# CS2105

## An *Awesome* Introduction to Computer Networks

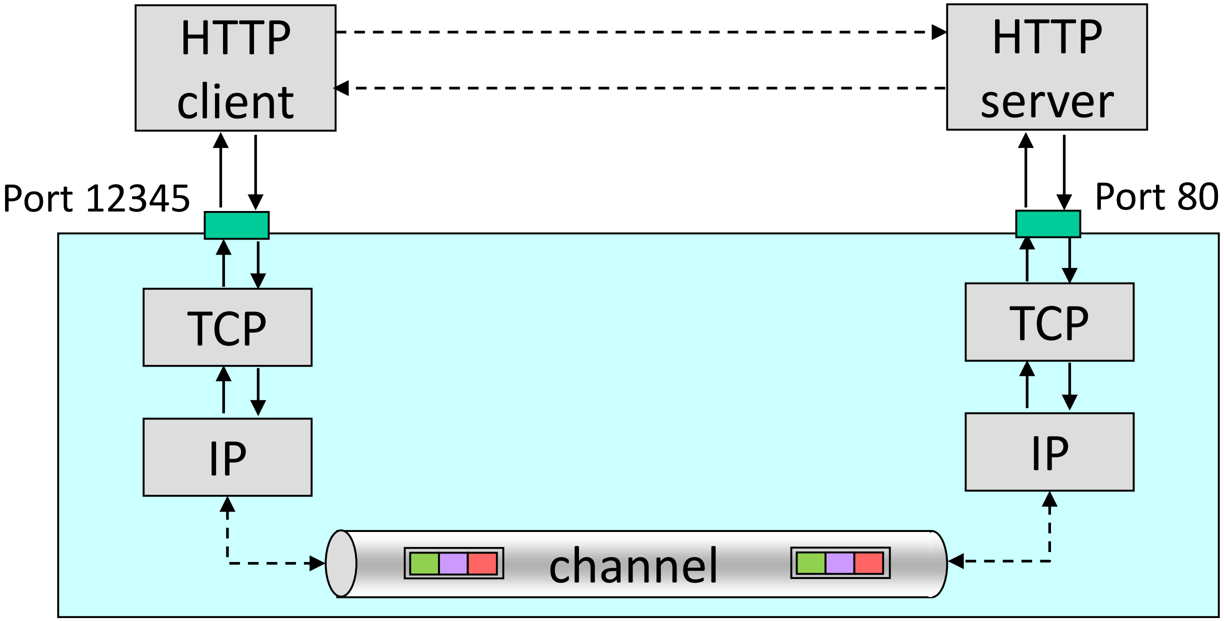
NUS
National University
of Singapore

Department of Computer Science
School of Computing

PREVIOUS LECTURE

# Web and HTTP

- A Web page consists of a *base HTML file* and *some other objects* referenced by the HTML file.
- HTTP uses TCP as transport service.
  - TCP, in turn, uses service provided by IP!



PREVIOUS LECTURE

# HTTP Connections

*HTTP 1.0: non-persistent*

- At most one object is sent over one TCP connection.
  - connection is then closed.
- Downloading multiple objects requires multiple connections.
  - TCP connections may be launched in parallel

*HTTP 1.1: persistent*

- Server leaves connection open after sending a Web object.
- Multiple objects can be sent over a single TCP connection.
  - Requests may be sent in parallel

# Lecture 3: Socket Programming

*After this class, you are expected to:*

- understand the concept of socket.
- be able to write simple client/server programs through socket programming.

# Lecture 3: Roadmap

2.1 Principles of Network Applications

2.2 Web and HTTP

2.5 DNS

2.7 Socket programming with TCP

2.8 Socket programming with UDP

# Processes

- Applications runs in hosts as **processes**.
  - Within the same host, two processes communicate using inter-process communication (defined by OS).
  - Processes in different hosts communicate by exchanging messages (according to protocols).

**In C/S model**

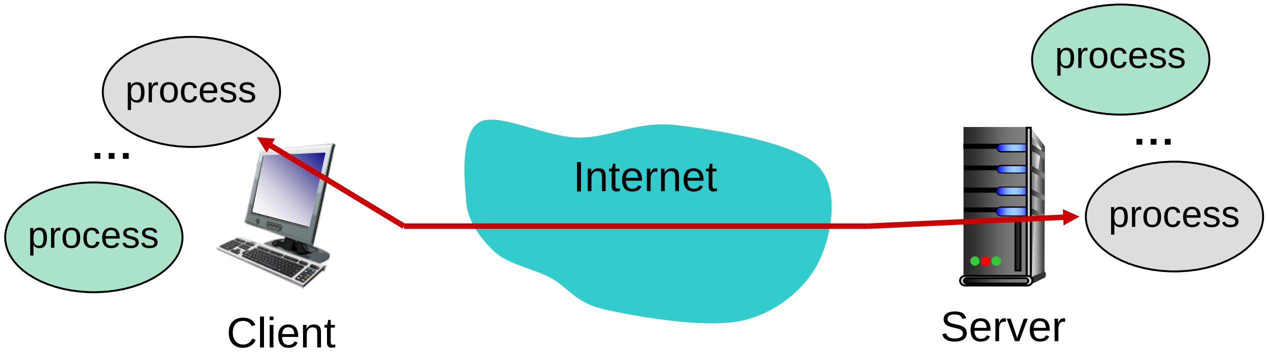
*server process* waits to be contacted

*client process* initiates the communication

# Addressing Processes

- IP address is used to identify a host device
  - A 32-bit integer (e.g. 137.132.21.27)
- Question: is IP address of a host suffice to identify a process running inside that host?

A: no, many processes may run concurrently in a host.



# Addressing Processes

- A process is identified by (IP address, port number).
  - Port number is 16-bit integer (1-1023 are reserved for standard use).
- Example port numbers
  - HTTP server: 80
  - Mail server: 25
- IANA coordinates the assignment of port number:
  - http://www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.xhtml

# Analogy

*Postal service:*

- *deliver letter to the doorstep:* home address

- *dispatch letter to the right person in the house:* name of the receiver as stated on the letter

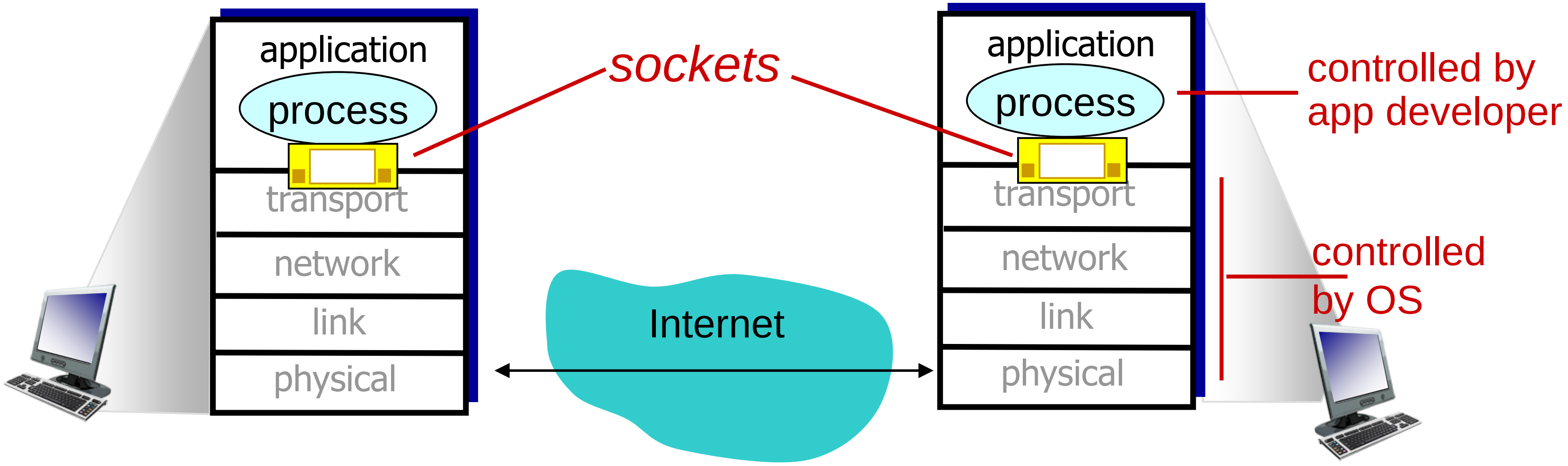
*Protocol service:*

- *deliver packet to the right host:* IP address of the host

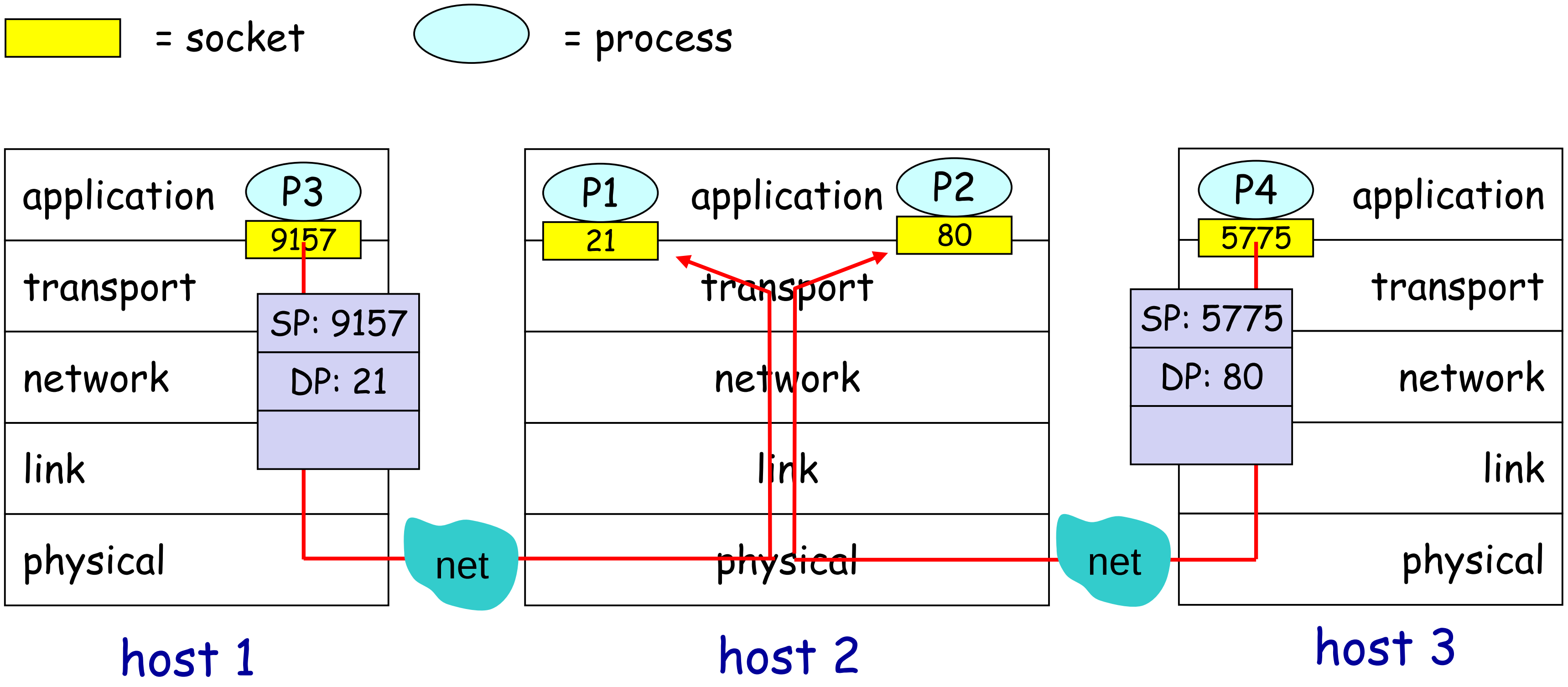
- *dispatch packet to the right process in the host:* port number of the process

# Sockets

- Socket is the software interface between app processes and transport layer protocols.
  - Process sends/receives messages to/from its socket.
  - Programming-wise: a set of API calls



# Multiplexing / de-multiplexing



use IP address + port number to locate a process

# Socket Programming

- Applications (or processes) treat the Internet as a black box, sending and receiving messages through sockets.

- Two types of sockets
  - stream socket (aka TCP socket) that uses TCP as its transport layer protocol.
    - Connection-oriented, reliable
  - datagram socket (aka UDP socket) that uses UDP.
    - Connection-less, unreliable (transmitted data may be lost, corrupted or received out-of-order)

# TCP Socket and UDP Socket

- Now let's write a simple client/server application that client sends a line of text to server, and server echoes it.
  - We will demo both TCP socket version and UDP socket version.

Client must contact server

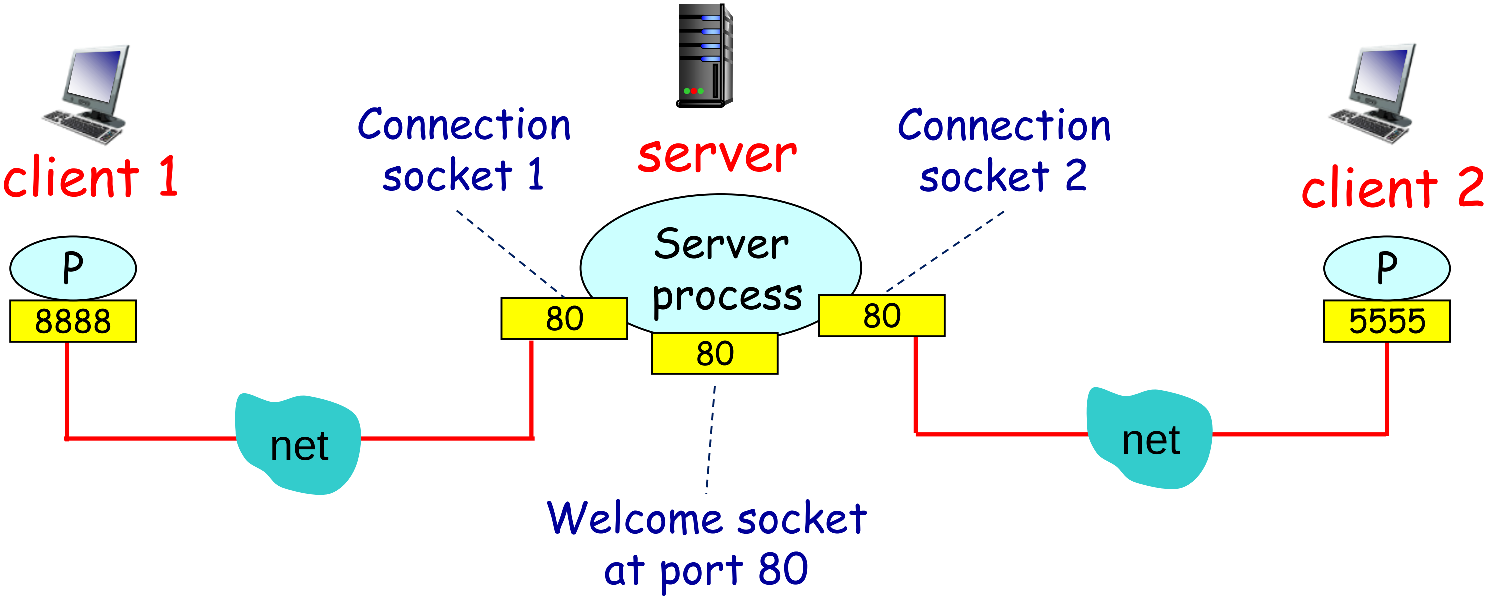
- server process must first be running
- server must have created socket that waits for client's contact

Client contacts server by

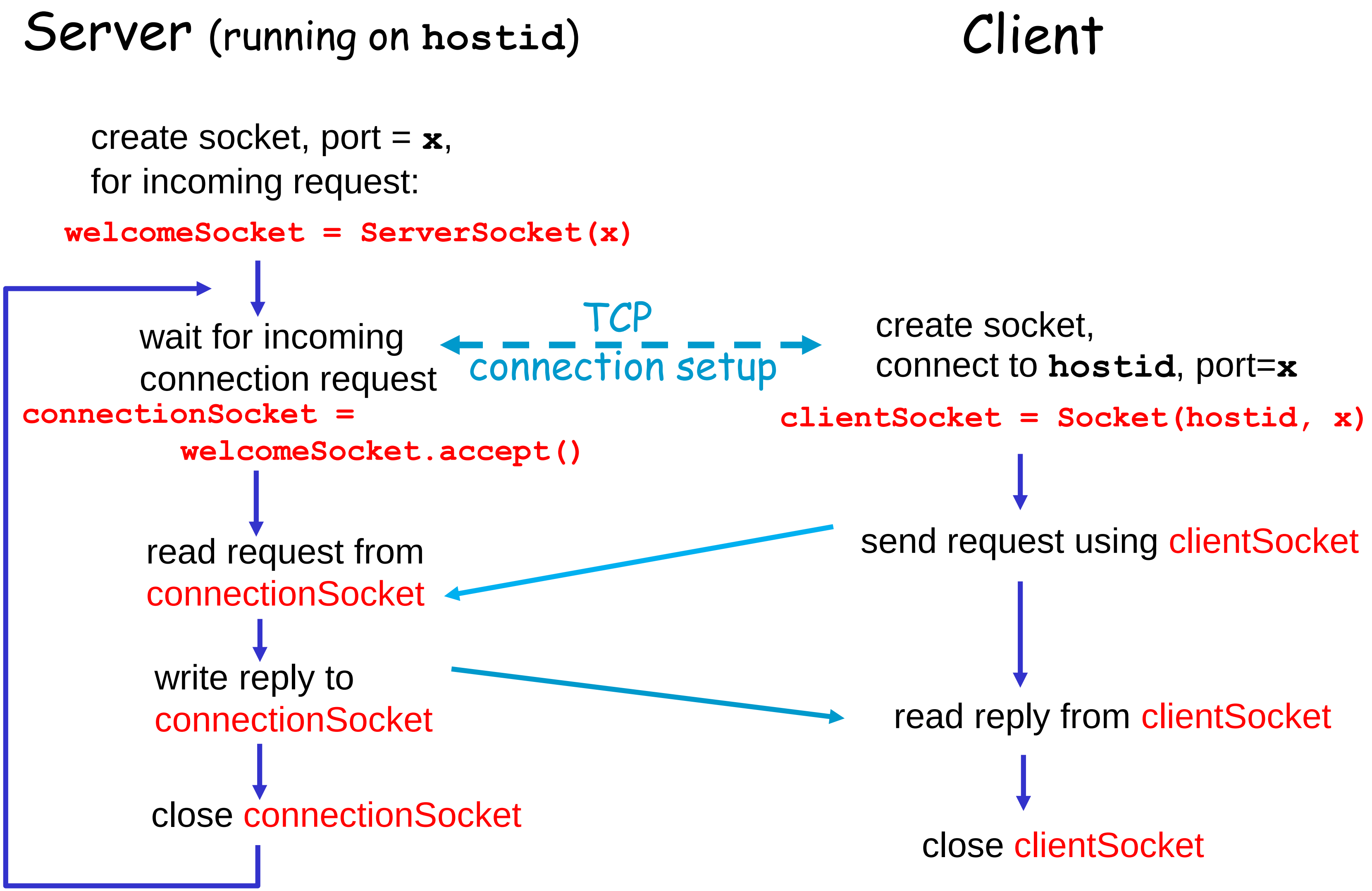
- creating client-local socket
- specifying IP address and port number of server process

# Socket Programming with *TCP*

- With TCP sockets, a process establishes a connection to another process.
- While the connection is in place, data flows between the processes in continuous streams.
- When contacted by client, server TCP creates a new socket for server process to communicate with client.



# TCP: Client/server Socket Interaction



Server (running on **hostid**)

Client

create socket, port = **x**,
for incoming request:

**welcomeSocket = ServerSocket(x)**

wait for incoming
connection request

**connectionSocket =**
**welcomeSocket.accept()**

create socket,
connect to **hostid**, port=**x**

**clientSocket = Socket(hostid, x)**

send request using clientSocket

read request from
connectionSocket

write reply to
connectionSocket

read reply from clientSocket

close connectionSocket

close clientSocket

# Example: TCP Echo Server (1/2)

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

class SimpleTCPEchoServer {

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

        int port = 5678; // server listens to this example port

        // server is waiting
        ServerSocket welcomeSocket = new ServerSocket(port);

        while (true) {  // server is always alive
            Socket connectionSocket = welcomeSocket.accept();


       // to continue next page
```

This package defines Socket and ServerSocket classes

accept() method returns a *new* socket to communicate with client socket

# Example: TCP Echo Server (2/2)

```
            System.out.println("Connected to a client...");

            Scanner scanner = new
                 Scanner(connectionSocket.getInputStream());
            // read data from the connection socket
            String fromClient = scanner.nextLine();


            PrintWriter toClient = new PrintWriter(
                     connectionSocket.getOutputStream(), true);

            // write data to the connection socket
            toClient.println(fromClient);
        }
    }
}
```

read from socket

write to socket

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

# Example: TCP Echo Client (1/2)

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

class SimpleTCPEchoClient {

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

        String serverIP = "127.0.0.1";  // local host, example
        int serverPort = 5678;          // just an example

        // create a client socket and connect to the server
        Socket clientSocket = new Socket(serverIP, serverPort);

        // read user input from keyboard
        Scanner scanner = new Scanner(System.in);
        String fromKeyboard = scanner.nextLine();

       // to continue next page
```

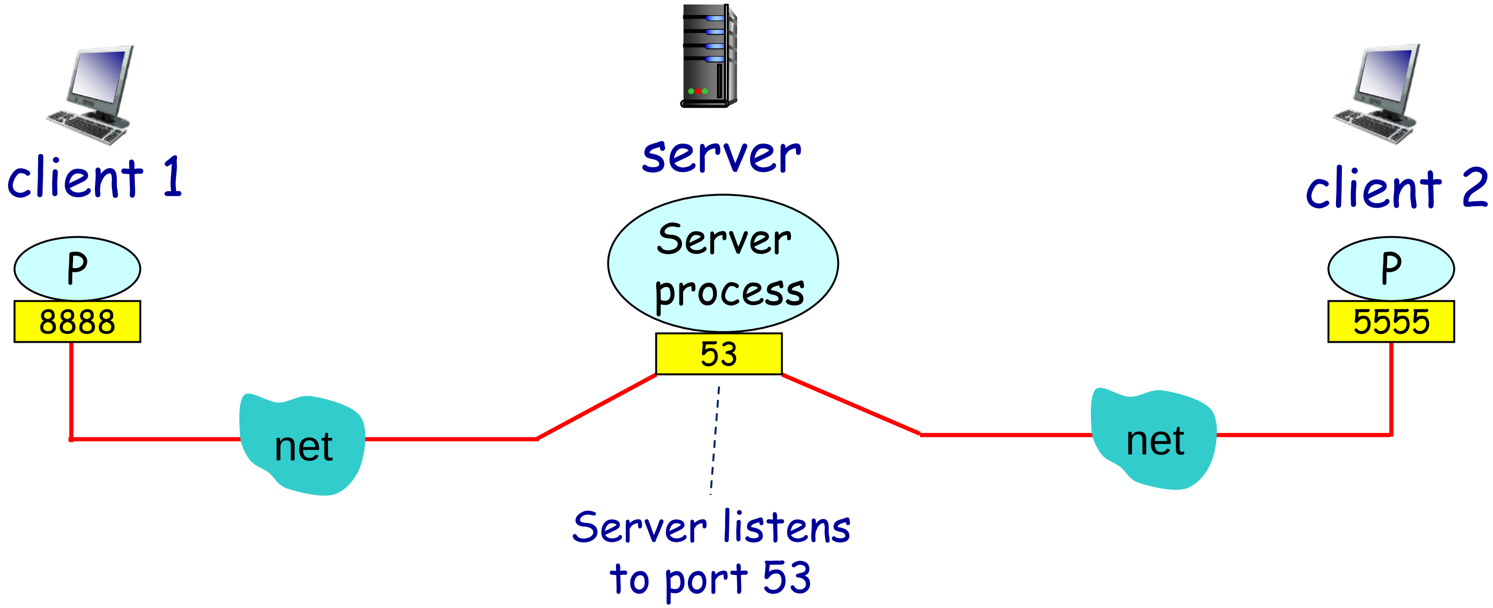
# Example: TCP Echo Client (2/2)

```
        // create output stream to server
        PrintWriter toServer = new
             PrintWriter(clientSocket.getOutputStream(), true);
        // write user input to the socket
        toServer.println(fromKeyboard);

        // create input stream from server
        Scanner sc =
                  new Scanner(clientSocket.getInputStream());
        // read server reply from the socket
        String fromServer = sc.nextLine();

        // show on screen
        System.out.println("Echo from server: " + fromServer);

        clientSocket.close();
    }
}
```

# Socket Programming with *UDP*

UDP: no "connection" between client and server

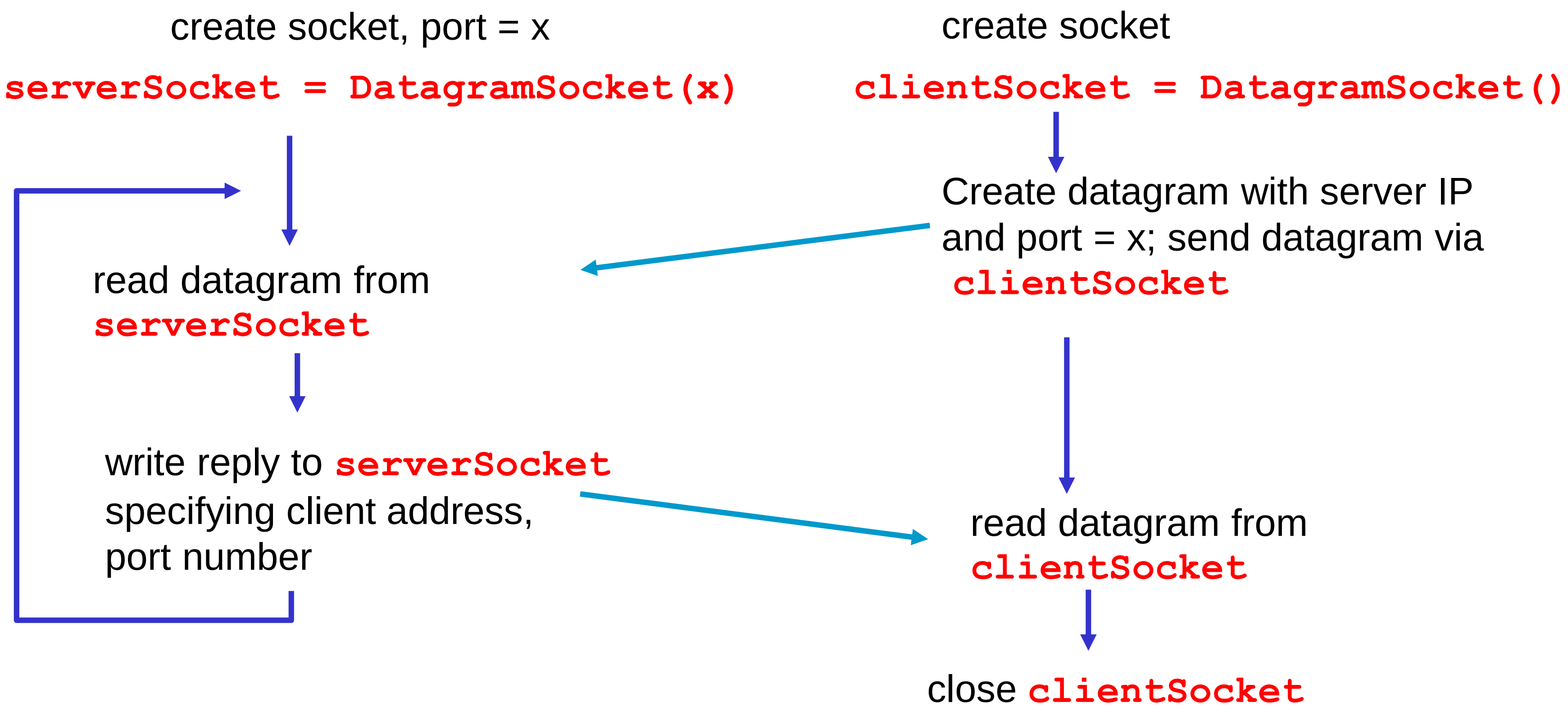
- Sender (client) explicitly attaches destination IP address and port number to every packet.
- Receiver (server) extracts sender IP address and port number from the received packet.



# UDP: Client/server Socket Interaction

Server (running on `hostid`)

Client



create socket, port = x

`serverSocket = DatagramSocket(x)`

read datagram from `serverSocket`

write reply to `serverSocket` specifying client address, port number

create socket

`clientSocket = DatagramSocket()`

Create datagram with server IP and port = x; send datagram via `clientSocket`

read datagram from `clientSocket`

close `clientSocket`

# Example: UDP Echo Server (1/2)

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

class SimpleUDPEchoServer {

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

        int port = 5678; // server listens to this example port
        DatagramSocket serverSocket = new DatagramSocket(port);

        byte[] rcvBuffer = new byte[1024];

        while (true) {  // server is always alive
            DatagramPacket rcvedPkt = new
                 DatagramPacket(rcvBuffer, rcvBuffer.length);

            serverSocket.receive(rcvedPkt);

       // to continue next page
```

receive() method blocks
till a packet is received

# Example: UDP Echo Server (2/2)

```
            String rcvedData = new String(rcvedPkt.getData(),
                                     0, rcvedPkt.getLength());

            InetAddress clientAddress = rcvedPkt.getAddress();

            int clientPort = rcvedPkt.getPort();

            byte[] sendData = rcvedData.getBytes();

            DatagramPacket sendPkt =
                 new DatagramPacket(sendData, sendData.length,
                                    clientAddress, clientPort);

            serverSocket.send(sendPkt);
        }
    }
}
```

extract client
address and
port number

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

# Example: UDP Echo Client (1/2)

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

class SimpleUDPEchoClient {

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

        InetAddress serverAddress =         // server IP address
                    InetAddress.getByName("localhost");
        int serverPort = 5678;               // just an example

        // create a client socket
        DatagramSocket clientSocket = new DatagramSocket();

        // read user input from keyboard
        Scanner scanner = new Scanner(System.in);
        String fromKeyboard = scanner.nextLine();

       // to continue next page
```

translate hostname to IP address using DNS

create a client socket

# Example: UDP Echo Client (2/2)

```
        // create a datagram and send to server
        byte[] sendData = fromKeyboard.getBytes();
        DatagramPacket sendPkt = new DatagramPacket(sendData,
                   sendData.length, serverAddress, serverPort);

        clientSocket.send(sendPkt);
```

create a datagram to send

```
        // receive a packet sent by server from socket
        byte[] rcvBuffer = new byte[1024];
        DatagramPacket rcvedPkt = new DatagramPacket(rcvBuffer,
                                             rcvBuffer.length);
        clientSocket.receive(rcvedPkt);
```

read a datagram
from server

```
        System.out.println("Echo from server: " +
                           new String(rcvedPkt.getData(), 0,
                                      rcvedPkt.getLength()));

        clientSocket.close();
    }
}
```

# TCP Socket vs. UDP Socket

- In TCP, two processes communicate as if there is a pipe between them. The pipe remains in place until one of the two processes closes it.
  - When one of the processes wants to send more bytes to the other process, it simply writes data to that pipe.
  - The sending process doesn’t need to attach a destination address and the port number to the bytes in each sending attempt as the logical pipe has been established (which is also reliable).
- In UDP, programmers need to form UDP datagram packets explicitly and attach destination IP address / port number to every packet.

# Lecture 3: Summary

- Socket programming
  - TCP socket
    - When contacted by client, server TCP creates new socket.
    - Server uses (client IP + port #) to distinguish clients.
    - When client creates its socket, client TCP establishes connection to server TCP.
  - UDP socket
    - Server use one socket to serve all clients.
    - No connection is established before sending data.
    - Sender explicitly attaches destination IP address and port # to each packet.
    - Transmitted data may be lost or received out-of-order.