Client Server Application Programming

Week 6: Transmission Control Protocol (ServerSocket Class, Creating a TCPServer.java Specific-Socket Exception handling etc.)

By Elubu Joseph - MSc.IS

Lecturer

Department of Information Technology

Kumi University

Email: josebulinda@gmail.com

jose@kumiuniversity.ac.ug

Summary of previous Lecture

1. Overview of Transmission Control Protocol
2. Advantages of TCP over UDP
3. Communication between Applications Using Ports
4. Socket Operations
5. Socket Class,
6. TCP Socket and java
7. Creating a TCP Client,

Agenda

1. Transmission Control Protocol - ServerSocket Class,
2. Creating a TCP Server,
3. Running TCPClient against TCPServer
4. Exception Handling and
5. Socket-Specific Exceptions

ServerSocket Class

ServerSocket Class is a java class that belong to java.net.*; package. This class has and controls all the necessary resources used for the creation and the maintenance of the functions of the Special network Socket that provides services to the client(s).

ServerSocket is a special type of socket used to provide TCP services. Client sockets bind to any free port on the local machine, and connect to a specific server port and host.

The difference with Socket is that they bind to a specific port on the local machine, so that remote clients may locate a service. Client socket connections will connect to only one machine, whereas server sockets are capable of fulfilling the requests of multiple clients.

The way it works is simple—clients are aware of a service running on a particular port. So when the client initiates connection, the server uses accept() method to accepts the client's request, then the server create a specific dedicated socket for serving the client.

Multiple connections can be accepted at the same time, or a server may choose to accept only one connection at any given moment. Once accepted, the connection is represented as a normal socket, in the form of a Socket object.

ServerSocket Class

once you have mastered the ServerSocket class, it becomes almost as simple to write servers as it is with clients. The only difference between a server and a client is that the server binds to a specific port, using a ServerSocket object.

This ServerSocket object acts as a factory for client connections—you don't need to create instances of the Socket class yourself. These connections are modeled as a normal socket, so you can connect input and output filter streams (or even a reader and writer) to the connection.

ServerSocket Class Constructors

As clearly defined before, a constructor is a method whose name is the same as the name of the class. Constructors allow additional customization of the functionality or capability of a class. Like other classes seen, ServerSocket class also has it's own constructors, all are marked as public. They include: -

1. ServerSocket(int port) throws java.io.IOException, java.lang.SecurityException— binds the server socket to the specified port number, so that remote clients may locate the TCP service. If a value of zero is passed, any free port will be used—however, clients will be unable to access the service unless notified somehow of the port number. By default, the queue size is set to 50, but an alternate constructor is provided that allows modification of this setting. If the port is already bound, or security restrictions (such as security polices or operating system restrictions on well-known port prevent access, an exception is thrown.

2. ServerSocket(int port, int numberOfClients) throws java.io.IOException, java.lang.SecurityException— binds the server socket to the specified port number and allocates sufficient space to the queue to support the specified number of client sockets. This is an overloaded version of the ServerSocket(int port) constructor, and if the port is already bound or security restrictions prevent access,an exception is thrown.

ServerSocket Class Constructors+

3. ServerSocket(int port, int numberOfClients, InetAddress address)throws java.io.IOException, java.lang.SecurityException— binds the server socket to the specified port number, and allocates sufficient space to the queue to support the specified number of client sockets. This is an overloaded version of the ServerSocket(int port, int numberOfClients) constructor that allows a server socket to bind to a specific IP address, in the case of a multihomed machine. For example, a machine may have two network cards, or may be configured to represent itself as several machines by using virtual IP addresses. Specifying a null value for the address will cause the server socket to accept requests on all local addresses. If the port is already bound or security restrictions prevent access, an exception is thrown.

Creating a ServerSocket

Once a server socket is created, it will be bound to a local port and ready to accept incoming connections. When clients attempt to connect, they are placed into a queue. Once all free space in the queue is exhausted, further clients will be refused.

The simplest way to create a server socket is to bind to a local address, which is specified as the only parameter, using a constructor. For example, to provide a service on port 80 (usually used for Web servers), the following snippet of code would be used:

```
try
{
      // Bind to port 80, to provide a TCP service (like HTTP)
      ServerSocket myServer = new ServerSocket ( 80 );

      // ......
}
catch (IOException ioe)
{
      System.err.println ("I/O error - " + ioe);
}
```

ServerSocket Class Methods

While the Socket class is fairly versatile, and has many methods, the Server Socket class doesn't really do that much, other than accept connections and act as a factory for Socket objects that model the connection between client and server.

The most important method is the accept() method, which accepts client connection requests, but there are several others that developers may find useful, as seen listed below.

1. Socket accept() throws java.io.IOException, java.lang.SecurityException— waits for a client to request a connection to the server socket, and accepts it. This is a blocking I/O operation, and will not return until a connection is made (unless the timeout socket option is set). When a connection is established, it will be returned as a Socket object. When accepting connections, each client request will be verified by the default security manager, which makes it possible to accept certain IP addresses and block others, causing an exception to be thrown. However, servers do not need to rely on the security manager to block or terminate connections—the identity of a client can be determined by calling the getInetAddress() method of the client socket.

ServerSocket Class Methods+

2. void close() throws java.io.IOException— closes the server socket, which unbinds the TCP port and allows other services to use it.
3. InetAddress getInetAddress()— returns the address of the server socket, which may be different from the local address in the case of a multihomed machine (i.e., a machine whose localhost is known by two or more IP addresses).
4. int getLocalPort()— returns the port number to which the server socket is bound.
5. int getSoTimeout() throws java.io.IOException— returns the value of the timeout socket option, which determines how many milliseconds an accept() operation can block for. If a value of zero is returned, the accept operation blocks indefinitely.
6. void implAccept(Socket socket) throws java.io.IOException— this method allows ServerSocket subclasses to pass an unconnected socket subclass, and to have that socket object accept an incoming request. Using the implAccept method to accept the connection, an overridden ServerSocket.accept() method can return a connected socket. Few developers will want to subclass the ServerSocket, and using this should be avoided unless required

ServerSocket Class Methods++

static void setSocketFactory (SocketImplFactory factory) throws java.io.IOException, java.net.SocketException, java.lang.Security Exception — assigns a server socket factory for the JVM. This is a static method, and should be called only once during the lifetime of a JVM. If assigning a new socket factory is prohibited, or one has already been assigned, an exception is thrown.

void setSoTimeout(int timeout) throws java.net.SocketException—assigns a timeout value (specified in milliseconds) for the blocking accept() operation. If a value of zero is specified, timeouts are disabled and the operation will block indefinitely. Providing timeouts are enabled, however, whenever the accept() method is called a timer starts. When the timer expires, a java.io.InterruptedIOException is thrown, which allows a server to then take further actions.

Accepting and Processing Requests from TCP Clients

The most important function of a server socket is to accept client sockets. Once a client socket is obtained, the server can perform all the "real work" of server programming, which involves reading from and writing to the socket to implement a network protocol. The exact data that is sent or received is dependent on the details of the protocol.

For example, a mail server that provides access to stored messages would listen to commands and send back message contents. A telnet server would listen for keystrokes and pass these to a log-in shell, and send back output to the network client. Protocol-specific actions are less network- and more programming-oriented.

Accepting and Processing Requests from TCP Clients+

The following snippet shows how client sockets are accepted, and how I/O streams may be connected to the client:

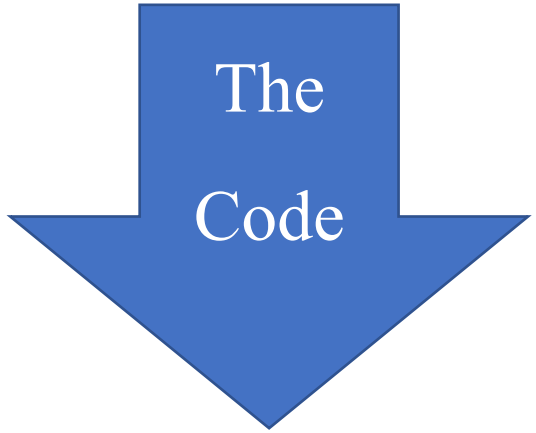
```
// Perform a blocking read operation, to read the next socket
// connection
Socket nextSocket = someServerSocket.accept();

// Connect a filter reader and writer to the stream
BufferedReader reader = new BufferedReader (new
                        InputStreamReader
                        (nextSocket.getInputStream() ) );
PrintWriter writer = new PrintWriter( new
                        OutputStreamWriter
                        (nextSocket.getOutputStream() ) );
```

Creating a TCP Server

One of the most enjoyable parts of networking is writing a network server. Clients send requests and respond to data sent back, but the server performs most of the real work.

Like promised, during lecture 5, this other example is a server side application called TCPServer that provides day time service to the TCPClient.



Creating a TCPServer Code

```
package cs; import java.net.*;import java.io.*;
public class TCPServer {
    public static final int SERVICE_PORT = 13;
    public static void main(String args[]){
        try{
        // Bind to the service port, to grant clients access to the TCP daytime service
        ServerSocket server = new ServerSocket(SERVICE_PORT);
        System.out.println ( "TCP Server started“ );
        // Loop indefinitely, accepting clients
        for (;;){ // Get the next TCP client
        Socket nextClient = server.accept();
        // Display connection details
        System.out.println ( "Received request from " +nextClient.getInetAddress() + ":" + nextClient.getPort() );
```

Creating a TCPServer Code+

```
// Don't read, just write the message
        OutputStream out = nextClient.getOutputStream();
        PrintStream pout = new PrintStream (out);
       // Write the current date out to the user
        pout.print( new java.util.Date() );
        // Flush unsent bytes
        out.flush(); // Close stream
        out.close(); // Close the connection
        nextClient.close();
        }
    }
```

Creating a TCPServer Code++

```
catch (BindException be){
        System.err.println ( "Service already running on port " + SERVICE_PORT );
    }
    catch (IOException ioe){
        System.err.println ( "I/O error - " + ioe);
    }
}
}
```

Creating a TCPServer Code Explained

Here's a step-by-step explanation of the given code:

1.Import the required packages:

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

The code uses classes from the java.net package for networking operations and classes from the java.io package for input/output operations.

2. Define the TCPServer class:

```
public class TCPServer {
```

.

This is the main class that represents the TCP server

Creating a TCPServer Code Explained+

3. Declare a constant integer **SERVICE_PORT:**

```
public static final int SERVICE_PORT = 13;
```

This constant variable represents the port number on which the server will listen for incoming client requests. In this case, it is set to 13, which is the well-known port number for the daytime service.

4. Wrap the code in a **try-catch block:**

```
try {
```

This Is used to handle any exceptions that may occur during runtime.

5. Create a **ServerSocket** object and bind it to the **SERVICE_PORT:**

```
ServerSocket server = new ServerSocket(SERVICE_PORT);
```

This creates a server socket that listens for incoming client requests on the specified SERVICE_PORT.

Creating a TCPServer Code Explained ++

6. Print a message to indicate that the TCP server has started:

```
System.out.println("TCP Server started");
```

7.Enter an infinite loop to accept incoming client requests:

```
for (;;) {
```

This creates an infinite loop that continuously listens for incoming client requests.

8. Accept a client connection:

```
Socket nextClient = server.accept();
```

This blocks and waits for a client to connect. Once a connection is established, it returns a Socket object that represents the client's socket.

Creating a TCPServer Code Explained+++

9. Print the details of the client connection:

```
System.out.println("Received request from " +
nextClient.getInetAddress() + ":" + nextClient.getPort());
```

This prints the IP address and port number of the client that has connected to the server.

10. Get the output stream of the client's socket and write the current date to it:

```
OutputStream out = nextClient.getOutputStream();
PrintStream pout = new PrintStream(out);
pout.print(new java.util.Date());
```

This writes the current date to the output stream, which will be sent to the client as a response

Creating a TCPServer Code Explained++++

11. Flush the unsent bytes in the output stream and close it:

```
out.flush();
out.close();
```

This ensures that all the data in the output stream is sent to the client and then closes the output stream.

12. Close the client's socket:

```
nextClient.close();
```

This closes the socket of the client that has connected to the server.

Creating a TCPServer Code Explained+++++

13. Handle exceptions:

```
} catch (BindException be) {
    System.err.println("Service already running on port " + SERVICE_PORT);
} catch (IOException ioe) {
    System.err.println("I/O error - " + ioe);
}
```

This catches and handles any exceptions that may occur during the execution of the code, such as if the service is already running on the specified port or if there is an I/O error.

Creating a TCPServer Code+NetBeans

```
1  package cs; import java.net.*;import java.io.*;
2  public class TCPServer {
3      public static final int SERVICE_PORT = 13;
4      public static void main(String args[]){
5          try{
6          // Bind to the service port, to grant clients access to the TCP daytime service
7          ServerSocket server = new ServerSocket(SERVICE_PORT);
8          System.out.println ("TCP Server started");
9          // Loop indefinitely, accepting clients
10         for (;;){// Get the next TCP client
           Socket nextClient = server.accept();
12         // Display connection details
13         System.out.println ("Received request from " +
14         nextClient.getInetAddress() + ":" + nextClient.getPort() );
15         // Don't read, just write the message
16         OutputStream out = nextClient.getOutputStream();
17         PrintStream pout = new PrintStream (out);
18         // Write the current date out to the user
19         pout.print( new java.util.Date() );
20         // Flush unsent bytes
21         out.flush();// Close stream
22         out.close();// Close the connection
23         nextClient.close();
```

Creating a TCPServer Code+NetBeans Output

```
24              }
25          }
26          catch (BindException be){
27              System.err.println ("Service already running on port " + SERVICE_PORT );
28          }
29          catch (IOException ioe){
30              System.err.println ("I/O error - " + ioe);
31          }
32      }}
```

TCPServer is running and waiting for any client application that may need a day time service.

Creating a TCP Client- TCPClient.java code int NetBeans

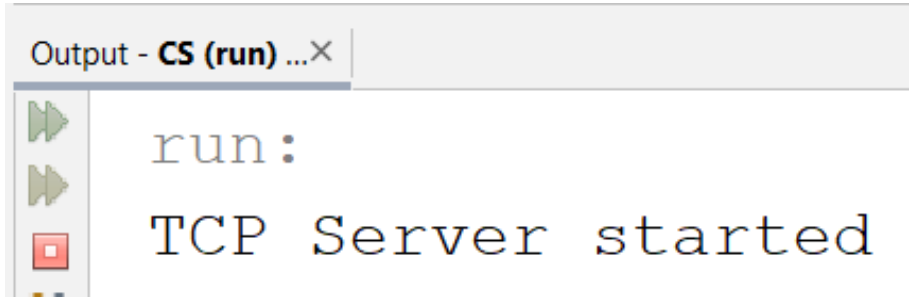
```java
package cs;import java.io.*; import java.net.*;
public class TCPClient {
    public static final int SERVICE_PORT = 13; // Added missing constant SERVICE_PORT
    public static void main(String[] args) {
        // Set the host name
       String hostname = "localhost";
        try {
            // Get a socket to the daytime service
            Socket socket = new Socket(hostname, SERVICE_PORT);
            System.out.println("Connection established");
            // Set the socket option just in case server stalls
            socket.setSoTimeout(2000);
            // Read from the server
            BufferedReader reader = new BufferedReader(new InputStreamReader(socket.getInputStream())
            System.out.println("Results: " + reader.readLine());
            // Close the connection
            socket.close();
        } catch (IOException ioe){
            System.err.println("Error: No Server found.\n " + ioe); //Updated error message
        }
    }
}
```

Running Complete TCPClient.java and TCPServer.java

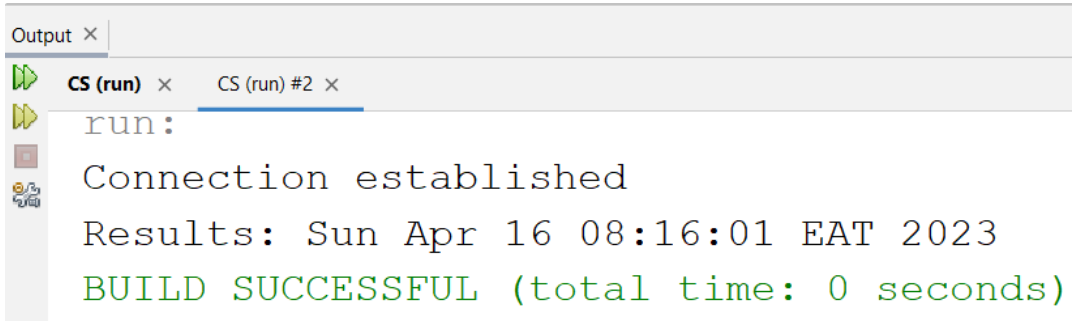
First- Run the TCPServer Application then

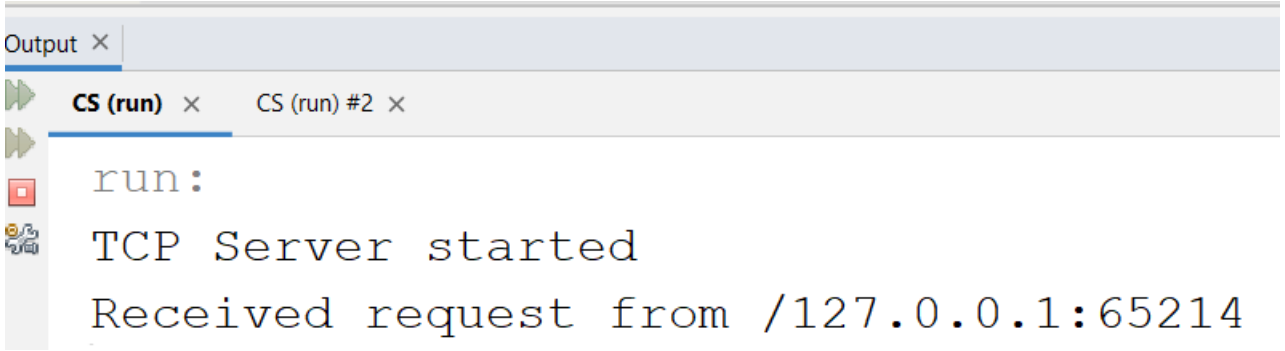
Second - Run the TCPClient Application for you to get correct output

Server running and waiting for client



Client Connected to the server





Server Connected to the TCPClient

Exception Handling: Socket-Specific Exceptions

As a medium for communication, networks are fraught with problems. With so many machines connected to the global Internet, the prospect of encountering a host whose **hostname cannot be resolved**, one that **is disconnected from the network**, or **one that locks up during a connection**, is very likely in the lifetime of a software application.

It is important, therefore, to be aware of the conditions that might cause such problems to arise in an application and to deal with them gracefully.

Of course, not every application will require precise control, and in simple applications you'll probably want to handle everything with a generic handler.

For those more advanced applications, however, it is important to be aware of the socket-specific exceptions that can be thrown at runtime.

Exception Handling: Socket-Specific Exceptions

All socket-specific exceptions extend from SocketException class, so by simply catching that exception, you catch all of the socket-specific ones and write a single generic handler.

In addition, SocketException extends from java.io.IOException if you want to provide a catchall for any I/O exception.

SocketException

The java.net.SocketException represents a generic socket error, which can represent a range of specific error conditions. For finer-grained control, applications should catch the subclasses discussed below.

1. BindException class
2. ConnectException class
3. NoRouteToHostException class
4. InterruptedIOException class

BindException

The java.net.BindException represents an inability to bind a socket to a local port.

The most common reason for this will be that the local port is already in use.

ConnectException

The java.net.ConnectException occurs when a socket can't connect to a specific remote host and port. There can be several reasons for this, such as:-

1. that the remote server does not have a service bound to that port, or
2. that it is so swamped by queued connections,

In that case therefore, it cannot accept any further connections.

NoRouteToHostException

The java.net.NoRouteToHostException is thrown due to a network error, when it is impossible to find a route to the remote host. The possible cause of this may be that;

-

1. either local network on which the software application is running is having a temporary gateway or router problem, or may be the fault of the remote network to which the socket is trying to connect.
2. firewalls and routers are blocking the client software, which is usually a permanent condition.

InterruptedIOException

The java.net.InterruptedIOException occurs when a read operation is blocked for sufficient time to cause a network timeout, as discussed earlier in the chapter.

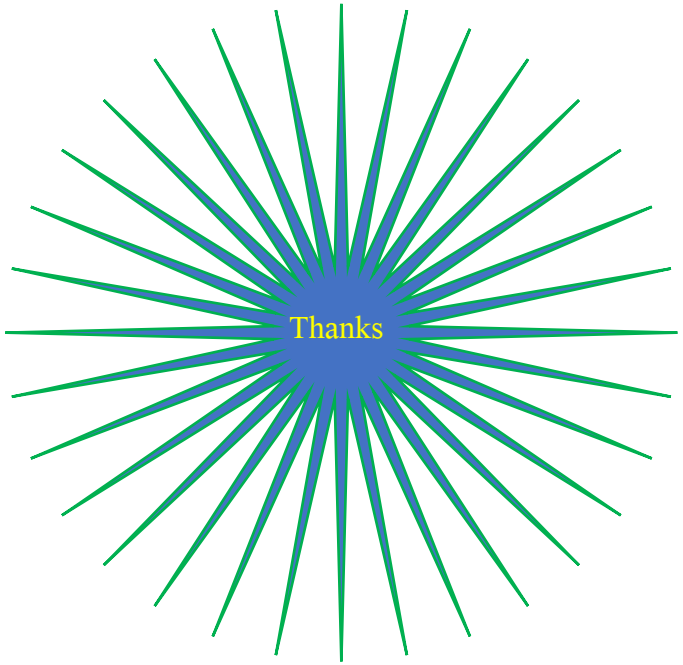
Handling timeouts is a good way to make your code more robust and reliable.

Summary

Summary

1. Transmission Control Protocol ServerSocket Class,
2. Creating a TCP Server,
3. Running TCPClient against TCPServer
4. Exception Handling and
5. Socket-Specific Exceptions(BindException class, ConnectException classNoRouteToHostException class etc.)

Thank you for Listening



References

Java Network Programming and Distributed Computing, David R, Michael R (2002), Publisher: Addison Wesley, ISBN: 0-201-71037-4