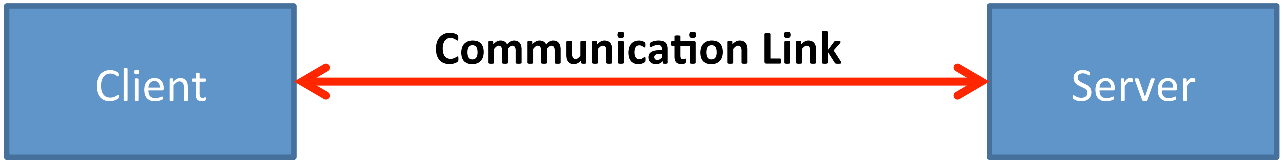
Programming with Sockets

CS3524 Distributed Systems

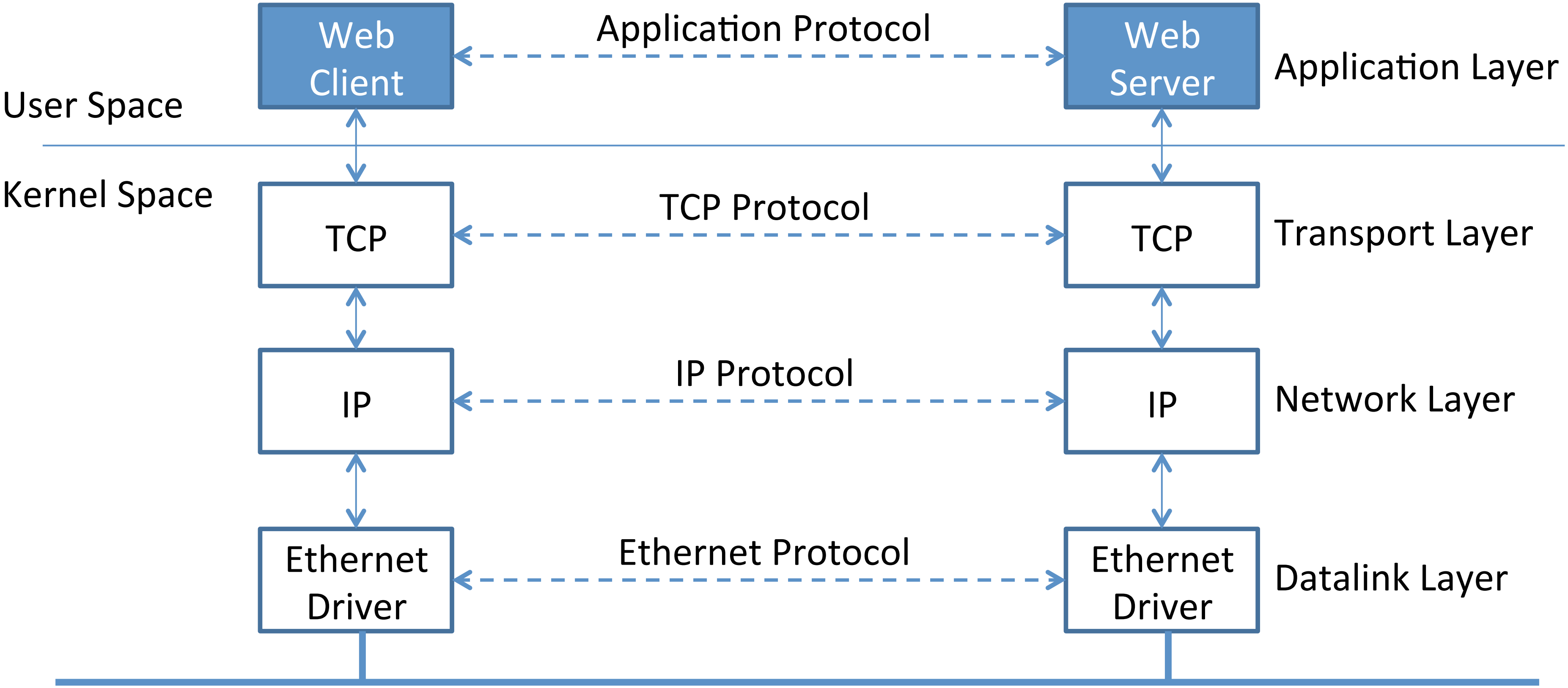
Lecture 06

Clients and Servers



- The communication between a client and a server involves networking protocols
- We focus on the TCP/IP protocol suite
- Programmatic means for process communication via TCP/IP: sockets

TCP/IP Protocol Stack



User Space

Kernel Space

Web Client

Application Protocol

Web Server

Application Layer

TCP

TCP Protocol

TCP

Transport Layer

IP

IP Protocol

IP

Network Layer

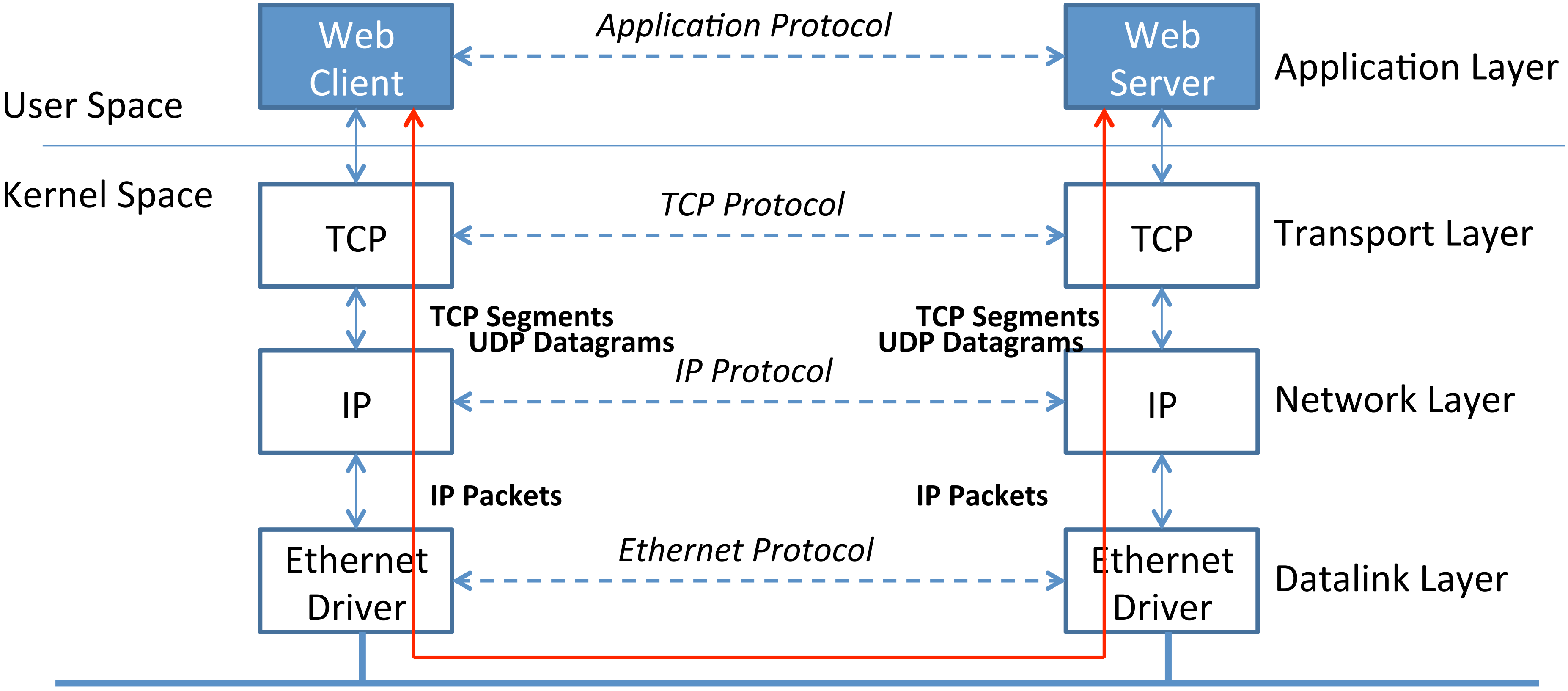
Ethernet Driver

Ethernet Protocol

Ethernet Driver

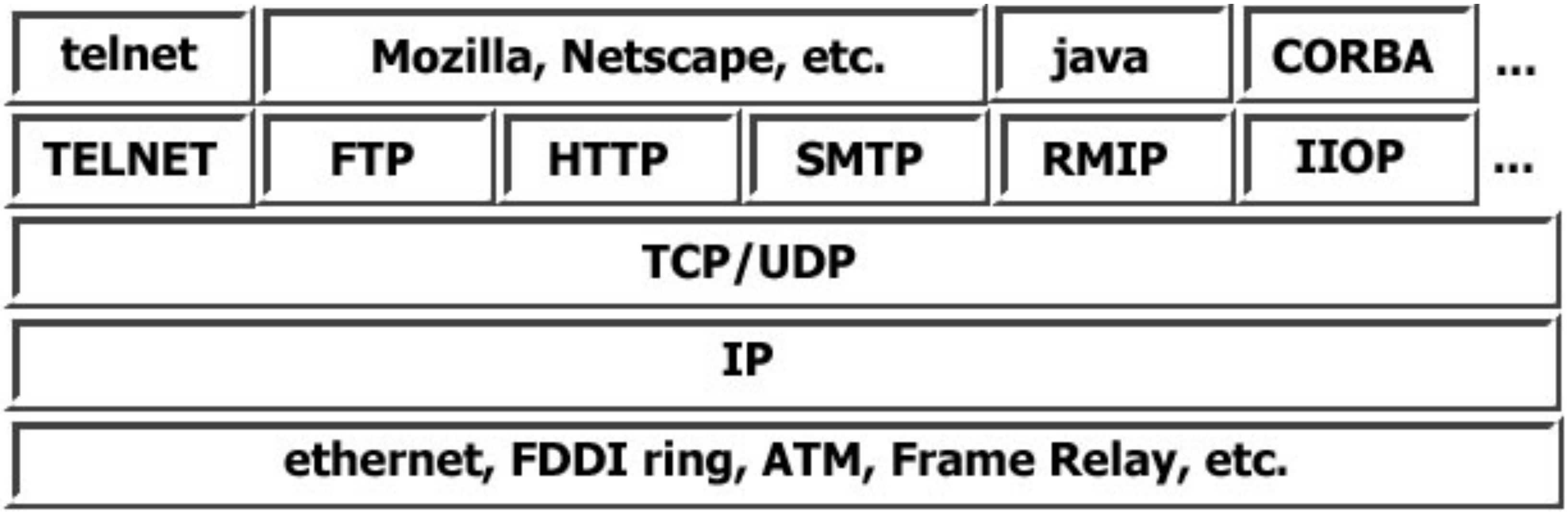
Datalink Layer

TCP/IP Protocol Stack



Protocol Layers

- The Internet is perceived
  - By users as a space of applications (application layer)
  - By application programmers as a single TCP/UDP interface (transport layer)
  - By transport protocol designers as a single IP interface
  - By network technology as the standard (IP) they support



telnet | Mozilla, Netscape, etc. | java | CORBA | ...

TELNET | FTP | HTTP | SMTP | RMIP | IIOP | ...

TCP/UDP

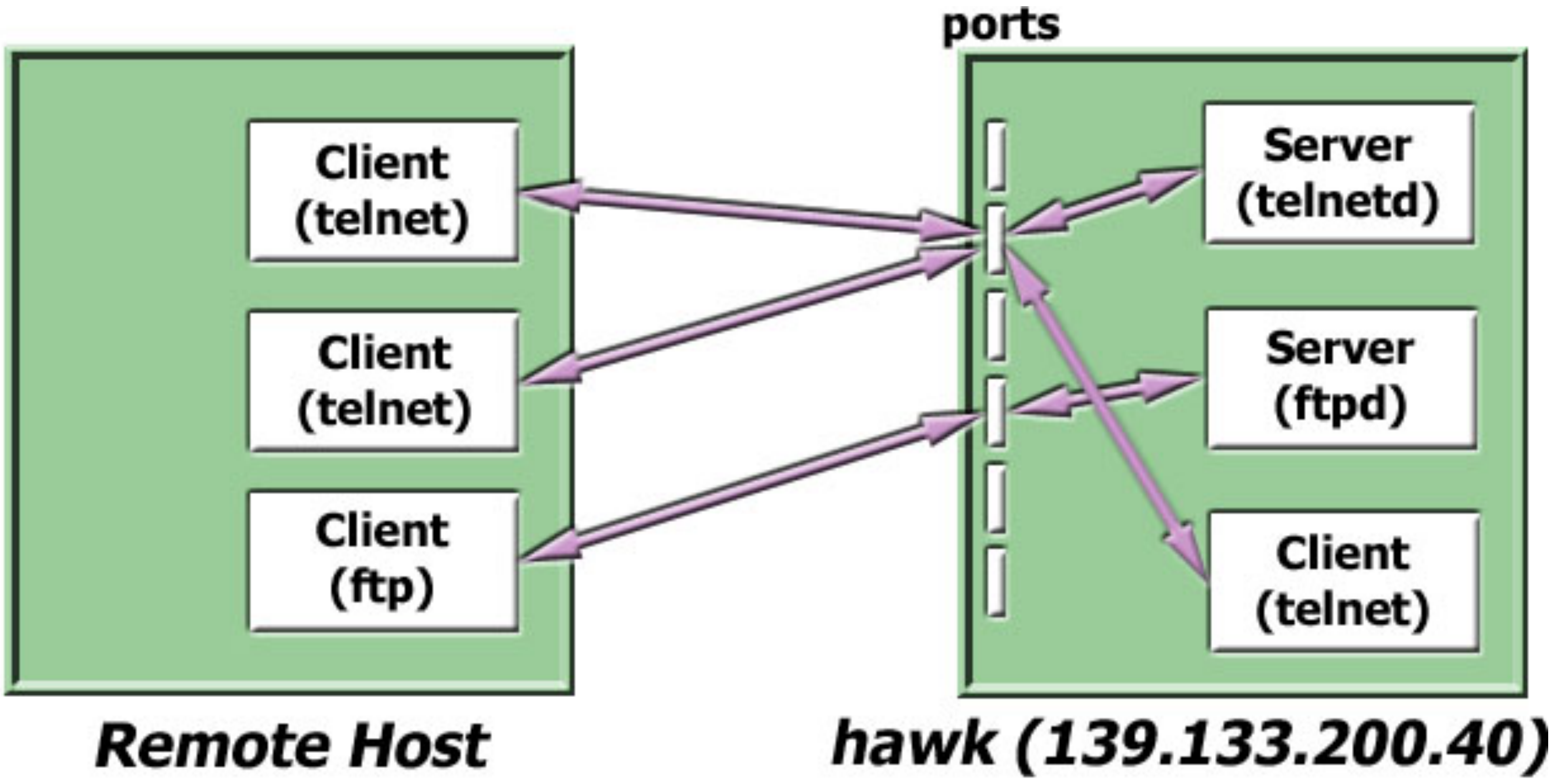
IP

ethernet, FDDI ring, ATM, Frame Relay, etc.

Ports

- Communication endpoint used in Transport Layer protocols
  - It is identified by
    - Its port number (such as, e.g., 50010),
    - The IP address it is associated with, and
    - The transport protocol used for communication
- To contact a server, a client needs to know the communication end-point
  - The IP address of the host where the server software runs, e.g. 139.133.200.40
  - The server's port number, e.g. 50010
- Most servers use reserved ports, so that clients know where to find them on an arbitrary host
- Reserved ports use numbers less than 1024

Clients and Servers



Unix: `cat /etc/services`

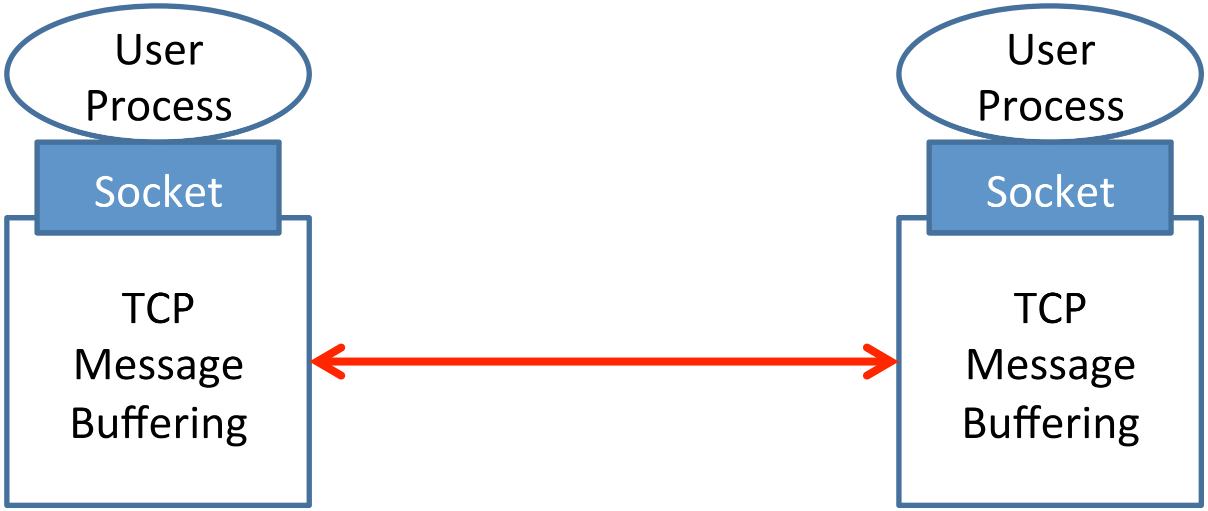
```
echo       7/tcp        Echos whatever is sent to it
...
daytime    13/tcp
daytime    13/udp       Gives the time of day
...
ftp-data   20/tcp
ftp        21/tcp       File Transfer Protocol
telnet     23/tcp
smtp       25/tcp       Simple Mail Transfer Protocol
...
nntp       119/tcp      Usenet News
...   (The rest listed here are Unix-specific functions)
exec       512/tcp
login      513/tcp
shell      514/tcp      Command
printer    515/tcp      Spooler
```

Transport Layer

- We will concentrate on two transport layer protocols:
  - TCP (Transmission Control Protocol):
    - Connection-oriented protocol
      - Establishes a connection between two endpoints
    - Is a reliable byte-stream protocol
    - Guarantees ordered delivery of a stream of bytes between a client and a server
  - UDP (User Datagram Protocol):
    - Is a connection-less protocol
    - Allows the exchange of messages, called datagrams, between clients and server
    - Does not guarantee the ordered delivery of a stream of bytes
    - Is fast, but unreliable

Network Programming: Sockets

Network Programming: Sockets



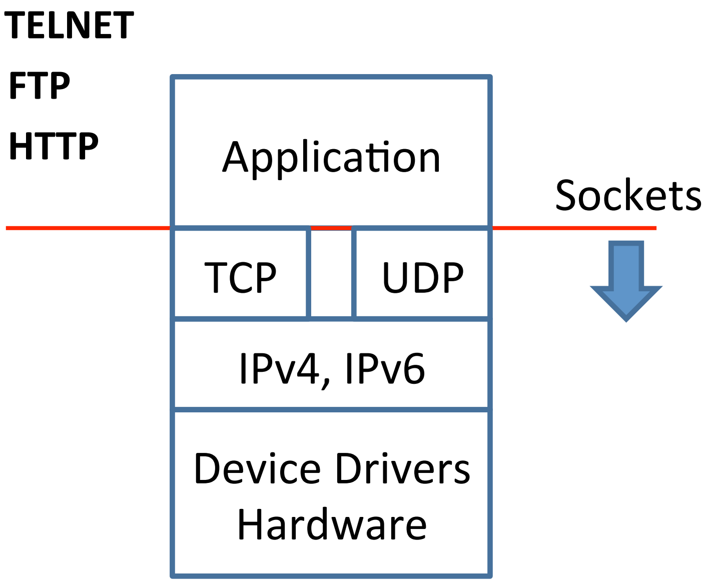
- Sockets are the programmatic interface between a user process and the transport layer
- Communication operations are based on socket pairs
  - Sockets are communication endpoints that can be used to connect a client and a server for communication
- Sockets are a common means of communication in distributed systems

Sockets

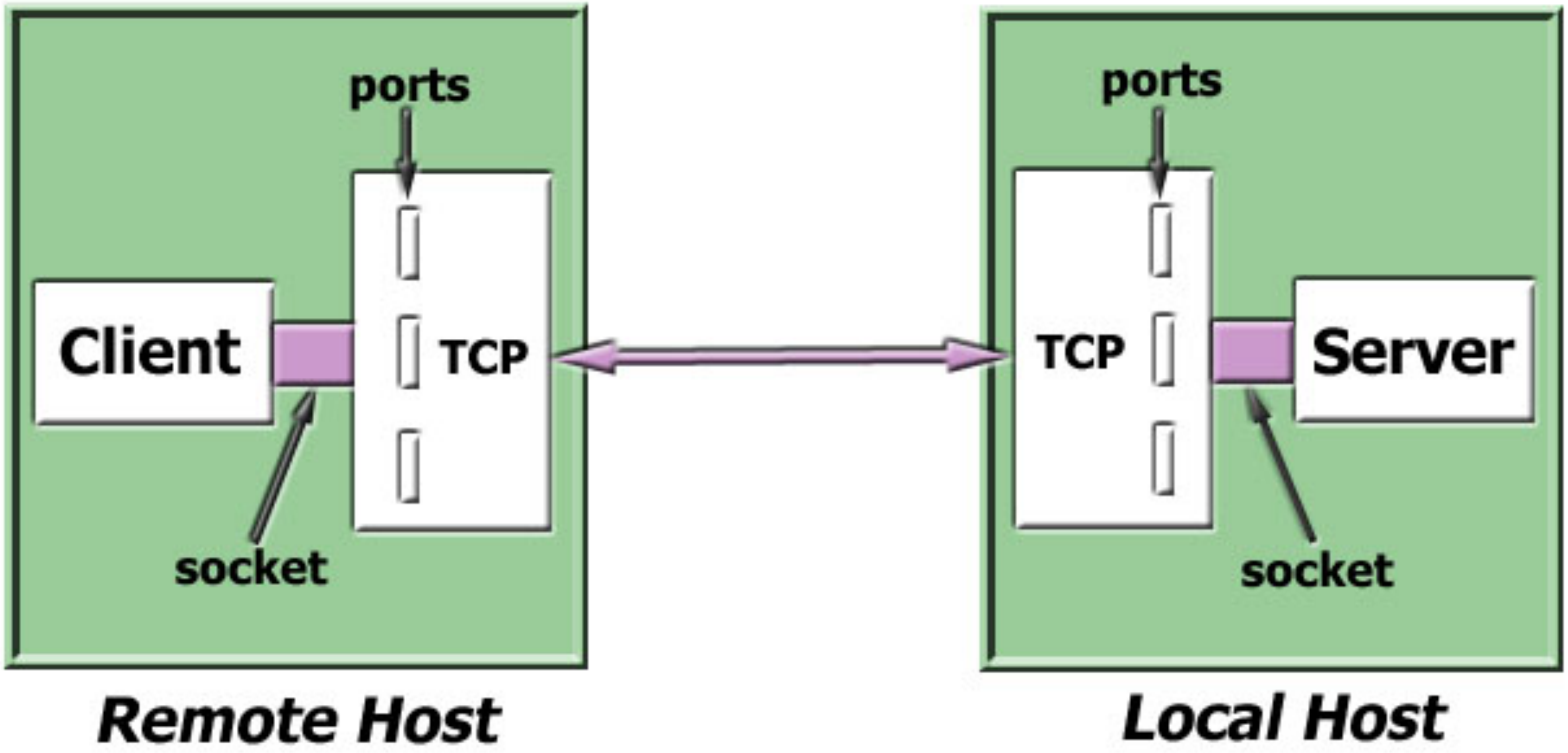
- Sockets are identified by their socket address
  - Consists of an IP address and a port number
- Messages are transmitted from socket of sending process to socket of receiving process
  - At sending socket: messages are queued until the underlying network protocol has transmitted them
  - At receiving socket: messages are queued until the receiving process has consumed them

Internet Applications using Sockets

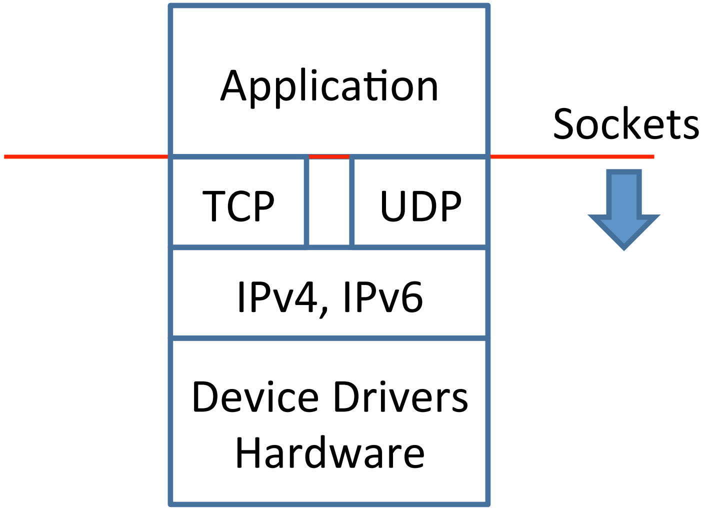
- TELNET (server port 23; TCP)
  - Virtual interactive terminals, can negotiate terminal type
- FTP (server port 21 control, 20 for data; TCP)
  - File Transfer Protocol, included in most Web browsers
- HTTP (server port 80; TCP)
  - Hyper-Text Transfer Protocol; carries Web objects (HTML pages, plain text files, graphics files, applets, etc.)
- SMTP (server port 25; TCP)
  - Simple Mail Transfer Protocol, for sending / receiving email
- SNMP
  - Simple Network Management Protocol, allows "remote control" of routers, etc. (update of routing information)



Sockets: A Transport Layer Interface



Network Programming: Sockets



- Sockets are the programming interface to the transport layer
- A socket is a "communication end-point"
- Sockets are bi-directional and can be used for communication between different hosts
- TCP and UDP sockets

TCP Multiplexing

- The TCP connection abstraction explains why this endpoint can service multiple clients
  - Imagine a client running on raven, transmitting on port 12345 that establishes a connection to the server. This connection is defined by the endpoints (139.133.200.90, 12345) and (139.133.200.40, 17777)
  - Suppose another connection is established from port 2222 on host cerberus. The connection is: (139.133.200.203, 2222) and (139.133.200.40, 17777)
- The server is able to deal with each connection entirely separately, as they are distinct software entities:
  - It is “multiplexing” between data arriving from either (139.133.200.90, 12345) or (139.133.200.203, 2222)

TCP Sockets

- TCP sockets are an example of stream sockets
- The amount of data passed to the IP layer is called a segment
- TCP provides the following functionality
  - Acknowledgements
  - Timeout, estimation of roundtrip time (RTT)
  - Retransmission
- TCP is reliable:
  - Expects acknowledgement from receiver
  - Retransmits data, if acknowledgement is not received
- TCP sequences transmitted data by associating a sequence number with each byte sent

UDP Sockets

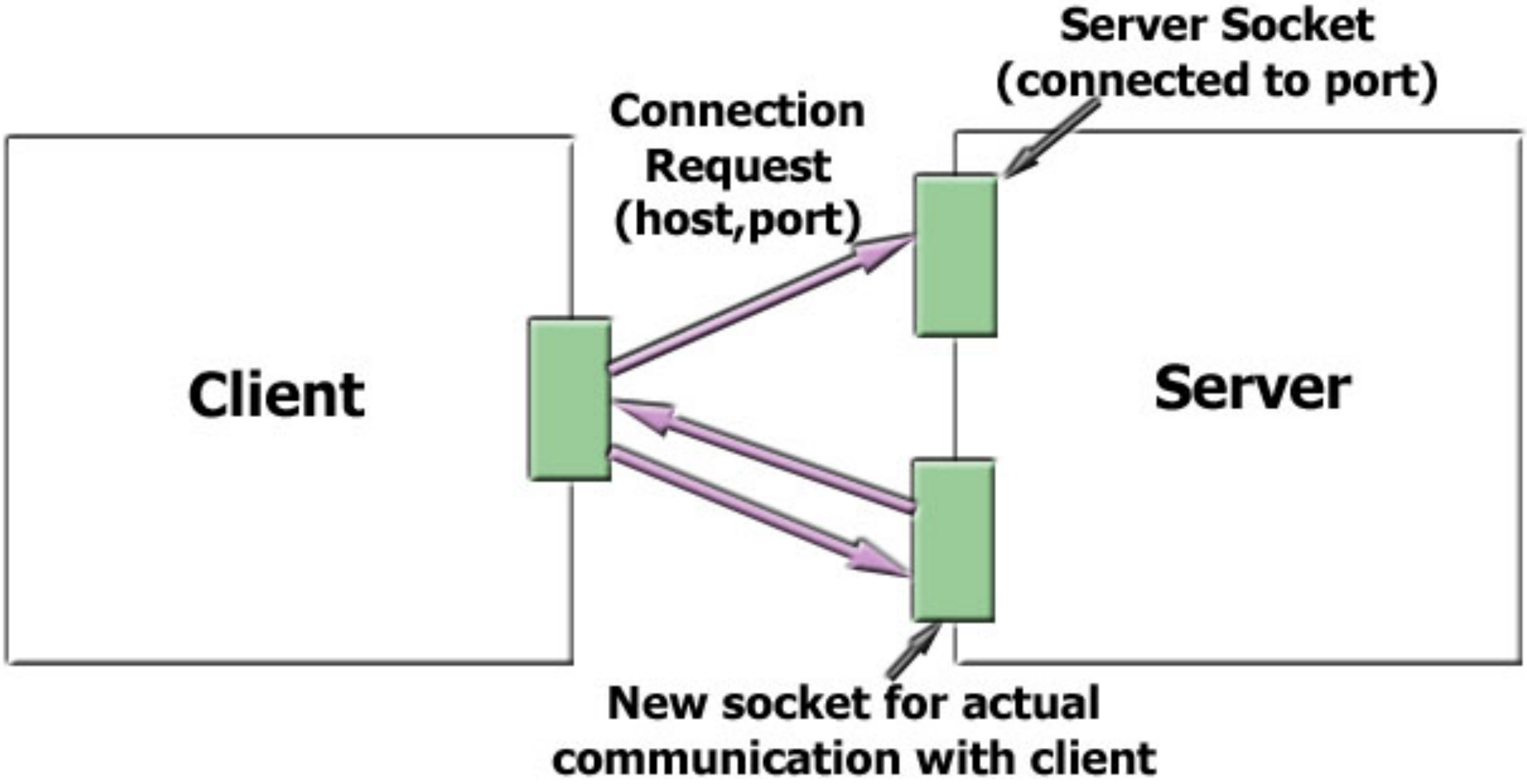
- Example of datagram sockets
- UDP is a simple transport layer protocol
- Operation
  - Application writes a datagram to the UDP socket, which is then encapsulated into an IP datagram
- Problem: lack of reliability
- Connectionless
  - Client may use same UDP socket to send messages to different servers
    - Addressee (IP address / port) is part of the message
  - Server may receive messages from different clients via the same UDP socket

Java Sockets

Common Java Classes for Socket Communication

- The socket interface classes of Java are provided by the java.net package:
  - `ServerSocket`
    - Allows to establish a socket on which a server "listens" for connection requests to establish a Stream (connection-oriented) communication channel
  - `Socket`
    - Stream (connection-oriented) communication channel
  - `DatagramSocket`
    - Datagram (connectionless) communication channel
  - `InetAddress`
    - Specifies target host of a communication channel
- Abstract classes from java.io to create Input/Output streams over sockets
  - `InputStream` : byte stream, incoming from socket
  - `OutputStream` : byte stream, outgoing to socket

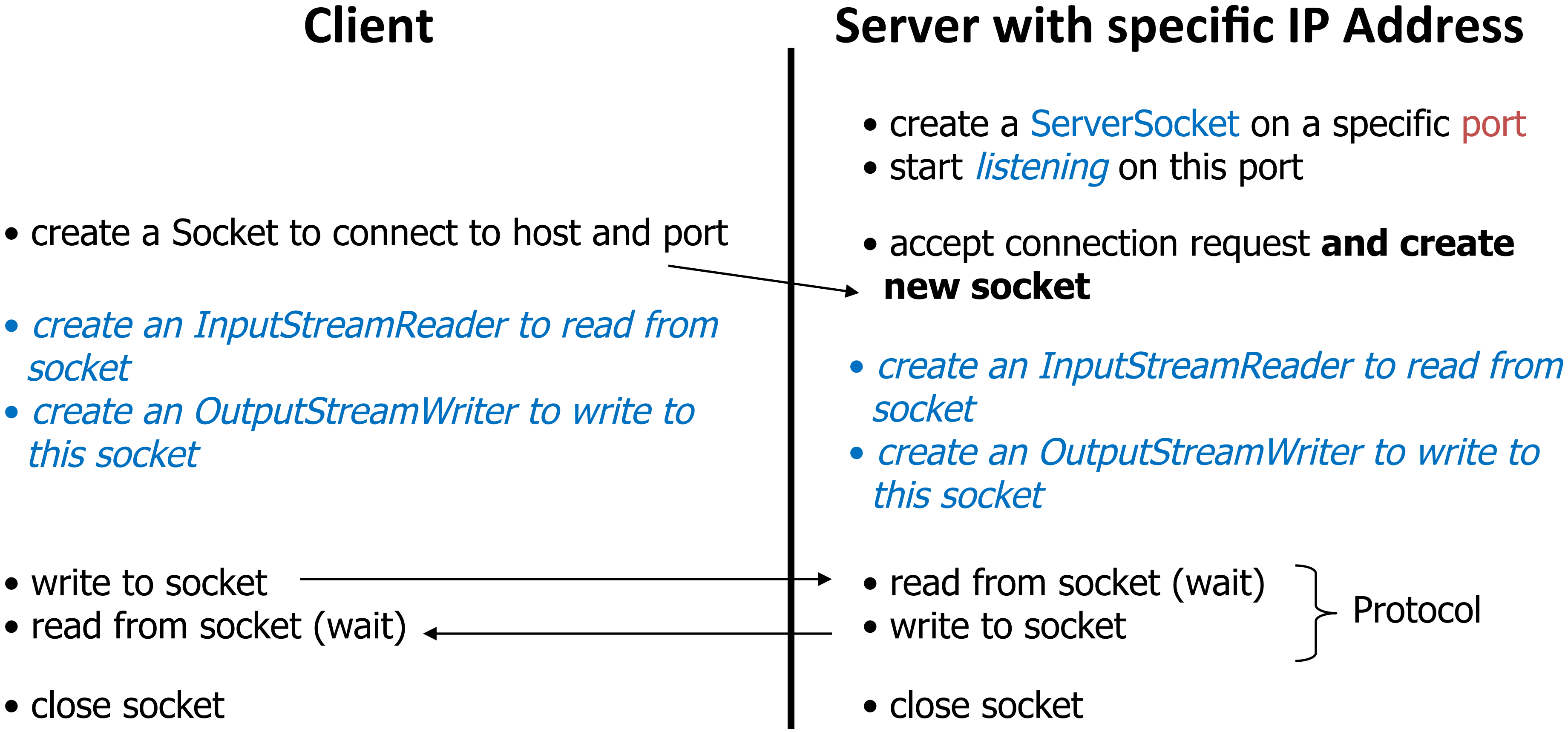
TCP Sockets: Connection-oriented Transport



TCP Connection Establishment

- Server creates a ServerSocket and continues to listen for incoming connection requests
  - This is called a “passive open” of a socket
  - The socket is “bound” to a port und IP address
- Client creates a socket and specifies the server endpoint as its destination
  - This is an “active open” of a socket by making a connection request
  - Three-way handshake for establishing a TCP connection (hidden from programmer):
    - TCP layer of Client sends synchronisation messages to server
    - TCP layer of Server acknowledges synchronisation message
    - TCP layer of Client sends final acknowledgement

TCP Sockets Example



Client

- create a Socket to connect to host and port
- *create an InputStreamReader to read from socket*
- *create an OutputStreamWriter to write to this socket*
- write to socket
- read from socket (wait)
- close socket

Server with specific IP Address

- create a ServerSocket on a specific port
- start *listening* on this port
- accept connection request **and create new socket**
- *create an InputStreamReader to read from socket*
- *create an OutputStreamWriter to write to this socket*
- read from socket (wait)
- write to socket
- close socket

Protocol

ShoutServer.java

```
import java.io.*;
import java.net.*;

public class ShoutServer
{
  // NB: IOException must be caught or declared to be
   thrown.

   public static void main (String args[]) throws IOException
   {
      // Create a ServerSocket to listen on the communication
      // end-point specified by the IP address of this
      // machine and the port provided as an argument.

     ServerSocket listener = new ServerSocket(
                                        Integer.parseInt(args[0]) );
```

ShoutServer.java

```
while (true)
{
    // Block until a connection request is received at
    // the ServerSocket.

   Socket client = listener.accept();

    // Get the I/O streams and wrap them in
    // appropriate reader/writer objects.

   . . .
   . . .
```

- Server will return from the accept() method as soon as a connection request was received

ShoutServer.java

```
 while (true)
 {
     // Block until a connection request is received at
     // the ServerSocket.

    Socket client = listener.accept();

     // Get the I/O streams and wrap them in
     // appropriate reader/writer objects.

    BufferedReader in = new BufferedReader(
                               new InputStreamReader(
                                   client.getInputStream() ));
    PrintWriter out = new PrintWriter(
                               new OutputStreamWriter(
                             client.getOutputStream() ),
true);
```

ShoutServer.java

```
      // Interact with the client and close the connection.
      out.println("Welcome to ShoutServer");
      String msg = in.readLine();
      msg = msg.toUpperCase(); // do something
      out.println( msg );
      client.close();
    }
  }
}
```

ShoutClient.java

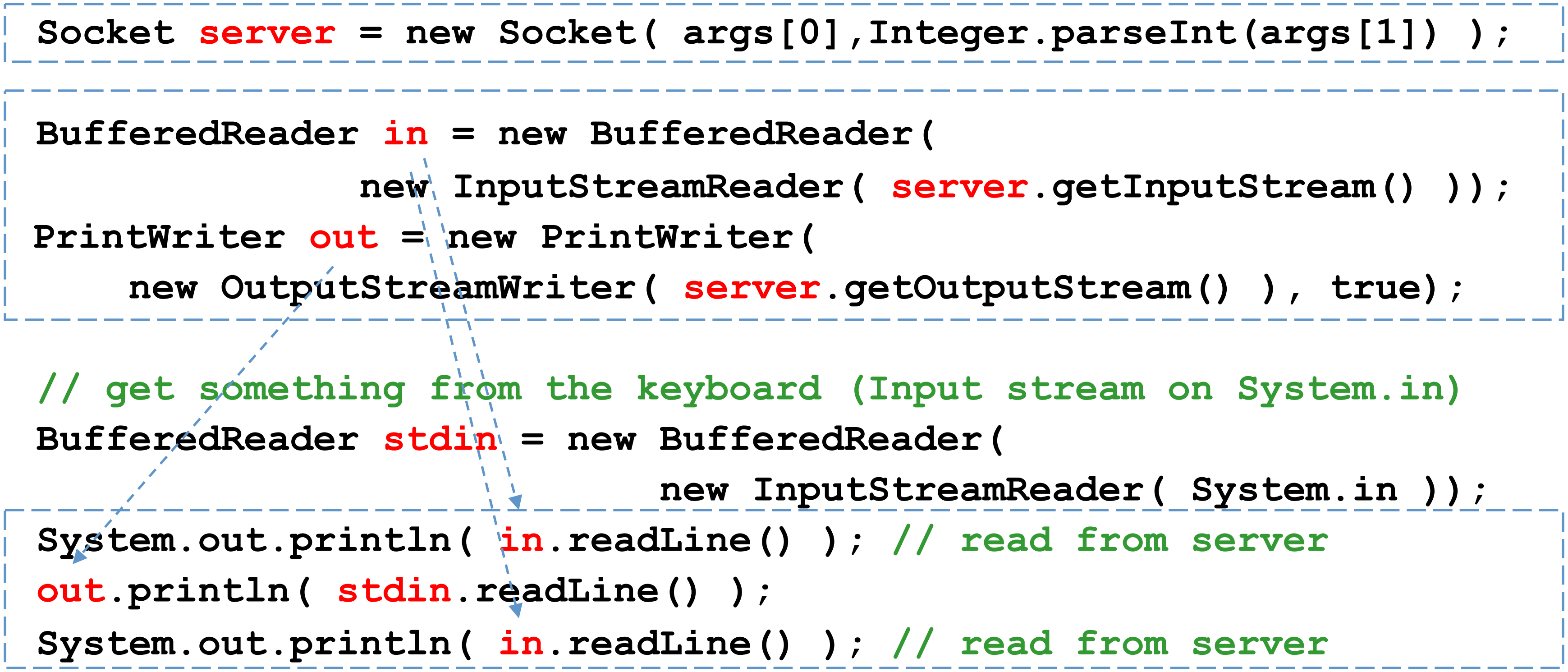
```
public class ShoutClient
{
   public static void main (String args[])throws IOException
   {
    Socket server = new Socket( args[0],                    // host
                                Integer.parseInt(args[1]) ); // port
     . . .
   }
}
```

Port, where server is listening

Specifies the name of the server host

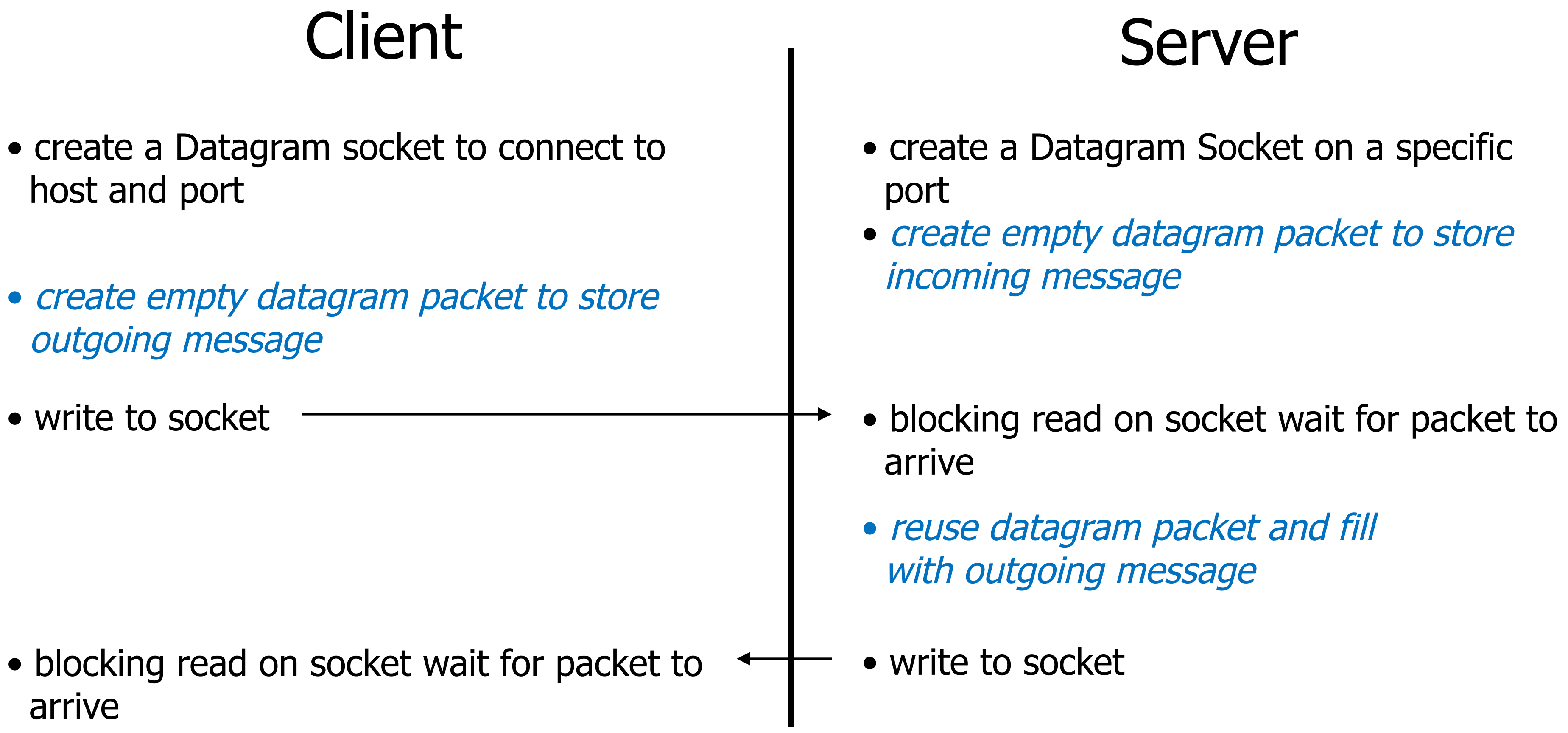
- The instantiation of the socket sends a connection request to the server

ShoutClient.java



```
public class ShoutClient
{
  public static void main (String args[])throws IOException
  {
    Socket server = new Socket( args[0],Integer.parseInt(args[1]) );

    BufferedReader in = new BufferedReader(
                   new InputStreamReader( server.getInputStream() ));
    PrintWriter out = new PrintWriter(
         new OutputStreamWriter( server.getOutputStream() ), true);

    // get something from the keyboard (Input stream on System.in)
    BufferedReader stdin = new BufferedReader(
                                    new InputStreamReader( System.in ));
    System.out.println( in.readLine() ); // read from server
    out.println( stdin.readLine() );
    System.out.println( in.readLine() ); // read from server
  }
}
```

UDP Sockets Example



Client

- create a Datagram socket to connect to host and port
- *create empty datagram packet to store outgoing message*
- write to socket
- blocking read on socket wait for packet to arrive

Server

- create a Datagram Socket on a specific port
- *create empty datagram packet to store incoming message*
- blocking read on socket wait for packet to arrive
- *reuse datagram packet and fill with outgoing message*
- write to socket

ShoutServerUDP.java

```
import java.io.*;
import java.net.*;

public class ShoutServerUDP
{
  // main throws IOException
  public static void main(String args[]) throws IOException
  {
    // Create a datagram socket.  This listens on a port on
    // the current host and will receive datagram packets.
    // It will not establish a connection.

    DatagramSocket socket = new DatagramSocket(
                               Integer.parseInt(args[0]) ) ;
```

ShoutServerUDP.java

```
while (true)
{
  // Create an empty datagram packet to store incoming
  // UDP packets.  Assume that they are no larger than
  // 1024 bytes.

  DatagramPacket packet = new DatagramPacket(
    new byte[1024], 1024 );

  // Block until a packet is received.  This may, of
  // course, throw IOExceptions if there's a problem.

  socket.receive( packet );
```

ShoutServerUDP.java

```
      // Do the shout server thing: get the data stored in
      // the packet received, convert it to upper case and
      // send it back as a datagram packet to the client.

      String msg = new String ( packet.getData() ) ;

      msg         = msg.toUpperCase();
      byte[] data = msg.getBytes()   ;
      packet.setData( data );
      packet.setLength( data.length );
      socket.send( packet );
      // NB: no connection to be closed.
    }
  }
}
```

ShoutClientUDP.java

```
import java.net.*;
import java.io.*;

public class ShoutClientUDP
{
  // Rather than catching IOExceptions, just declare that
  // this main method throws them (you could do either).
  public static void main (String args[]) throws IOException
  {
    // The protocol is slightly different here; we are not
    // expecting a welcome message.
    // we want to type a message on the keyboard
    BufferedReader stdin = new BufferedReader(
                       new InputStreamReader( System.in ) );
```

ShoutClientUDP.java

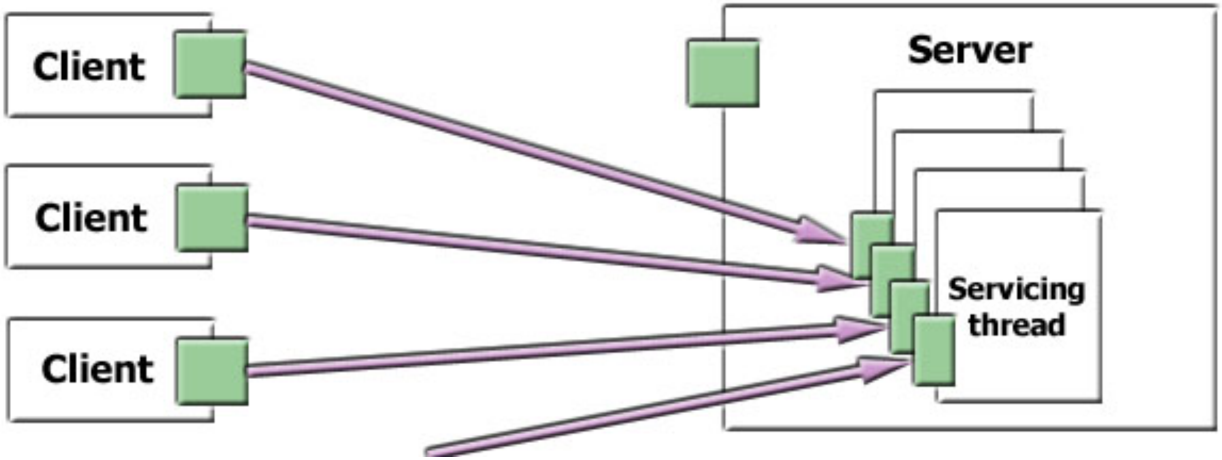
```
// Build a byte array with the user's input.
String msg  = stdin.readLine(); // user input
byte[] data = msg.getBytes();

// Obtain the IP address of the server using the
// Java interface to the DNS service. args[0] contains
// the hostname
InetAddress addr = InetAddress.getByName ( args[0] ) ;

// Construct a DatagramPacket with the byte array
// containing the user input and its destination. args[1]
// contains port

DatagramPacket packet =
  new DatagramPacket ( data, data.length, addr,
                       Integer.parseInt( args[1]) ) ;
```

ShoutClientUDP.java

```
    // Create a new socket to manage the transmission of
    // datagram packets.   NB. We are not establishing a
    // connection to the server.

    DatagramSocket socket = new DatagramSocket();

    // Send the user input and receive the response.

    socket.send( packet );
    socket.receive( packet );

    System.out.println( packet.getData() );
  }
}
```

What we really want

- One Server – Multiple Clients:
  - How can a server service multiple clients at the same time?
  - This can be introduced with a combination of Sockets and Threads



MultiShoutServer.java

- Create a ServerSocket (port no. given):

```
ServerSocket listener =
         new ServerSocket( Integer.parseInt(args[0]) );
```

- Listen for a connection request from a client, create a thread of control:

```
while (true)
{
    new ShoutServerConnection(
                           listener.accept() ).start();
}
```

MultiShoutServer.java

```
import java.io.*;
import java.net.*;

public class MultiShoutServer {
    public static void main( String argv[] )throws IOException
    {
        ServerSocket listener = new ServerSocket(
                                    Integer.parseInt( argv[0] ));
        while (true)
        {
            new ShoutServerConnection(
                                    listener.accept() ).start();
        }
    }
}
```

Client Socket

MultiShoutServer.java

- For each client, a separate thread is created

```
class ShoutServerConnection extends Thread
{
    Socket client;

    ShoutServerConnection( Socket client ) throws SocketException
    {
        this.client = client;
        setPriority( NORM_PRIORITY - 1 );
    }
```

MultiShoutServer.java

```
    public void run() {
        try {
            BufferedReader in = new BufferedReader(
                new InputStreamReader(
                client.getInputStream() ));
            PrintWriter out = new PrintWriter(
                new OutputStreamWriter(
                client.getOutputStream() ), true);
            out.println("Welcome to ShoutServer");
            String msg = in.readLine();
            msg = msg.toUpperCase();
            out.println(msg);
            client.close();
        } catch ( IOException e ) {
            System.out.println( "I/O Error: " + e );
        }
    }
}
```