

TCP/IP NETWORKING

<https://docs.oracle.com/javase/tutorial/essential/io/index.html>

Extending I/O Streams

- An *I/O Stream* represents an input source or an output destination.
- A stream can represent many different kinds of sources and destinations, including disk files, devices, other programs, and memory arrays.
- Streams support many different kinds of data, including simple bytes, primitive data types, localized characters, and objects.
- Some streams simply pass on data; others manipulate and transform the data in useful ways.

Overview

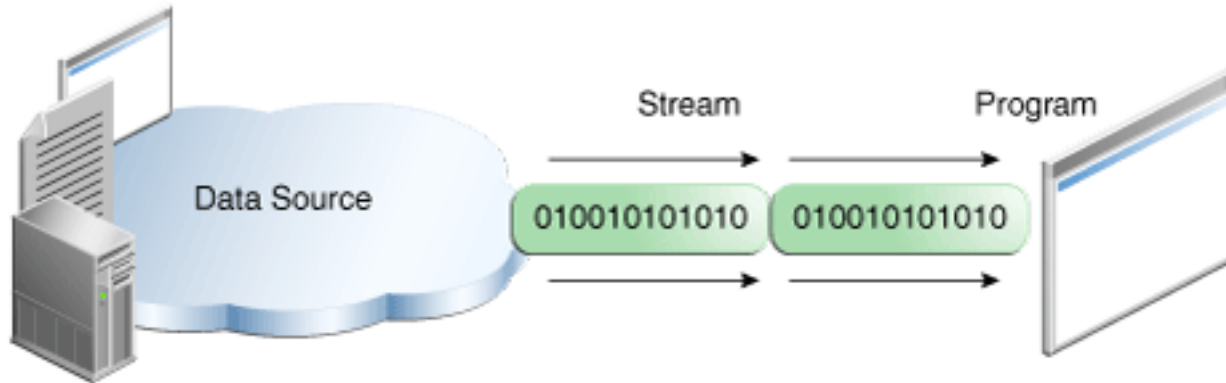
- **Byte Streams** handle I/O of raw binary data.
- **Character Streams** handle I/O of character data, automatically handling translation to and from the local character set.
- **Buffered Streams** optimize input and output by reducing the number of calls to the native API.
- ..

Overview ^{con't}

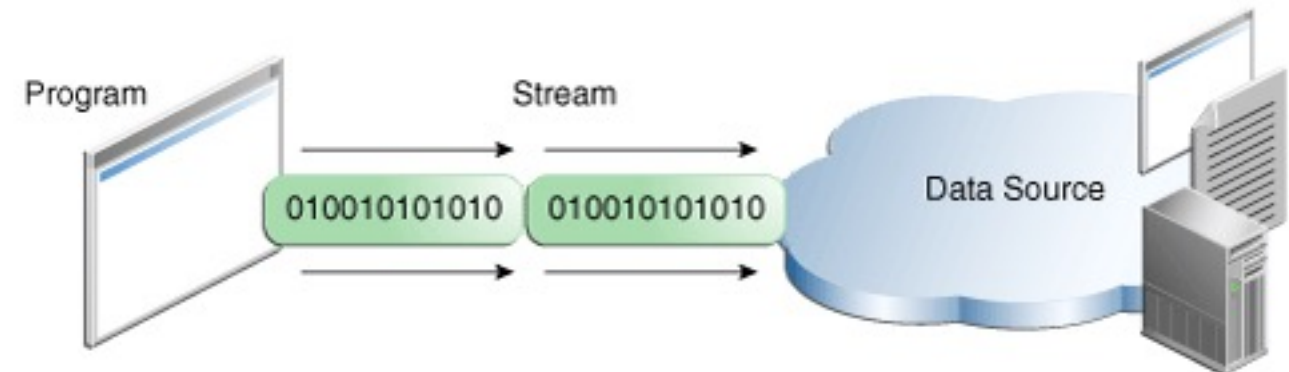
- **Scanning and Formatting** allows a program to read and write formatted text.
- **I/O from the Command Line** describes the Standard Streams and the Console object.
- **Data Streams** handle binary I/O of primitive data type and String values.
- **Object Streams** handle binary I/O of objects.

I/O Streams

- A stream is a sequence of data. A program uses an *input stream* to read data from a source, one item at a time:



- A program uses an *output stream* to write data to a destination, one item at a time:

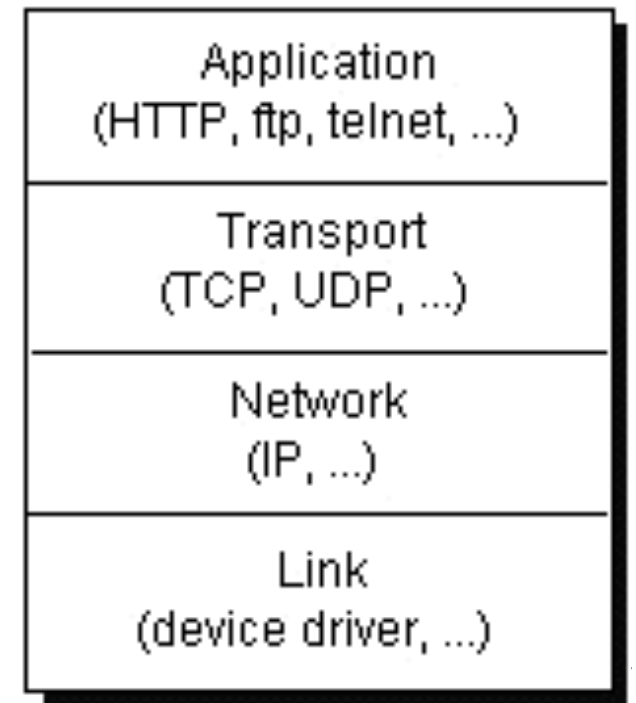


Networking

- Computers running on the Internet communicate to each other using either
 - *the Transmission Control Protocol (TCP)*
 - *or the User Datagram Protocol (UDP)*

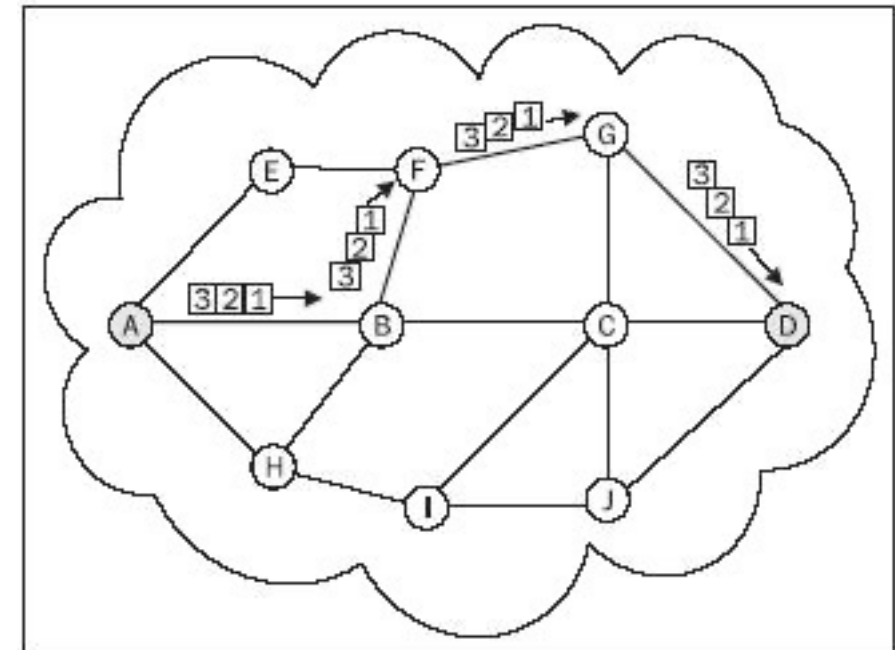
Java programs that communicate over the network are at the application layer.

No need to concern with the TCP & UDP layers



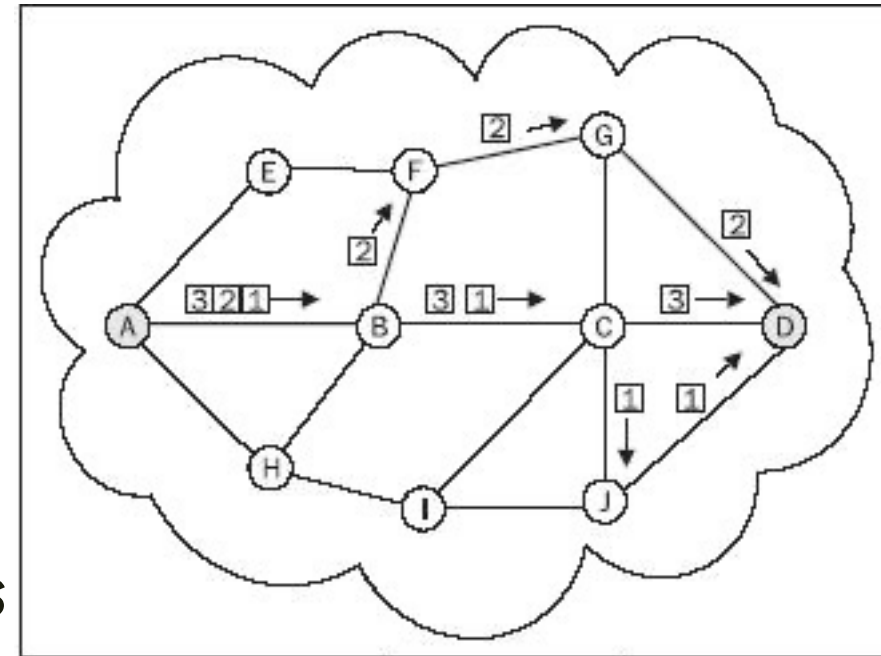
TCP

- TCP – Transmission Control Protocol
 - *reliable delivery/communication between two computers*
 - *establish a connection*
 - Virtual Circuit
 - (maintained over time, ping, ping)
 - *A base for HTTP, FTP*



UDP

- UDP – User datagram protocol
 - *no guarantee on delivery/arrival of packets*
 - *not connection-based*
 - *independent packets*
 - (grams – international unit for weight)
 - *possible to have multiple receivers - multicast*

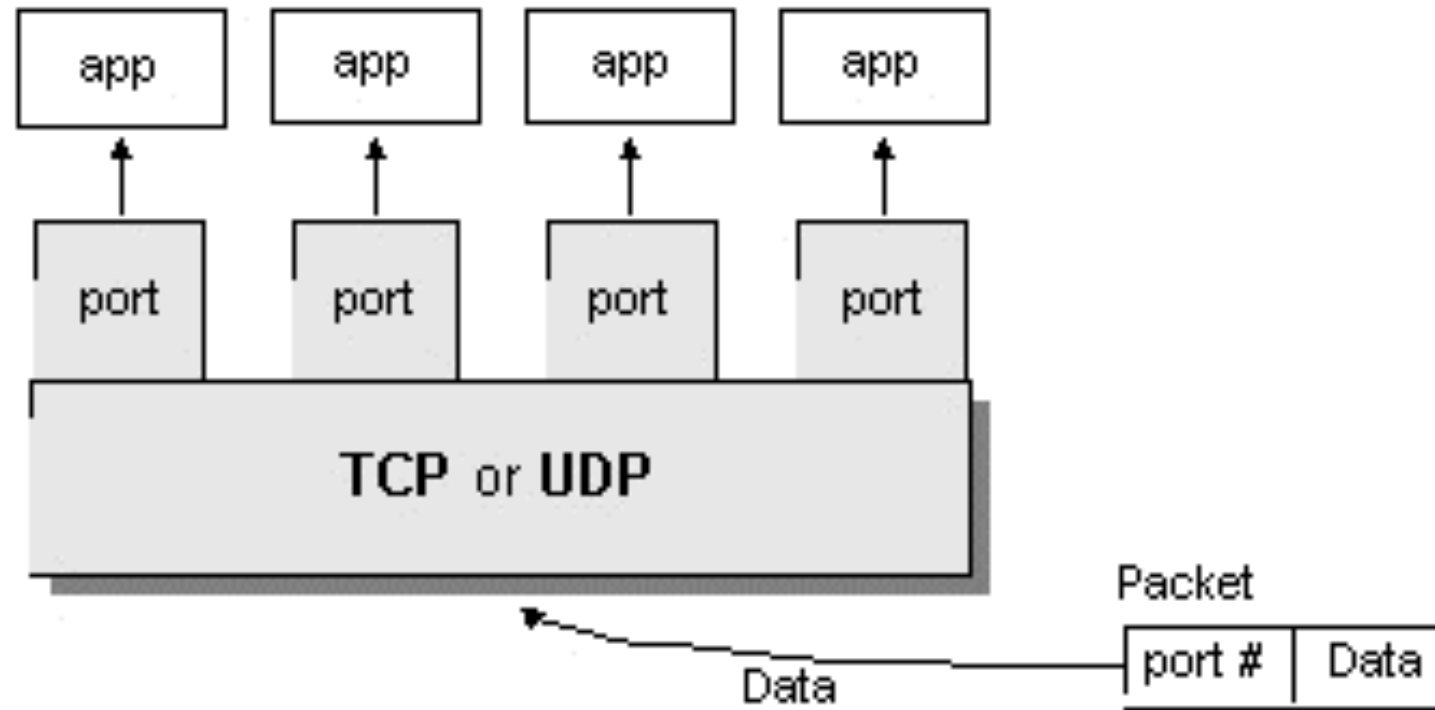


Communication

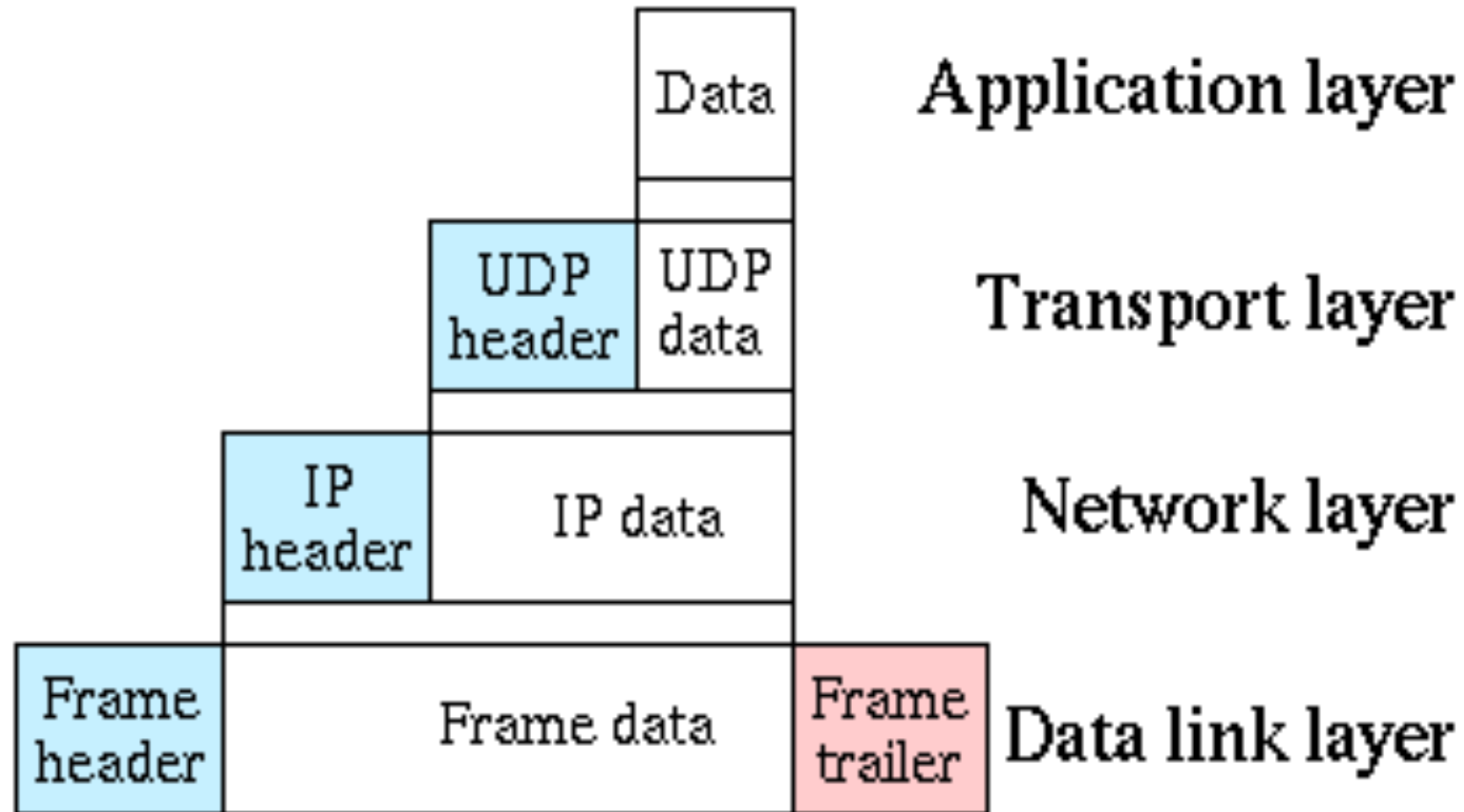
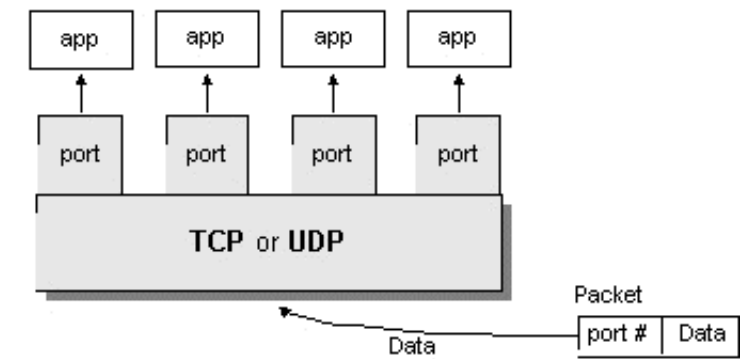
- Two peers
 - *e.g. two computers or two programs on the same computer*
- A computer has assigned **IP address**
- A computer hosts multiple **programs**
- Each program has assigned a **port** in given computer
- The pair of {**IP:port**} identifies a running program in the network
- E.g. Webserver has port 80, ssh has port 22
- Port is a 16-bit number. || IP is a 32-bit address
 - *127.0.0.1:8888.*
 - *(A an IP address can have a host name e.g. tomas or localhost; tomas:4040)*

Communication

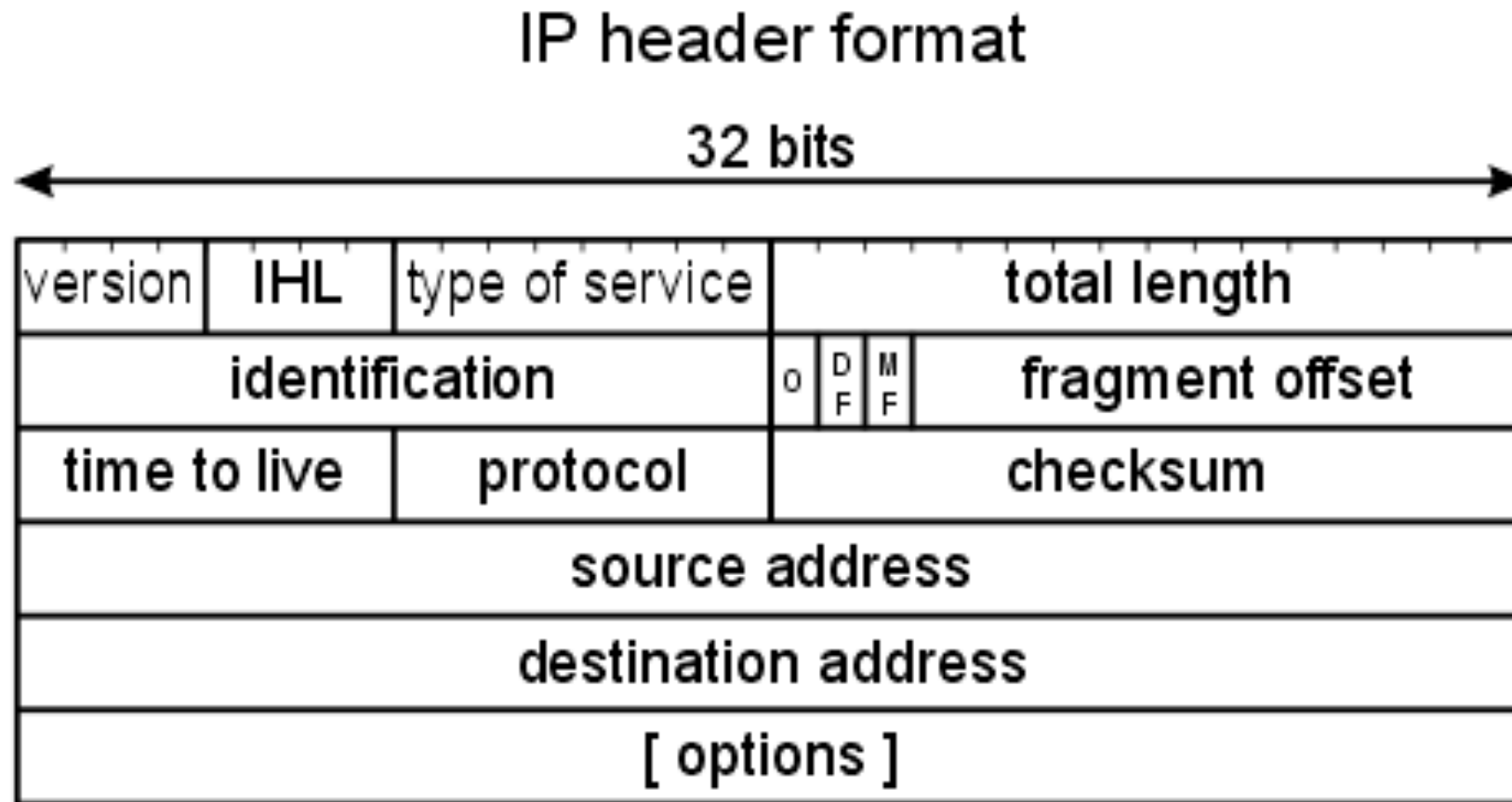
- How to find a destination (multiplex)?



Packets

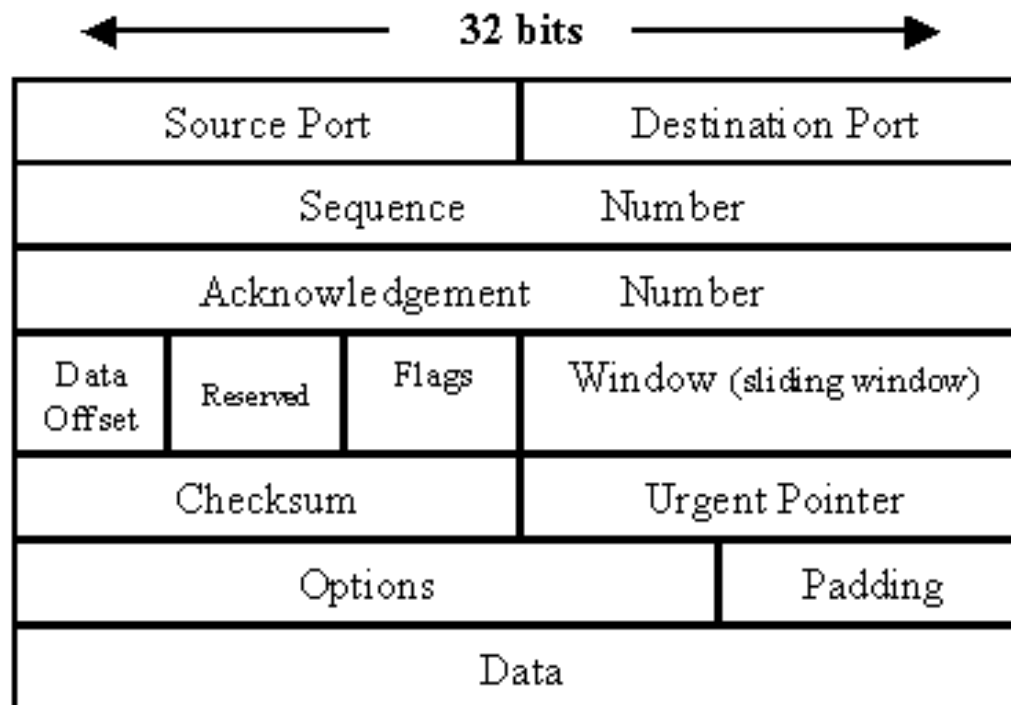


Packet IP header

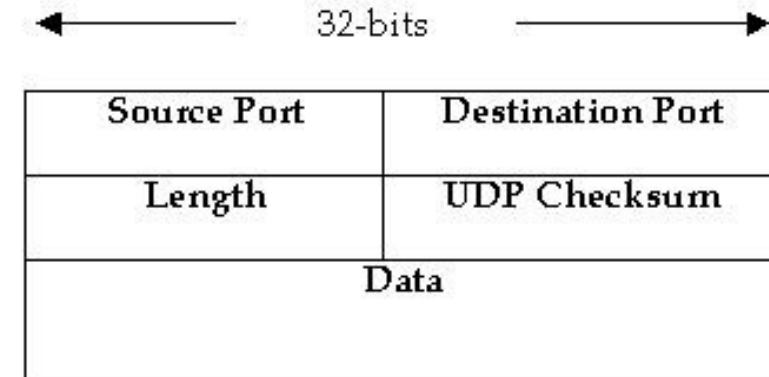


Packet TCP/UDP header

■ TCP



UDP



URL

- *Uniform Resource Locator* and is a reference (an address) to a resource on the Internet.
- URL `http://example.com`
 - *Protocol identifier: http*
 - *Resource name: example.com*
- The resource name is the complete address to the resource.
 - *Host Name* The name of the machine on which the resource lives.
 - *Filename* The pathname to the file on the machine.
 - *Port Number* The port number to which to connect (typically optional and defaults to 80).
 - *Reference* ~ anchor within a resource ~ identifies a specific location within a file (optional).
 - `tomas.cz:8080/foo/bar/file.html#intro`

URL

- Java
 - *URL myURL = new URL("http://example.com/pages/page1.html");*
- URL can have special characters
 - *URI uri = new URI("http", "example.com", "/hello world/", "");*

```
URL aURL = new URL(
    "http://example.com:80/docs/books/tutorial"
    +
    "/index.html?name=networking#DOWNLOADING");

out.println("protocol = " + aURL.getProtocol());
out.println("authority = " + aURL.getAuthority());
out.println("host = " + aURL.getHost());
out.println("port = " + aURL.getPort());
out.println("path = " + aURL.getPath());
out.println("query = " + aURL.getQuery());
out.println("filename = " + aURL.getFile());
out.println("ref = " + aURL.getRef());
```

```
protocol = http
authority = example.com:80
host = example.com
port = 80
path = /docs/../index.html
query = name=networking
filename = /doc../index.html
ref = ?name=networking#DOWNLOADING
```

URL

- Go and fetch via `openStream()` {internally `openConnection().getInputStream()`}

```
public class URLConnectionReader {  
    public static void main(String[] args) throws Exception {  
        URL url= new URL("http://www.sample.com/");  
        URLConnection urlConn = url.openConnection();  
        BufferedReader in = new BufferedReader(  
            new InputStreamReader(  
                urlConn.getInputStream()));  
  
        String inputLine;  
        while ((inputLine = in.readLine()) != null) {  
            System.out.println(inputLine);  
        }  
        in.close();  
    }  
}
```


Socket



Client

Tom's Office

Room 360.04



Server

GreenEnergy Provider

electricity meter ILN3334624

Socket



Client

IP 147.0.0.1

Port 16667



Server

IP 156.21.22.123

Port 18800

ServerSocket

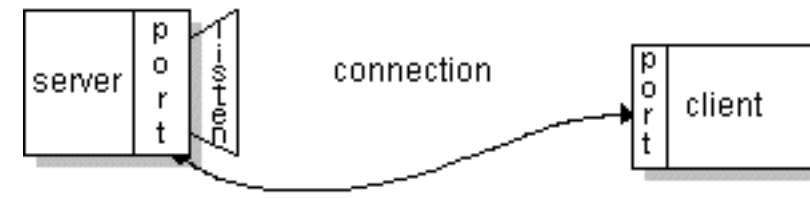
Server runs on a specific computer with socket bound to a specific port number.

The server just waits, listening to the socket for a client connection request.

The same as when you sell ice-cream waiting for customer to serve

ServerSocket

(2 sockets TYPES)



- The server accepts the connection.
- The server initiates a **new socket** bound to the client's port and IP address
 - *(by accepting the connection).*
- It needs a new socket, to continue to listen to the original socket for other new connection requests.

```
ServerSocket serverSocket = new ServerSocket(portNumber);  
Socket clientSocket = serverSocket.accept();
```

Socket - (step 1) On the client-side:

- The client knows the hostname
 - *Knows where the server is running and the port number on which the server is listening.*
- To make a connection request, the client tries to contact the server and port.
- The client posts its local port number and IP address with the request.
 - *This is usually assigned by the system.*



Socket - (step 2)

- On the client side, a socket is successfully created and the client can use the socket to communicate with the server.
- *Socket has associated input stream and output stream to the “other side”*

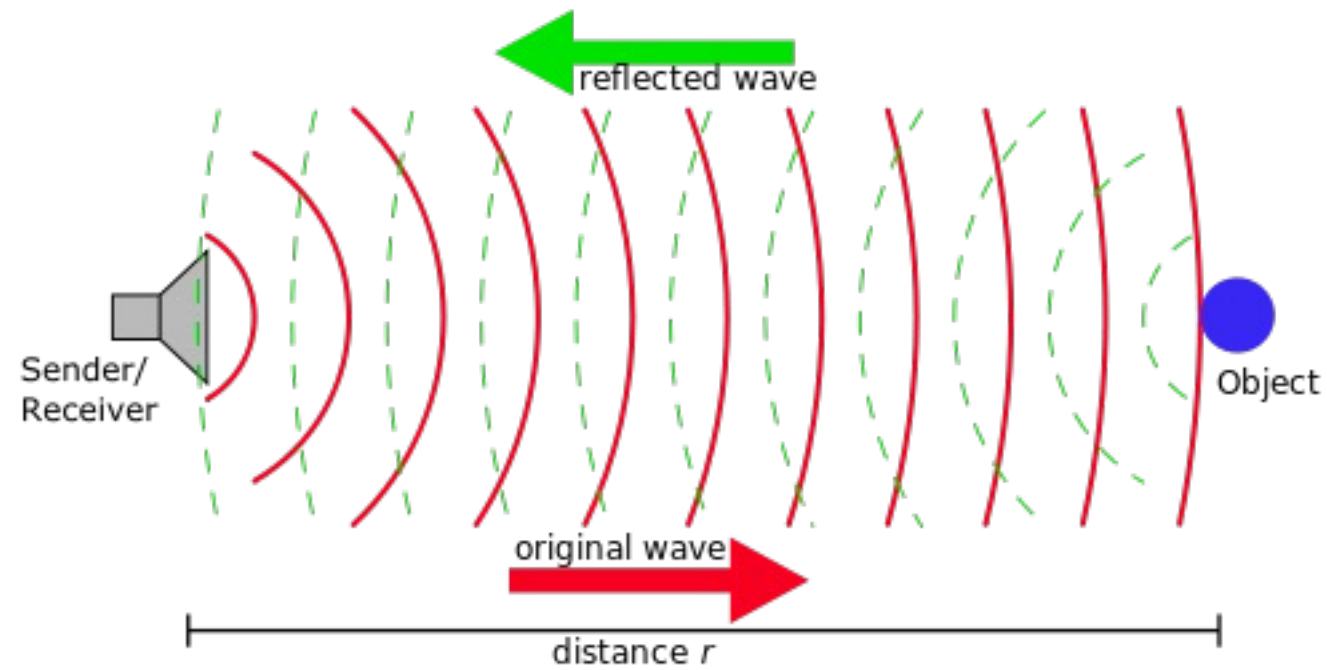
```
clientSocket.getOutputStream()
```

```
clientSocket.getInputStream()
```

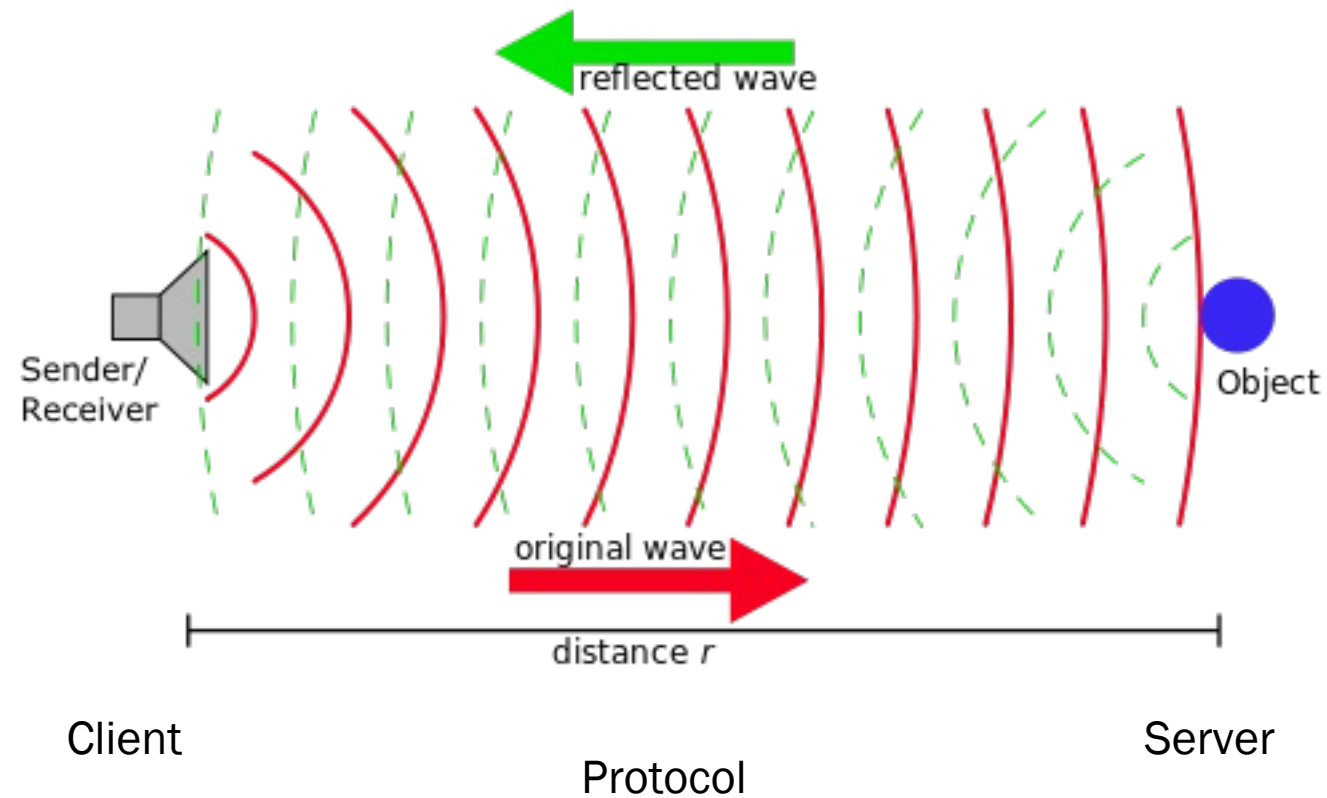
Socket

- A *socket* is one **endpoint** of a two-way communication link between two programs on the network.
 - A socket is bound to a port number so that the TCP layer can identify the application that data is meant to be sent to.
-
- An **endpoint** is a combination of an IP address and a port number.
 - Every TCP connection can be uniquely identified by its two endpoints.
 - That way you can have multiple connections between your host and the server.

Example: Echo



Example: Echo



Example : Echo

- Where do we start?

Example : Echo

- Where do we start?
 1. Protocol
 - *How to behave, what to wear*
 2. Server
 - *Pick a port (ServerSocket[ThisIP:port])*
 3. Client
- Know your server (Socket[IP:port])

```

public class EchoServer {
    public static void main(String[] args) throws IOException {
        int portNumber = Integer.parseInt(args[0]);
        try (
            ServerSocket serverSocket = new ServerSocket(portNumber);
            Socket clientSocket = serverSocket.accept(); // blocking
            PrintWriter sockOut =
                new PrintWriter(clientSocket.getOutputStream(), true);
            BufferedReader sockIn = new BufferedReader(
                new InputStreamReader(clientSocket.getInputStream()));
        ) {
            String inputLine;
            while ((inputLine = sockIn.readLine()) != null) {
                sockOut.println(inputLine);
            }
        } catch (IOException e) {
            System.out.println(e.getMessage());
        }
    }
}

```

```

public class EchoClient {
    public static void main(String[] args) throws IOException {
        String hostName = args[0];
        int portNumber = Integer.parseInt(args[1]);
        try (
            Socket echoSocket = new Socket(hostName, portNumber);
            PrintWriter sockOut = new PrintWriter(echoSocket.getOutputStream(), true);
            BufferedReader sockIn = new BufferedReader(
                new InputStreamReader(echoSocket.getInputStream()));
            BufferedReader stdIn = new BufferedReader(
                new InputStreamReader(System.in))
        ) {
            String userInput;
            while ((userInput = stdIn.readLine()) != null) {
                sockOut.println(userInput);
                System.out.println("echo: " + sockIn.readLine());
            }
        } catch (IOException e) {
            System.err.println(e.getMessage());
        }
    }
}

```

Example : Run

■ Two consoles

```
$ java EchoServer 8888
```

```
$ java EchoClient localhost 8888
```

```
##
```

1. Open a socket.
2. Open an input stream and output stream to the socket.
3. Read from and write to the stream according to the server's protocol.
4. Close the streams.
5. Close the socket.

Example 2 Knock knock protocol

Server: "Knock knock!"

Client: "Who's there?"

Server: "Dexter."

Client: "Dexter who?"

Server: "Dexter halls with boughs of holly."

Client: "Groan."

..

What is a protocol?

- the official procedure or system of rules governing affairs of state or diplomatic occasions.
- "protocol forbids the prince from making any public statement in his defense"

We do not have a diplomatic occasion here – agreed communication – who goes first, what, etc.

How many states do we have here?

Important for the protocol.

Who initiates the connection?

```
public class KnockKnockClient {
    public static void main(String[] args) throws IOException {
        if (args.length != 2) {
            System.err.println(
                "Usage: java EchoClient <host name> <port number>");
            System.exit(1);
        }
        String hostName = args[0];
        int portNumber = Integer.parseInt(args[1]);

        try {
            .. // next slide
        } catch (UnknownHostException e) {
            System.err.println("Don't know about host " + hostName);
            System.exit(1);
        } catch (IOException e) {
            System.err.println("Couldn't get I/O for the connection to " +
                               hostName);
            System.exit(1);
        }
    }
}
```



```
try ( Socket kkSocket = new Socket(hostname, portNumber);
      PrintWriter out = new PrintWriter(kkSocket.getOutputStream(), true);
      BufferedReader in = new BufferedReader(
          new InputStreamReader(kkSocket.getInputStream()))
    ) {
        BufferedReader stdIn =
            new BufferedReader(new InputStreamReader(System.in));
        String fromServer, fromUser;

        while ((fromServer = in.readLine()) != null) {
            System.out.println("Server: " + fromServer);
            if (fromServer.equals("Bye.")) { break; }

            fromUser = stdIn.readLine();
            if (fromUser != null) {
                System.out.println("Client: " + fromUser);
                out.println(fromUser);
            }
        }
    } catch (UnknownHostException e) {
```

```

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

        if (args.length != 1) {
            System.err.println("Usage: java KnockKnockServer <port number>");
            System.exit(1);
        }

        int portNumber = Integer.parseInt(args[0]);

        try(..) {

            .. // next slide

        } catch (IOException e) {
            System.out.println("Exception caught when trying to listen on port "
                               + portNumber + " or listening for a connection");
            System.out.println(e.getMessage());
        }
    }
}

```


```
try (  
    ServerSocket serverSocket = new ServerSocket(portNumber);  
    Socket clientSocket = serverSocket.accept();  
    PrintWriter sockOut =  
        new PrintWriter(clientSocket.getOutputStream(), true);  
    BufferedReader sockIn = new BufferedReader(  
        new InputStreamReader(clientSocket.getInputStream()));  
    ) {  
        String inputLine, outputLine;  
        // Initiate conversation with client  
        KnockKnockProtocol kkp = new KnockKnockProtocol();  
        outputLine = kkp.processInput(null);  
        sockOut.println(outputLine);  
  
        while ((inputLine = sockIn.readLine()) != null) {  
            outputLine = kkp.processInput(inputLine);  
            sockOut.println(outputLine);  
            if (outputLine.equals("Bye.")) { break; }  
        }  
    } catch (IOException e) { ..
```

```
public class KnockKnockProtocol {
    private static final int WAITING = 0;
    private static final int SENTKNOCKKNOCK = 1;
    private static final int SENTCLUE = 2;
    private static final int ANOTHER = 3;

    private static final int NUMJOKES = 5;

    private int state = WAITING;
    private int currentJoke = 0;

    private String[] clues = {"Turnip", "Little Old Lady", "Atch", "Who", "Who"};
    private String[] answers = {"Turnip the heat, it's cold in here!",
                                "I didn't know you could yodel!",
                                "Bless you!",
                                "Is there an owl in here?",
                                "Is there an echo in here?" };
}
```



```
// protocol!
public String processInput(String theInput) {
    String theOutput = null;
    ..

    // Waiting
    // Sent KnockKnock
    // Sent Clue
    // wants Another
    // All at the next slide

    ..
    return theOutput;
}
```

Example Knock Protocol

```
if (state == WAITING) {                                // Waiting
    theOutput = "Knock! Knock!";
    state = SENTKNOCKKNOCK;
} else if (state == SENTKNOCKKNOCK) {                  // Sent Knock Knock
    if (theInput.equalsIgnoreCase("Who's there?")) {
        theOutput = clues[currentJoke];
        state = SENTCLUE;
    } else {
        theOutput = "You're supposed to say \"Who's there?\"! \" +
            "Try again. Knock! Knock!";
    }
} else if (state == SENTCLUE) {                         // Sent clue
    if (theInput.equalsIgnoreCase(clues[currentJoke] + " who?")) {
        theOutput = answers[currentJoke] + " Want another? (y/n)";
        state = ANOTHER;
    } else {
        theOutput = "You're supposed to say \"" + clues[currentJoke] +
            " who?\" + \"! Try again. Knock! Knock!";
        state = SENTKNOCKKNOCK;
    }
}
```

Example Knock Protocol

```
// Another?
    } else if (state == ANOTHER) {
        if (theInput.equalsIgnoreCase("y")) {
            theOutput = "Knock! Knock!";
            if (currentJoke == (NUMJOKES - 1))
                currentJoke = 0;
            else
                currentJoke++;
            state = SENTKNOCKKNOCK;
        } else {
            theOutput = "Bye.";
            state = WAITING;
        }
    }
}
```