Socket Programming in Java

Required reading:
Kurose 2.7, 2.8

CSE 4213, Fall 2006
Instructor: N. Vlajic

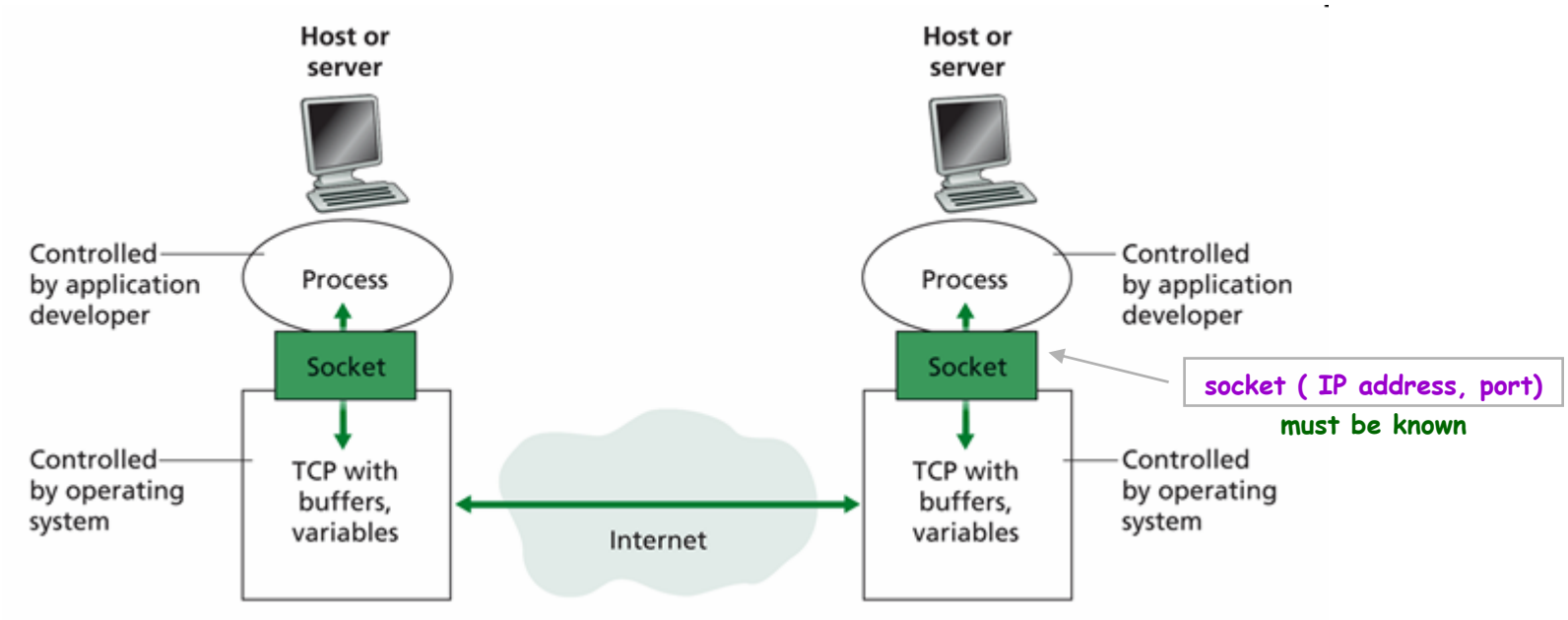
Socket Programming

Socket – a local-host, application created, OS controlled interface (a "door") into which application process can send/receive messages to/from another application process

- also, a door between application process and end-to-end transport protocol TCP or UDP
- each TCP/IP socket is uniquely identified with two pieces of information

  (1) name or address of the host (IP address)

  (2) identifier of the given process in the destination host (port number)



Socket Programming (cont.)

Socket Programming – **development of client/server application(s) that communicate using sockets**

- **developer has control of everything on application side but has little control of transport side of socket**
- **only control on transport-layer side is**
  - **(1) choice of transport protocol (TCP or UDP)**
  - **(2) control over a few transport-layer parameters e.g. max buffer and max segment size**

Socket programming refers to programming at the application level/layer!

TCP vs. UDP in Socket Programming – **to decide which transport-layer protocol, i.e. which type of socket, our application should use, we need to understand how TCP and UDP differ in terms of**

- **reliability**
- **timing**
- **overhead**

Socket Programming (cont.)

TCP vs. UDP Reliability

- UDP - there is no guarantee that the sent datagrams will be received by the receiving socket 👎
- TCP - it is guaranteed that the sent packets will be received in exactly the same order in which they were sent 👍

TCP vs. UDP Timing

- UDP - does not include a congestion-control mechanism, so a sending process can pump data into network at any rate it pleases (although not all the data may make it to the receiving socket) 👍
- TCP - TCP congestion control mechanism throttles a sending process when the network is congested – TCP guarantees that data will eventually arrive at the receiving process, but there no limit on how long it may take 👎

TCP vs. UDP Overhead

- UDP - every time a datagram is sent, the local and receiving socket address need to be sent along with it (packet/bandwidth overhead) 👎
- TCP - a connection must be established before communications between the pair of sockets start (connection setup time overhead) 👎

Socket Programming (cont.)

TCP vs. UDP in Socket Programming (cont.)

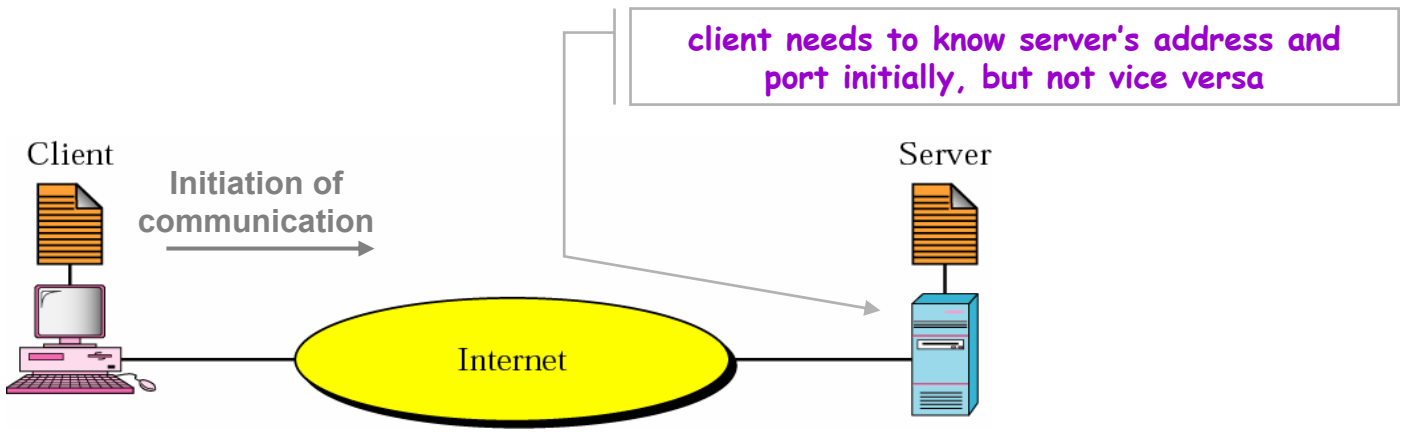
- **TCP is useful when indefinite amount of data need to be transferred 'in order' and reliably**
  - otherwise, we end up with jumbled files or invalid information
  - examples: HTTP, ftp, telnet, …
- **UDP is useful when data transfer should not be slowed down by extra overhead of reliable TCP connection**
  - examples: real-time applications
  - e.g. consider a clock server that sends the current time to its client – if the client misses a packet, it doesn't make sense to resend it because the time will be incorrect when the client receives it on the second try

In socket programming we pick transport-layer protocol that has services that best match the needs of our application.

Client-Server Model

Client-Server Model – most common form of network communication in the Internet whose purpose is to enable/provide various types of service to users

- **CLIENT**: process that initiates communication, requests service, and receives response
  - although request-response part can be repeated several times, whole process is finite and eventually comes to an end
- **SERVER**: process that passively waits to be contacted and subsequently provides service to clients
  - runs infinitely
  - can be iterative or concurrent

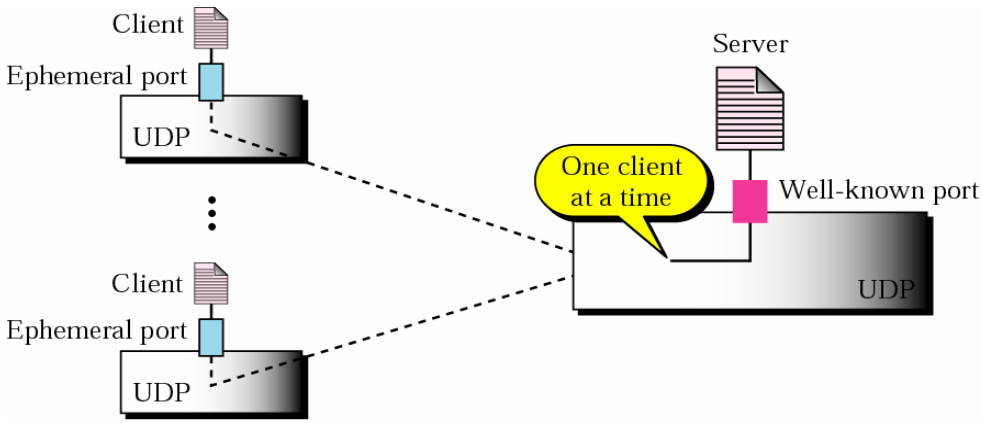


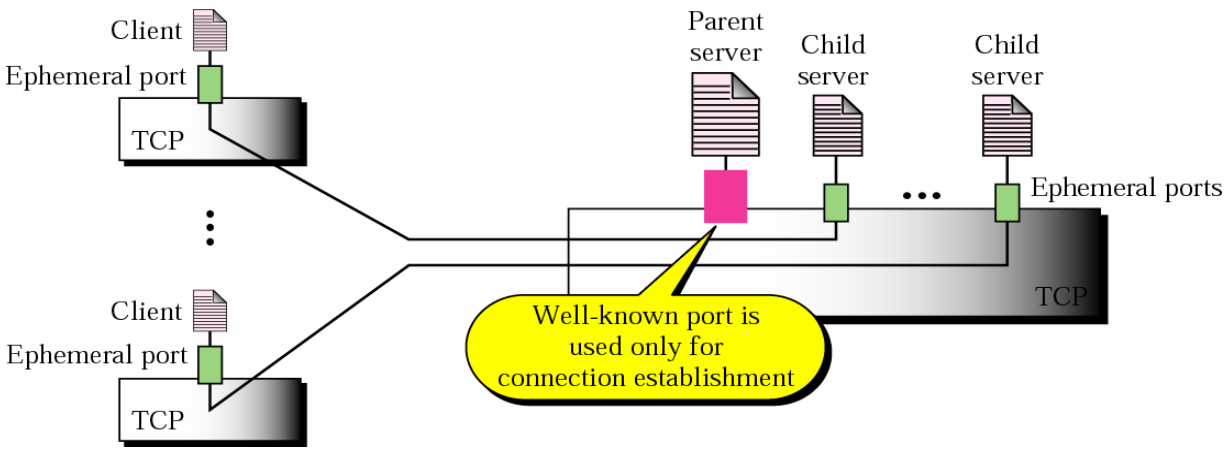
Client-Server Model (cont.)

Example [iterative vs. concurrent servers]

An iterative server can process only one request at a time – it receives a request, processes it, and sends the response to the requestor before handling another request.

The servers that use UDP are normally iterative.





A concurrent server can process many requests at the same time.

The servers that use TCP are normally concurrent.

Client-Server Model (cont.)

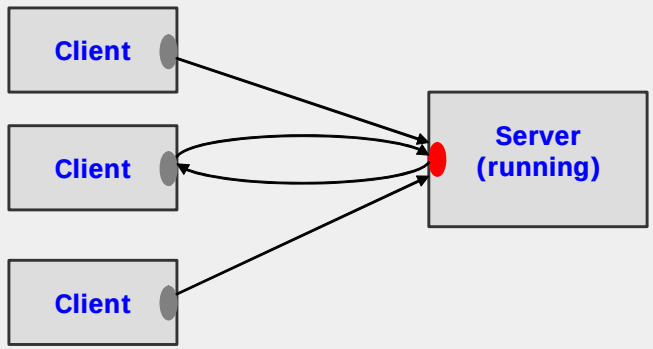
Principles of Client-Server Communication with UDP

- **CLIENT**:

(1) create socket

(2) repeat:
  (2.a) send
  (2.b) receive
  (2.c) process

(3) destroy



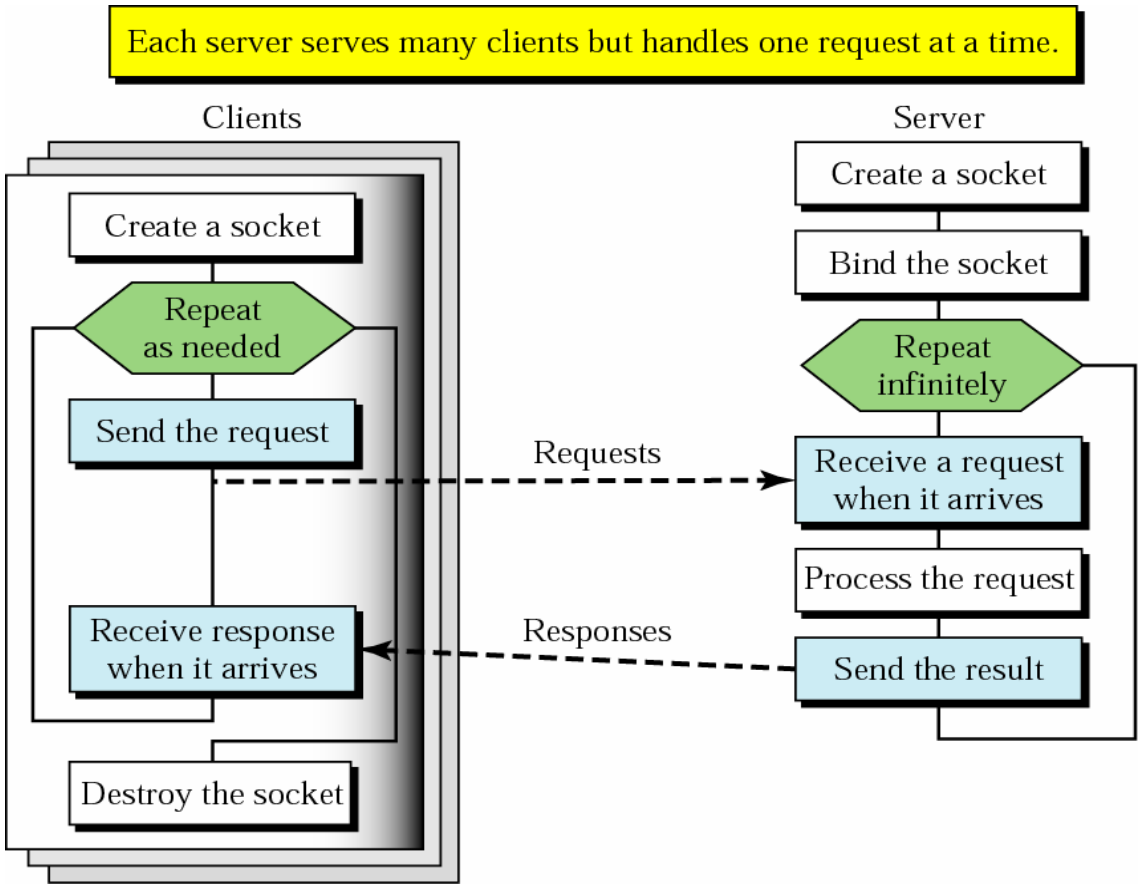
- **SERVER**:

(1) create socket

(2) repeat:
  (2.a) receive
  (2.b) process
  (2.c) send

- all clients use the same socket to communicate with server
- clients and server exchange packets (datagrams)
- no handshaking
- sender explicitly attaches IP address and port of destination to each packet
- server must extract IP and port of sender from received packet to be able to send its response back

Principles of Client-Server Communication with UDP (cont.)

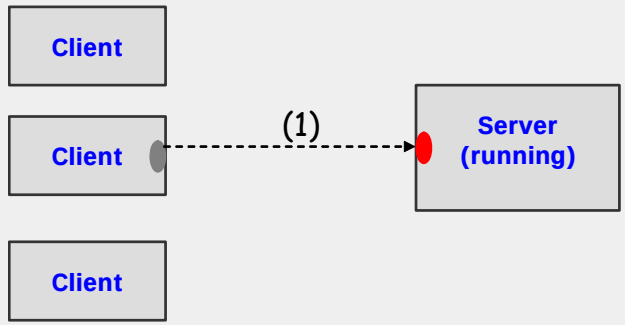


Client-Server Model (cont.)

Principles of Client-Server Communication with TCP

- **CLIENT**:

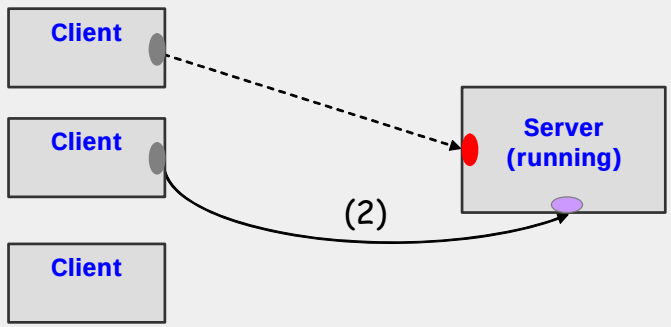
(1) create socket



(2) connect

(3) repeat:
  (3.a) write
  (3.b) read

(4) destroy



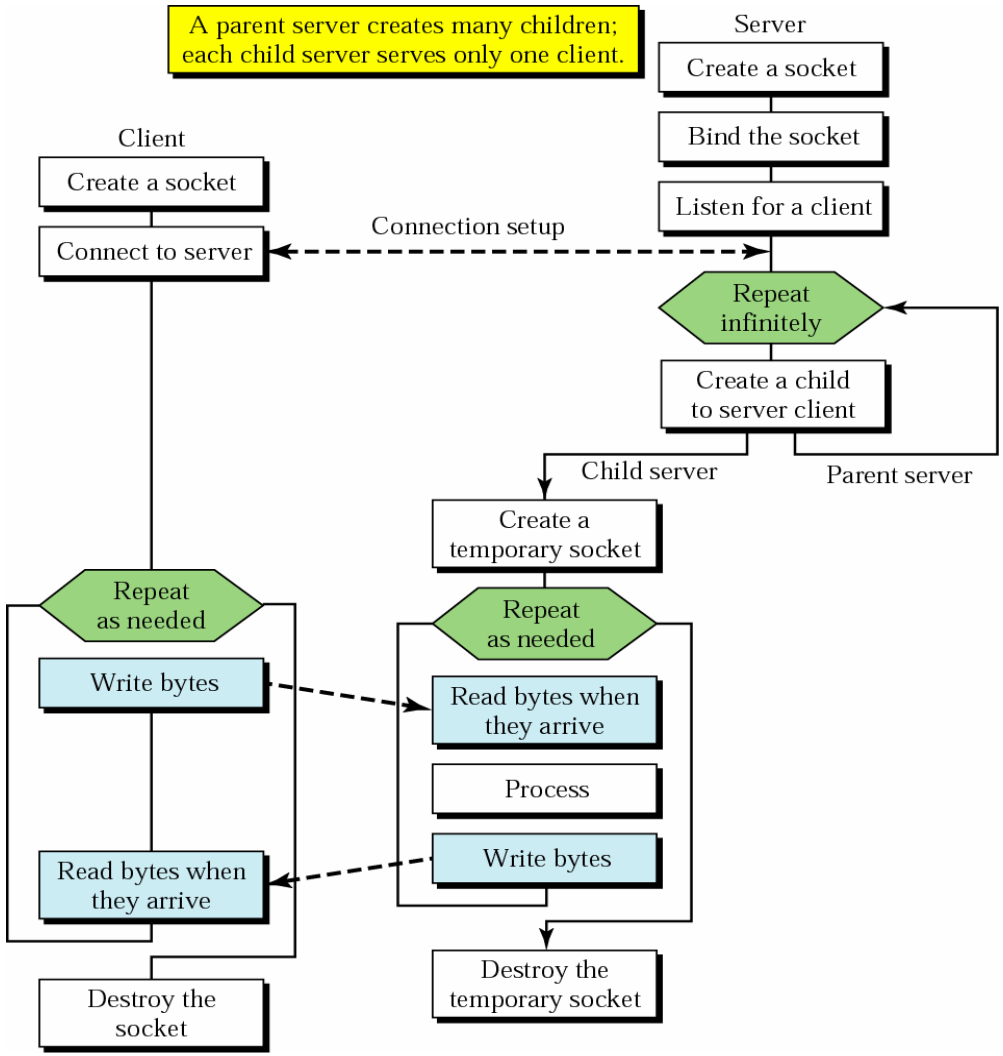
- **SERVER**:

(1) create socket

(2) listen

(3) repeat
  (3.a) create new socket
  (3.b) repeat:
    (3.b.1) read
    (3.b.2) process
    (3.b.3) write
  (3.c) destroy socket

Principles of Client-Server Communication with TCP (cont.)



Java Socket Programming

Advantages of Socket Programming in Java

- applications are more neatly and cleanly written in Java than in C or C++
  - there are fewer lines of code and each line can be explained to novice programmer without much difficulty
- Java keeps all socket transport-layer complexity "under the cover"
  - developer can focus on application rather than worrying about how network and transport layer operate
- Java does not rely on native code $\Rightarrow$ programs can communicate over network (the Internet) in platform-independent fashion

Disadvantages of Socket Programming in Java

- Java does not expose the full range of socket possibilities to developer

Java Socket Programming (cont.)

Example [Java vs. C socket programming]

C code to establish a socket

```
int set_up_socket(u_short port) {
    char myname[MAXHOSTNAME+1];
    int s;
    struct sockaddr_in sa;
    struct hostent *he;
    bzero(&sa,sizeof(struct sockaddr_in));          /* clear the address */
    gethostname(myname,MAXHOSTNAME);                /* establish identity */
    he= gethostbyname(myname);                      /* get our address */
    if (he == NULL)                                 /* if addr not found... */
        return(-1);
    sa.sin_family= he->h_addrtype;                  /* host address */
    sa.sin_port= htons(port);                       /* port number */
    if ((s= socket(AF_INET,SOCK_STREAM,0)) <0)      /* finally, create socket */
        return(-1);
    if (bind(s, &sa, sizeof(sa), 0) < 0) {
        close(s);
        return(-1);                                 /* bind address to socket */
    }
    listen(s, 3);                                   /* max queued connections */
    return(s);
}
```

Java code to establish a socket

```
ServerSocket servsock = new ServerSocket(port, backlog, bindAddr);
```

Java Socket Programming (cont.)

java.net package

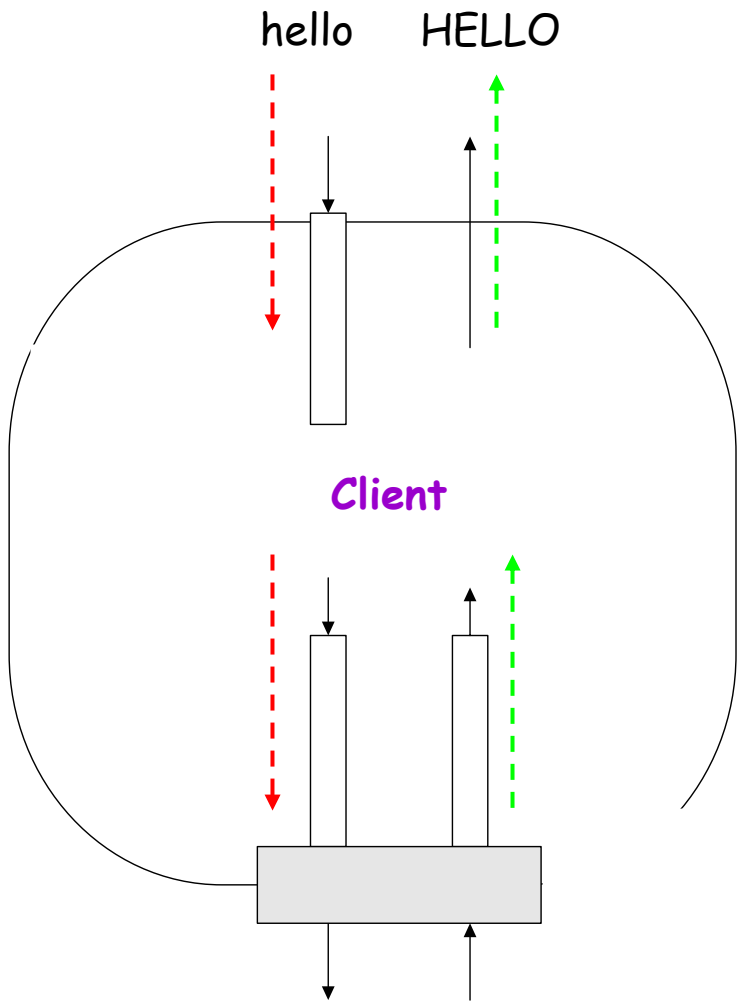
InetAddress class	represents IP address
ServerSocket class	**passive TCP (server) socket** – used on server side to wait for client connection requests
Socket class	**active TCP socket** – can be used as communication end point both on client and server side
DatagramSocket class	**connectionless (UDP) socket** – used for sending and receiving datagrams
DatagramPacket class	datagram packet – in addition to data also contains IP address and port information – **used in UDP!**
MulticastSocket class	**subclass of DatagramSocket** – can used for sending and receiving packets to/from multiple users

Java Socket Programming (cont.)

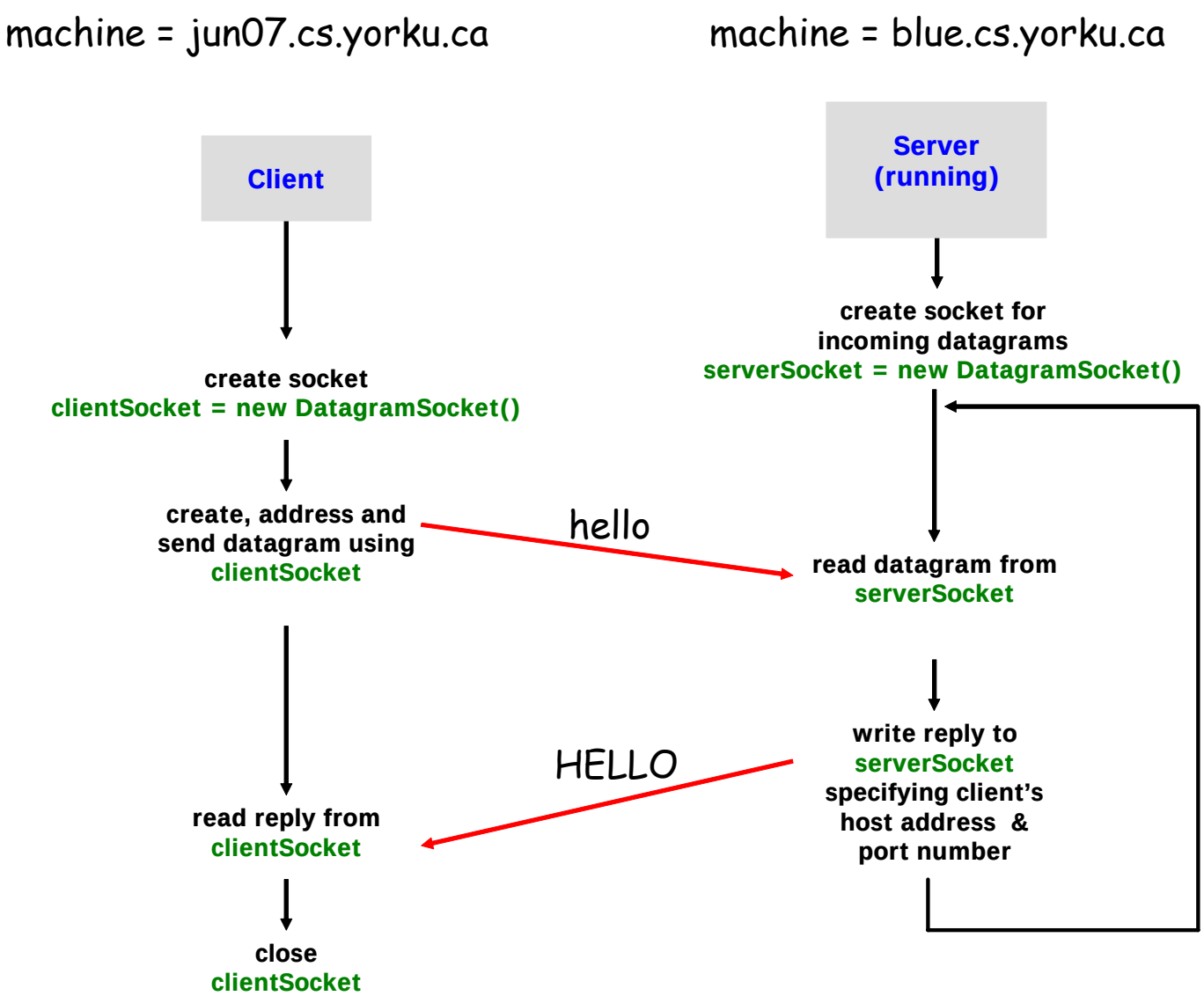
Example [Java socket programming – unicast communication]

Use the following simple client/server application to demonstrate socket programming for both TCP and UDP:

1) **A client reads a line from its standard input (keyboard) and sends line out through its socket to the server.**
2) **The server reads a line from its connection socket.**
3) **The server converts the line to upper case.**
4) **The server sends the modified line out through its socket to the client.**
5) **The client reads the modified line from its socket and prints the line on its standard output (monitor).**



Java Socket Programming with UDP



Java Socket Programming with UDP (cont.)

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

class UDPClient {

    public static void main (String argv[]) throws Exception {

        BufferedReader inFromUser = new BufferedReader (new
        InputStreamReader(System.in));

        byte[] sendData = new byte[1204];
        byte[] receiveData = new byte[1204];

        DatagramSocket clientSocket = new DatagramSocket();

        InetAddress IPAddress = InetAddress.getByName("blue.cs.yorku.ca");
        .....

        String sentence = inFromUser.readLine();

        sendData = sentence.getBytes();
```

create input stream attached to keyboard

byte arrays sendData and receiveData will hold data that client sends and receives in datagrams

create client socket – host does NOT contact server upon execution of this line!

translate hostname to IP address using DNS

store inFromUser to sendData buffer

server runs on blue

Java Socket Programming with UDP (cont.)

server runs port 7777

construct datagram with data, length, server IP address and port number

```
DatagramPacket sendPacket =
new DatagramPacket(sendData, sendData.length, IPAddress, 7777);
```

send datagram

```
clientSocket.send(sendPacket);
```

while waiting for response, create placeholder for packet

```
DatagramPacket receivePacket =
new DatagramPacket(receiveData, receiveData.length);
```

```
clientSocket.receive(receivePacket);
```

read datagram

```
String modifiedSentence = new String(receivePacket.getData());
```

extract data from receivePacket buffer and perform type conversion

```
System.out.println("FROM SERVER: "+modifiedSentence.trim());
```

```
clientSocket.close();
    }
}
```

Java Socket Programming with UDP (cont.)

server on blue.cs.yorku.ca …

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

class UDPServer {

    public static void main (String argv[]) throws Exception {

        DatagramSocket serverSocket = new DatagramSocket(7777);

        byte[] receiveData = new byte[1024];
        byte[] sendData = new byte[1024];

        ..........................................................

        while(true) {

            DatagramPacket receivePacket =
            new DatagramPacket(receiveData, receiveData.length);

            serverSocket.receive(receivePacket);
```

create datagram socket at port 7777

Why do we have to specify port number in this case?!

Java Socket Programming with UDP (cont.)

```
String sentence = new String(receivePacket.getData());

InetAddress IPAddress = receivePacket.getAddress();

int port = receivePacket.getPort();

String capitalizedSentence = sentence.toUpperCase() + '\n';

sendData = capitalizedSentence.getBytes();

DatagramPacket sendPacket = new DatagramPacket(sendData,
sendData.length, IPAddress, port);

serverSocket.send(sendPacket);

        }
    }
}
```

get IP address of the sender

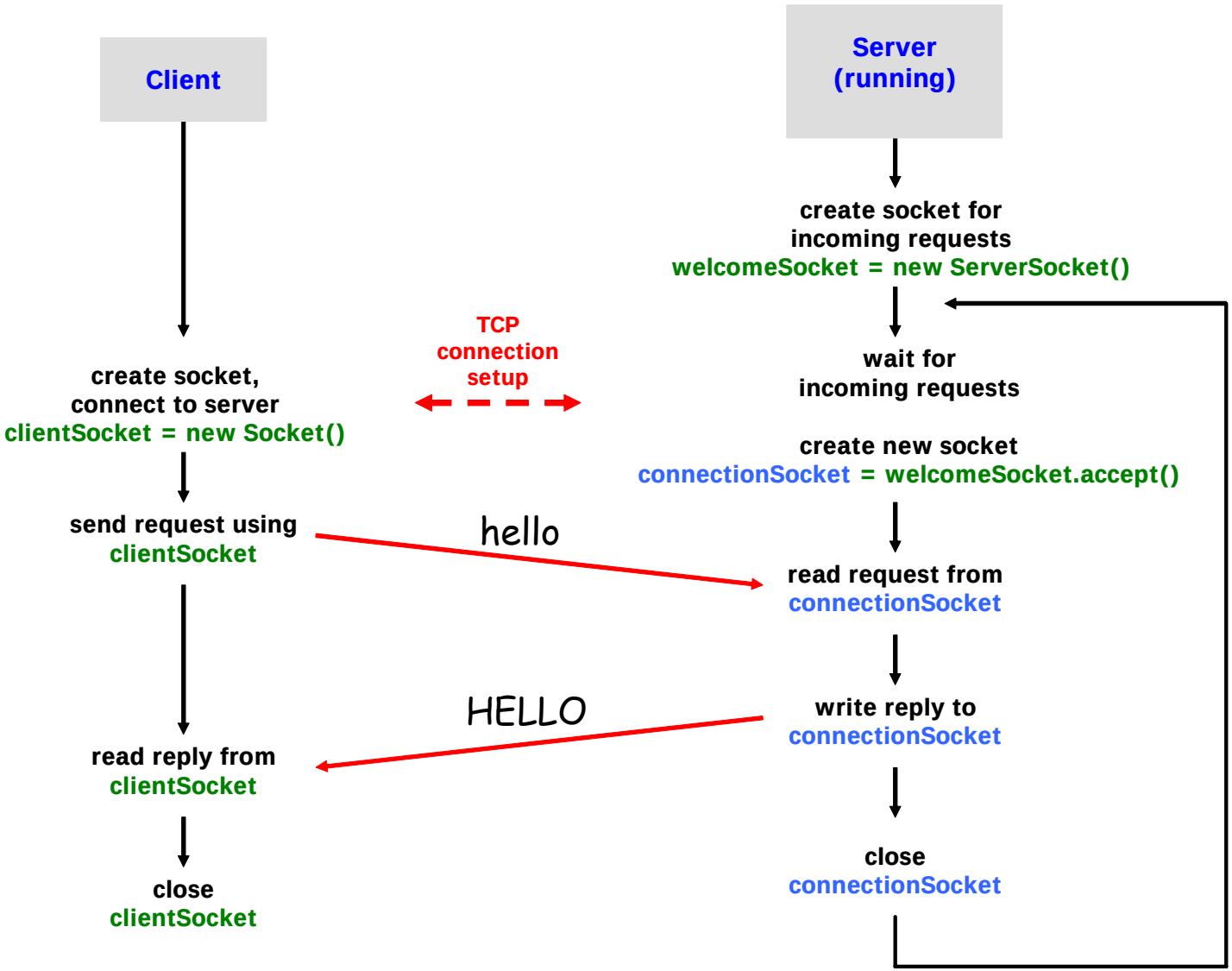
get port number of the sender

create datagram to send to client

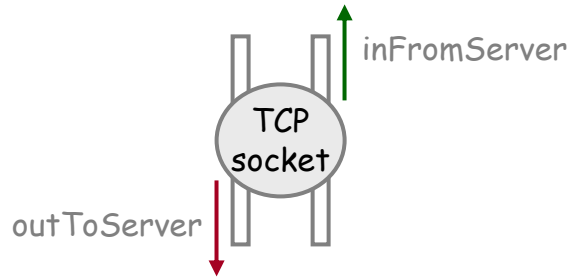
write datagram to socket

loop back and wait for another datagram

Java Socket Programming with TCP



Java Socket Programming with TCP (cont.)




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

class TCPClient {

    public static void main (String argv[]) throws Exception {

        String sentence;
        String modifiedSentence;

        BufferedReader inFromUser = new BufferedReader (new
        InputStreamReader(System.in));

        Socket clientSocket = new Socket("blue.cs.yorku.ca", 5555);

        DataOutputStream outToServer =
        new DataOutputStream (clientSocket.getOutputStream());

        BufferedReader inFromServer = new BufferedReader (new
        InputStreamReader(clientSocket.getInputStream()));
```

create input stream attached to keyboard

create socket; connect it to server

server runs on blue - port 5555

create output stream attached to socket

create input stream attached to socket

Java Socket Programming with TCP (cont.)

place line typed by user into 'sentence'; 'sentence' continues to gather characters until a carriage return

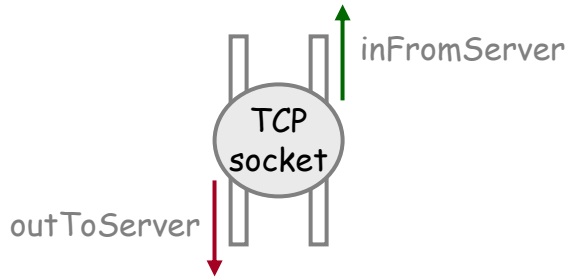
send line to server

print to monitor line read from server

```
                sentence = inFromUser.readLine();

                outToServer.writeBytes(sentence + '\n');

                modifiedSentence = inFromServer.readLine();

                System.out.println("FROM SERVER: "+modifiedSentence);

                clientSocket.close();

            }
        }
```

Java Socket Programming with TCP (cont.)

server on blue.cs.yorku.ca …




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

class TCPServer {

    public static void main (String argv[]) throws Exception {

        String clientSentence;
        String capitalizedSentence;

        ServerSocket welcomeSocket = new ServerSocket(5555);

        while(true) {

            Socket connectionSocket = welcomeSocket.accept();

            BufferedReader inFromClient = new BufferedReader (new
            InputStreamReader(connectionSocket.getInputStream()));
```

create welcoming socket at port 5555

wait for contact-request by clients

once a request arrives, allocate new socket

create & attach input stream to new socket

Java Socket Programming with TCP (cont.)

create & attach output stream to new socket

```
DataOutputStream outToClient =
new DataOutputStream(connectionSocket.getOutputStream());
```

read from socket

```
clientSentence = inFromClient.readLine();
```

write to socket

```
capitalizedSentence = clientSentence.toUpperCase() + '\n';

outToClient.writeBytes(capitalizedSentence);
```

```
            }
        }
    }
```

end of while loop – wait for another client to connect

NOTE: This version of TCP Server is NOT actually serve clients concurrently, but it can be easily modified (with threads) to do so.

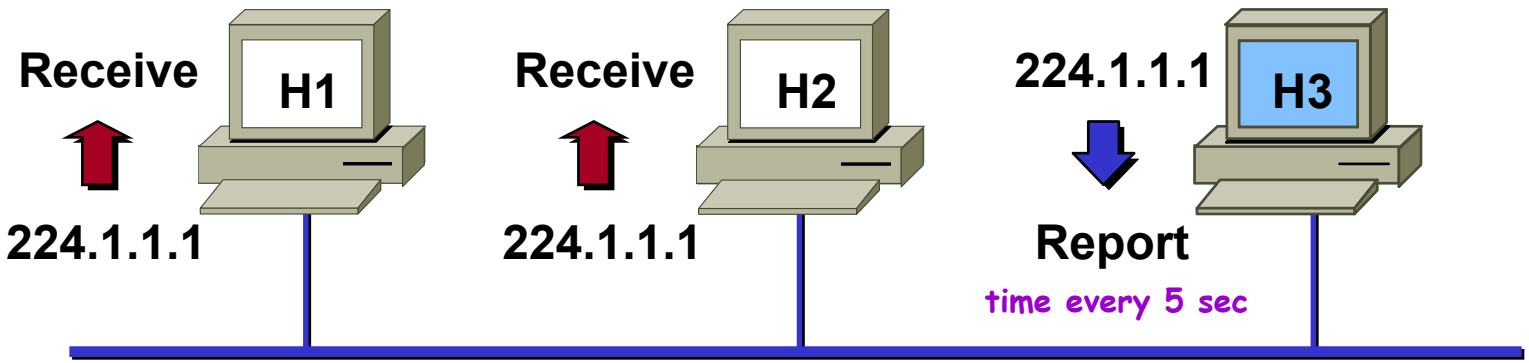
Java Socket Programming – Multicast

Example **[Java socket programming – multicast time/date server]**

Use the following simple application to demonstrate multicast/broadcast socket programming in Java:

MulticastServer uses an arbitrary multicast IP address (any address in the interval 224.0.0.1 to 239.255.255.255) **to broadcast current date and time to all 'subscribed' users. The date and time is broadcasted every 5 [sec].**

A user subscribes to the service by 'tuning in' to the right multicast IP address.



Source: http://www.cswl.com/whiteppr/tutorials/modified.html

Java Socket Programming – Multicast (cont.)

```
import java.io.*;
import java.util.*;

class MulticastServer {

    public static final int destPORT = 1200;

    public static void main(String args[]) throws Exception {

    MulticastSocket socket;
    DatagramPacket packet;
    InetAddress address = InetAddress.getByName("224.1.1.1");

    socket = new MulticastSocket();

    // join a Multicast group and send the group salutations

    socket.joinGroup(address);
```

EXTENDS DatagramSocket class!!!

create multicast socket – socket is bound to any available local port

join multicast group

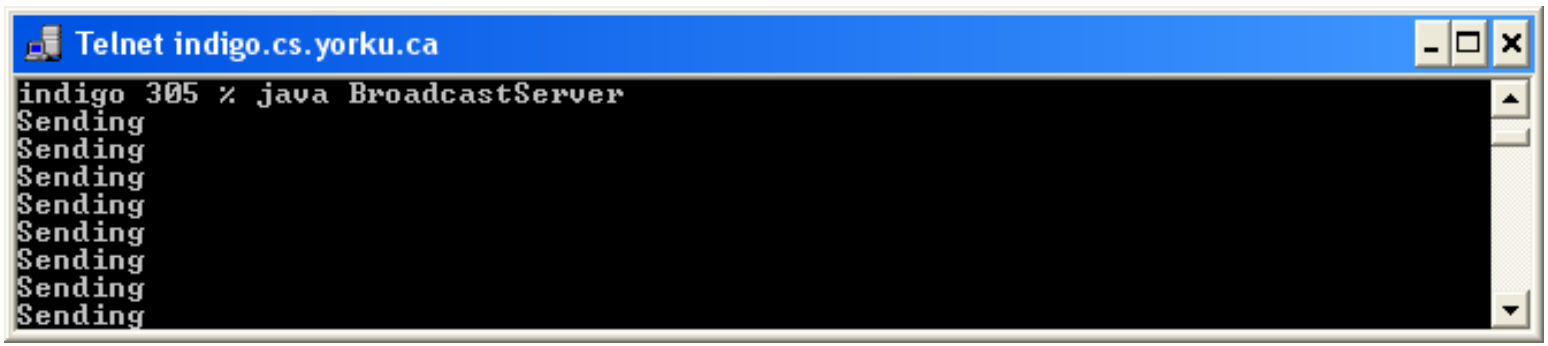
Is this step really necessary?!

Java Socket Programming – Multicast (cont.)

```
byte[] data = new byte[1024];

while(true){
        Thread.sleep(5000);
        System.out.println("Sending ");
        String str = (new Date()).toString();
        data = str.getBytes();
        packet = new DatagramPacket
                (data, str.length(), address, destPORT);
        socket.send(packet);
        }
    }
}
```

datagram destined to all clients listening to destPort who are members of 224.1.1.1 multicast group

send datagram



Java Socket Programming – Multicast (cont.)

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

class MulticastClient {

    public static final int destPORT = 1200;

    public static void main(String args[]) throws Exception{

MulticastSocket socket;
DatagramPacket packet;
InetAddress address;

address = InetAddress.getByName(args[0]);
socket = new MulticastSocket(PORT);

//join a Multicast group and send the group salutations

socket.joinGroup(address);
```

accept multicast address from initial command-line

construct multicast socket and bind it to specific port (another possibility)

join multicast group

Java Socket Programming – Multicast (cont.)

```
byte[] data = new byte[1024];
packet = new DatagramPacket(data,data.length);

while(true){
    socket.receive(packet);
    String str = new String(packet.getData());
    System.out.println(" Time signal received from"+
        packet.getAddress() + " Time is : " +str);
  }
 }
}
```

print sender's unicast IP address and received message

extract sender's IP address

```
Telnet zebra.cs.yorku.ca
zebra 305 % java BroadcastClient 224.1.1.1
 Time signal received from/130.63.92.157 Time is : Thu Aug 26 15:40:33 EDT 2004
 Time signal received from/130.63.92.157 Time is : Thu Aug 26 15:40:38 EDT 2004
 Time signal received from/130.63.92.157 Time is : Thu Aug 26 15:40:43 EDT 2004
 Time signal received from/130.63.92.157 Time is : Thu Aug 26 15:40:48 EDT 2004
 Time signal received from/130.63.92.157 Time is : Thu Aug 26 15:40:53 EDT 2004
 Time signal received from/130.63.92.157 Time is : Thu Aug 26 15:40:58 EDT 2004
 Time signal received from/130.63.92.157 Time is : Thu Aug 26 15:41:03 EDT 2004
```

Exercise

1. **Check for yourself whether the server in “multicast time/date server” could use a simple DatagramSocket instead of MulticastSocket?**
   Would the application performance change?
   Justify your observation!