



PGT: Sockets

Mohamed Elshaikh

Faculty of Electronics Engineering Technology – UniMAP (FTKEN-UniMAP)

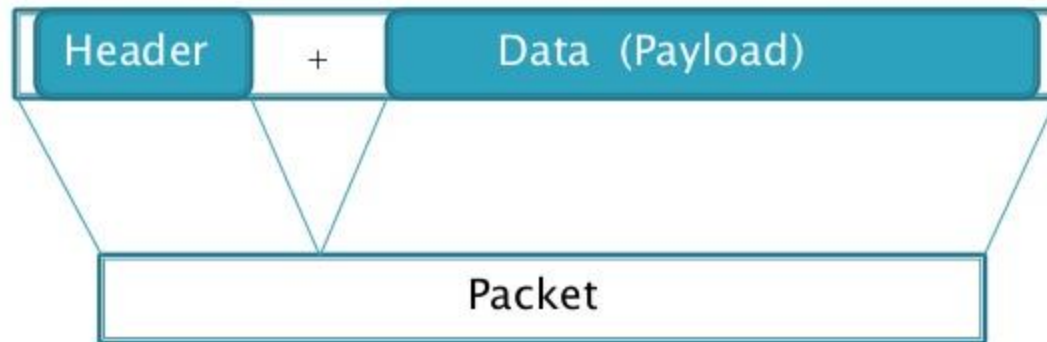


Objectives

- ◎ **CREATE** and **SHOW** how to write network clients that interact with TCP servers, how to use the socket server and to protect client server communications using the Secure Sockets Layer (SSL).
 - ❖ Sockets for Clients
 - ❖ Sockets for Servers
 - ❖ Secure Sockets

Introduction

- ⦿ Data is transmitted across the Internet in packets of finite size called *datagrams*.
- ⦿ *Each* datagram contains a *header* and a *payload*.



- ⦿ However, because datagrams have a finite length, it's often necessary to split the data across multiple packets and reassemble it at the destination.

Abstraction

- ◎ Datagrams are mostly hidden from the Java programmer.
- ◎ The host's native networking software transparently splits data into packets on the sending end of a connection, and then reassembles packets on the receiving end.
- ◎ Instead, the Java programmer is presented with a higher level abstraction called a *socket*.

Sockets

- ◎ A socket is a **reliable connection** for the transmission of data between two hosts.
- ◎ Sockets isolate programmers from the details of packet encodings, lost and retransmitted packets, and packets that arrive out of order.
- ◎ There are limits. Sockets are more likely to throw `IOExceptions` than files, for example.

Using Sockets

◎ A socket is a connection between two hosts. It can perform 7 basic operations:

1. **Connect to a remote machine**

2. **Send data**

3. **Receive data**

4. **Close a connection**

5. **Bind to a port**

6. **Listen for incoming data**

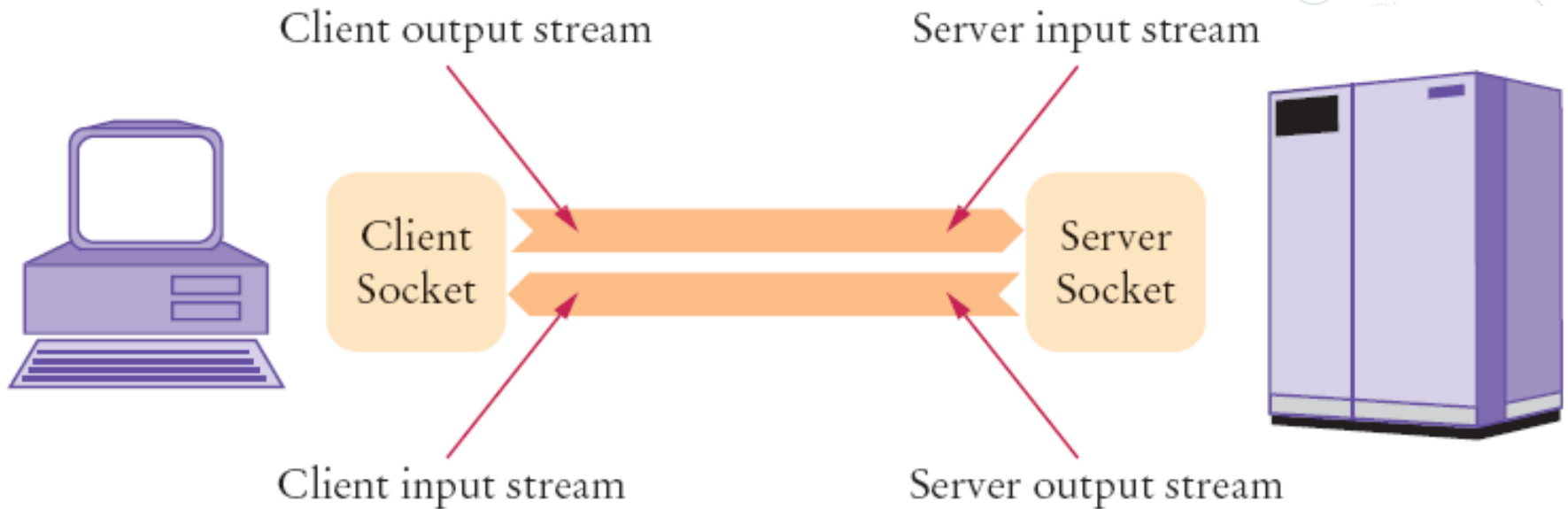
7. **Accept connections from remote machines on the bound port**

By client
(Socket class)

By server
(ServerSocket class)

◎ A socket may not be connected to more than one host at a time.

Client and Server Sockets



- ❖ Once the connection is established, the local and remote hosts get input and output streams from the socket and use those streams to send data to each other.
- ❖ This connection is *full-duplex*. Both hosts can send and receive data *simultaneously*.

The `java.net.Socket` class

- ◎ The `java.net.Socket` class allows you to create socket objects that perform 4 fundamental socket operations.
- ◎ You can **connect** to remote machines; you can **send** data; you can **receive** data; you can **close** the connection.
- ◎ Each **Socket** object is associated with exactly one remote host. To connect to a different host, you must create a new **Socket** object.

Constructing a Socket

- ◎ Connection is accomplished through the constructors. There are 4 available constructors in class Socket:

```
public Socket(String host, int port) throws  
UnknownHostException, IOException
```

```
public Socket(InetAddress address, int port) throws  
IOException
```

```
public Socket(String host, int port, InetAddress localAddr,  
int localPort) throws IOException
```

```
public Socket(InetAddress address, int port, InetAddress  
localAddr, int localPort) throws IOException
```

Opening Sockets

- ◎ The `Socket ()` constructors do not just create a `Socket` object. They also **attempt to connect** the underlying socket to the remote server.
- ◎ All the constructors throw an `IOException` if the connection can't be made for any reason.
- ◎ You must at least specify the remote host and port to connect to.
- ◎ The host may be specified as either a string like "unimap.edu.my" or as an `InetAddress` object.
- ◎ The port should be an `int` between 1 and 65535.
- ◎ **Example:**

```
Socket webMetalab = new Socket("unimap.edu.my", 80);
```

Picking an IP address

- ◎ The last two constructors also specify the **host and port you're connecting from**.
- ◎ On a system with multiple IP addresses, like many web servers, this allows you to pick your network interface and IP address.

Port Number

- ◎ The port number field of an IP packet is specified as a 16-bit unsigned integer. This means that valid port numbers range from 1 through 65535. (Port number 0 is reserved and can't be used).
- ◎ Port numbers from 0-1023 are not available for user programs in Java systems.
 - ❖ In practice, port numbers **below 1024** are **reserved for predefined services**, and you should not use them unless communicating with one of those services (such as telnet, SMTP mail, ftp, and so on).
 - ❖ Client ports are allocated by the host OS to something not in use, while server port numbers are specified by the programmer, and are used to identify a particular service.

Example:

A Custom Method to Determine Listening Port

You **cannot** just connect to any port on any host. **The remote host must actually be listening for connections on that port.**

You can use the constructors to determine which ports on a host are listening for connections.

```
public static void scan(InetAddress remote) {
    String hostname = remote.getHostName();
    for (int port = 0; port < 65536; port++) {
        try {
            Socket s = new Socket(remote, port);
            System.out.println("A server is listening on port "
                + port + " of " + hostname);
            s.close();
        } catch (IOException e) {
            // The remote host is not listening on this port
        }
    }
}
```

Choosing a Local Port

- ◎ You can also specify a local port number,
- ◎ Setting the port to **0** tells the system to randomly choose an available port.
- ◎ If you need to know the port you're connecting from, you can always get it with `getLocalPort()`.
- ◎ Example:

```
Socket webMetalab =  
    new Socket("metalab.unc.edu", 80,  
              "calzone.oit.unc.edu", 0);
```

Sending and Receiving Data

- ◎ Data is sent and received with output and input streams.
- ◎ There are methods to get an **input stream** for a socket and an **output stream** for the socket.

```
public InputStream getInputStream() throws IOException  
public OutputStream getOutputStream() throws IOException
```

- ◎ There's also a method to **close a socket**.

```
public synchronized void close() throws IOException
```

Reading Input from a Socket

- ◎ The `getInputStream()` method returns an `InputStream` which reads data from the socket.
- ◎ You can use all the normal methods of the `InputStream` class to read this data.
- ◎ Most of the time you'll chain the input stream to some other input stream or reader object to more easily handle the data.

Example: Read input

The following code fragment connects to the daytime server on port 13 of metalab.unc.edu, and displays the data it sends.

```
try{
    Socket s = new Socket("metalab.unc.edu", 13);
    InputStream is = s.getInputStream();
    InputStreamReader isr = new InputStreamReader(is);
    BufferedReader br = new BufferedReader(isr);
    String theTime = br.readLine();
    System.out.println(theTime);
}catch(IOException e) {
    return(new Date()).toString();
}finally{ //this is not needed in java 7 above
    if(s != null) {
        try {
            s.close();
        } catch (IOException ex) {
            //ignore
        }
    }
}
```

Writing Output to a Socket

- ◎ The `getOutputStream()` method returns an output stream which writes data to the socket.
- ◎ Most of the time you'll chain the raw output stream to some other output stream or writer class to more easily handle the data.

Example: Write Output

```
byte[] b = new byte[128];

try {
    Socket s = new Socket("metalab.unc.edu", 9);
    OutputStream theOutput = s.getOutputStream();
    while (true) {
        int n = theInput.available();
        if (n > b.length) n = b.length;
        int m = theInput.read(b, 0, n);
        if (m == -1) break;
        theOutput.write(b, 0, n);
    }
    s.close();
} catch (IOException e) {}
```

Reading and Writing to a Socket

- ◎ It's unusual to only read from a socket. It's even more unusual to only write to a socket.
- ◎ Most protocols require the client to both read and write.
- ◎ Java places no restrictions on reading and writing to sockets.
- ◎ One thread can read from a socket while another thread writes to the socket at the same time.

Example

```
try {
    URL u = new URL(args[i]);
    if (u.getPort() != -1) port = u.getPort();
    if (!(u.getProtocol().equalsIgnoreCase("http"))) {
        System.err.println("I only understand http.");
    }
    Socket s = new Socket(u.getHost(), u.getPort());
    OutputStream theOutput = s.getOutputStream();
    PrintWriter pw = new PrintWriter(theOutput, false);
    pw.print("GET " + u.getFile() + " HTTP/1.0\r\n");
    pw.print("Accept: text/plain, text/html, text/*\r\n");
    pw.print("\r\n");
    pw.flush();
    InputStream theInput = s.getInputStream();
    InputStreamReader isr = new InputStreamReader(theInput);
    BufferedReader br = new BufferedReader(isr);
    String theLine;
    while ((theLine = br.readLine()) != null) {
        System.out.println(theLine);
    }
} catch (MalformedURLException e) {
    System.err.println(args[i] + " is not a valid URL");
} catch (IOException e) {
    System.err.println(e);
}
```

Socket Methods

- ◎ Several methods set various socket options. Most of the time the defaults are fine.


```
public void setTcpNoDelay(boolean on) throws SocketException
public boolean getTcpNoDelay() throws SocketException
public void setSoLinger(boolean on, int val) throws SocketException
public int getSoLinger() throws SocketException
public synchronized void setSoTimeout(int timeout) throws SocketException
public synchronized int getSoTimeout() throws SocketException
```

- ◎ These methods to return information about the socket:

```
public InetAddress getInetAddress()
public InetAddress getLocalAddress()
public int getPort()
public int getLocalPort()
```

Servers



- ◎ There are two ends to each connection: the client, that is the host that initiates the connection, and the server, that is the host that responds to the connection.
 - ◎ Clients and servers are connected by sockets.
 - ◎ A server, rather than connecting to a remote host, is a program that waits for other hosts to connect to it.
- 

Server Sockets

- ◎ A server socket **binds to a particular port** on the local machine.
- ◎ Once it has successfully bound to a port, it **listens** for incoming connection attempts.
- ◎ When a server detects a connection attempt, it ***accepts*** the connection. This creates a socket between the client and the server over which the client and the server communicate.

Multiple Clients

- ◎ Multiple clients can connect to the same port on the server at the same time.
- ◎ Incoming data is distinguished by **the port to which it is addressed** and the **client host and port from which it came**.
- ◎ The server can tell for which service (like http or ftp) the data is intended by inspecting the port.
- ◎ It can tell which open socket on that service the data is intended for by looking at the client address and port stored with the data.

Threading

- ◎ No more than one server socket can listen to a particular port at one time.
- ◎ Since a server may need to handle many connections at once, server programs tend to be heavily multi-threaded.
- ◎ Generally the server socket passes off the actual processing of connections to a separate thread.

Queueing

- ◎ Incoming connections are stored in a **queue** until the server can accept them.
- ◎ On most systems the default queue length is between 5 and 50.
- ◎ Once the queue fills up further incoming connections are refused until space in the queue opens up.

The `java.net.ServerSocket` Class

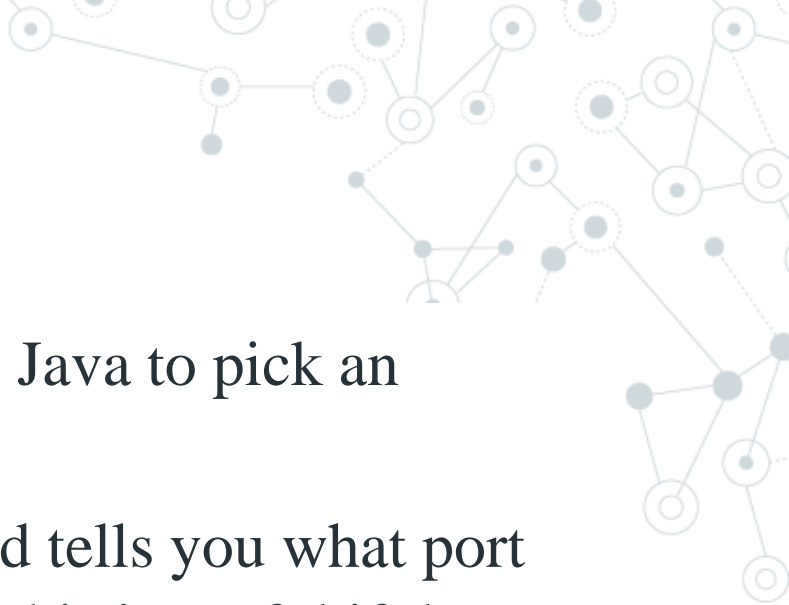

- ◎ The `java.net.ServerSocket` class represents a server socket.
- ◎ A `ServerSocket` object is constructed on a particular local port. Then it calls `accept()` to listen for incoming connections.
- ◎ `accept()` blocks until a connection is detected. Then it returns a `java.net.Socket` object that performs the actual communication with the client.

Constructing Server Sockets

- ◎ Normally you only specify the port you want to listen on, like this:

```
try {
    ServerSocket ss = new ServerSocket(80);
} catch (IOException e) {
    System.err.println(e);
}
```

- ◎ When a `ServerSocket` object is created, it attempts to *bind* to the **port on the local host** given by the port argument.
- ◎ **If another server socket is already listening to the port**, then a `java.net.BindException`, a subclass of `IOException`, is thrown.
- ◎ No more than one process or thread can listen to a particular port at a time. This includes non-Java processes or threads.
- ◎ For example, if there's already an HTTP server running on port 80, you won't be able to bind to port 80.

- 
- ◎ 0 is a special port number. It tells Java to pick an available port.
 - ◎ The `getLocalPort()` method tells you what port the server socket is listening on. This is useful if the client and the server have already established a separate channel of communication over which the chosen port number can be communicated.
 - ◎ The `accept()` and `close()` methods provide the basic functionality of a server socket.
 - ◎ A server socket can't be reopened after it's closed
- 

Reading Data with a ServerSocket

- ◎ ServerSocket objects use their `accept ()` method to connect to a client.
- ◎ There are **no** `getInputStream ()` or `getOutputStream ()` methods for ServerSocket.
- ◎ `accept ()` returns a Socket object, and its `getInputStream ()` and `getOutputStream ()` methods provide streams.

Example: Print the input from client

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

public class SimpleServer{
    public static void main(String[] args){
        ServerSocket ss = null;
        try{
            ss = new ServerSocket(5432);
            Socket s = ss.accept();
            DataInputStream dis = new DataInputStream(s.getInputStream());

            //read the data in byte, cast the data to String
            String str =(String)dis.readUTF();

            //print the input message by client
            System.out.println("message= " + str);

            s.close();
        }catch(IOException e){
            e.printStackTrace();
        }
    }
}
```

Example:

Writing Data to a Client

```
try {
    //register service on port 2345
    ServerSocket ss = new ServerSocket(2345);

    //establish connection
    Socket s = ss.accept();

    //create a PrintWriter object associated with the socket
    PrintWriter pw = new PrintWriter(s.getOutputStream());
    pw.println("Hello There!");
    pw.println("Goodbye now.");

    //close the connection
    s.close();
}
catch (IOException e) {
    System.err.println(e);
}
```

Example:

Writing Data to a Client

```
try {
    ServerSocket ss = new ServerSocket(port);
    while (true) {
        try {
            Socket s = ss.accept();
            PrintWriter pw = new PrintWriter(s.getOutputStream());
            pw.print("Hello " + s.getInetAddress() + " on port "
                    + s.getPort() + "\r\n");
            pw.print("This is " + s.getLocalAddress() + " on port "
                    + s.getLocalPort() + "\r\n");

            pw.flush();
            s.close();
        } catch (IOException e) {}
    }
} catch (IOException e) {
    System.err.println(e);
}
```



Thank You

