Java Socket Programming

Amirhossein Sadr

Course: AP SPRING 2024

Instructor: Dr. Mojtaba Vahidi Asl

Recap: Client-Server Communication Paradigm

Typical network app has two pieces: *client* and *server*

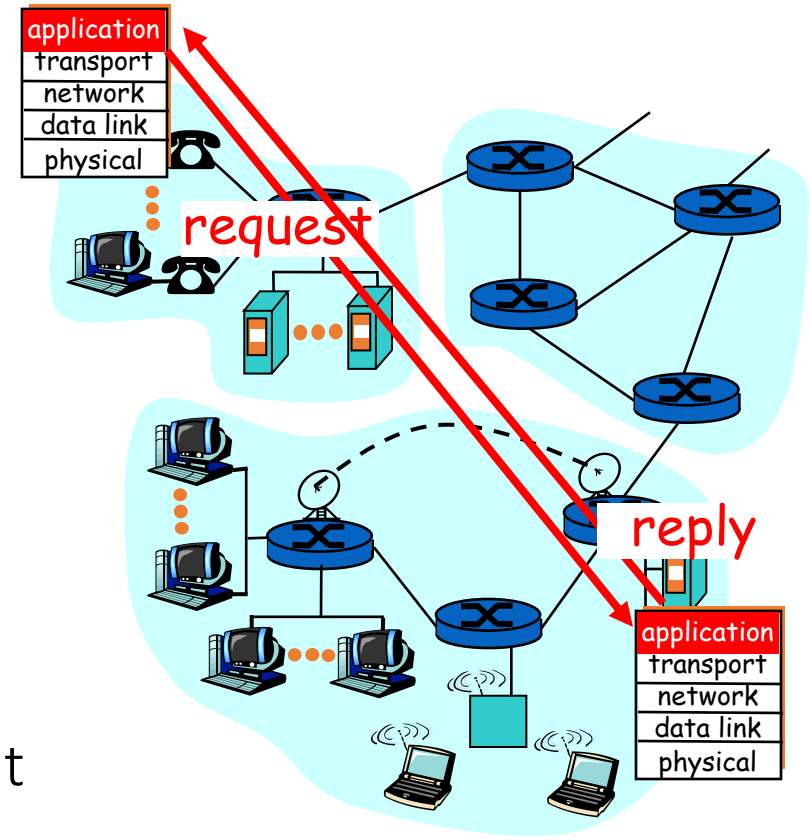
Client:

initiates contact with server ("speaks first")

typically requests service from server

Server:

provides requested service to client



How do clients and servers communicate?

API: application programming interface

- defines interface between application and transport layer
- socket: Internet API
  - two processes communicate by sending data into socket, reading data out of socket

Question: how does a process "identify" the other process with which it wants to communicate?

- IP address of host running other process
- "port number" - allows receiving host to determine to which local process the message should be delivered

… more on this later.

Recap: IP & Ports

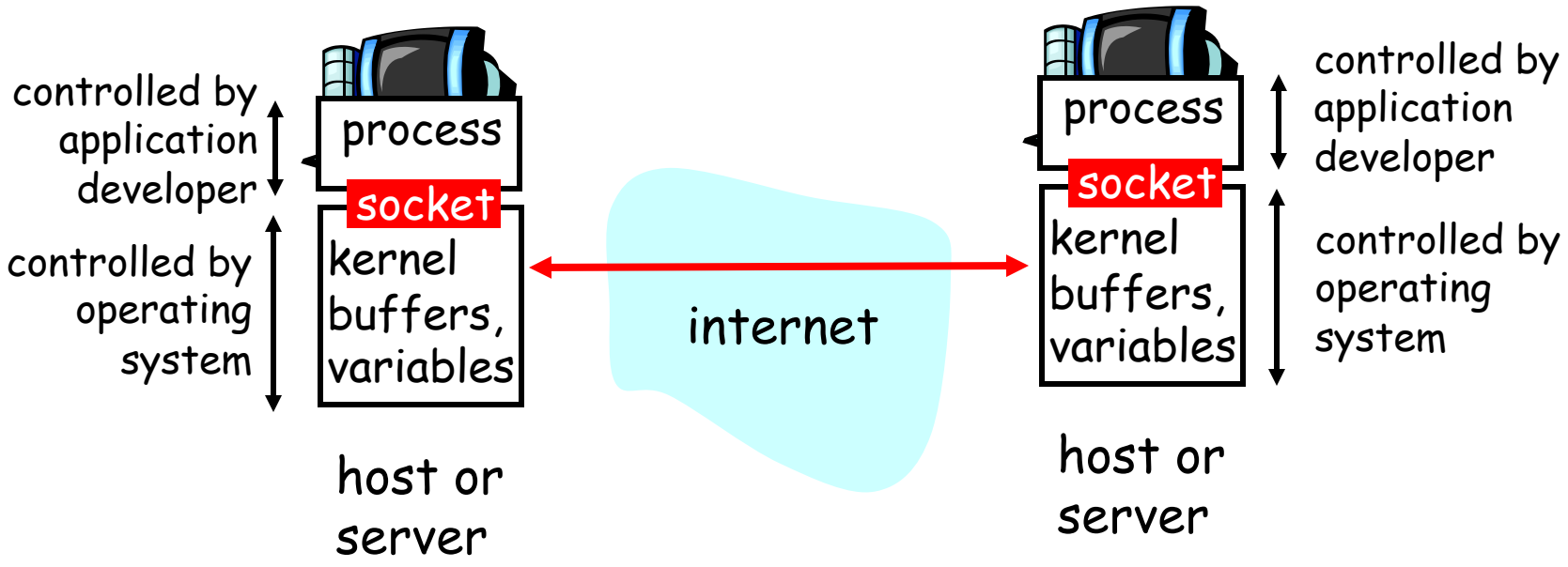
- In computer networking, a port or port number is a number assigned to uniquely identify a connection endpoint and to direct data to a specific service. At the software level, within an operating system, a port is a logical construct that identifies a specific process or a type of network service. A port at the software level is identified for each transport protocol and address combination by the port number assigned to it. The most common transport protocols that use port numbers are the Transmission Control Protocol (TCP) and the User Datagram Protocol (UDP); those port numbers are 16-bit unsigned numbers. port numbers lower than 1024 identify the historically most commonly used services and are called the well-known port numbers.

Recap: IP & Ports

- The Internet Protocol (IP) is the network layer communications protocol in the Internet Protocol Suite for relaying datagrams across network boundaries.
- IP has the task of delivering packets from the source host to the destination host solely based on the IP addresses in the packet headers.
- An Internet Protocol address (IP address) is a numerical label such as *192.0.2.1* that is assigned to a device connected to a computer network that uses the Internet Protocol for communication. IP addresses serve two main functions: network interface identification, and location addressing.

Sockets

Socket: a door between application process and end-to-end transport protocol (UCP or TCP)



Decisions

- Before you go to write socket code, decide
  - Do you want a TCP-style reliable, full duplex, connection oriented channel? Or do you want a UDP-style, unreliable, message oriented channel?
  - Will the code you are writing be the client or the server?
    - Client: you assume that there is a process already running on another machines that you need to connect to.
    - Server: you will just start up and wait to be contacted

Socket Programming is Easy

- Create socket much like you open a file
- Once open, you can read from it and write to it
- Operating System hides most of the details

Socket Programming

Two socket types for two transport services:

- *UDP:* unreliable datagram
- *TCP:* reliable, byte stream-oriented

Application Example:

- client reads a line of characters (data) from its keyboard and sends data to server
- server receives the data and converts characters to uppercase
- server sends modified data to client
- client receives modified data and displays line on its screen

Socket Programming: Basics

- The server application must be running before the client can send anything.
- The server must have a socket through which it sends and receives messages. The client also need a socket.
- Locally, a socket is identified by a port number.
- In order to send messages to the server, the client needs to know the IP address and the port number of the server.

Port number is analogous to an apartment number. All doors (sockets) lead into the building, but the client only has access to one of them, located at the provided number.

OVERVIEW: TCP vs UDP

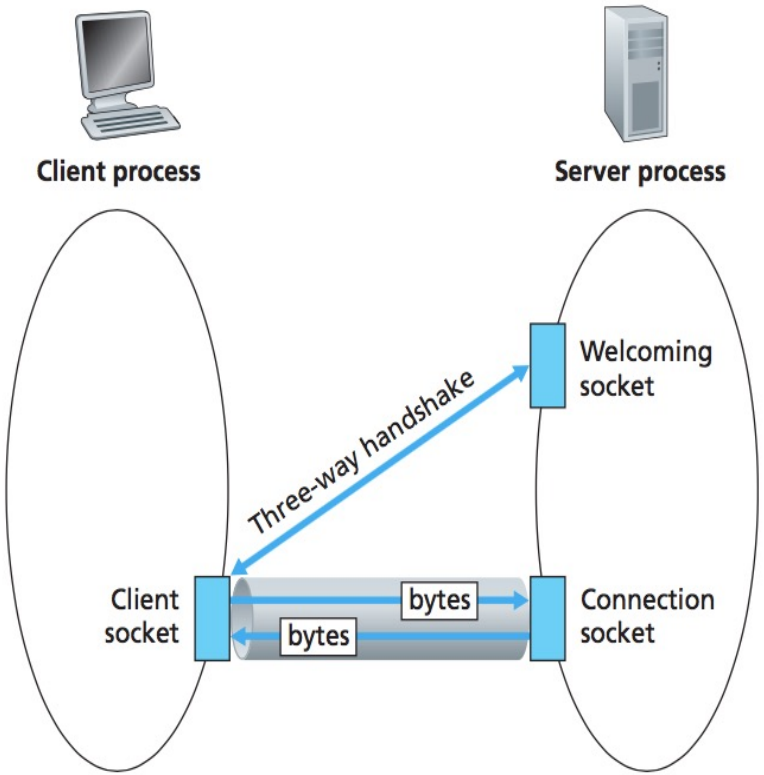
TCP service:

- *connection-oriented:* setup required between client, server
- *reliable transport* between sending and receiving process
- *flow control:* sender won't overwhelm receiver
- *congestion control:* throttle sender when network overloaded
- *does not provide:* timing or minimum bandwidth guarantees

UDP service:

- unreliable data transfer between sending and receiving process
- does not provide: connection setup, reliability, flow control, congestion control, timing, or bandwidth guarantee

Socket Programming *with TCP*



Three-Way Handshake: SYN – SYN/ACK - ACK

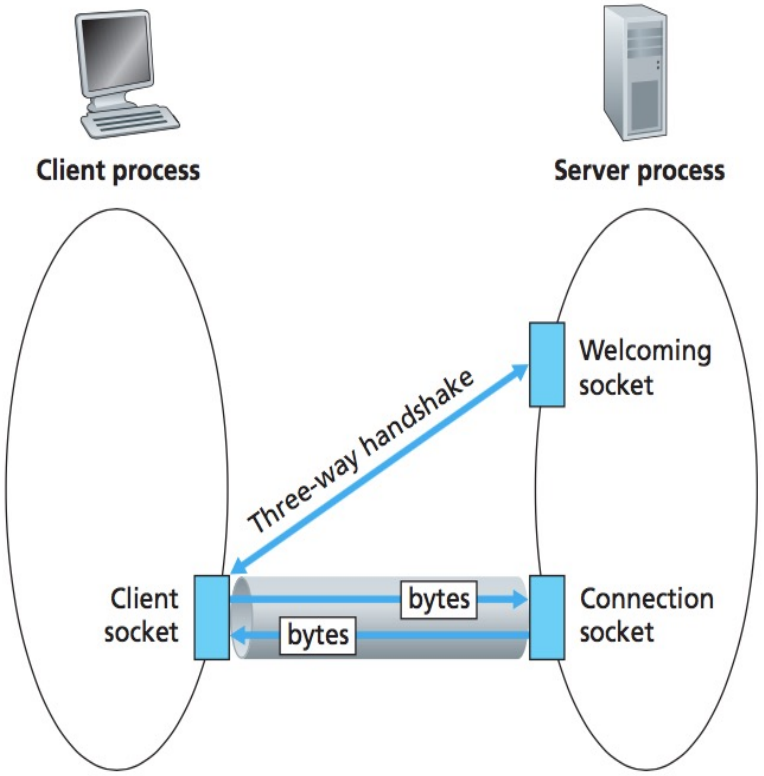
client must contact server

- server process must first be running
- server must have created socket (door) that welcomes client's contact

client contacts server by:

- Creating TCP socket, specifying IP address, port number of server process
- *when client creates socket:* client TCP establishes connection to server TCP

Socket Programming *with TCP*



application viewpoint:

TCP provides reliable, in-order byte-stream transfer ("pipe") between client and server

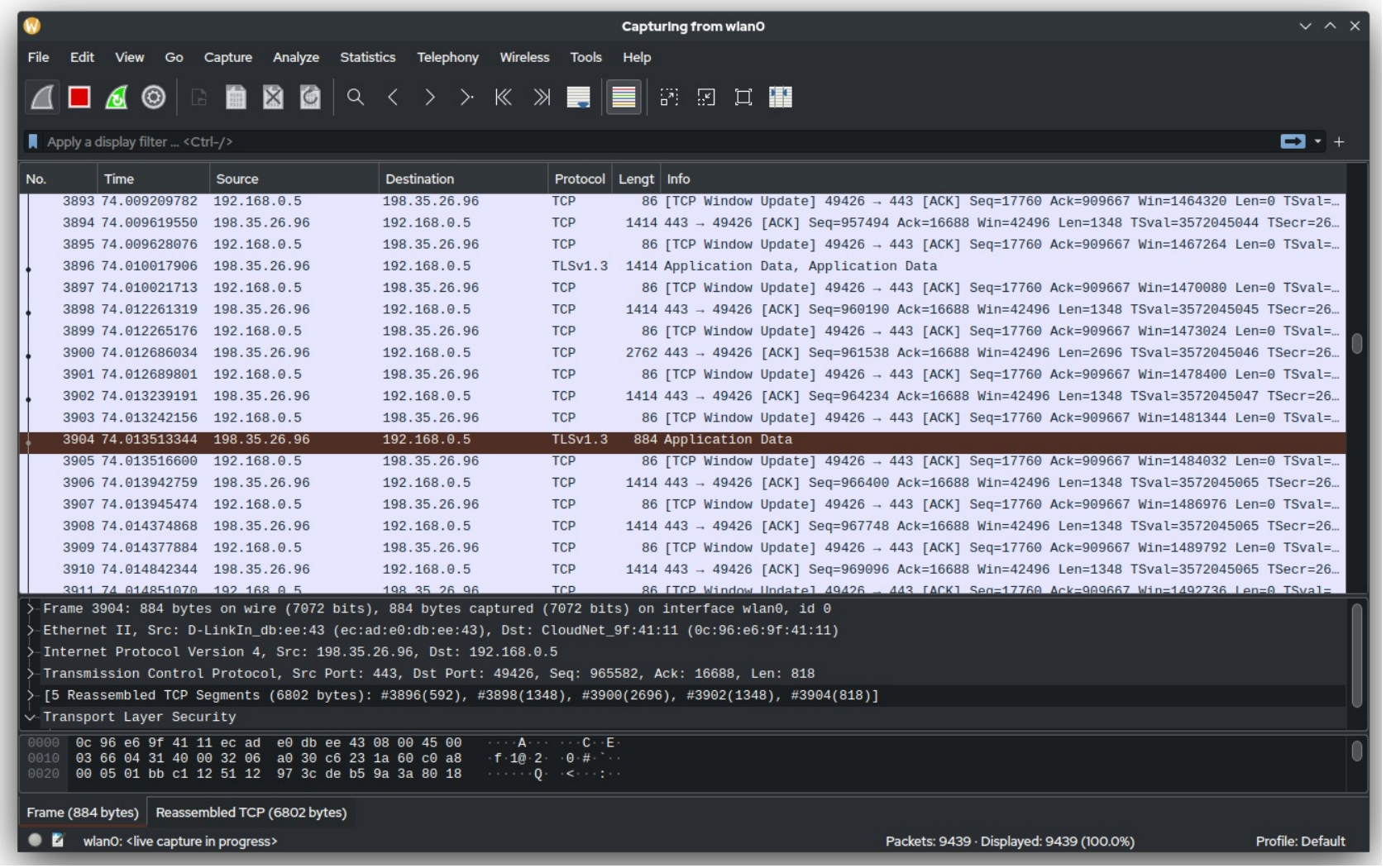
- when contacted by client, *server TCP creates new socket* for server process to communicate with that particular client
  - allows server to talk with multiple clients
  - source port numbers used to distinguish clients (more in Chap 3)

Additional: Wireshark

How to install?

https://www.geeksforgeeks.org/how-to-install-wireshark-on-windows/

Additional: Wireshark



Pseudo code TCP server

- Create socket (doorbellSocket)
- Bind socket to a specific port where clients can contact you
- Register with the kernel your willingness to listen that on socket for client to contact you
- Loop

    Listen to doorbell Socket for an incoming connection, get a connectSocket

    Read and Write Data Into connectSocket to Communicate with client

    Close connectSocket
- End Loop
- Close doorbellSocket

Pseudo code TCP client

- Create socket, connectSocket
- Do an active connect specifying the IP address and port number of server
- Read and Write Data Into connectSocket to Communicate with server
- Close connectSocket

Socket Programming *with UDP*

UDP: no "connection" between client & server

- no handshaking before sending data
- sender explicitly attaches IP destination address and port # to each packet
- receiver extracts sender IP address and port# from received packet

UDP: transmitted data may be lost or received out-of-order

Application viewpoint:

UDP provides *unreliable* transfer of groups of bytes ("datagrams") between client and server

Pseudo code UDP server

- Create socket
- Bind socket to a specific port where clients can contact you
- Loop

  (Receive UDP Message from client x)+

  (Send UDP Reply to client x)*
- Close Socket

Pseudo code UDP client

- Create socket

- Loop

  (Send Message To Well-known port of server)+

  (Receive Message From Server)

- Close Socket

Two Different Server Behaviors

- Iterative server
  - At any time, only handles one client request

```
for (;;) {
   accept a client request;
   handle it
}
```

- Concurrent server
  - Multiple client requests can be handled simultaneously
  - create a new process/thread to handle each request

```
for (;;) {
   accept a client request;
   create a new  process / thread  to
         handle request;
   parent process / thread continues
}
```

Pseudo code concurrent TCP server

- Create socket doorbellSocket
- Bind
- Listen
- Loop

  Accept the connection, connectSocket

  Fork

  If I am the child

  Read/Write connectSocket

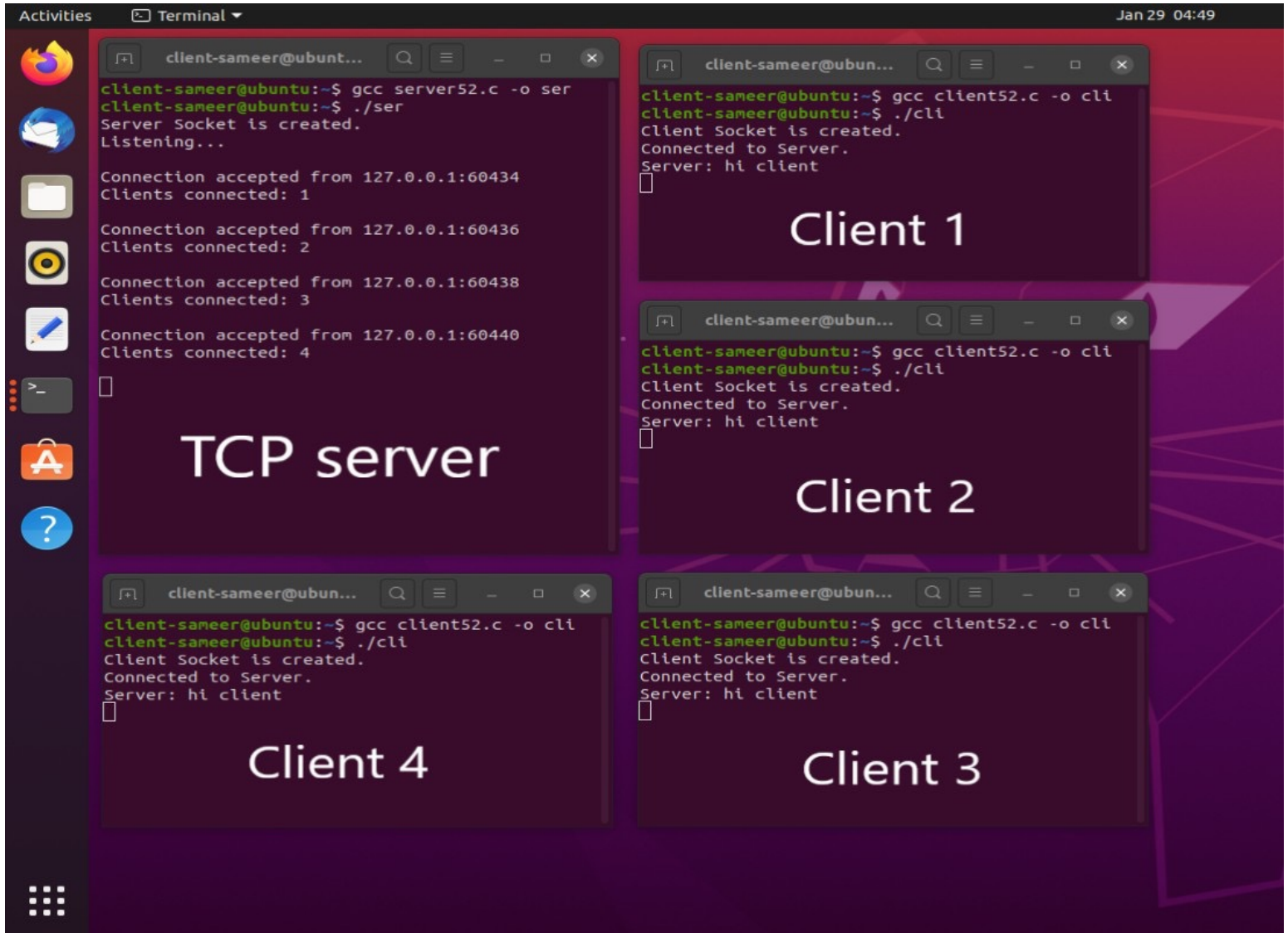
  Close connectSocket

  exit
- EndLoop
- Close doorbellSocket

Related Links

- https://www.geeksforgeeks.org/design-a-concurrent-server-for-handling-multiple-clients-using-fork/
- https://www.geeksforgeeks.org/introducing-threads-socket-programming-java/
- https://www.geeksforgeeks.org/multithreaded-servers-in-java/

Related Links



TCP vs UDP

- TCP can use read/write (or recv/send) and source and destination are implied by the connection; UDP must specify destination for each datagram
  - Sendto, recevfrom include address of other party
- TCP server and client code look quite different; UDP server and client code vary mostly in who sends first

Java Sockets Programming

- The package java.net provides support for sockets programming (and more).

- Typically you import everything defined in this package with:

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

Java Socket Programming API

- Class InetAddress
  - Represents an Internet Protocol (IP) Address
- Class ServerSocket
  - Connection-oriented server side socket
- Class Socket
  - Regular connection-oriented socket (client)
- Class DatagramSocket
  - Connectionless socket

InetAddress class

- static methods you can use to create new InetAddress objects.
  - getByName(String host)
  - getHostAddress(String host)
  - getLocalHost()

```
InetAddress x = InetAddress.getByName(
                    "cse.unr.edu");
```

- Throws **`UnknownHostException`**

Sample Code: Lookup.java

- Uses InetAddress class to lookup hostnames found on command line.

```
> java Lookup www.yahoo.com
www.yahoo.com:209.131.36.158
```

```
try {

  InetAddress a = InetAddress.getByName(hostname);

  System.out.println(hostname + ":" +
                     a.getHostAddress());

} catch (UnknownHostException e) {

  System.out.println("No address found for " +
                     hostname);

}
```

Socket class

- Corresponds to active TCP sockets only!
  - client sockets
  - socket returned by accept();
- Passive sockets are supported by a different class:
  - ServerSocket
- UDP sockets are supported by
  - DatagramSocket

Java TCP Sockets

- java.net.Socket
  - Implements client sockets (also called just “sockets”).
  - An endpoint for communication between two machines.
  - Constructor and Methods
    - Socket(String host, int port): Creates a stream socket and connects it to the specified port number on the named host.
    - InputStream getInputStream()
    - OutputStream getOutputStream()
    - close()

Java TCP Sockets

- java.net.ServerSocket
  - Implements server sockets.
  - Waits for requests to come in over the network.
  - Performs some operation based on the request.
  - Constructor and Methods
    - ServerSocket(int port)
    - Socket Accept(): Listens for a connection to be made to this socket and accepts it. This method blocks until a connection is made.

Socket Constructors

- Constructor creates a TCP connection to a named TCP server.
  - There are a number of constructors:

```
Socket(InetAddress server, int port);

Socket(InetAddress server, int port,
       InetAddress local, int localport);

Socket(String hostname, int port);
```

Socket Methods

```
void close();
InetAddress getInetAddress();
InetAddress getLocalAddress();
InputStream getInputStream();
OutputStream getOutputStream();
```

- Lots more (setting/getting socket options, partial close, etc.)

Socket I/O

- Socket I/O is based on the Java I/O support
  - in the package **`java.io`**

- InputStream and OutputStream are abstract classes
  - common operations defined for all kinds of InputStreams, OutputStreams…

InputStream Basics

```
// reads some number of bytes and
// puts in buffer array b
int read(byte[] b);

// reads up to len bytes
int read(byte[] b, int off, int len);
```

Both methods can throw **`IOException`**.

Both return –1 on EOF.

OutputStream Basics

```
// writes b.length bytes
void write(byte[] b);

// writes len bytes starting
// at offset off
void write(byte[] b, int off, int len);
```

Both methods can throw **`IOException`**.

ServerSocket Class (TCP Passive Socket)

- Constructors:

```
ServerSocket(int port);

ServerSocket(int port, int backlog);

ServerSocket(int port, int backlog,
             InetAddress bindAddr);
```

ServerSocket Methods

```
Socket accept();

void close();

InetAddress getInetAddress();

int getLocalPort();
```

throw **`IOException, SecurityException`**

Example: Java server (TCP)

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

class TCPServer {

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

      ServerSocket doorbellSocket = new ServerSocket(6789);

      while(true) {

        Socket connectSocket = doorbellSocket.accept();

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

Create welcoming socket at port 6789

Wait, on welcoming socket for contact by client

Create input stream, attached to socket

Example: Java server (TCP), cont

Create output stream, attached to socket

Read in line from socket

Write out line to socket

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

clientSentence = inFromClient.readLine();

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

outToClient.writeBytes(capitalizedSentence);
            }
        }
    }
```

End of while loop, loop back and wait for another client connection

Example: Java client (TCP)

```
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 connectSocket = new Socket("hostname", 6789);

        DataOutputStream outToServer =
          new DataOutputStream(connectSocket.getOutputStream());
```

Create input stream

Create client socket, connect to server

Create output stream attached to socket

Example: Java client (TCP), cont.

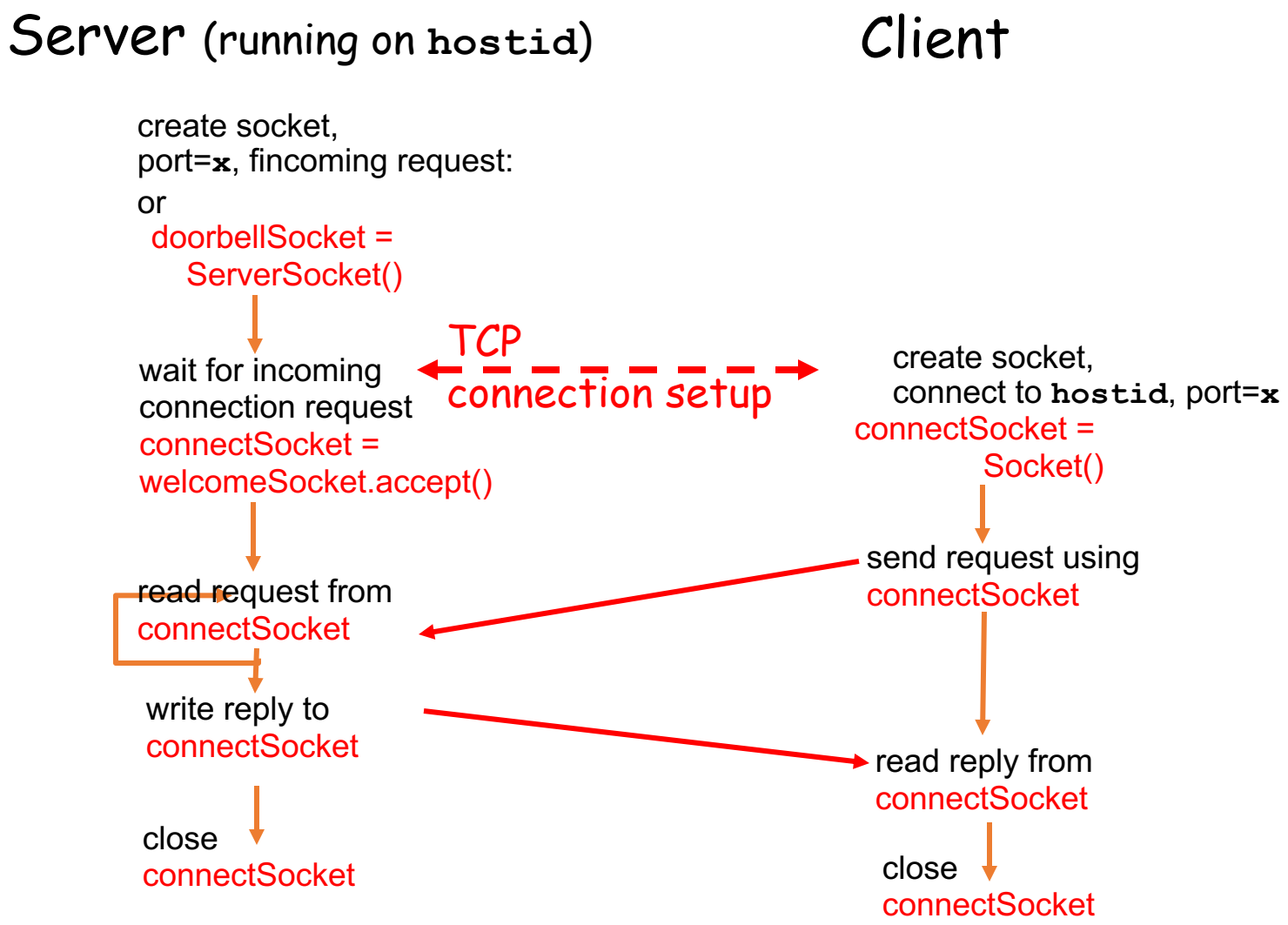
Create input stream attached to socket

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

sentence = inFromUser.readLine();

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

modifiedSentence = inFromServer.readLine();

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

connectSocket.close();

    }
}
```

Send line to server

Read line from server

Client/server socket interaction: TCP (Java)



TCP Server vs Client

- Server waits to accept connection on well known port
- Client initiates contact with the server
- Accept call returns a new socket for this client connection, freeing welcoming socket for other incoming connections
- Read and write only (addresses implied by the connection)

UDP Sockets

- DatagramSocket class

- DatagramPacket class needed to specify the payload
  - incoming or outgoing

Java UDP Sockets

- In Package java.net
  - java.net.DatagramSocket
    - A socket for sending and receiving datagram packets.
    - Constructor and Methods
      - DatagramSocket(int port): Constructs a datagram socket and binds it to the specified port on the local host machine.
      - void receive(DatagramPacket p)
      - void send(DatagramPacket p)
      - void close()

DatagramSocket Constructors

```
DatagramSocket();

DatagramSocket(int port);

DatagramSocket(int port, InetAddress a);
```

All can throw SocketException or SecurityException

Datagram Methods

```
void connect(InetAddress, int port);

void close();

void receive(DatagramPacket p);

void send(DatagramPacket p);
```

Lots more!

Datagram Packet

- Contain the payload
  - (a byte array)

- Can also be used to specify the destination address
  - when not using connected mode UDP

DatagramPacket Constructors

For receiving:

```
DatagramPacket( byte[] buf, int len);
```

For sending:

```
DatagramPacket( byte[] buf, int len
                InetAddress a, int port);
```

DatagramPacket methods

```
byte[] getData();
void setData(byte[] buf);

void setAddress(InetAddress a);
void setPort(int port);

InetAddress getAddress();
int getPort();
```

Example: Java server (UDP)

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

class UDPServer {
  public static void main(String args[]) throws Exception
    {
      DatagramSocket serverSocket = new DatagramSocket(9876);

      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 9876

Create space for received datagram

Receive datagram

Example: Java server (UDP), cont

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

Get IP addr
port #, of      InetAddress IPAddress = receivePacket.getAddress();
sender
                int port = receivePacket.getPort();

                        String capitalizedSentence = sentence.toUpperCase();

                sendData = capitalizedSentence.getBytes();

Create datagram
to send to client  DatagramPacket sendPacket =
                     new DatagramPacket(sendData, sendData.length, IPAddress,
                                        port);

Write out
datagram        serverSocket.send(sendPacket);
to socket
              }
                }

            }
```

End of while loop, loop back and wait for another datagram

Example: Java client (UDP)

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

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

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

      DatagramSocket clientSocket = new DatagramSocket();

      InetAddress IPAddress = InetAddress.getByName("hostname");

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

      String sentence = inFromUser.readLine();

      sendData = sentence.getBytes();
```

Create input stream

Create client socket, DatagramSocket, but no port

Translate hostname to IP address using DNS

Example: Java client (UDP), cont.

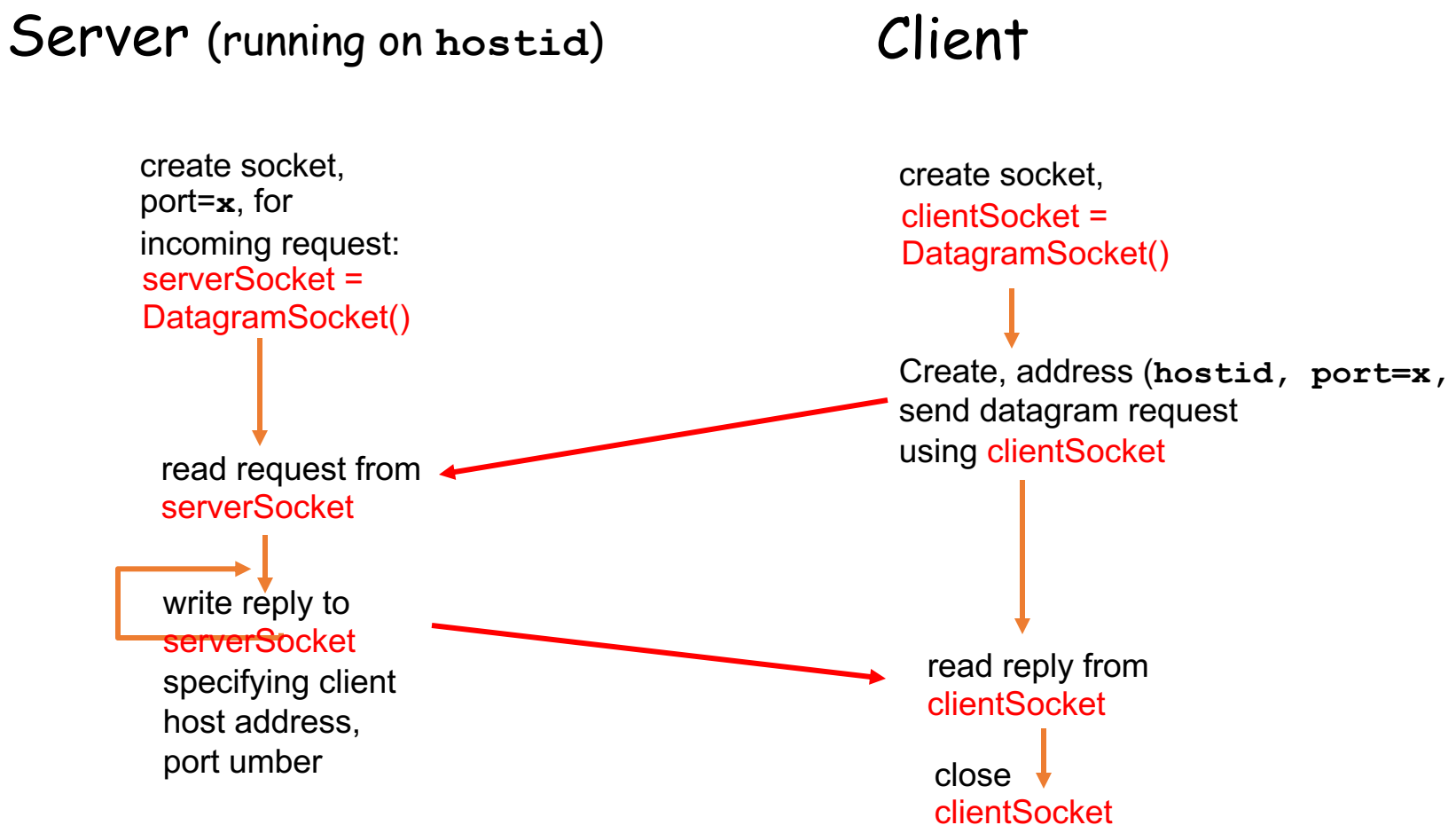
Create datagram with data-to-send, length, IP addr, port

Send datagram to server

Read datagram from server

```
DatagramPacket sendPacket =
   new DatagramPacket(sendData, sendData.length, IPAddress, 9876);

clientSocket.send(sendPacket);

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

clientSocket.receive(receivePacket);

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

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

}
```

Client/server socket interaction: UDP



UDP Server vs Client

- Server has a well-known port number
- Client initiates contact with the server
- Less difference between server and client code than in TCP
  - Both client and server bind to a UDP socket, server specifies port while client does not
  - Not accept for server and connect for client
- Client send to the well-known server port; server extracts the client's address from the datagram it receives

Socket Programming in the Real World

- Download some open source implementations of network applications
  - Web browsers (Firefox)
  - DNS Servers and resolvers (BIND)
  - Email clients/servers (sendmail, qmail, pine)
  - telnet
- Can you find the socket code? The protocol processing? What percentage of the code is it? What does the rest of the code do?

Questions?

Resources

- Introduction to Computer Networks Slides By Amirhossein Sadr
- Dr. Mojtaba Vahidi Asl Slides
- Dr. Sadegh Aliakbari Slides
- Other Universites Slides

Thanks!