UNIT 6
SOCKET FOR CLIENTS

LH - 5HRS

PREPARED BY: **ER. SHARAT MAHARJAN**

NETWORK PROGRAMMING

PRIME COLLEGE, NAYABAZAAR

CONTENTS (LH-5HRS)

6.1 Introduction to Socket

6.2 Using Sockets: Investigating Protocols with telnet, Reading from Servers with Sockets, Writing to Servers with Sockets

6.3 Constructing and connecting Sockets: Basic Constructors, Picking a Local Interface to Connect From, Constructing Without Connecting, Socket Addresses and Proxy Servers

6.4 Getting Information about a Socket: Closed or Connected, toString()

6.5 Setting Socket Options: TCP_NODELAY, SO_LINGER, SO_TIMEOUT, SO_RCVBUF and SO_SNDBUF, SO_KEEPALIVE, SO_OOBINLINE, SO_REUSEADDER and IP_TOS class of Services

6.6 Socket in GUI Applications: Whois and A Network Client Library

6.1 Introduction to Socket

A socket is a **connection between two hosts**. It can perform **seven basic operations**:

- Connect to a remote machine
- Send data
- Receive data
- Close a connection
- Bind to a port
- Listen for incoming data
- Accept connections from remote machines on the bound port

Java's Socket class, which is **used by both clients and servers**, has **methods** that correspond to the **first four of these operations**. The **last three operations are needed only by servers**, which wait for clients to connect to them. Java programs normally use client sockets in the following fashion:

1. The program **creates a new socket with a constructor**.
2. The **socket attempts to connect** to the **remote host**.
3. 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**.
4. When the **transmission** of data is **complete**, one or both sides **close the connection**. Some protocols, such as **HTTP 1.0, require the connection to be closed after each request** is serviced. Others, such as **FTP, allow multiple requests to be processed in a single connection**.

6.2 Using Sockets

Investigating Protocols With Telnet

✓telnet

✓open localhost 25

- This requests a **connection to port 25**, the **SMTP port**, on the **local machine**; SMTP is the protocol used to **transfer email** between servers or between a mail client and a server. If you know the commands to interact with an SMTP server, you can send email without going through a mail program.

Reading from and Writing to Servers with Sockets

```
try{
 //connect a socket to some host machine and port
 Socket socket = new Socket(somehost, someport);
 //connect a BufferedReader to read
 BufferedReader reader = new BufferedReader(new
        InputStreamReader(socket.getInputStream()));
//connect a PrintWriter to write
 PrintWriter writer = new
PrintWriter(socket.getOutputStream, true);//auto flush
}catch(Exception e){System.err.println("Error - "+e);}
```

Half-closed sockets

- The **close() method shuts down both** input and output from the socket. On occasion, you may want to **shut down only half** of the connection, either **input or output**. The **shutdownInput()** and **shutdownOutput()** methods close only half the connection:

public void shutdownInput() throws IOException

public void shutdownOutput() throws IOException

6.3 Constructing and connecting Sockets

- The **java.net.Socket** class is **Java's fundamental class** for performing client-side **TCP operations**.
- The methods of the Socket class **set up and tear down connections** and **set various socket options**.
- The **interface** that the **Socket class provides** to the programmer **is streams** .
- The **actual reading and writing of data** over the socket is accomplished via the **familiar stream classes**.

Basic Constructors

- The **public Socket constructors** are simple. Each lets you **specify the host and the port** you want to connect to.
- **Hosts** may be specified as an **InetAddress or a String** . **Ports** are always specified as **int values from 0 to 65,535**.

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

- This constructor **creates a TCP socket** to the **specified port** on the **specified host** and attempts to connect to the remote host.

For example:

```
try {   Socket toOReilly = new Socket("www.oreilly.com", 80);
        //send and receive data... } catch(UnknownHostException ex) {
System.err.println(ex);} catch (IOException ex) {   System.err.println(ex); }
```

- If the **domain name server cannot resolve the hostname** or is not functioning, the constructor throws an **UnknownHostException** . If the **socket cannot be opened** for some other reason, the constructor throws an **IOException**.

LAB 1: Find out which of the first 1,024 ports seem to be hosting TCP servers on a specified host.(PortScanner)

```
import java.io.*;
import java.net.*;
public class LowPortScanner {
 public static void main(String[] args) {
  for(int i=1;i<1024;i++) {
   try {
    Socket socket = new Socket(”localhost",i);//creates socket object //only if no errors(valid host and open port)-errors(unknown host or close port)
    System.out.println("There is a server on port "+i+" of localhost.");
    } catch (UnknownHostException e) {
     System.out.println(e);
    }catch(IOException e) {
     System.out.println("There is a server off port "+i+" of localhost");
}}}}
```

public Socket(InetAddress host, int port) throws IOException

- Like the previous constructor, this constructor **creates a TCP socket** to the specified port on the specified host and tries to connect. It **differs by using an InetAddress object** to specify the host rather than a hostname.
- **For example:**

```
try{
 InetAddress oreilly = InetAddress.getByName("www.oreilly.com");
Socket oreillySocket = new Socket(oreilly , 80);
// send and receive data... } catch (UnknownHostException ex) {
System.err.println(ex); } catch (IOException ex) {   System.err.println(ex);
}
```

- This technique helps to improve on the efficiency of previous lab.

LAB 2: Find out which of the ports at or above 1,024 seem to be hosting TCP servers.

```
import java.io.*;
import java.net.*;
public class HighPortScanner {
 public static void main(String[] args) {
  try {
   InetAddress address = InetAddress.getByName("localhost");
   for(int i=1024;i<65536;i++) {
    try {
     Socket socket = new Socket(address, i);
     System.out.println("There is a server on port "+i+" of localhost.");
    }catch(IOException e) {
      //System.out.println("There is a server off port "+i+" of localhost");
     }}} catch (UnknownHostException e1) {
          System.out.println("Unknown Host.");
}}}
```

Picking a Local Interface to Connect From

public Socket(String host, int port, InetAddress interface, int localPort) throws IOException, UnknownHostException

- This constructor creates a socket to the specified port on the specified host and tries to connect.
- It **connects to** the host and port specified in the first two arguments.
- It **connects from** the local network interface and port specified by the last two arguments.
- The network interface may be either physical (e.g., a different Ethernet card) or virtual (a multihomed host).
- If 0 is passed for the localPort argument, Java chooses a random available port between 1,024 and 65,535.

public Socket(InetAddress host, int port, InetAddress interface, int localPort) throws IOException

This constructor is identical to the previous one except that the host to connect to is passed as an InetAddress , not a String .

It creates a TCP socket to the specified port on the specified host from the specified interface and local port, and tries to connect.

If it fails, it throws an IOException.

Constructing Without Connecting

protected Socket( )

- The Socket class also has **two** (three in Java 1.5) **constructors that create an object without connecting the socket**. You **use these if you're subclassing Socket** , perhaps **to implement a special kind of socket that encrypts transactions or understands your local proxy server.**

protected Socket(SocketImpl impl)

- This constructor **installs the SocketImpl object** impl **when** it **creates the new Socket object**. The **Socket object is created but is not connected**. This constructor is usually called by subclasses of java.net.Socket . You can pass null to this constructor if you don't need a SocketImpl .

Socket Addresses

- The SocketAddress class represents a connection endpoint.
- The primary purpose of the SocketAddress class is to provide a store for socket connection information such as the IP address and port that can be reused to create new sockets, even after the original socket is disconnected.
- Socket class offers two methods:

public SocketAddress getRemoteSocketAddress()

public SocketAddress getLocalSocketAddress()

- For example, first you might connect to Yahoo! then store its address:

```
Socket socket = new Socket("www.yahoo.com",80);
SocketAddress yahoo = socket.getRemoteSocketAddress();
socket.close();
```

- Later, you could connect to Yahoo! using this address:

```
Socket socket2 = new Socket();
socket2.connect(yahoo);
```

Proxy Servers

- The last constructor **creates an unconnected socket** that **connects through a specified proxy server**:

public Socket(Proxy proxy)

- For example, this code fragment **uses the SOCKS proxy server** at **myproxy.example.com** to **connect to** the host **login.ibiblio.org**:

```
SocketAddress proxyAddress= new
InetSocketAddress("myproxy.example.com",1080);
Proxy proxy = new  Proxy(Proxy.Type.SOCKS, proxyAddress);
Socket socket = new Socket(proxy);
SocketAddress host = new InetSocketAddress("login.ibiblio.org",25);
socket.connect(host);
```

6.4 Getting Information about a Socket

- To the programmer, Socket objects **appear to have several private fields** that are **accessible** through various **getter methods.**
- Actually, sockets have **only one field, a SocketImpl** ; the fields that appear to belong to the Socket actually **reflect native code in the SocketImpl**.
- This way, **socket implementations can be changed without disturbing the program** for example, **to support firewalls and proxy servers**.

1. public InetAddress getInetAddress( )

Given a Socket object, the **getInetAddress( ) method** tells you **which remote host the Socket is connected** to or, if the **connection** is now **closed**, **which host the Socket was connected to** when it was connected.

For example:

```
try {
Socket socket= new Socket(“www.prime.edu.np", 80);
InetAddress host = socket.getInetAddress( );
System.out.println("Connected to remote host " + host); }
catch (UnknownHostException ex) {   System.err.println(ex); }
catch (IOException ex) {   System.err.println(ex); }
```

2. public int getPort( )

The getPort( ) method tells you which port the Socket is (or was or will be) connected to on the remote host.

For example:

```
 try {
Socket socket = new Socket("java.sun.com", 80);
int port = socket.getPort( );
System.out.println("Connected on remote port " + port); }
catch (UnknownHostException ex) {   System.err.println(ex); }
catch (IOException ex) {   System.err.println(ex); }
```

3. public int getLocalPort( )

There are two ends to a connection: the remote host and the local host. To **find the port number for the local end** of a connection, call **getLocalPort( ).**

For example:

```
 try {
Socket socket = new Socket(“www.prime.edu.np", 80, true);
int localPort = socket.getLocalPort( );
System.out.println("Connecting from local port " + localPort);
} catch (UnknownHostException ex) {
System.err.println(ex); } catch (IOException ex) {
System.err.println(ex); }
```

4. public InetAddress getLocalAddress( )

The **getLocalAddress() method tells** you **which network interface a socket is bound to**. You normally use this on a multihomed host, or **one with multiple network interfaces**.

For example:

```
try {
Socket socket = new Socket(hostname, 80);
InetAddress localAddress = socket.getLocalAddress( );
System.out.println("Connecting from local address  “
+localAddress); }catch (UnknownHostException ex) {
System.err.println(ex); } catch (IOException ex) {
System.err.println(ex); }
```

LAB 3: WAP to attempt to open a socket for www.prime.edu.np or any other host, and then uses these four methods to print the remote host, the remote port, the local address, and the local port.

```
import java.io.*;
import java.net.*;
public class Example3 {
public static void main(String[] args) {
try {
Socket socket = new Socket("www.prime.edu.np", 80);
System.out.println("Connected to " + socket.getInetAddress( )+ " on port "  +
socket.getPort( ) + " from port "+ socket.getLocalPort( ) + " of " +
socket.getLocalAddress( ));}
catch (UnknownHostException ex) {System.err.println("I can't find the host." );
}
catch (SocketException ex) {System.err.println("Could not connect to host.");
}
catch (IOException ex) {System.err.println(ex);}}}
```

5. public InputStream getInputStream( ) throws IOException

- The **getInputStream( ) method returns an input stream** that can **read data from the socket into a program**. You usually **chain this InputStream** to a filter stream or reader that **offers more functionality DataInputStream or InputStreamReader**.
- For performance reasons, it's also a **very good idea to buffer the input** by **chaining it to** a BufferedInputStream and/or a **BufferedReader** .

6. public OutputStream getOutputStream( ) throws IOException

- The **getOutputStream() method** returns a **raw OutputStream for writing data from your application** to the other end of the socket.
- You usually chain this stream to a more convenient class like DataOutputStream or OutputStreamWriter or **PrintWriter** before using it.

LAB 4: WAP for EchoClient.

```
import java.io.*;
import java.net.*;
public class EchoClient {
        public static void main(String[] args) {
                try {
                        Socket socket = new Socket("localhost",4567);
                        BufferedReader in_socket = new BufferedReader(new
                                                        InputStreamReader(socket.getInputStream()));
                        BufferedReader in_user = new BufferedReader(new InputStreamReader(System.in));
                        PrintWriter out_socket = new PrintWriter(socket.getOutputStream(),true);
                        String message;
                        while(true) {
                                message = in_user.readLine();
                                if(message.endsWith(".")) break;
                                out_socket.println(message);
                                System.out.println(in_socket.readLine());}
                        in_socket.close();
                        out_socket.close();
                        socket.close();
                } catch (UnknownHostException e) {
                        System.err.println("Unknown Host.");
                }catch(IOException e) {
                        System.err.println("Socket can't be created.");}}}
```

- Java 1.4 adds an **isClosed() method** that returns **true if the socket has been closed, false if it isn't**:

public boolean isClosed( ) // Java 1.4

- If you're uncertain about a socket's state, you can check it with this method rather than risking an IOException . For example,

if (socket.isClosed( )) { // do something... } else { // do something else... }

- However, this is **not a perfect test**. If the **socket has never been connected in the first place, isClosed( ) returns false**, even though the socket isn't exactly open.
- Java 1.4 **also adds an isConnected() method**:

public boolean isConnected( ) // Java 1.4

- The name is a **little misleading**. It **does not tell** you **if the socket is currently connected to a remote host (that is, if it is unclosed**). **Instead it tells you whether the socket has ever been connected to a remote host**. If the socket was able to connect to the remote host at all, then this method **returns true, even after that socket has been closed**. **To tell if a socket is currently open, you need to check that isConnected( ) returns true and isClosed() returns false**. For example:

boolean connected = socket.isConnected( ) && ! socket.isClosed( );

The Object Methods:

public String toString()

- The toString() method produces a string that looks like this:

```
Socket[addr=www.prime.edu.np/104.21.95.59,port=80,localport=62859]
```

6.5 Setting Socket Options

Socket options **specify how the native sockets** on which the Java Socket class **relies send and receive data**. You can set four options in Java 1.1, six in Java 1.2, seven in Java 1.3, and eight in Java 1.4:

1. **TCP_NODELAY**
2. **SO_LINGER**
3. **SO_TIMEOUT**
4. **SO_RCVBUF (Java 1.2 and later)**
5. **SO_SNDBUF (Java 1.2 and later) //socket option send buffer size**
6. **SO_KEEPALIVE (Java 1.3 and later)**
7. **OOBINLINE (Java 1.4 and later)**

1. TCP_NODELAY

public void setTcpNoDelay(boolean on) throws SocketException

public boolean getTcpNoDelay( ) throws SocketException

- Setting **TCP_NODELAY to true** ensures that **packets are sent as quickly as possible regardless of their size**.
- Before **sending another packet**, the local host **waits to receive acknowledgment** of the previous packet from the remote system. This is known as **Nagle's algorithm**. The **problem with Nagle's algorithm** is that **if the remote system doesn't send acknowledgments back to the local system fast enough**, **applications** that depend on the steady transfer of small bits of information **may slow down**.
- Setting **TCP_NODELAY to true defeats** this **buffering scheme**, so that all **packets are sent as soon as they're ready**.
- SocketException will be thrown only if the underlying socket implementation **doesn't support the TCP_ NODELAY option**.
- For example, the following fragment turns on TCP_NODELAY for the socket s: **s.setTcpNoDelay(true);**

2. SO_LINGER

public void setSoLinger(boolean on, int seconds) throws SocketException

public int getSoLinger( ) throws SocketException

- The SO_LINGER option **specifies what to do with datagrams that have not yet been sent** when a **socket is closed(during sending)**. If the linger **time is set to zero**, **any unsent packets are thrown away when the socket is closed**. If the linger **time is any positive value, the close( ) method blocks while waiting the specified number of seconds** for the **data to be sent and the acknowledgments to be received**. When that **number of seconds has passed**, the **socket is closed** and **neither any remaining data is sent nor any acknowledgment is received**.
- These **two methods each throw a SocketException** if the underlying socket implementation **does not support the SO_LINGER option**. The **setSoLinger()** method can also **throw an IllegalArgumentException** if you try to set the **linger time to a negative value**. However, the **getSoLinger( ) method** may return **-1 to indicate that this option is disabled**,
- For example, to set the linger timeout for the **Socket s to four minutes**, if it's not already set to some other value:

```
if (s.getTcpSoLinger( ) == -1) s.setSoLinger(true, 240);
```

3. SO_TIMEOUT

public void setSoTimeout(int milliseconds) throws SocketException
public int getSoTimeout( ) throws SocketException

- Normally when you **try to read data from a socket**, the **readLine() call blocks** as long as necessary to get enough bytes. By setting SO_TIMEOUT, you ensure that the **call will not block for more than a fixed number of milliseconds**. Although this **readLine( ) call failed**, you can try to **read from the socket again**. The next call may succeed.
- Timeouts are given in **milliseconds**. **Zero is** interpreted as an **infinite timeout**; it is the default value. For example, to set the timeout value of the **Socket object s to 3 minutes** if it isn't already set, specify 180,000 milliseconds:

```
if (s.getSoTimeout( ) == 0) s.setSoTimeout(180000);
```

- These two methods each throw a SocketException if the underlying socket implementation does not support the SO_TIMEOUT option.

4. SO_RCVBUF

- Most TCP stacks **use buffers to improve network performance**. Larger buffers tend to improve performance for reasonably fast connections.

public void setReceiveBufferSize(int size) throws SocketException, IllegalArgumentException

public int getReceiveBufferSize( ) throws SocketException

- The **getReceiveBufferSize() method** returns the **number of bytes in the buffer** that can be **used for input from** this socket.
- The **setReceiveBufferSize() method suggests a number of bytes to use for buffering output** on this socket.
- These methods throw a SocketException if the underlying socket implementation does not recognize the SO_RCVBUF option.

5. SO_SNDBUF

- Starting in Java 1.2, there are methods to get and set the suggested send buffer size used for network output:

public void setSendBufferSize(int size)throws SocketException, IllegalArgumentException

public int getSendBufferSize( ) throws SocketException

- The **getSendBufferSize()** method **returns the number of bytes in the buffer used for output** on this socket.
- The **setSendBufferSize( )** method **suggests a number of bytes to use for buffering output** on this socket.
- These methods throw a **SocketException** if the underlying socket implementation doesn't understand the SO_SNDBUF option.

6. SO_KEEPALIVE

- If SO_KEEPALIVE is turned **on**, the client will **occasionally send a data packet over an idle connection** (most commonly once every two hours), just to make sure the **server hasn't crashed**. If the **server fails to respond** to this packet, the client keeps **trying for** a little more than **11 minutes** until it receives a response. If it **doesn't receive a response within 12 minutes,** the client **closes the socket**.
- Java 1.3 adds methods to turn SO_KEEPALIVE on and off and to determine its current state:

public void setKeepAlive(boolean on) throws SocketException

public boolean getKeepAlive( ) throws SocketException

- The default for SO_KEEPALIVE is false. This code fragment turns SO_KEEPALIVE off, if it's turned on:

```
if (s.getKeepAlive( )) s.setKeepAlive(false);
```

7. OOBINLINE

- TCP includes a feature that **sends a single byte of "urgent" data**. This data is sent immediately. Furthermore, the **receiver is notified when the urgent data is received** and may elect to process the urgent data before it processes any other data that has already been received.
- Java 1.4 adds support for both **sending and receiving such urgent data**. The sending method is named, obviously enough, **sendUrgentData( ) :**

public void sendUrgentData(int data) throws IOException

- This method sends the **lowest order byte** of its argument almost immediately. If necessary, any currently cached data is flushed first.

6.6 Socket in GUI Applications

PRACTICAL IN LAB.

THANK YOU FOR YOUR ATTENTION