Unit 7 – Socket for Server

7.1 Using ServerSockets: Serving Binary Data, Multithreaded Servers, Writing to Servers with Sockets and Closing Server Sockets

7.2 Logging: What to Log and How to Log

7.3 Constructing Server Sockets: Constructing Without Binding

7.4 Getting Information about Server Socket

7.5 Socket Options: SO_TIMEOUT, SO_RSUMEADDR, SO_RCVBUF and Class of Service

7.6 HTTP Servers: A Single File Server, A Redirector and A Full-Fledged HTTP Server

INTRODUCTION TO SERVER SOCKET

- However, client sockets themselves aren't enough; clients aren't much use unless they can talk to a server.
- When you're writing a server, you don't know in advance who will contact you, and even if you did, you wouldn't know when that host wanted to contact you. In other words, servers are like receptionists who sit by the phone and wait for incoming calls. They don't know who will call or when, only that when the phone rings, they have to pick it up and talk to whoever is there.
- Java provides a ServerSocket class to allow programmers to write servers.
- ServerSocket runs on the server and listens for incoming TCP connections. Each ServerSocket listens on a particular port on the server machine.
- Server sockets wait for connections while client sockets initiate connections. Once the server socket has set up the connection, the server uses a regular Socket object to send data to the client. Data always travels over the regular socket.

ServerSocket Class

- The ServerSocket class contains everything you need to write servers in Java. It has constructors that create new ServerSocket objects, methods that listen for connections on a specified port, and methods that return a Socket object when a connection is made so that you can send and receive data. The basic life cycle of a server is:

1. A new ServerSocket is created on a particular port using a ServerSocket( ) constructor.
2. The ServerSocket listens for incoming connection attempts on that port using its accept() method. accept( ) blocks until a client attempts to make a connection, at which point accept( ) returns a Socket object connecting the client and the server.
3. Depending on the type of server, either the Socket's getInputStream( ) method, getOutputStream( ) method, or both are called to get input and output streams that communicate with the client.
4. The server and the client interact according to an agreed-upon protocol until it is time to close the connection.
5. The server, the client, or both close the connection.
6. The server returns to step 2 and waits for the next connection.

USING SOCKETS: INVESTIGATING PROTOCOLS WITH TELNET

- clients that use sockets to communicate with a number of well-known Internet services such as HTTP, echo, and more. The sockets themselves are simple enough; however, the protocols to communicate with different servers make life complex.
- To get a feel for how a protocol operates, you can use Telnet to connect to a server, type different commands to it, and watch its responses. By default, Telnet attempts to connect to port 23. To connect to servers on different ports, specify the port you want to connect to like this:

```
telnet localhost 25
```

- This example 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.

ServerSocket constructors

- `public ServerSocket(int port) throws IOException, BindException`
- `public ServerSocket(int port, int queueLength) throws IOException, BindException`
- `public ServerSocket(int port, int queueLength, InetAddress bindAddress)`
  `throws IOException`

- These constructors let you specify the port, the length of the queue used to hold incoming connection requests, and the local network interface to bind to.

ServerSocket constructors

public ServerSocket(int port) throws IOException, BindException

- This constructor creates a server socket on the port specified by the argument. If you pass for the port number, the system selects an available port for you. For example, to create a server socket that would be used by an HTTP server on port 80, you would write:

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

- The constructor throws an IOException (specifically, a BindException) if the socket cannot be created and bound to the requested port.

ServerSocket constructors

public ServerSocket(int port, int queueLength) throws IOException, BindException

- This constructor creates a ServerSocket on the specified port with a queue length of your choosing. If the machine has multiple network interfaces or IP addresses, then it listens on this port on all those interfaces and IP addresses. The queueLength argument sets the length of the queue for incoming connection requests—that is, how many incoming connections can be stored at one time before the host starts refusing connections.

```
try { ServerSocket httpd = new ServerSocket(5776, 100); }
catch (IOException e) { System.err.println(e); }
```

- The constructor throws an IOException (specifically, a BindException) if the socket cannot be created and bound to the requested port.

ServerSocket constructors

public ServerSocket(int port, int queueLength, InetAddress bindAddress) throws IOException,

- This constructor creates a ServerSocket on the specified port with the specified queue length. This ServerSocket binds only to the specified local IP address. This constructor is useful for servers that run on systems with several IP addresses (a common practice at web server farms) because it allows you to choose the address to which you'll listen.
- To create a server socket that listens on port 5,776 of metalab.unc.edu but not on port 5,776 of www.gigabit-ethernet.org, you would write:

```
try {
    ServerSocket httpd = new ServerSocket(5776, 10,
    InetAddress.getHostByName("metalab.unc.edu"));
}
catch (IOException e) {  System.err.println(e); }
```

Serving Binary Data

- Sending binary, nontext data is not significantly harder. You just use an OutputStream that writes a byte array rather than a Writer that writes a String.
- Example A time server - demonstrates with an iterative time server that follows the time protocol outlined in RFC 868. When a client connects, the server sends a 4-byte, big-endian, unsigned integer specifying the number of seconds that have passed since 12:00 A.M., January 1, 1900,GMT (the epoch). Once again, the current time is found by creating a new Date object. However, because Java's Date class counts milliseconds since 12:00 A.M., January 1,1970, GMT rather than seconds since 12:00 A.M., January 1, 1900, GMT, some conversion is necessary.

Serving Binary Data

```
import java.io.*;
import java.net.*;
import java.util.Date;
public class TimeServerTest {
    public final static int PORT = 37;
    public static void main(String[] args) {
        // The time protocol sets the epoch at 1900, the Date class at 1970.
        // This number converts between them.
        long differenceBetweenEpochs = 2208988800L;
        try ( ServerSocket server = new ServerSocket(PORT)) {
            while (true) {
```

```
try ( Socket connection = server.accept()) {
                    OutputStream out = connection.getOutputStream();
                    Date now = new Date();
                    long msSince1970 = now.getTime();
                    long secondsSince1970 = msSince1970 / 1000;
                    long secondsSince1900 = secondsSince1970
                            + differenceBetweenEpochs;
                    byte[] time = new byte[4];
                    time[0]
                            = (byte) ((secondsSince1900 & 0x00000000FF000000L) >> 24);
                    time[1]
                            = (byte) ((secondsSince1900 & 0x0000000000FF0000L) >> 16);
                    time[2]
                            = (byte) ((secondsSince1900 & 0x000000000000FF00L) >> 8);
                    time[3] = (byte) (secondsSince1900 & 0x00000000000000FFL);
                    out.write(time);
                    out.flush();
                } catch (IOException ex) { System.err.println(ex.getMessage()); }
            }
        } catch (IOException ex) { System.err.println(ex); }
    }
}
```

Multithreaded Servers

- All developers are familiar with writing sequential programs , each sequential programs has a beginning, an execution sequence, and an end.
- A thread is a single sequential flow of control within a program. It's an independent path of execution through program code. When multiple threads execute, one thread's path through the same code usually differs from the others.
- Every thread in Java is created and controlled by the java.lang.Thread Class.
- There are two ways to create thread in java:
  - Implement the Runnable interface (java.lang.Runnable)
  - By Extending the Thread class (java.lang.Thread)
- Multithreading in java is a process of executing multiple threads simultaneously. A multi-threaded program contains two or more process that can run concurrently and each process can handle a different task at the same time making optimal use of the available resources specially when your computer has multiple CPUs. The process of executing multiple threads simultaneously is known as multithreading .

ServerClientThread

```
import java.net.*;
import java.io.*;
class ServerClientThread extends Thread {
    Socket serverClient;
    int clientNo;
    int squre;
    ServerClientThread(Socket inSocket, int counter) {
        serverClient = inSocket;
        clientNo = counter;
    }
    public void run() {
        try {
            DataInputStream inStream = new                    DataInputStream
                    (serverClient.getInputStream());
            DataOutputStream outStream = new DataOutputStream
                    (serverClient.getOutputStream());
            String clientMessage = "", serverMessage = "";
```

ServerClientThread

```
    public void run() {
        try {
            DataInputStream inStream = new DataInputStream(serverClient.getInputStream());
            DataOutputStream outStream = new DataOutputStream(serverClient.getOutputStream());
            String clientMessage = "", serverMessage = "";
            while (!clientMessage.equals("bye")) {
                clientMessage = inStream.readUTF();
                System.out.println("From Client-" + clientNo + ": Number is :" + clientMessage);
                squre = Integer.parseInt(clientMessage) * Integer.parseInt(clientMessage);
                serverMessage = "From Server to Client-" + clientNo + " Square of " +
                                clientMessage + " is " + squre;
                outStream.writeUTF(serverMessage);
                outStream.flush();
            }
            inStream.close();
            outStream.close();
            serverClient.close();
        } catch (Exception ex) {  System.out.println(ex);  }
        finally {   System.out.println("Client -" + clientNo + " exit!! ");     }
    }
}
```

MultithreadedSocketServer

```
import java.net.*;
import java.io.*;
public class MultithreadedSocketServer {
    public static void main(String[] args) throws Exception {
        try {
            ServerSocket server = new ServerSocket(8889);
            int counter = 0;
            System.out.println("Server Started ....");
            while (true) {
                counter++;
                //server accept the client connection request
                Socket serverClient = server.accept();
                System.out.println(" >> " + "Client No:" + counter + " started!");
                //send  the request to a separate thread
                ServerClientThread sct = new ServerClientThread(serverClient, counter);
                sct.start();
            }
        } catch (Exception e) {    System.out.println(e);    }
    }
}
```

```java
public class TCPClient {
    public static void main(String[] args) throws Exception {
        try {
            Socket socket = new Socket("127.0.0.1", 8889);
            DataInputStream inStream = new DataInputStream(socket.getInputStream());
            DataOutputStream outStream = new DataOutputStream(socket.getOutputStream());
            BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
            String clientMessage = "", serverMessage = "";
            while (!clientMessage.equals("bye")) {
                System.out.println("Enter number :");
                clientMessage = br.readLine();
                outStream.writeUTF(clientMessage);
                outStream.flush();
                serverMessage = inStream.readUTF();
                System.out.println(serverMessage);
            }
            outStream.close();
            outStream.close();
            socket.close();
        } catch (Exception e) { System.out.println(e); }
    }
}
```

USING SOCKETS: WRITING TO SERVERS WITH SOCKETS

- Because this protocol is text based, more specifically UTF-8 based, it's convenient to wrap this in a Writer:

```
Writer writer = new OutputStreamWriter(out, "UTF-8");
```

- Now write the command over the socket:

```
writer.write("DEFINE eng-lat gold\r\n");
```

- Finally, flush the output so you'll be sure the command is sent over the network:

```
writer.flush();
```

Writing to Servers with Sockets

- Most protocols, however, require the server to do both – read and write. You'll accept a connection as before, but this time ask for both an InputStream and an OutputStream.
- Read from the client using the InputStream and write to it using the OutputStream. The main trick is understanding the protocol: when to write and when to read.
- The echo protocol, defined in RFC 862, is one of the simplest interactive TCP services. The client opens a socket to port 7 on the echo server and sends data.
- The server sends the data back. This continues until the client closes the connection.
- The echo protocol is useful for testing the network to make sure that data is not mangled by a misbehaving router or firewall.

Closing Server Sockets

- If you're finished with a server socket, you should close it, especially if the program is going to continue to run for some time. This frees up the port for other programs that may wish to use it. Closing a ServerSocket should not be confused with closing a Socket.
- Closing a ServerSocket frees a port on the local host, allowing another server to bind to the port; it also breaks all currently open sockets that the ServerSocket has accepted. Server sockets are closed automatically when a program dies, so it's not absolutely necessary to close them in programs that terminate shortly after the ServerSocket is no longer needed.

```
ServerSocket server = null;
try {
  server = new ServerSocket(port);  // ... work with the server socket
} finally {
  if (server != null) {
    try {       server.close();   }
     catch (IOException ex)  { // ignore }
  }
}
```

LOGGING: WHAT TO LOG AND HOW TO LOG

- In Java, logging is an important feature that helps developers to trace out the errors. It provides a Logging API that was introduced in Java 1.4 version.
- It provides the ability to capture the log file.
- When an application generates the logging call, the Logger records the event in the LogRecord. After that, it sends to the corresponding handlers or appenders. Before sending it to the console or file, the appenders format that log record by using the formatter or layouts.
- Reasons why we may need to capture the application activity.
  - Tracing information of the application.
  - Recording unusual circumstances or errors that may occur in the program
  - Getting the info about whats going in the application

WHAT TO LOG

- There are two primary things to store in logs:
  - Requests
  - Server errors
- Indeed, servers often keep two different logfiles for these two different items. The audit log usually contains one entry for each connection made to the server. Servers that perform multiple operations per connection may have one entry per operation instead. For instance, a dict server might log one entry for each word a client looks up.
- The error log contains mostly unexpected exceptions that occurred while the server was running. For instance, any NullPointerException that happens should be logged here because it indicates a bug in the server you'll need to fix.
- The error log does not contain client errors, such as a client that unexpectedly disconnects or sends a malformed request. These go into the request log. The error log is exclusively for unexpected exceptions.

Java Loggin Levels

- The log levels control the logging details. They determine the extent to which depth the log files are generated. Each level is associated with a numeric value and there are 7 basic log levels. We need to specify the desired level of logging every time, we seek to interact with the log system. The basic logging levels are:

Level	Value	Used for
SEVERE	1000	Indicates some serious failure
WARNING	900	Potential Problem
INFO	800	General Info
CONFIG	700	Configuration Info
FINE	500	General developer info
FINER	400	Detailed developer info
FINEST	300	Specialized Developer Info

Logger Class

- The logger class provides methods for logging. Since LogManager is the one doing actual logging, its instances are accessed using the LogManager‘s getLogger method.
- The global logger instance

is accessed through Logger class’

static field GLOBAL_LOGGER_NAME.

- It is provided as a convenience

for making casual use of the

Logging package.

- Ex – Logger Server using ServerSocket

```
import java.io.*;
import java.net.*;
import java.util.logging.Level;
import java.util.logging.Logger;
import java.util.logging.*;

public class LoggingServer {
  private ServerSocket serverSocket = null;
  private Socket socket = null;
  public LoggingServer(int port) {
    try {
      serverSocket = new ServerSocket(port);
      socket = serverSocket.accept();
    } catch (IOException ioe) {
      ioe.printStackTrace();
    }
  }
```

```
  public void acceptMessage() {
    try {
        // Generating some log messages through the function defined above
        LogManager lgmngr = LogManager.getLogManager();
        // lgmngr now contains a reference to the log manager.
        Logger log = lgmngr.getLogger(Logger.GLOBAL_LOGGER_NAME);
        InputStream inStream = socket.getInputStream();
        BufferedReader reader = new BufferedReader(new InputStreamReader(inStream))
        String str = null;
        while ((str = reader.readLine()) != null) {
                    System.out.println(str);
                    log.log(Level.INFO, str);
        }
    }
    catch (IOException ioe) {  ioe.printStackTrace();  }
  }
  public static void main(String args[]) {
    LoggingServer server = new LoggingServer(2021);
    server.acceptMessage();
  }
}
```

CONSTRUCTING SERVER SOCKETS: CONSTRUCTING WITHOUT BINDING

- These constructors specify the port, the length of the queue used to hold incoming connection requests, and the local network interface to bind to.
- There are four public ServerSocket constructors:
  - public ServerSocket(int port) throws BindException, IOException
  - public ServerSocket(int port, int queueLength) throws BindException, IOException
  - public ServerSocket(int port, int queueLength, InetAddress bindAddress)

    throws IOException
  - public ServerSocket() throws IOException
- For example, to create a server socket that would be used by an HTTP server on port 80, you would write.

```
ServerSocket httpd = new ServerSocket(80);
```

CONSTRUCTING SERVER SOCKETS: CONSTRUCTING WITHOUT BINDING

- Example - Look for local ports, checks for ports on the local machine by attempting to create ServerSocket objects

```
import java.io.*;
import java.net.*;
public class LocalPortScanner {
  public static void main(String[] args) {
    for (int port = 1; port <= 65535; port++) {
      try {
        // the next line will fail and drop into the catch block if
        // there is already a server running on the port
        ServerSocket server = new ServerSocket(port);
      } catch (IOException ex) {
        System.out.println("There is a server on port " + port + ".");
      }
    }
  }
}
```

Constructing Without Binding

- The noargs constructor creates a ServerSocket object but does not actually bind it toa port, so it cannot initially accept any connections. It can be bound later using the bind() methods:

```
public void bind(SocketAddress endpoint) throws IOExceptionpublic
void bind(SocketAddress endpoint, int queueLength) throws IOException
```

- The primary use for this feature is to allow programs to set server socket options before binding to a port. Some options are fixed after the server socket has been bound. The general pattern looks like this:

```
ServerSocket ss = new ServerSocket(); // set socket options...
SocketAddress  http = new InetSocketAddress(80);
ss.bind(http);
```

- You can also pass null for the SocketAddress to select an arbitrary port. This is like passing 0 for the port number in the other constructors.

GETTING INFORMATION ABOUT SERVER SOCKET

- The ServerSocket class provides two getter methods that tell you the local address and port occupied by the server socket. These are useful if you've opened a server socket on an anonymous port and/or an unspecified network interface.

public InetAddress getInetAddress()

- This method returns the address being used by the server (the local host). If the localhost has a single IP address (as most do), this is the address returned by InetAddress.getLocalHost(). If the local host has more than one IP address, the specific address returned is one of the host's IP addresses. You can't predict which address you will get. For example:

```
ServerSocket httpd = new ServerSocket(80);
InetAddress ia = httpd.getInetAddress();
```

- If the ServerSocket has not yet bound to a network interface, this method returns null

GETTING INFORMATION ABOUT SERVER SOCKET

public int getLocalPort()

- The ServerSocket constructors allow you to listen on an unspecified port by passing 0 for the port number. This method lets you find out what port you're listening on. You might use this in a peer-to-peer multi socket program where you already have a means to inform other peers of your location.
- Or a server might spawn several smaller servers to perform particular operations. The well-known server could inform clients on what ports they can find the smaller servers. Of course, you can also use getLocalPort() to find a non anonymous port.

```
try {
      ServerSocket server = new ServerSocket(0);
      System.out.println("This server runs on port "
          + server.getLocalPort());
    } catch (IOException ex) {   System.err.println(ex);   }
```

SOCKET OPTIONS:

- Socket options specify how the native sockets on which the ServerSocket class relies send and receive data. For server sockets, Java supports three options:
  - SO_TIMEOUT
  - SO_REUSEADDR
  - SO_RCVBUF
- It also allows you to set performance preferences for the socket's packets.

SOCKET OPTIONS:

SO_TIMEOUT

- SO_TIMEOUT is the amount of time, in milliseconds, that accept() waits for an in-coming connection before throwing a java.io.InterruptedIOException. If SO_TIMEOUT is 0, accept() will never time out. The default is to never timeout.
- Setting SO_TIMEOUT is uncommon. You might need it if you were implementing a complicated and secure protocol that required multiple connections between the client and the server where responses needed to occur within a fixed amount of time. However, most servers are designed to run for indefinite periods of time and therefore just use the default timeout value, 0 (never time out). If you want to change this, the setSoTimeout() method sets the SO_TIMEOUT field for this server socket object:
  - `public void setSoTimeout(int timeout) throws SocketException`
  - `public int  getSoTimeout() throws IOException`

SOCKET OPTIONS:

SO_REUSEADDR

- The SO_REUSEADDR option for server sockets is very similar to the same option for client sockets. It determines whether a new socket will be allowed to bind to a previously used port while there might still be data traversing the network addressed to the old socket. As you probably expect, there are two methods to get and set this option:

```
public boolean getReuseAddress() throws SocketException
public void setReuseAddress(boolean on) throws SocketException
```

- The default value is platform dependent. This code fragment determines the default value by creating a new ServerSocket and then calling getReuseAddress():

```
ServerSocket ss = new ServerSocket(10240);
System.out.println("Reusable: " + ss.getReuseAddress());
```

SOCKET OPTIONS:

SO_RCVBUF

- The SO_RCVBUF option sets the default receive buffer size for client sockets accepted by the server socket. It's read and written by these two methods:

```
public int  getReceiveBufferSize() throws SocketException
public void setReceiveBufferSize(int size) throws SocketException
```

- Setting SO_RCVBUF on a server socket is like calling setReceiveBufferSize() on each individual socket returned by accept()

Class of Service

- Different types of Internet services have different performance needs. For instance, live streaming video of sports needs relatively high bandwidth. On the other hand, a movie might still need high bandwidth but be able to tolerate more delay and latency. Email can be passed over low-bandwidth connections and even held up for several hours without major harm. Four general traffic classes are defined for TCP:
  - Low cost
  - High reliability
  - Maximum throughput
  - Minimum delay
- These traffic classes can be requested for a given Socket. The setPerformancePreferences() method expresses the relative preferences given to connection time, latency, and bandwidth for sockets accepted on this server.

Class of Service

- Different types of Internet services have different performance needs. For instance,

```
public void setPerformancePreferences(int connectionTime,int latency,int bandwidth)
```

- For instance, by setting connectionTime to 2, latency to 1, and bandwidth to 3, you indicate that maximum bandwidth is the most important characteristic, minimum latency is the least important, and connection time is in the middle:

```
ss.setPerformancePreferences(2, 1, 3);
```

- Exactly how any given VM implements this is implementation dependent. The under-lying socket implementation is not required to respect any of these requests. They only provide a hint to the TCP stack about the desired policy.

HTTP SERVERS

- HTTP is a large protocol. A full-featured HTTP server must respond to requests for files, convert URLs into filenames on the local system, respond to POST and GET requests, handle requests for files that don't exist, interpret MIME types, launch CGI programs, and much, much more.
- However, many HTTP servers don't need all of these features. For example, many sites simply display an "under construction" message. Clearly, Apache is overkill for a site like this. Such a site is a candidate for a custom server that does only one thing. Java's network class library makes writing simple servers like this almost trivial.

A single-file server

- a server that always sends out the same file, no matter who or what the request. This is shown in Example - SingleFileHTTPServer. The filename, local port, and content encoding are read from the command line. If the port is omitted, port 80 is assumed. If the encoding is omitted, ASCII is assumed.

A redirector

- Another simple but useful application for a special-purpose HTTP server is redirection. Here, we develop a server that redirects users from one web site to another—for example, from cnet.com to home.cnet.com.
- Example below show reading a URL and a port number from the command-line, opens a server socket on the port, then redirects all requests that it receives to the site indicated by the new URL, using a 302 FOUND code.
- Chances are this server is fast enough not to require multiple threads. Nonetheless, threads might be mildly advantageous, especially on a high volume site on a slow network connection. And this server does a lot of string processing, one of Java's most notorious performance bottlenecks. But really for purposes of example more than anything, we have made the server multithreaded. In this example, we chose to use a new thread rather than a thread pool for each connection. This is perhaps a little simpler to code and understand but somewhat less efficient.

A Full-Fledged HTTP Server

- Enough special-purpose HTTP servers. This next section develops a full-blown HTTP server, called JHTTP, that can serve an entire document tree, including images, applets, HTML files, text files, and more. It will be very similar to the SingleFileHTTPServer, except that it pays attention to the GET requests.
- This server is still fairly light weight; after looking at the code, we'll discuss other features you might want to add. Because this server may have to read and serve large files from the filesystem over potentially slow network connections, you'll change its approach. Rather than processing each request as it arrives in the main thread of execution, you'll place incoming connections in a pool. Separate instances of a Request Processor class will remove the connections from the pool and process them.
- Example - The JHTTP web server - showing the main JHTTP class, the main() method of JHTTP handles initialization, but other programs can use this class to run basic web servers.