UCL DEPARTMENT OF ELECTRONIC & ELECTRICAL ENGINEERING

ELEC0021 - PROGRAMMING II
OBJECT-ORIENTED PROGRAMMING

NETWORKING

Prof. George Pavlou
Communication and Information Systems Group
Dept. of Electronic and Electrical Engineering
http://www.ee.ucl.ac.uk/~gpavlou/
g.pavlou@ucl.ac.uk
Tel: 020 76793985

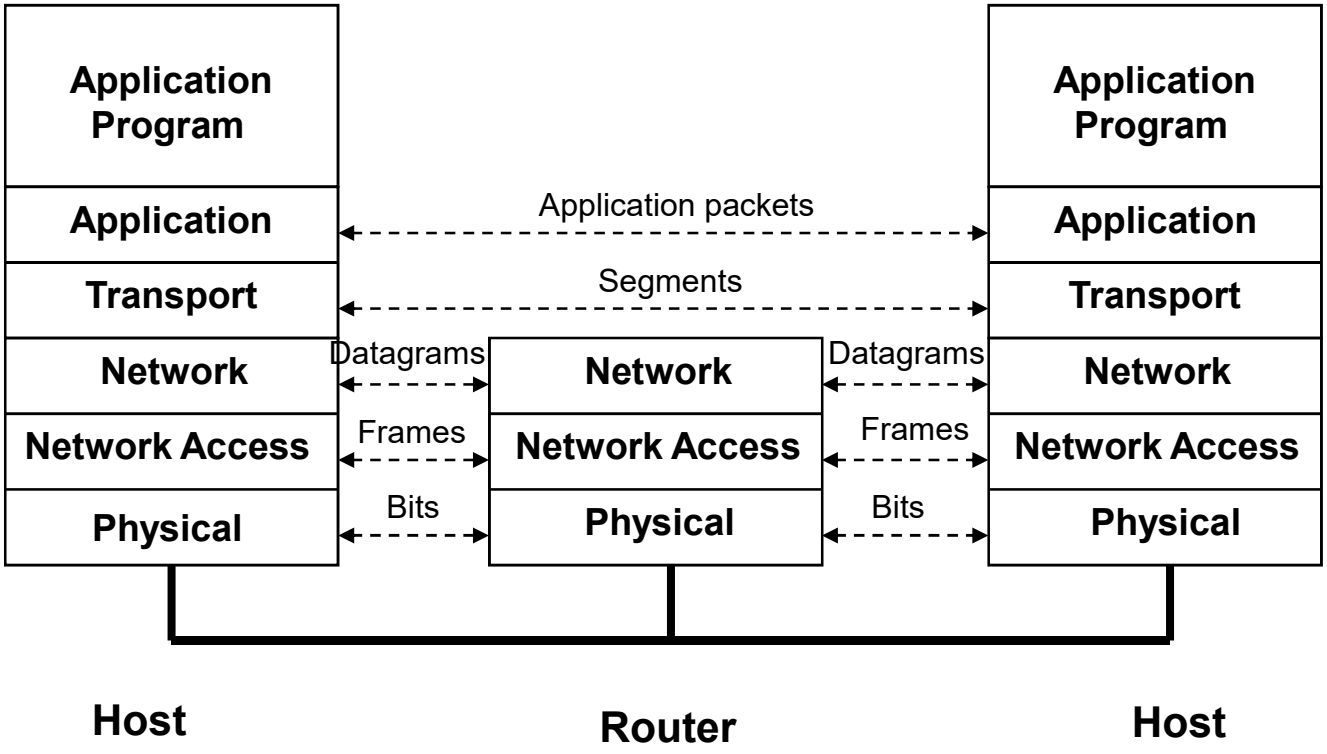
The Internet and TCP/UDP/IP

- The Internet is based on the TCP/IP protocol architecture
- The Internet Protocol (IP) is the uniform network level protocol whose key functionality is packet routing and forwarding
  - An IP packet is also called a datagram
  - IP runs in everywhere i.e. in both end systems or hosts and intermediate systems or routers
- Two transport protocols run above IP, the Transmission Control Protocol (TCP) and the User Datagram Protocol (UDP)
  - TCP and UDP run only in the end systems / hosts

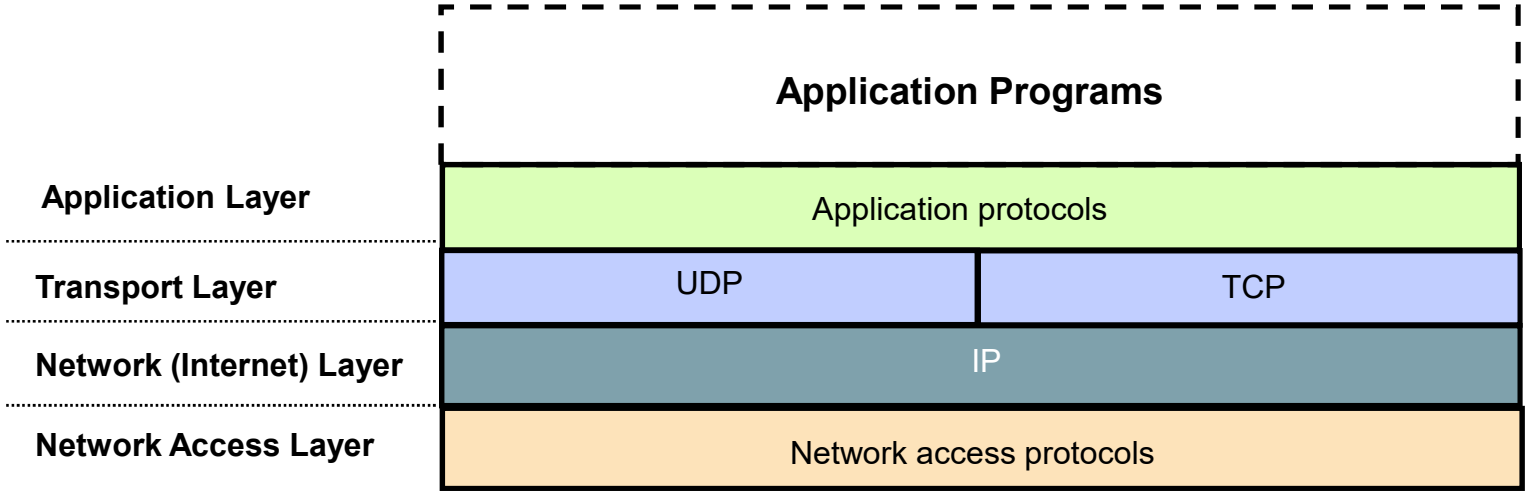
TCP and UDP

- The Transmission Control Protocol (TCP) is a stream-based connection-oriented reliable transport protocol
  - Stream-based: data is treated as a stream of bytes with no packet boundaries, data may be buffered before transmitted at the sender or before it is passed to the application in the receiver
  - Connection-oriented: connection establishment is required before data transfer – and connection release after finishing
  - Reliable: transmitted packets (called “segments”) are guaranteed to be received intact and in sequence
- The User Datagram Protocol (UDP) is a datagram-oriented connectionless unreliable transport protocol
  - Datagram-oriented: the transport unit is a packet (datagram) which is transmitted and delivered immediately at the sender and receiver
  - Connectionless: no connection establishment/release is required
  - Unreliable: transmitted datagrams may be lost or arrive out-of-sequence in the receiver
- Most data applications use TCP and most real-time streaming applications use UDP, although this is not always the case

The Internet Layered Model



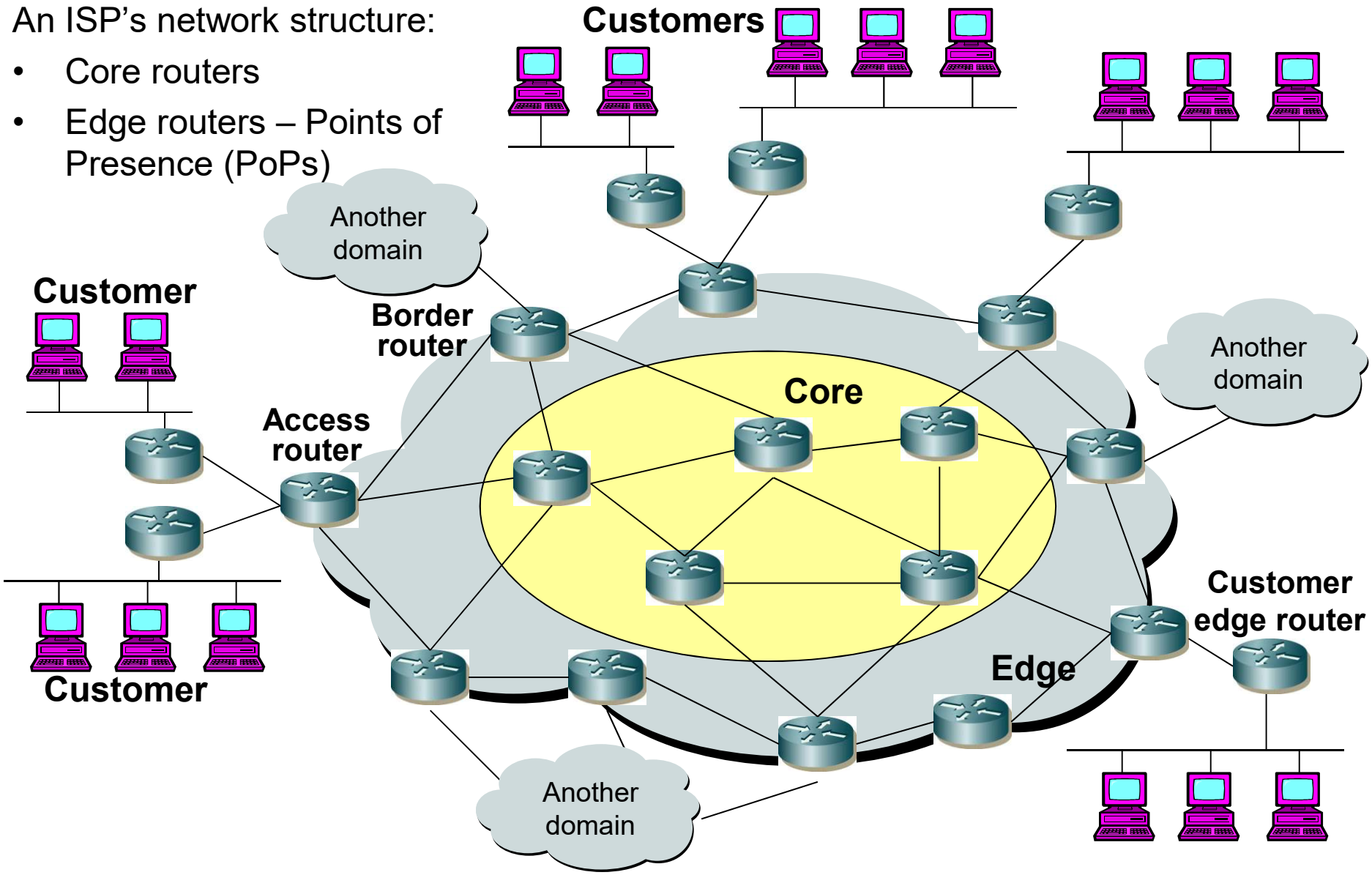
The Internet Layered Model (cont'd)



Typical ISP Network Structure

An ISP's network structure:

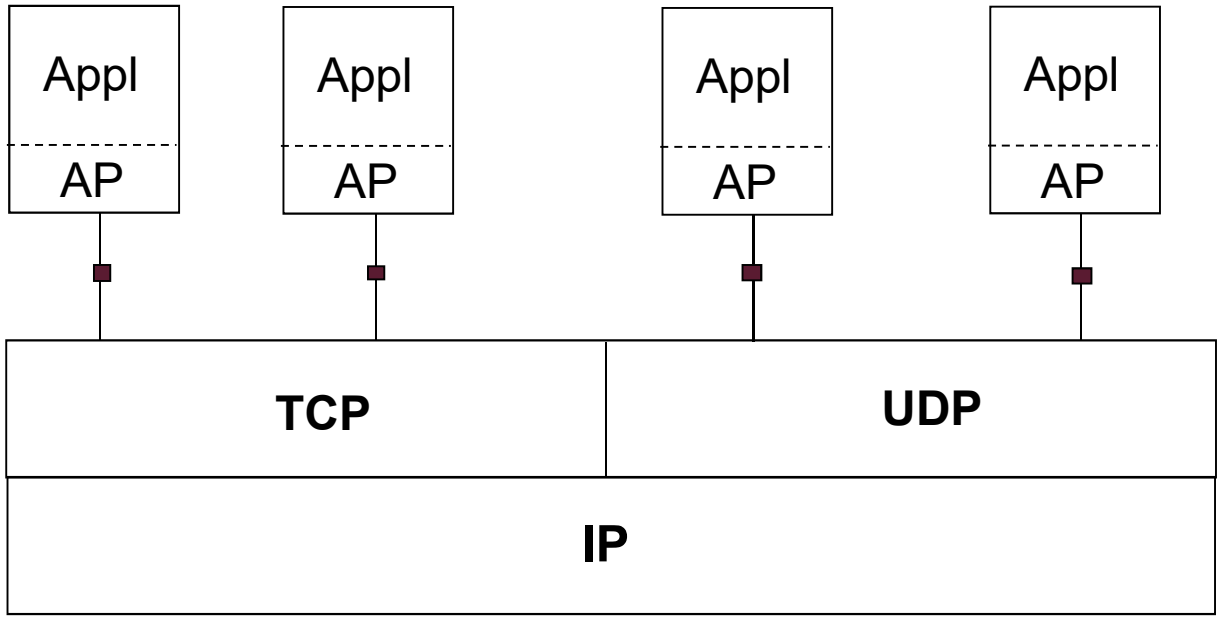
- Core routers
- Edge routers – Points of Presence (PoPs)



Host and Application Addresses

- In IPv4 (i.e. the vast majority of the currently deployed IP systems), a host address is a 32-bit quantity typically displayed as 4 8-bit numbers in “dot notation”
  - For example, 128.40.42.3 for host dublin.ee.ucl.ac.uk
  - The address 127.0.0.1 is the loopback address which effectively means “the same host”
  - Host names such as dublin.ee.ucl.ac.uk are mapped to network addresses through the Domain Name System (DNS)
- Given that typically many applications run on a single host, distinct port numbers identify them
  - Ports below 1024 are used for well-known applications
  - Given that there exist two transport protocols, the same port number in TCP and UDP can be used by a different application
  - So an application address is in fact the combination of
    {IP address, transport protocol, port number }
    e.g. {128.40.42.3, TCP, 80 } identifies the Web server on host dublin

Applications and Ports



AP: Application-level Protocol

■ : Port

The Socket API

- The Socket Application Program Interface was pioneered in the early 1980's as part of the Berkeley Unix release and was implemented in the C programming language
  - This is an API in the operating system kernel / user space interface that allows applications to use TCP and UDP functionality (applications use implicitly IP through TCP or UDP)
  - It allowed Internet applications to be developed for the Unix environment and was fundamental for Internet evolution and success
- In the early 1990's it was ported to the Windows environment, the port known as Winsock
  - It has been fundamental for the Internet capabilities of Windows PCs, whose number has increased exponentially over the years

Java Sockets

- The Socket API is available in Java through the java.net package
  - A key class is InetAddress that models an IP address
- TCP sockets are known as “stream sockets” and UDP sockets as “datagram sockets”
  - Key classes for stream sockets are ServerSocket and Socket
  - Key classes for datagram sockets are DatagramSocket and DatagramPacket
- We will examine how to write server and client applications that use TCP through stream sockets

Server Using Stream Sockets - 1

- Establishing a simple stream socket server in Java entails the following steps
- 1 - Create a server socket object:
  - ServerSocket serverSocket = new ServerSocket(portNo);
  - This call throws an IOException if unsuccessful e.g. another program is listening on that port number
  - This calls “binds” the server program to that port - only one program at a time can be bound to a port
- 2 – Do a blocking listen to the established server socket using the ServerSocket accept method until a connection is established (object of class Socket):
  - Socket connection = serverSocket.accept();
  - This call will return when a connection request arrives and the connection is established with the client program
  - It throws an IOException if unsuccessful

Server Using Stream Sockets - 2

- 3 – Get the input and output stream objects that enable the server to receive and send data using the Socket getInputStream & getOutputStream methods
  - ObjectInputStream input = connection.getInputStream();
  - ObjectOutputStream output = connection.getOutputStream();
  - The BufferedReader class can be used as input stream for efficiency (with the help of the InputStreamReader class) and the PrintWriter as formatted output stream:
  - BufferedReader input = new BufferedReader (new InputStreamReader(connection.getInputStream());
  - PrintWriter output = new PrintWriter (connection.getOutputStream(), true);
  - The latter two are better for **text** exchanges, the *true* value in the last method enables automatic line flushing

Server Using Stream Sockets - 3

- 4 – Communicate with the client by receiving and sending data using the input and output stream objects
  - Input can be read using e.g. the BufferedReader methods read – reads a single char, readLine – reads a line, or read with parameters – reads an arbitrary no of chars
  - These methods block if there is no (enough) data to read and they throw an IOException if unsuccessful
  - Output can be written using e.g. the PrintWriter methods print, println or write – see online documentation
- 5 – Close the connection and the associated input and output streams when finished with it; close also the server socket before exiting the program
  - input.close(); output.close(); connection.close();
  - serverSocket.close();

Sockets and Blocking Calls

- The ServerSocket accept method blocks until a connection request arrives and is accepted
  - This means that a multi-threaded server is required if we need to deal with an existing connection while still listening for new ones
  - In this case the main thread could listen for new connections and spawn a new thread to deal with each newly established connection
- When reading data from a socket using the BufferedReader read or readLine methods, these block until input appears
  - Multi-threading is again required in a program that has to read input simultaneously from more than one endpoints e.g. sockets, the standard input, etc.

Client Using Stream Sockets - 1

- Establishing a simple stream socket client in Java entails the following steps
- 1 – Establish a connection to the server by creating a socket object:
  - Socket connection = new Socket(serverIPAddr, portNo);
  - This call throws an IOException if unsuccessful
    e.g. it cannot connect to the specified address and port
  - If the host is known by name, the InetAddress static getByName method can be used to return the IP address:
  - InetAddress serverIPAddr = InetAddress.getByName (serverHostName);
  - This call throws an UnknownHostException if unsuccessful

Client Using Stream Sockets - 2

- 2 – Get the input and output stream objects that enable the server to receive and send data using the Socket getInputStream & getOutputStream methods
- 3 - Communicate with the server by sending and receiving data using the input and output stream objects
- 4 - Close the connection and the associated input and output streams when finished with it

- Steps 2, 3 and 4 use exactly the same calls as steps 3, 4 and 5 for the server case

Simple Echo Application

- We will describe next a simple echo application consisting of two programs:
  - EchoServer and EchoClient
- The EchoServer is started first waiting for the EchoClient to connect; it then simply echoes back any string messages received from the EchoClient
- The EchoClient receives string messages the user types on the terminal, sends them to the EchoServer, receives them back and prints them out
  - The message “quit” causes both programs to terminate
- The two programs may run on different computers

Echo Server - 1

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

public class EchoServer {
    private ServerSocket serverSocket;
    private Socket connection;
    private BufferedReader input;
    private PrintWriter output;

    public EchoServer () throws IOException {
        final int portNumber = 4433;

        try {
            serverSocket = new ServerSocket(portNumber);
        } catch (IOException ioexc) {
            System.err.println("cannot listen on port: " + portNumber);
            System.exit(1);
        }
        System.out.println("listening on port: " + portNumber);

// continues on the next page
```

Echo Server - 2

```
// continues from the previous page

        try {
            connection = serverSocket.accept();
        } catch (IOException ioexc) {
            System.err.println("accept failed on socket for port: "
                                + portNumber);
            System.exit(1);
        }
        // use Socket getInetAddress and InetAddress getHostName
        // to print out the name of the host we received a connection from
        System.out.println("connection from: " +
                        (connection.getInetAddress()).getHostName());

        // get read/write streams to send and receive data
        output = new PrintWriter(connection.getOutputStream(), true);
        input = new BufferedReader(
                    new InputStreamReader(connection.getInputStream()));
    }  // end constructor EchoServer

// continues on the next page
```

Echo Server - 3

```
// continues from the previous page

    public void run () throws IOException
    {
        String inputLine;
        while ((inputLine = input.readLine()) != null) {
            if (inputLine.equalsIgnoreCase("quit"))
                break;
            System.out.println("received: " + inputLine);
            output.println(inputLine);  // echo it back
        }

        System.out.println("received quit, exiting");
        input.close();
        output.close();
        connection.close();
        serverSocket.close();
    } // end method run

// continues on the next page
```

Echo Server - 4

```
// continues from the previous page

    // the main method creates an echo server object that accepts
    // a connection and prepares the input/ouput streams;
    // it then calls its run method which reads from the input and
    // copies to the output stream until it receives “quit” and cleans up

    public static void main (String args[]) {
        try {
            EchoServer echoServer = new EchoServer();
            echoServer.run();
        } catch (IOException ioexc) {
            System.err.println(“IOException in main()”);
        }
    } // end method main

} // end class EchoServer
```

Echo Client - 1

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

public class EchoClient {

    private Socket connection;
    private BufferedReader input;
    private PrintWriter output;

    public EchoClient (String host)
    {
        final int portNumber = 4433;

        try {
            connection = new Socket(InetAddress.getByName(host),
                                                                portNumber);
            output = new PrintWriter(connection.getOutputStream(), true);
            input = new BufferedReader(
                    new InputStreamReader(connection.getInputStream()));
        }

// continues on the next page
```

Echo Client - 2

```
// continues from the previous page

        catch (UnknownHostException uhe) {
            System.err.println("don't know about host " + host);
            System.exit(1);
        } catch (IOException ioexc) {
            System.err.println("cannot connect to " + host);
            System.exit(1);
        }
        System.out.println("connected successfully to: " + host);
    } // end constructor EchoClient

// continues on the next page
```

Echo Client - 3

```
// continues from the previous page

    public void run () throws IOException
    {
        System.out.println("starts to send/receive messages");
        // System.in is the standard input stream
        BufferedReader stdIn = new BufferedReader(
                               new InputStreamReader(System.in));
        String userInput;

        while ((userInput = stdIn.readLine()) != null) {
            output.println(userInput); // send it to EchoServer
            if (userInput.equalsIgnoreCase("quit"))
                break;
            System.out.println("received back: " + input.readLine());
        }

        output.close();
        input.close();
        stdIn.close();
        connection.close();
    } // end method run,  continues on the next page
```

Echo Client - 4

```
// continues from the previous page
    // the main method creates an echo client object; the constructor
    // connects to the server and  prepares the input/ouput streams;
    // after that the run method is called which reads from the keyboard,
    // copies to the output and listens to the input stream for the response;

    public static void main (String args[]) {
        EchoClient echoClient;

        // if the host name is not passed as an argument, the local loopback
        // address is used which effectively means “the same host”;
        if (args.length == 0)
            echoClient = new EchoClient("127.0.0.1");
        else
            echoClient = new EchoClient(args[0]);
        try {
            echoClient.run();
        } catch (IOException ioexc) {
            System.err.println(“IOException in EchoClient run()”);
        }
    } // end method main

} // end class EchoClient
```

Summary

- Java socket programming provides support for developing application level protocols and applications that use TCP and UDP transport services
- TCP sockets are known as stream sockets and are realised through the ServerSocket and Socket classes
- Listening to a socket is done through the blocking call serverSocket accept, so multithreading is required for programs that need to listen on multiple sockets
- We have examined the stream socket API in detail through a simple echo application