  
    BufferedReader socketInput =  
        new BufferedReader(new InputStreamReader(inStream));  
    // read a line from the data stream  
    String message = socketInput.readLine();  
    System.out.println("\t" + message);  
    mySocket.close();  
} // end try  
catch (Exception ex) {  
    System.out.println(ex);  
}
```

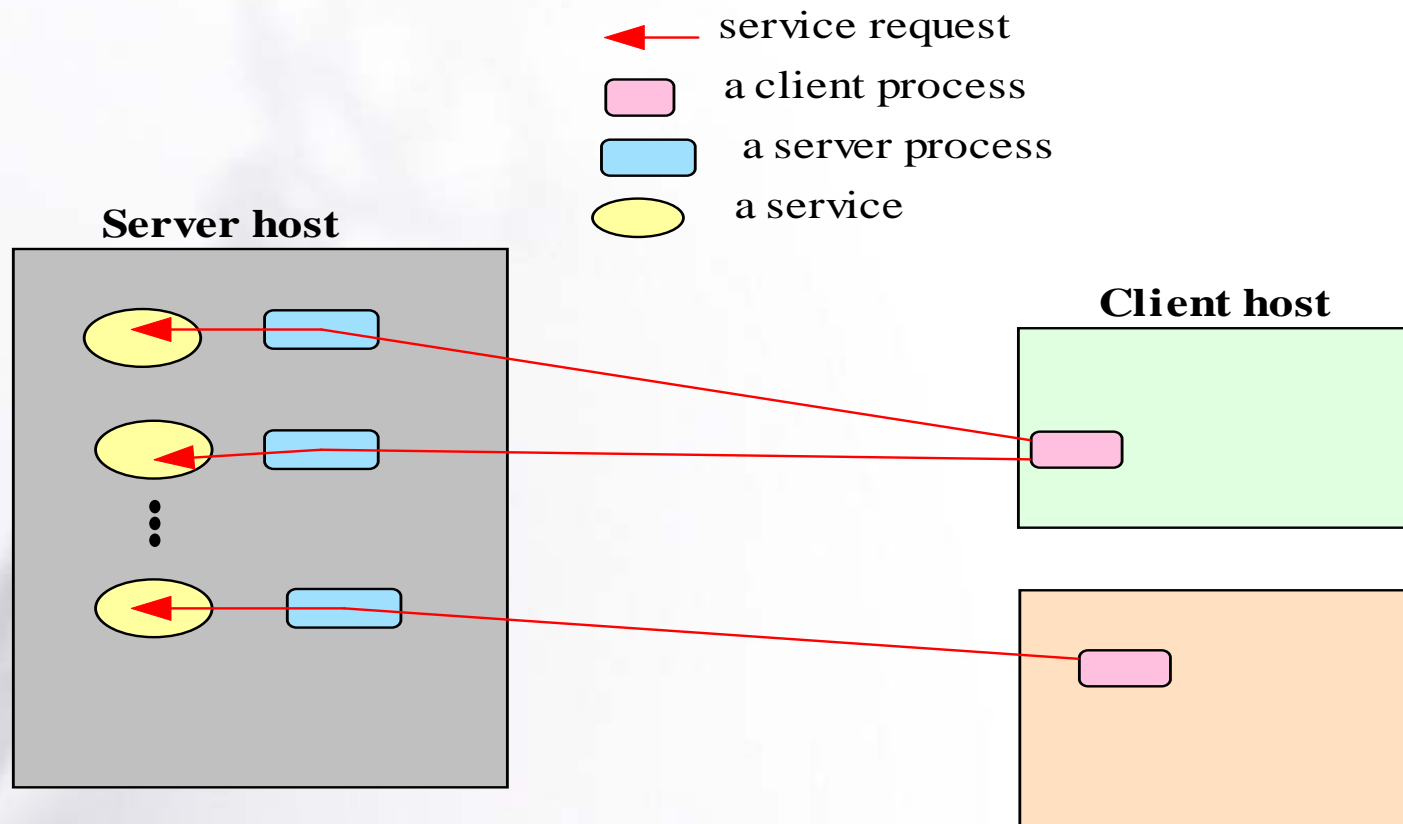

1. Socket API

2. Client-Server Model

- ➔ **The Client-Server paradigm is the most prevalent model for distributed computing protocols.**
- ➔ **It is the basis of all distributed computing paradigms at a higher level of abstraction.**
- ➔ **It is service-oriented, and employs a request-response protocol.**

The Client-Server Paradigm

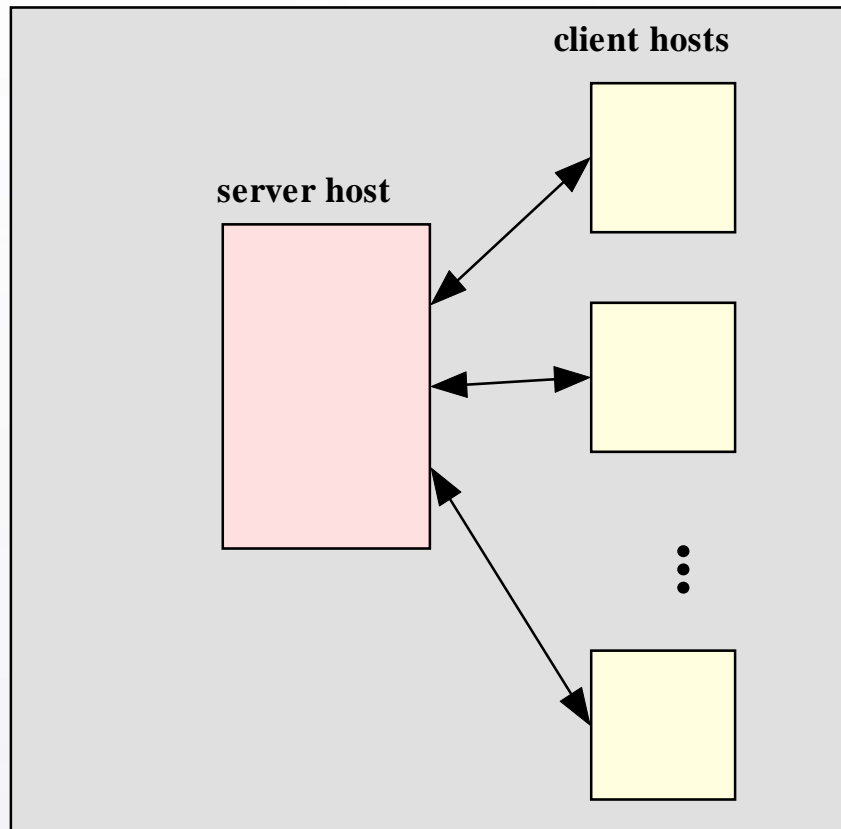
- ➔ A server process, running on a server host, provides access to a service.
- ➔ A client process, running on a client host, accesses the service via the server process.
- ➔ The interaction of the process proceeds according to a protocol.



- ➔ An application based on the client-server paradigm is a client-server application.
- ➔ On the Internet, many services are Client-server applications. These services are often known by the protocol that the application implements.
- ➔ Well known Internet services include HTTP, FTP, DNS, finger, gopher, etc.
- ➔ User applications may also be built using the client-server paradigm.

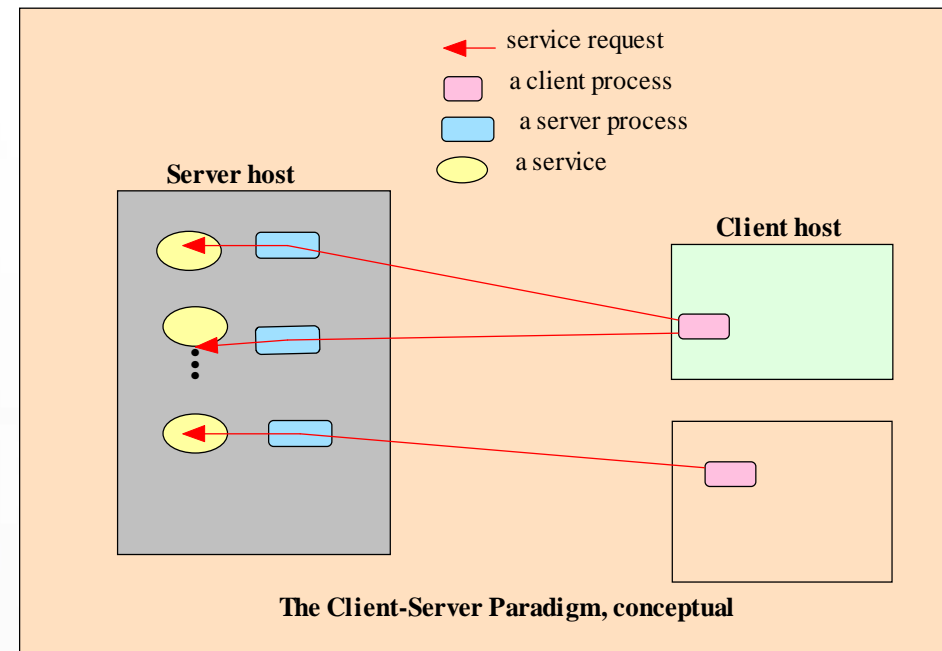
In the client-server system architecture, the terms clients and servers refer to computers, while in the client-server distributed computing paradigm, the terms refer to processes.

Client-server, an overloaded term



Client-Server System Architecture

Client hosts make use of services provided on a server host.



Client-Server Computing Paradigm

Client processes (objects) make use of a service provided by a server process (object) running on a server host.

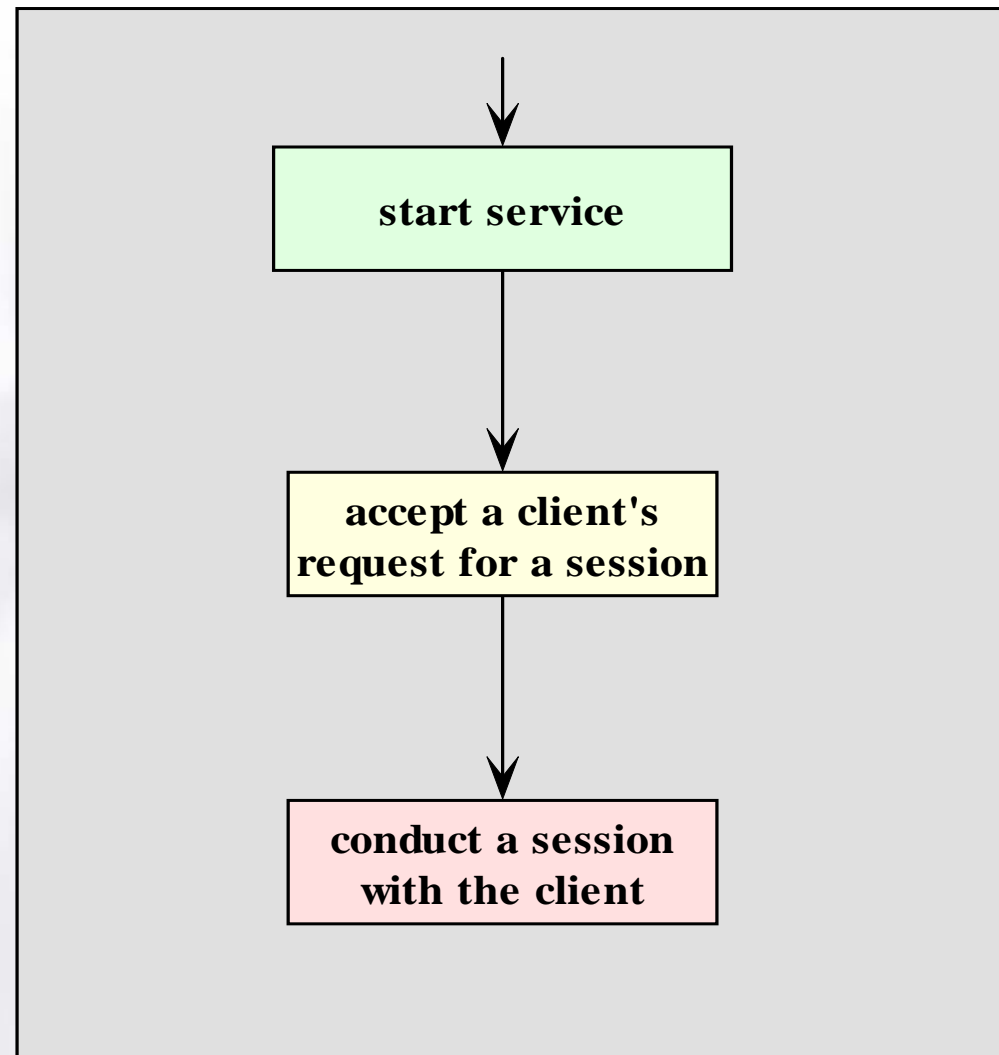
A protocol/service session

In the context of the client-server model, we will use the term session to refer to the interaction between the server and one client.

The service managed by a server may be accessed by multiple clients who desire the service, sometimes concurrently.

Each client, when serviced by the server, engages in a separate session with the server, during which it conducts a dialog with the server until the client has obtained the service it required

A service session



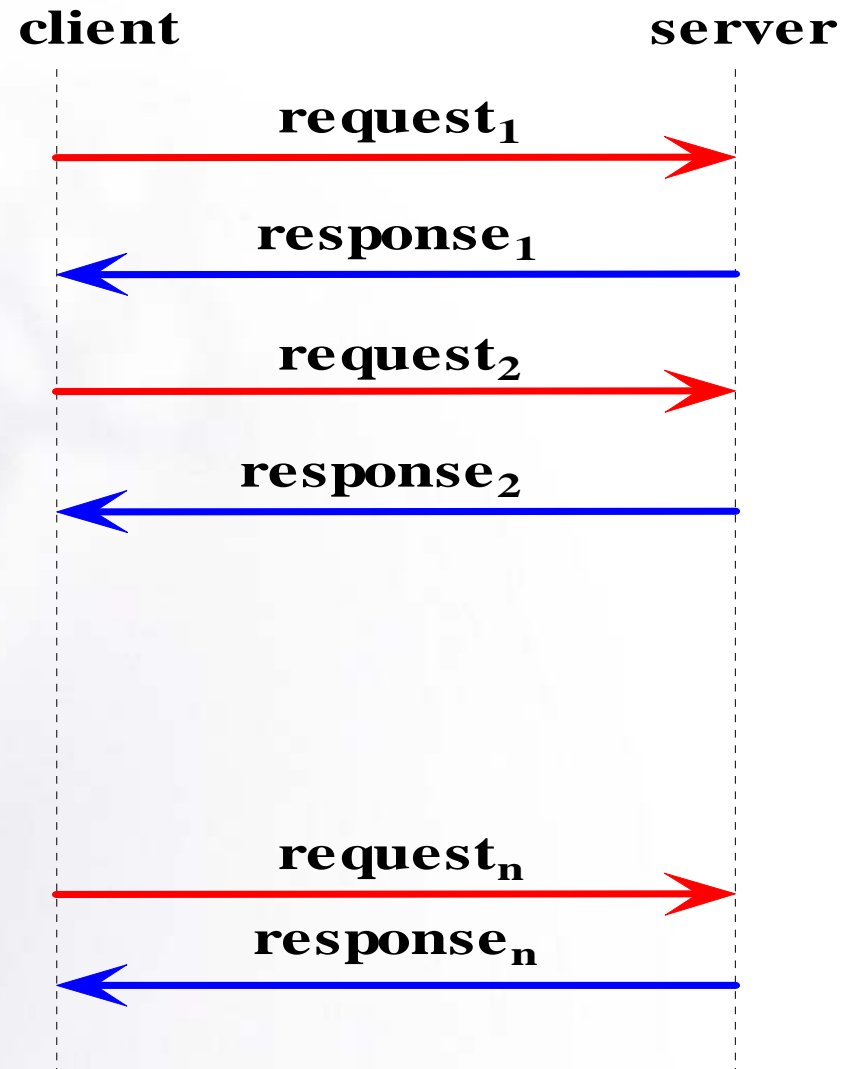
The Protocol for a Network Service

- ➔ A protocol is needed to specify the rules that must be observed by the client and the server during the conductin of a service.
- ➔ Such rules include specifications on matters such as
 - (i) how the service is to be located
 - (ii) the sequence of interprocess communication
 - (iii) the representation and interpretation of data exchanged with each IPC.
- ➔ On the Internet, such protocols are specified in the RFCs.

Locating the service

- A mechanism must be available to allow a client process to locate a server for a given service.
- A service can be located through the address of the server process, in terms of the host name and protocol port number assigned to the server process. This is the scheme for Internet services. Each Internet service is assigned to a specific port number. In particular, a well-known service such as ftp, HTTP, or telnet is assigned a default port number reserved on each Internet host for that service.
- At a higher level of abstraction, a service may be identified using a logical name registered with a registry, the logical name will need to be mapped to the physical location of the server process. If the mapping is performed at runtime (that is, when a client process is run), then it is possible for the service's location to be dynamic, or moveable.

The interprocess communications and event synchronization



Any implementation of the client or server program for this service is expected to adhere to the specification for the protocol, including how the dialogs of each session should proceed.

Among other things, the specification defines

- (i) which side (client or server) should speak first,
- (ii) the syntax and semantic of each request and response
- (iii) the action expected of each side upon receiving a particular request or response.

Session IPC examples

The dialog in each session follows a pattern prescribed in the protocol specified for the service.

Daytime service [RFC867]:

Client: Hello, <client address> here. May I have a timestamp please.

Server: Here it is: (time stamp follows)

World Wide Web session:

Client: Hello, <client address> here.

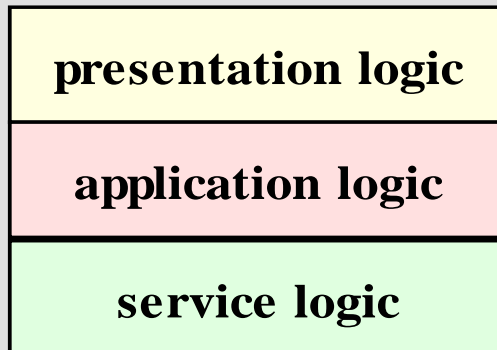
Server: Okay. I am a web server and speaks protocol HTTP1.0.

Client: Great, please get me the web page index.html at the root of your document tree.

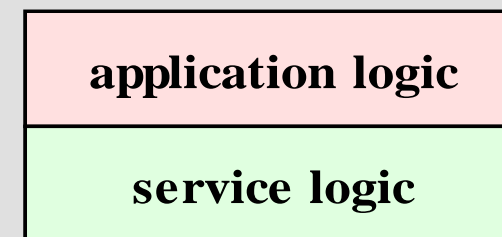
Server: Okay, here's what's in the page: (contents follows).

- ➔ **Part of the specification of a protocol is the syntax and semantics of each request and response.**
- ➔ **The choice of data representation depends on the nature and the needs of the protocol.**
- ➔ **Representing data using text (character strings) is common, as it facilitates data marshalling and allows the data to be readable by human.**
- ➔ **Most well known Internet protocols are client-server, request-response, and text-base.**

client-side software



server-side software



Client-side presentation logic

DaytimeClient1.java encapsulates the client-side presentation logic; that is, it provides the interface for a user of the client process. You will note that the code in this class is concerned with obtaining input (the server address) from the user, and displaying the output (the timestamp) to the user.

To obtain the timestamp, a method call to a “helper” class, *DaytimeClientHelper1.java*, is issued. This method hides the details of the application logic and the underlying service logic. In particular, the programmer of *DaytimeClient1.java* need not be aware of which socket types is used for the IPC.

Daytime Client-server using Connectionless Datagram Socket

Client-side Application logic

The *DaytimeClientHelper1.java* class (Figure 6b) encapsulates the client-side application logic. This module performs the IPC for sending a request and receiving a response, using a specialized class of the *DatagramSocket*, *myClientDatagramSocket*. Note that the details of using datagram sockets are hidden from this module. In particular, this module does not need to deal with the byte array for carrying the payload data.

Service logic

The *MyClientDatagram.java* (Figure 6c) class provides the details of the IPC service, in this case using the datagram socket API.

Server-side software

Presentation logic

Typically, there is very little presentation logic on the server-side. In this case, the only user input is for the server port, which, for simplicity, is handled using a command-line argument.

Application logic

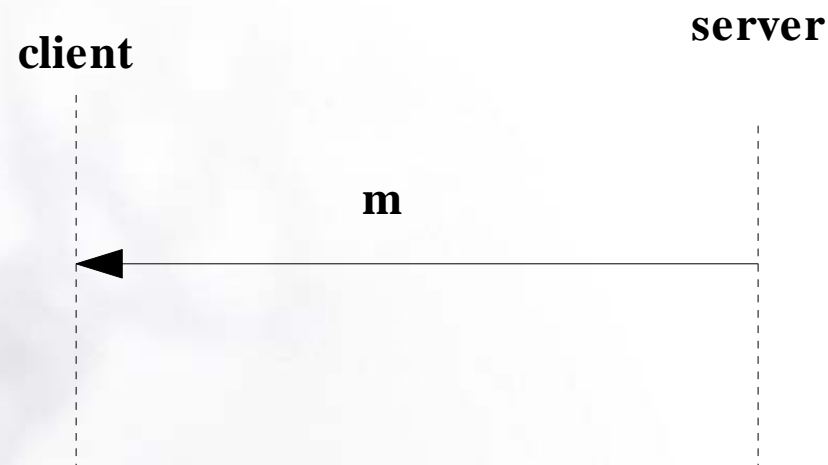
The *DaytimeServer1.java* class encapsulates the server-side application logic. This module executes in a forever loop, waiting for a request from a client and then conduct a service session for that client. The module performs the IPC for receiving a request and sending a response, using a specialized class of the *DatagramSocket*, *myServerDatagramSocket*. Note that the details of using datagram sockets are hidden from this module. In particular, this module does not need to deal with the byte array for carrying the payload data.

Service logic

The *MyServerDatagram.java* class provides the details of the IPC service, in this case using the datagram socket API.

Example protocol: daytime

Defined in [RFC867](#)



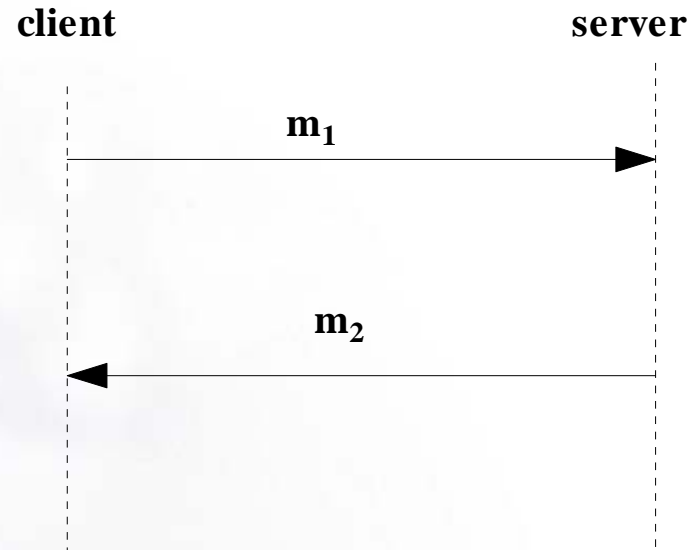
sequence diagram

data representation: text (character strings)

data format:

**m : contains a timestamp, in a format such as
Wed Jan 30 09:52:48 2002**

Daytime Protocol



sequence diagram

data representation: text (character strings)

data format:

m1; a null message - contents will be ignored.

m2 : contains a timestamp, in a format such as

Wed Jan 30 09:52:48 2002

Daytime Protocol Implementation

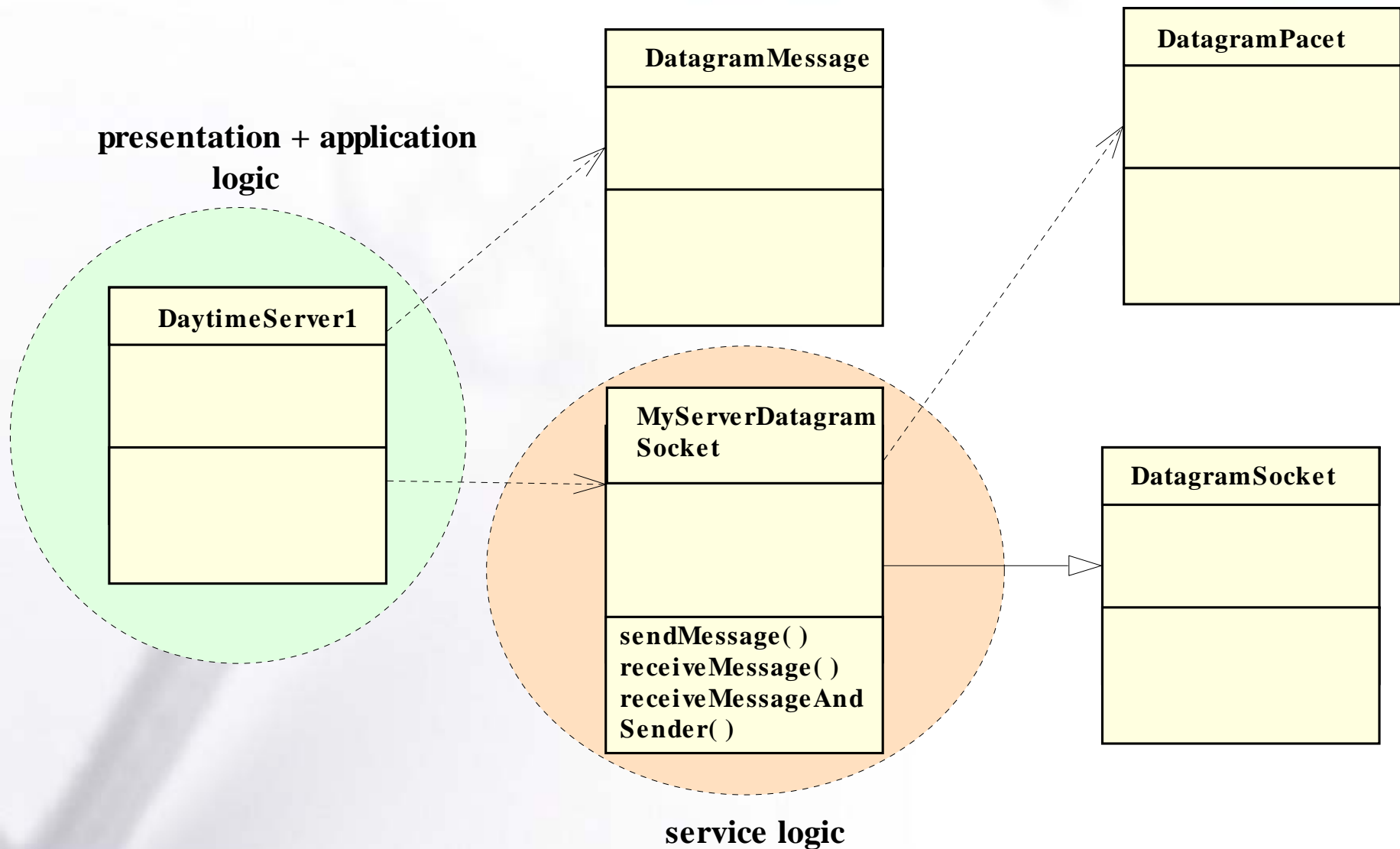
Sample 1 – using connectionless sockets:

DaytimeServer1.java

DaytimeClient1.java

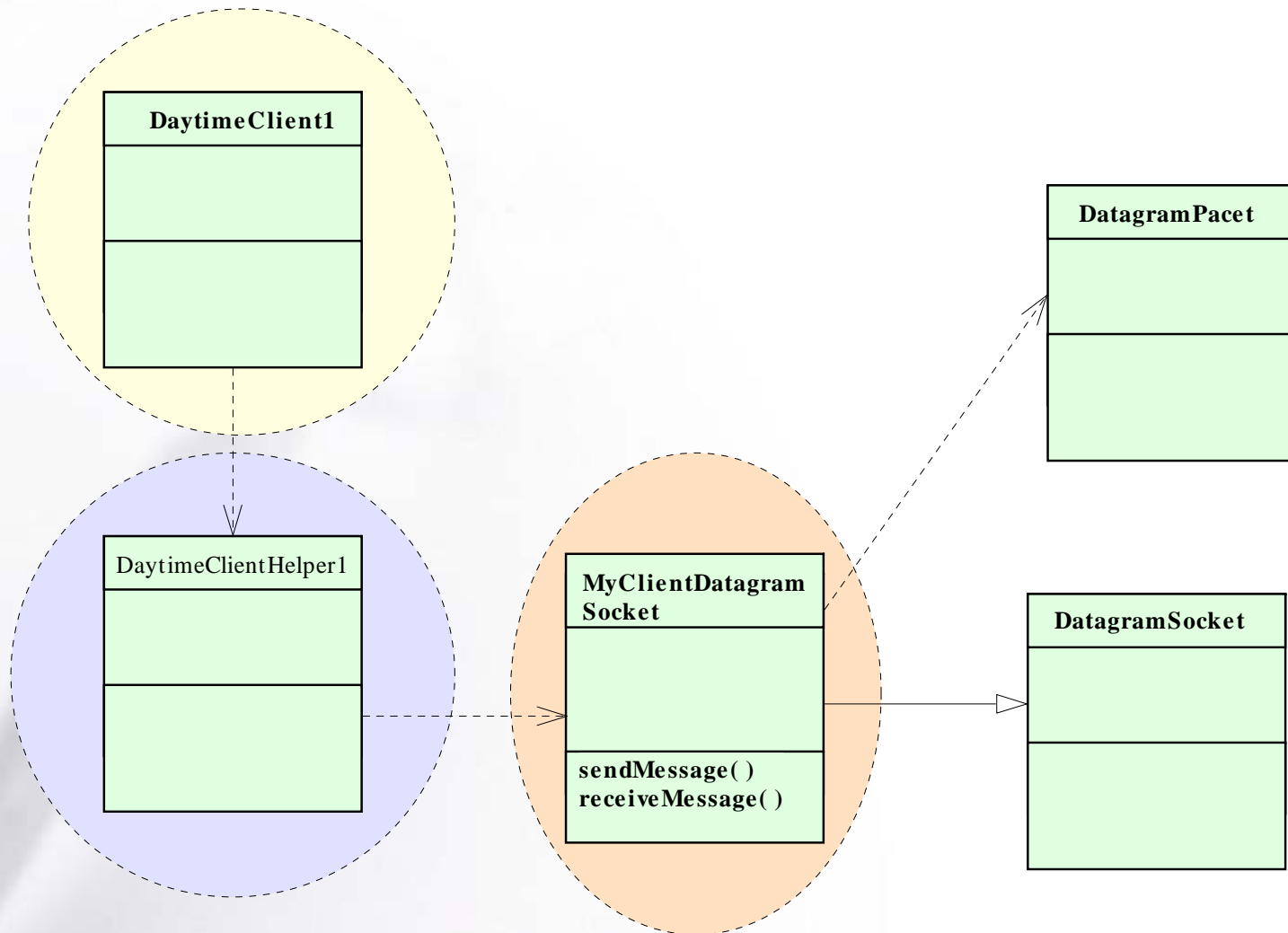
The getAddress and getPort Methods

Method (of DatagramPacket class)	Description
public InetAddress getAddress()	Return the IP address of the remote host from a socket of which the datagram was received.
public int getPort()	Return the port number on the remote host from a socket of which the datagram was received.



UML Diagram for the Datagram Daytime Client

presentation logic



application logic

Connection-oriented Daytime Server Client:

Sample 2 – using connection-oriented sockets:

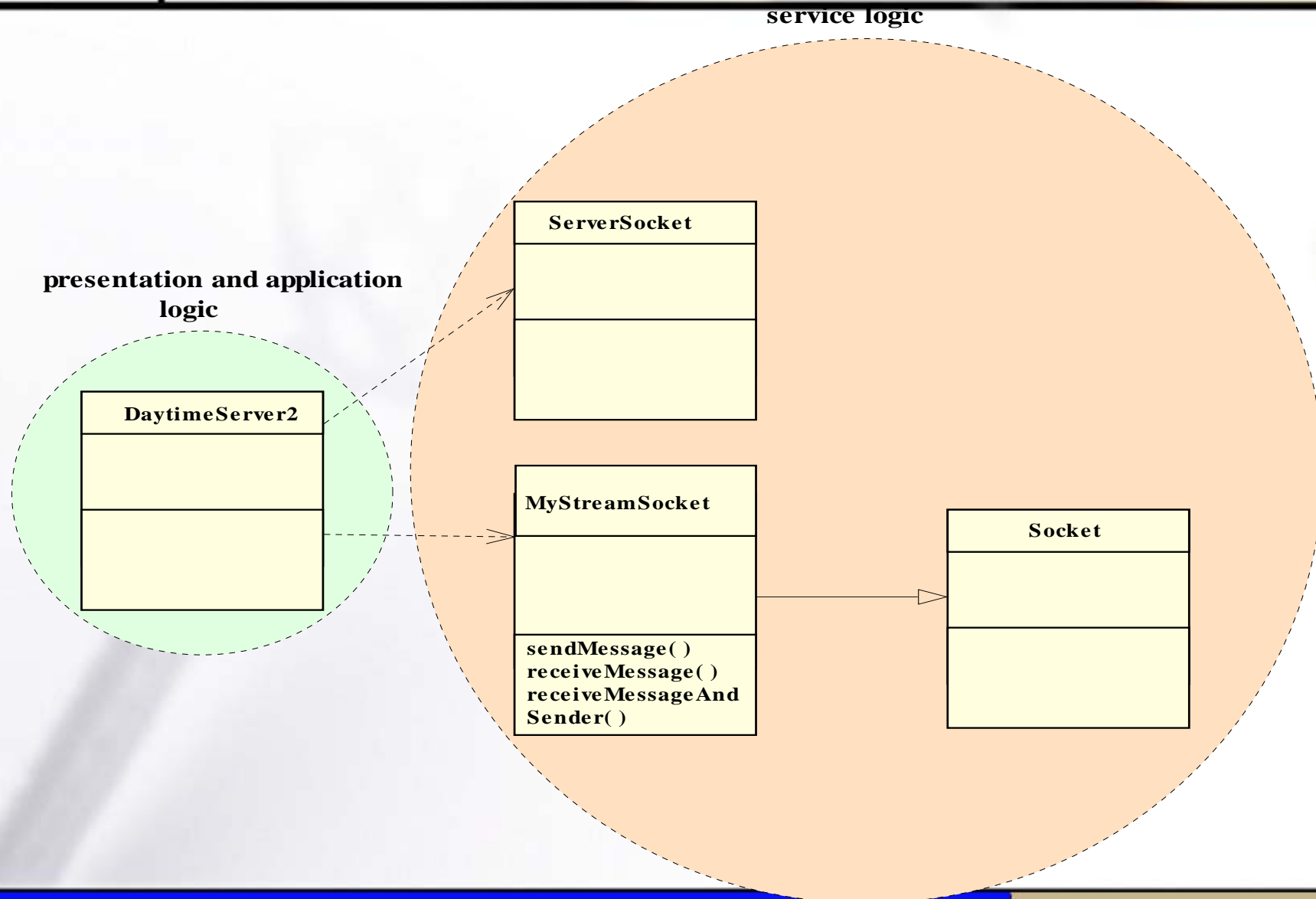
DaytimeServer2.java

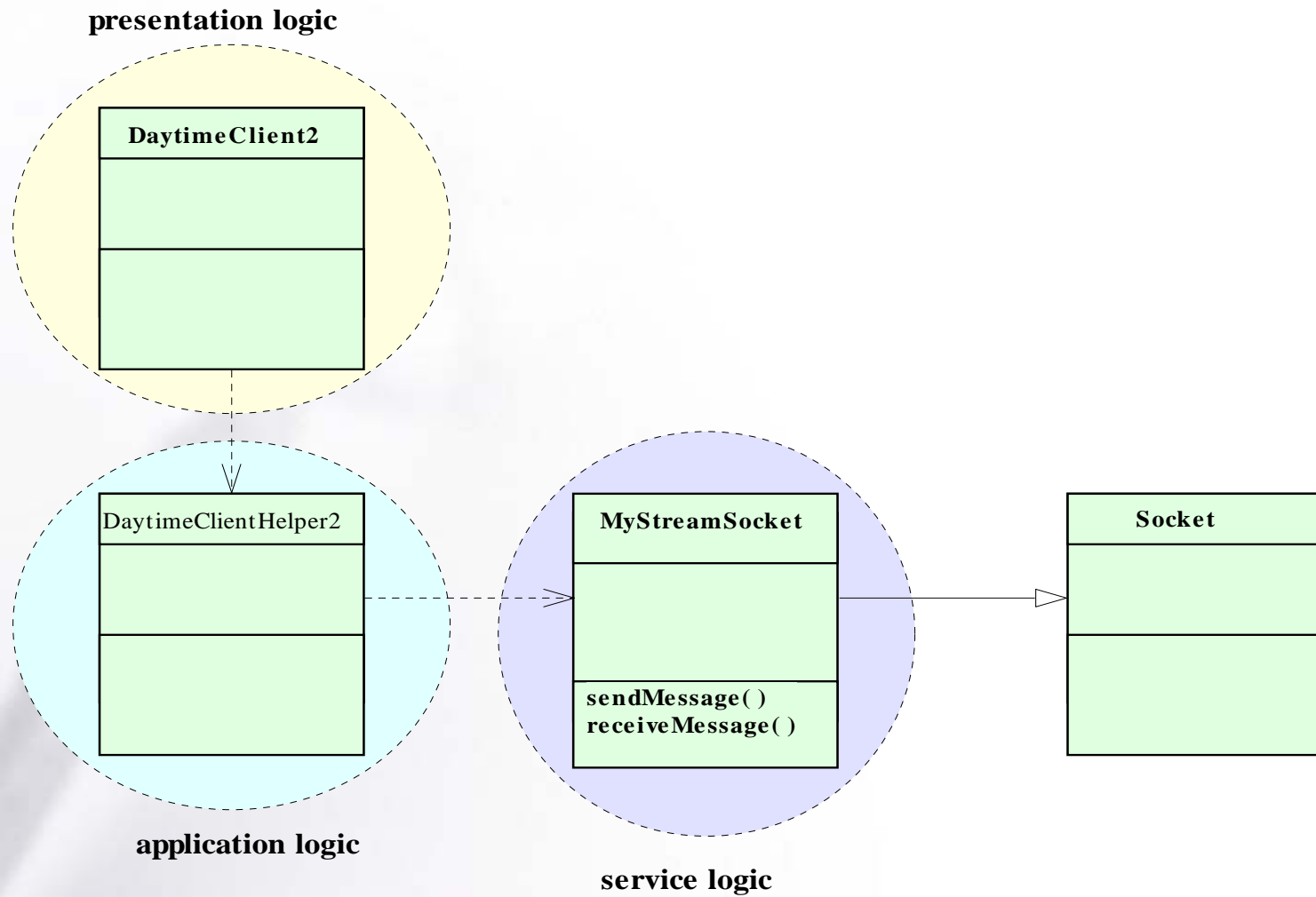
DaytimeClient2.java

DaytimeClientHelper2.java

MyStreamSocket.java

UML Diagram for stream mode Daytime Server





A connectionless server

**Uses a connectionless IPC API (e.g.,
connectionless datagram socket)**

Sessions with concurrent clients can be interleaved.

A connection-oriented server

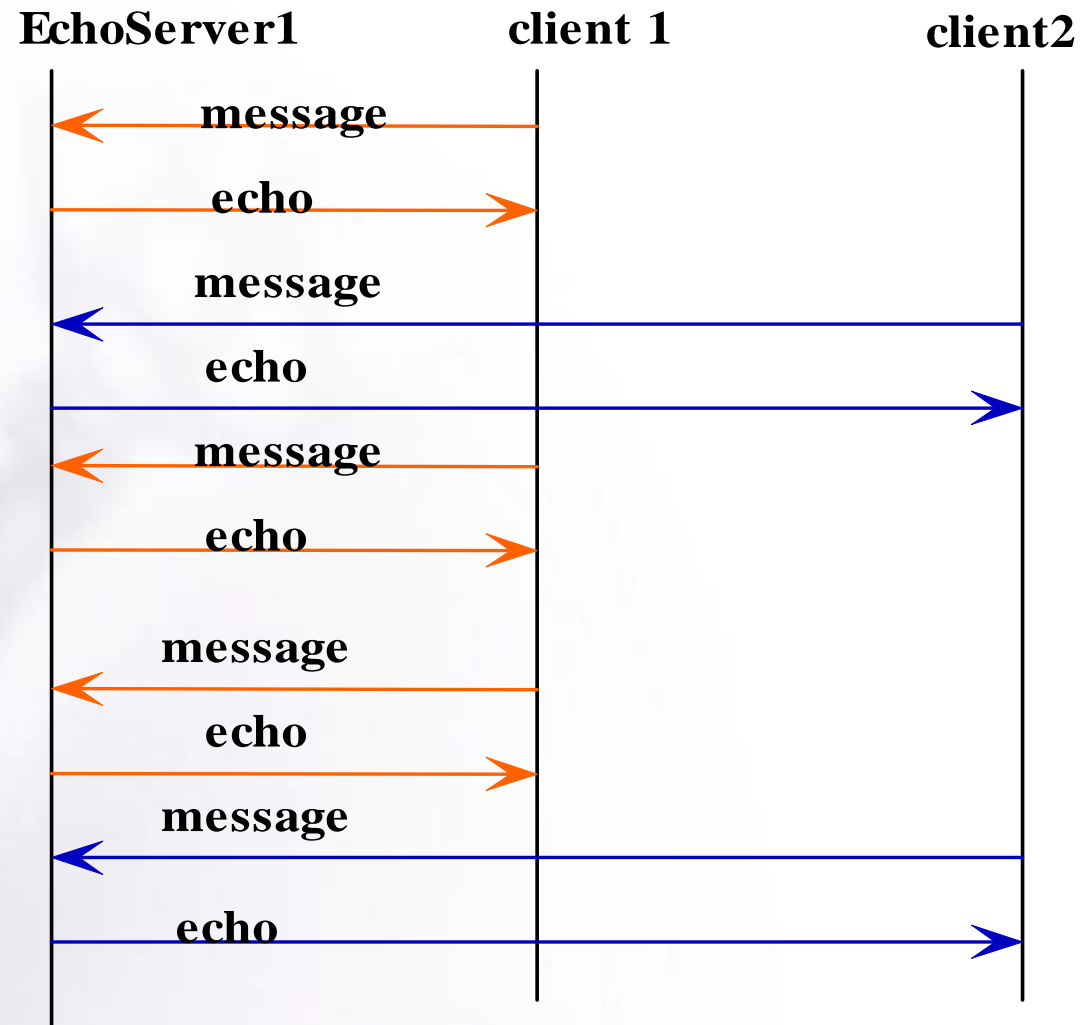
**Uses a connection-oriented IPC API (e.g. stream-
mode socket)**

Sessions with concurrent clients can only be
sequential unless the server is threaded

EchoServer 1(connectionless)

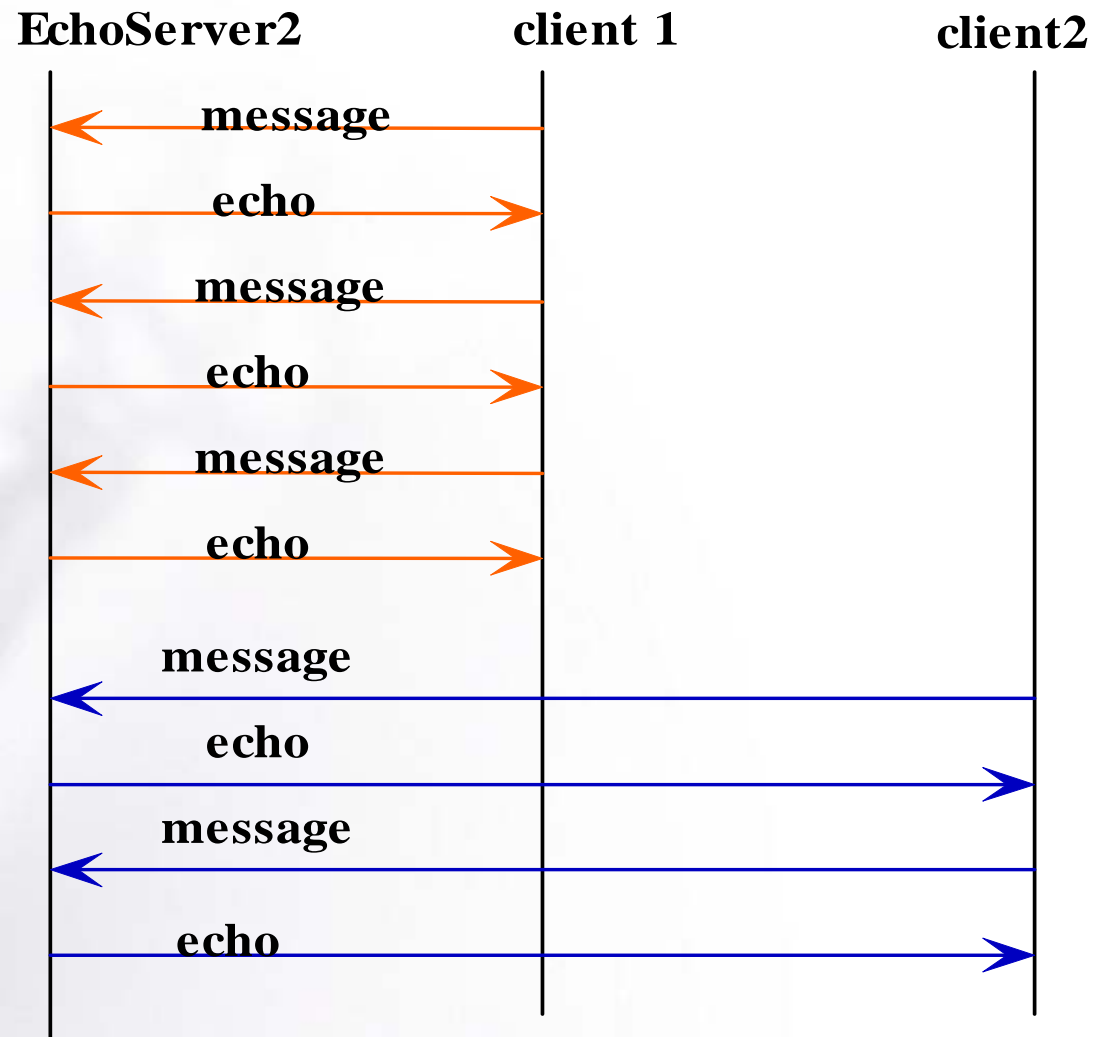
```
public class EchoServer1 {  
    public static void main(String[] args) {  
        ...  
        // instantiates a datagram socket for both sending  
        // and receiving data  
        MyServerDatagramSocket mySocket = new  
            MyServerDatagramSocket(serverPort);  
        while (true) { // forever loop  
            DatagramMessage request =  
                mySocket.receiveMessageAndSender();  
            String message = request.getMessage( );  
            mySocket.sendMessage(request.getAddress( ),  
                request.getPort( ), message);  
        } //end while  
    }  
}
```

Concurrent client sessions with EchoServer1



EchoServer2 (Connection-oriented)

```
ServerSocket myConnectionSocket = new ServerSocket(serverPort);
while (true) { // forever loop
    MyStreamSocket myDataSocket = new MyStreamSocket
        (myConnectionSocket.accept( ));
    boolean done = false;
    while (!done) {
        message = myDataSocket.receiveMessage( );
        if ((message.trim()).equals (endMessage)){
            myDataSocket.close( );
            done = true;
        } //end if
        else {
            myDataSocket.sendMessage(message);
        } //end else
    } //end while !done
} //end while forever
```

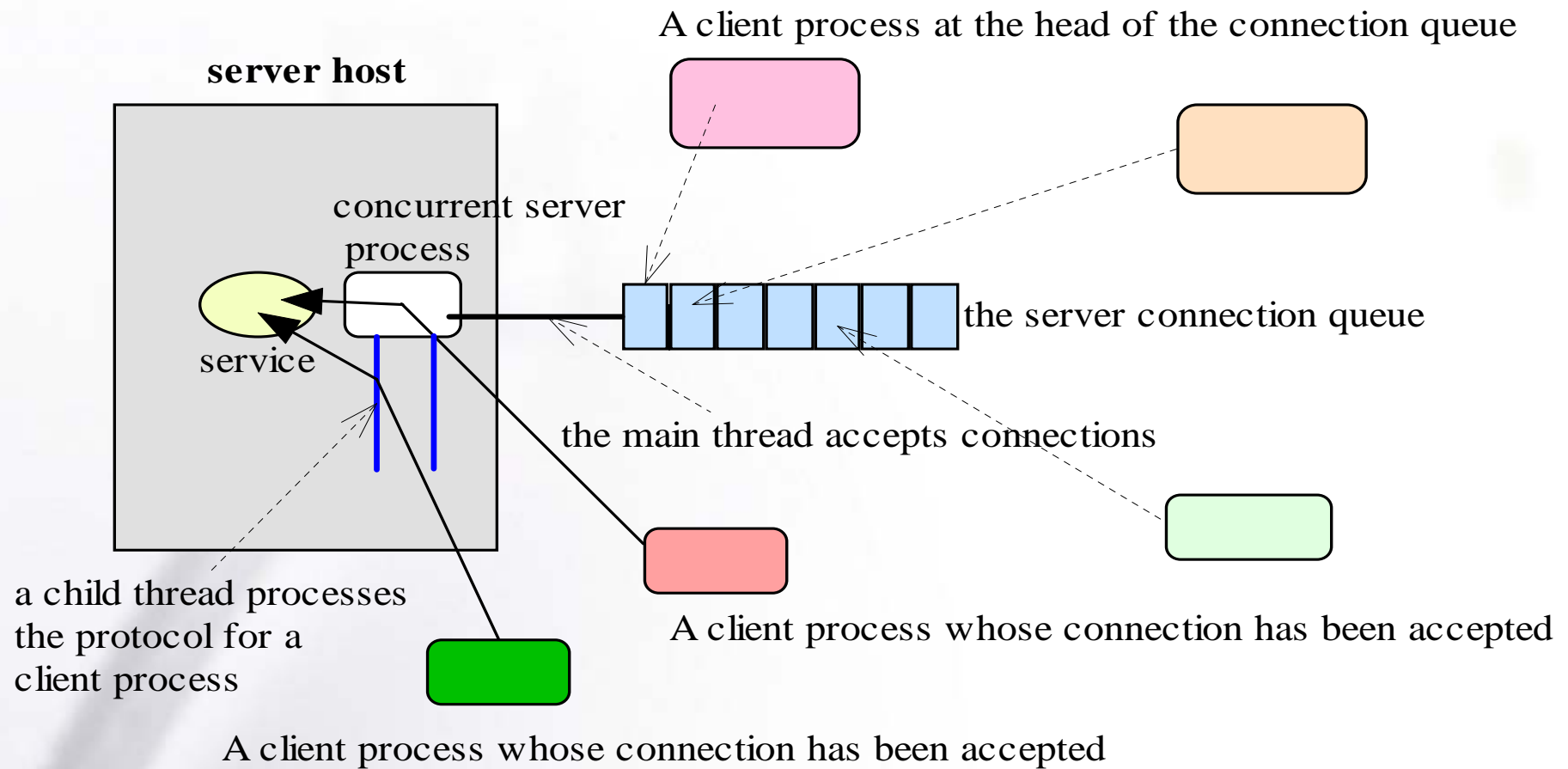


Iterative servers Vs. Concurrent servers

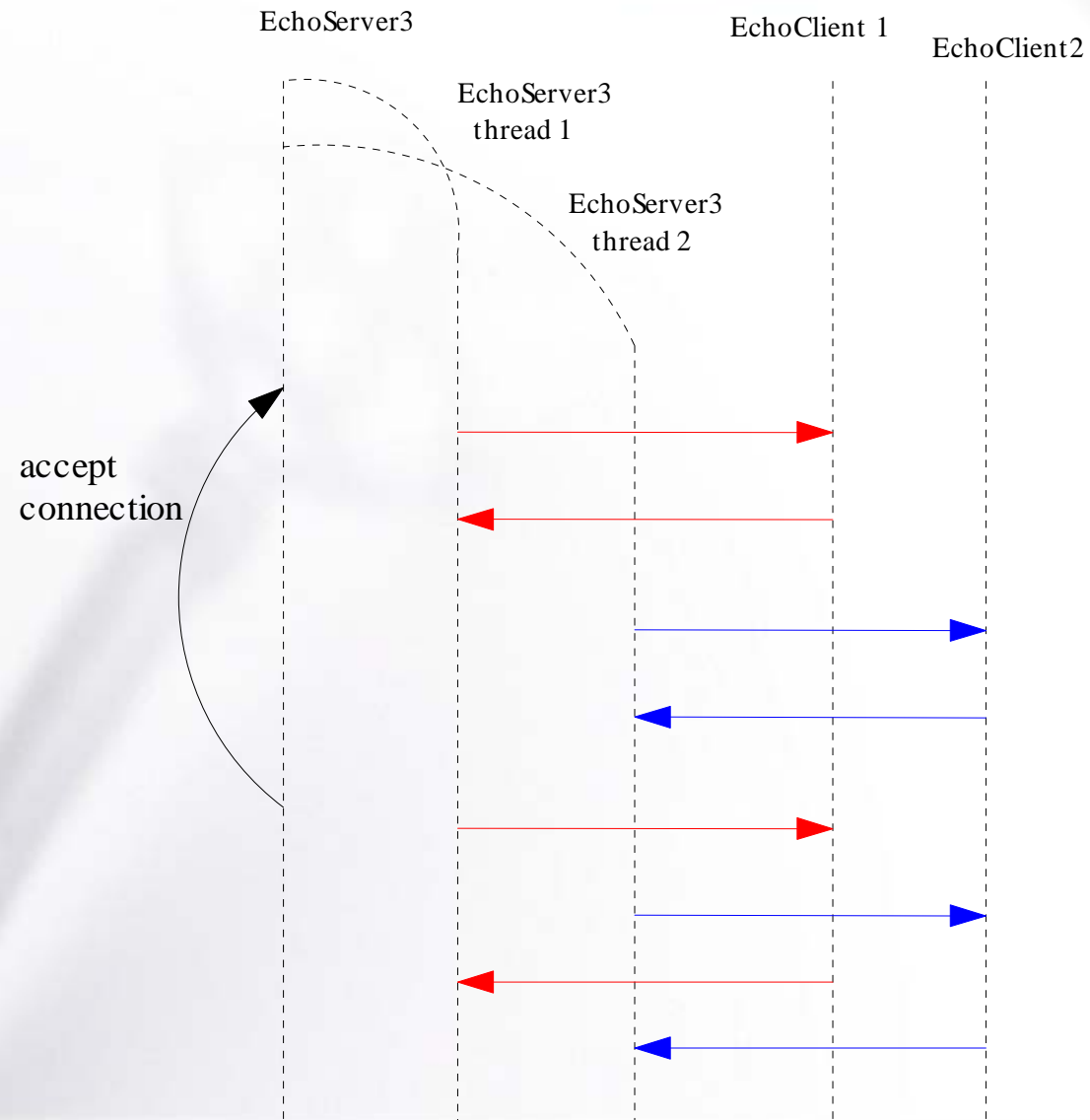
Concurrent Server

- ➔ A connection-oriented server can be threaded so that it can service multiple clients concurrently. Such a server is said to be a *concurrent server*.
- ➔ An unthreaded connection-oriented server is said to be an *iterative server*.

A Concurrent, Connection-oriented Server



Sequence diagram – EchoServer3



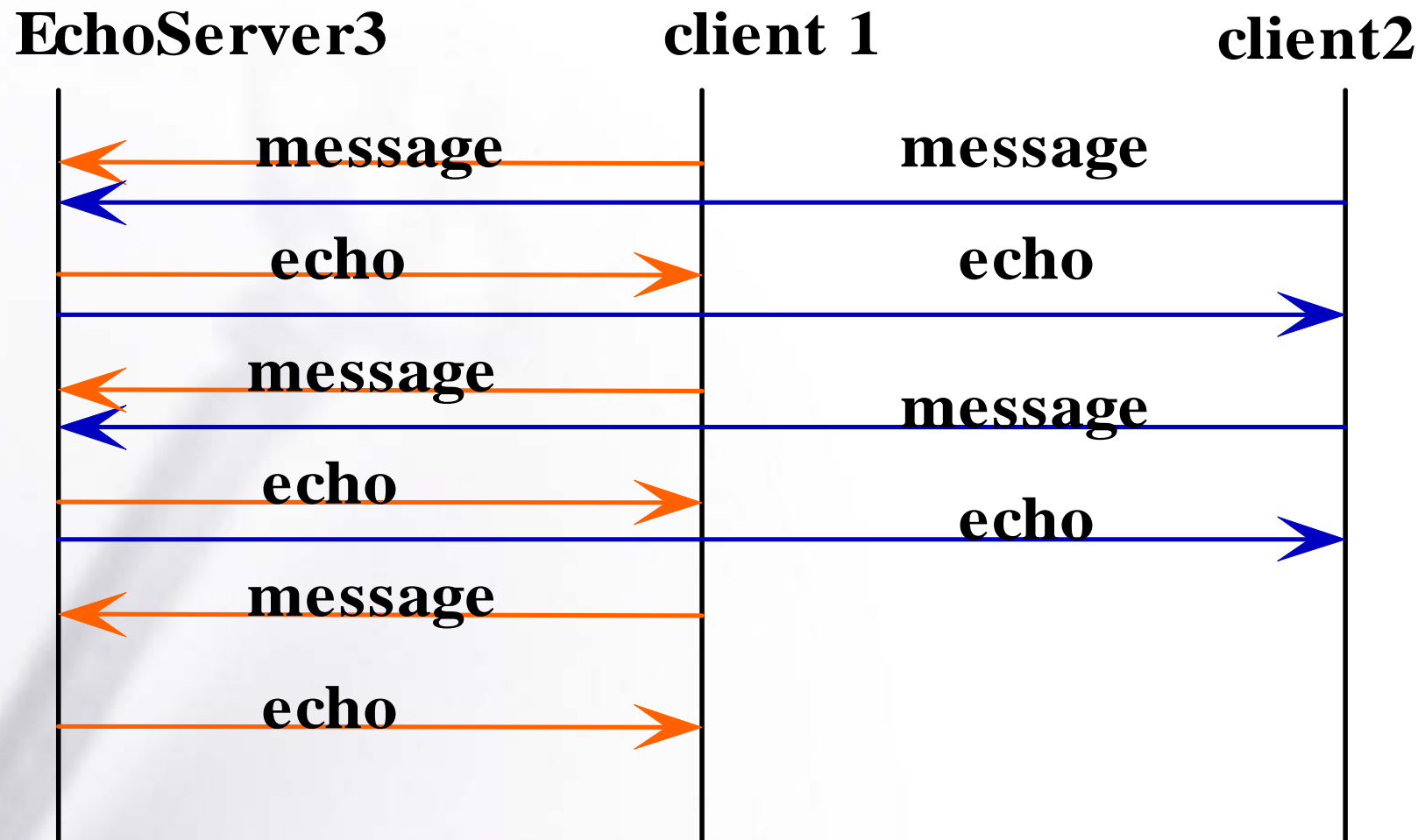
EchoServer3 (concurrent)

```
ServerSocket myConnectionSocket =  
    new ServerSocket(serverPort);  
    while (true) { // forever loop  
        MyStreamSocket myDataSocket = new  
            MyStreamSocket  
                (myConnectionSocket.accept( ));  
        Thread theThread =  
            new Thread(new  
                ServerThread(myDataSocket));  
        theThread.start();  
    } //end while forever
```

ServerThread.java excerpt

```
class ServerThread implements Runnable {  
    static final String endMessage = ".";  
    MyStreamSocket myDataSocket;  
    EchoServerThread(MyStreamSocket myDataSocket) {  
        this.myDataSocket = myDataSocket;  
    }  
    public void run( ) {  
        boolean done = false; String message;  
        try { // put in here the logic for each client session  
            while (!done) {  
                message = myDataSocket.receiveMessage( );  
                if ((message.trim()).equals (endMessage)){  
                    myDataSocket.close( ); done = true;  
                } //end if  
                else {  
                    myDataSocket.sendMessage(message);  
                } //end else  
            } //end while !done  
        } // end try  
        catch (Exception ex) {  
        }  
    } //end run  
} //end class
```

Echo3Server concurrent sessions



Server Thread class template

```
class ServerThread implements Runnable {  
    static final String endMessage = ".";  
    MyStreamSocket myDataSocket;  
    ServerThread(MyStreamSocket myDataSocket) {  
        this.myDataSocket = myDataSocket;  
    }  
    public void run( ) {  
        boolean done = false;  
        String message;  
        try {  
            //add code here  
        } // end try  
        catch (Exception ex) {  
            System.out.println("Exception caught in thread: " + ex);  
        }  
    } //end run  
} //end class
```


Stateful servers vs. Stateless servers

Session State Information

For some protocols or applications, a server must maintain information specific to a client during its service session.

Consider a network service such as file transfer. A file is typically transferred in blocks, requiring several rounds of data exchanges to complete the file transfer. The dialog during a session proceeds roughly as follows:

Client: Please send me the file foo in directory someDir.

Server: Okay. Here is block 1 of the file

Client: Got it.

Server. Okay. Here is block 2 of the file

Client: Got it.

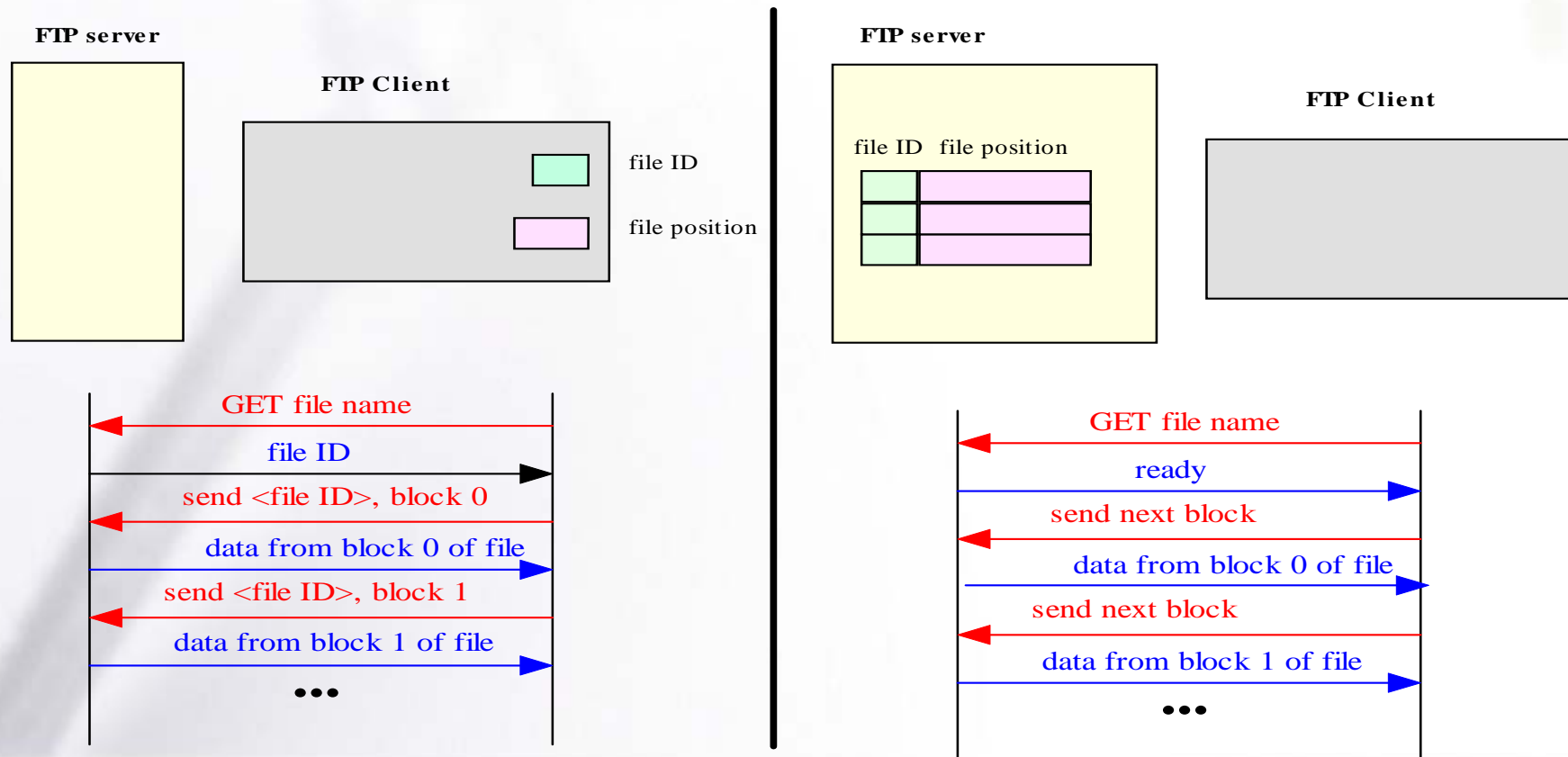
...

Server. Okay. Here is block 3 of the file

Client: Got it.

Stateful server

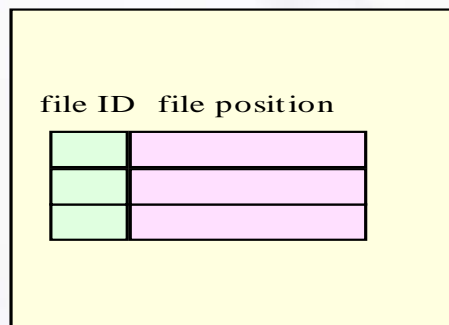
- ➔ A stateful server maintain stateful information on each active client.
- ➔ Stateful information can reduce the data exchanged, and thereby the response time.



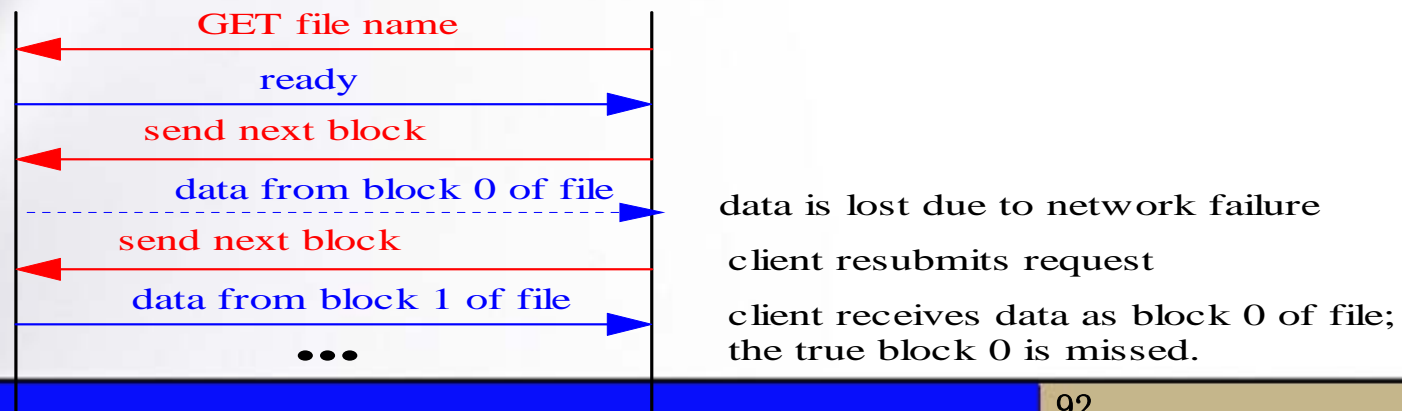
Stateful vs. Stateless server

- ➔ Stateless server is straightforward to code.
- ➔ Stateful server is harder to code, but the state information maintained by the server can reduce the data exchanged, and allows enhancements to a basic service.
- ➔ Maintaining stateful information is difficult in the presence of failures.

FTP server



FTP Client



Session State Information - 2

With a protocol such as ftp, there is a need for the server to keep track of the progress of the session, such as which block of the file needs to be fetched next. A server does so by maintaining a set of state for each session, known as the session state data. For the file transfer protocol, the session state data may include the name of the file being transferred, and the current block count.

Another example of a stateful protocol is one for a shopping cart application. Each session must maintain state data that keeps track of the identify of the shopper and the cumulative contents of the shopping cart.

State Data Storage

In our example, the state data – the sleep time interval - is stored in a local variable in the run method of each thread. Since each client is serviced by a separate thread, the local variable suffices as a storage for the state data. Using local variables in a thread to store session state data is adequate for a network service server. In complex network applications such as shopping carts, more complex mechanisms are needed for state data storage.

Stateful vs. stateless server

➔ In actual implementation, a server may be

Stateless

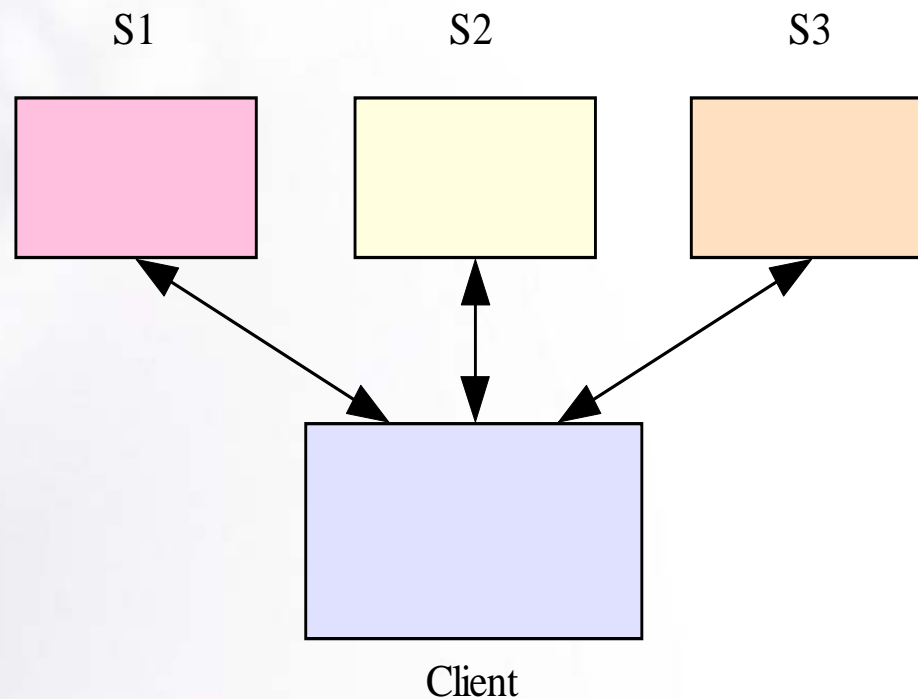
Stateful

A hybrid, wherein the state data is distributed on both the server-side and the client-side.

➔ Which type of server is chosen is a design issue.

A client can contact multiple servers

A process may require the service of multiple servers. For example, it may obtain a timestamp from a daytime server, data from a database server, and a file from a file server.



Summary

You have been introduced to the client-server paradigm in distributed computing.

Topics covered include:

- The difference between the client-server system architecture and the client-server distributed computing paradigm.
- Definition of the paradigm and why it is widely adopted in network services and network applications.
- The issues of service sessions, protocols, service location, interprocess communications, data representation, and event synchronization in the context of the client-server paradigm.
- The three-tier software architecture of network applications: Presentation logic, application logic, and service logic.
- Connectionless server versus connection-oriented server.
- Iterative server versus concurrent server and the effect on a client session.
- Stateful server versus stateless server.
- In the case of a stateful server: global state information versus session state information.