Multi-threaded Client/Server Applications

Sockets Programming in Java

What is a socket ?

A socket is the one end-point of a two-way communication link between two programs running over the network. Running over the network means that the programs run on different computers, usually referred as the local and the remote computers. However one can run the two programs on the same computer. Such communicating programs constitutes a client/server application. The server implements a dedicated logic, called **service**. The clients connect to the server to get served, for example, to obtain some data or to ask for the computation of some data. Different client/server applications implement different kind of services.

To distinguish different services, a numbering convention was proposed. This convention uses integer numbers, called port numbers, to denote the services. A server implementing a service assigns a specific port number to the entry point of the service. There are no specific physical entry points for the services in a computer. The port numbers for services are stored in configuration files and are used by the computer software to create network connections.

A socked is a complex data structure that contains an internet address and a port number. A socket, however, is referenced by its descriptor, like a file which is referenced by a file descriptor. That is why, the sockets are accessed via an application programming interface (API) similar to the file input/output API. This makes the programming of network applications very simple. The two-way communication link between the two programs running on different computers is done by reading from and writing to the sockets created on these computers. The data read from a socked is the data wrote into the other socket of the link. And reciprocally, the the data wrote into a socket in the data read from the other socket of the link. These two sockets are created and linked during the connection creation phase. The link between two sockets is like a pipe that is implemented using a stack of protocols. This linking of the sockets involves that internally a socket has a much more complex data structure, or more precisely, a collaboration of data structures. Thus, a socket data structure is more than just an internet address and a port number. You have to imagine a socket as a data structure that contains at least the internet address and the port number on the local computer, and the internet address and the port number on the remote computer.

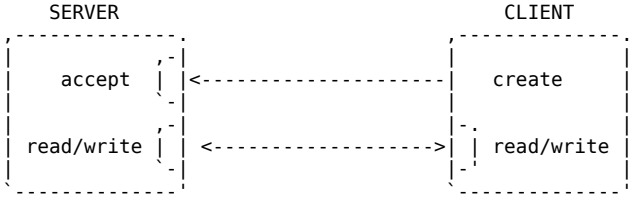
How a network connection is cr eated ?

A network connection is initiated by a client program when it creates a socket for the communication with the server. To create the socket in Java, the client calls the **Socket** constructor and passes the server address and the the specific server port number to it. At this stage the server must be started on the machine having the specified address and listening for connections on its specific port number.

The server uses a specific port dedicated only to listening for connection requests from clients. It can not use this specific port for data communication with the clients because the server must be able to accept the client connection at any instant. So, its specific port is dedicated only to listening for new connection requests. The server side socket associated with specific port is called server socket. When a connection request arrives on this socket from the client side, the client and the server establish a connection. This connection is established as follows:

1. When the server receives a connection request on its specific server port, it creates a new socket for it and binds a port number to it.
2. It sends the new port number to the client to inform it that the connection is established.
3. The server goes on now by listening on two ports:
   - it waits for new incoming connection requests on its specific port, and
   - it reads and writes messages on established connection (on new port) with the accepted client.

   The server communicates with the client by reading from and writing to the new port. If other connection requests arrive, the server accepts them in the similar way creating a new port for each new connection. Thus, at any instant, the server must be able to communicate simultaneously with many clients and to wait on the same time for incoming requests on its specific server port. The communication with each client is done via the sockets created for each communication.



The **java.net** package in the Java development environment provides the class **Socket** that implements the client side and the class **serverSocket** class that implements the server side sockets.

The client and the server must agree on a protocol. They must agree on the language of the information transferred back and forth through the socket. There are two communication protocols :

- stream communication protocol and,
- datagram communication protocol.

The stream communication protocol is known as TCP (transfer control protocol). TCP is a connection-oriented protocol. It works as described in this document. In order to communicate over the TCP protocol, a connection must

first be established between two sockets. While one of the sockets listens for a connection request (server), the other asks for a connection (client). Once the two sockets are connected, they can be used to transmit and/or to receive data. When we say "two sockets are connected" we mean the fact that the server accepted a connection. As it was explained above the server creates a new local socket for the new connection. The process of the new local socket creation, however, is transparent for the client.

The datagram communication protocol, known as UDP (user datagram protocol), is a connectionless protocol. No connection is established before sending the data. The data are sent in a packet called datagram. The datagram is sent like a request for establishing a connection. However, the datagram contains not only the addresses, it contains the user data also. Once it arrives to the destination the user data are read by the remote application and no connection is established. This protocol requires that each time a datagram is sent, the local socket and the remote socket addresses must also be sent in the datagram. These addresses are sent in each datagram.

The **java.net** package in the Java development environment provides the class **DatagramSocket** for programming datagram communications.

UDP is an unreliable protocol. There is no guarantee that the datagrams will be delivered in a good order to the destination socket. For, example, a long text, split in several pages and sent one page per datagram, can be received in a different page order. On the other side, TCP is a reliable protocol. TCP guarantee that the pages will be received in the order in which they are sent.

When programming TCP and UDP based applications in Java, different types of sockets are used. These sockets are implemented in different classes. The classes **ServerSocket** and **Socket** implement TCP based sockets and the class **DatagramSocket** implements UDP based sockets as follows:

- Stream socket to listen for client requests (TCP): the class **ServerSocket**.
- Stream socket (TCP): the class **Socket**.
- Datagram socket (UDP): the class **DatagramSocket**.

This document shows how to program TCP based client/server applications. The UDP oriented programming is not covered in document.

Opening a socket

The client side

When programming a client, a socket must be opened like below:

```
Socket MyClient;
MyClient = new Socket("MachineName", PortNumber);
```

This code, however, must be put in a **try/catch** block to catch the **IOException**:

```
Socket MyClient;
try {
      MyClient = new Socket("MachineName", PortNumber);
    }
    catch (IOException e) {
      System.out.println(e);
    }
```

where

- **MachineName** is the machine name to open a connection to and
- **PortNumber** is the port number on which the server to connect to is listening.

When selecting a port number, one has to keep in mind that the port numbers in the range from 0 to 1023 are reserved for standard services, such as email, FTP, HTTP, etc. For our service (the chat server) the port number should be chosen greater than 1023.

The server side

When programming a server, a server socket must be created first, like below:

```
ServerSocket MyService;
try {
      MyServerice = new ServerSocket(PortNumber);
    }
    catch (IOException e) {
      System.out.println(e);
    }
```

The server socket is dedicated to listen to and accept connections from clients. After accepting a request from a client the server creates a client socket to communicate (to send/receive data) with the client, like below :

```
Socket clientSocket = null;
try {
      serviceSocket = MyService.accept();
    }
    catch (IOException e) {
      System.out.println(e);
    }
```

Now the server can send/receive data to/from the clients. Since the sockets are like the file descriptors the send/receive operations are implemented like read/write file operations on the input/output streams.

Creating an input stream

On the client side, you can use the **DataInputStream** class to create an input stream to receive responses from the server:

```
DataInputStream input;
try {
   input = new DataInputStream(MyClient.getInputStream());
}
catch (IOException e) {
   System.out.println(e);
}
```

The class **DataInputStream** allows you to read lines of text and Java primitive data types in a portable way. It has several read methods such as **read**, **readChar**, **readInt**, **readDouble**, and **readLine**. One has to use whichever function depending on the type of data to receive from the server.

On the server side, the **DataInputStream** is used to receive inputs from the client:

```
DataInputStream input;
try {
     input = new DataInputStream(serviceSocket.getInputStream());
}
catch (IOException e) {
   System.out.println(e);
}
```

Create an output stream

On the client side, an output stream must be created to send the data to the server socket using the class **PrintStream** or **DataOutputStream** of **java.io** package:

```
PrintStream output;
try {
   output = new PrintStream(MyClient.getOutputStream());
}
catch (IOException e) {
   System.out.println(e);
}
```

The class **PrintStream** implements the methods for displaying Java primitive data types values, like **write** and **println** methods. Also, one may want to use the **DataOutputStream**:

```
DataOutputStream output;
try {
   output = new DataOutputStream(MyClient.getOutputStream());
}
catch (IOException e) {
   System.out.println(e);
}
```

The class **DataOutputStream** allows you to write Java primitive data types; many of its methods write a single Java primitive type to the output stream.

On the server side, one can use the class **PrintStream** to send data to the client.

```
PrintStream output;
try {
   output = new PrintStream(serviceSocket.getOutputStream());
}
catch (IOException e) {
   System.out.println(e);
}
```

Closing sockets

Closing a socked is like closing a file. You have to close a socket when you do not need it any more. The output and the input streams must be closed as well but before closing the socket.

On the client side you have to close the input and the output streams and the socket like below:

```
try {
   output.close();
   input.close();
   MyClient.close();
}
catch (IOException e) {
   System.out.println(e);
}
```

On the server you have to close the input and output streams and the two sockets as follows:

```
try {
   output.close();
   input.close();
   serviceSocket.close();
   MyService.close();
}
catch (IOException e) {
   System.out.println(e);
}
```

Usually, on the server side you need to close only the client socket after the client gets served. The server socket is kept open as long as the server is running. A new client can connect to the server on the server socket to establish a new connection, that is, a new client socket.

A simple Client/Server application

In this section we present a simple client/server application.

The client

This is a simple client which reads a line from the standard input and sends it to the echo server. The client keeps then reading from the socket till it receives the message **"Ok"** from the server. Once it receives the **"Ok"** message then it breaks.

```
//Example 23

import java.io.DataInputStream;
import java.io.PrintStream;
import java.io.BufferedInputStream;
import java.io.IOException;
import java.net.Socket;
import java.net.UnknownHostException;

public class Client {
  public static void main(String[] args) {

    Socket clientSocket = null;
    DataInputStream is = null;
    PrintStream os = null;
    DataInputStream inputLine = null;

    /*
     * Open a socket on port 2222. Open the input and the output streams.
     */
    try {
      clientSocket = new Socket("localhost", 2222);
      os = new PrintStream(clientSocket.getOutputStream());
      is = new DataInputStream(clientSocket.getInputStream());
      inputLine = new DataInputStream(new BufferedInputStream(System.in));
    } catch (UnknownHostException e) {
      System.err.println("Don't know about host");
    } catch (IOException e) {
      System.err.println("Couldn't get I/O for the connection to host");
    }

    /*
     * If everything has been initialized then we want to write some data to the
     * socket we have opened a connection to on port 2222.
     */
    if (clientSocket != null && os != null && is != null) {
      try {

        /*
         * Keep on reading from/to the socket till we receive the "Ok" from the
         * server, once we received that then we break.
         */
```

```
        System.out.println("The client started. Type any text. To quit it type 'Ok'.");
        String responseLine;
        os.println(inputLine.readLine());
        while ((responseLine = is.readLine()) != null) {
          System.out.println(responseLine);
          if (responseLine.indexOf("Ok") != -1) {
            break;
          }
          os.println(inputLine.readLine());
        }

        /*
         * Close the output stream, close the input stream, close the socket.
         */
        os.close();
        is.close();
        clientSocket.close();
      } catch (UnknownHostException e) {
        System.err.println("Trying to connect to unknown host: " + e);
      } catch (IOException e) {
        System.err.println("IOException:  " + e);
      }
    }
  }
}
```

The server

This is a simple echo server. The server is dedicated to echo messages received from clients. When it receives a message it sends the message back to the client. Also, it appends the string **"From server :"** in from of the echoed message.

```
//Example 24

import java.io.DataInputStream;
import java.io.PrintStream;
import java.io.IOException;
import java.net.Socket;
import java.net.ServerSocket;

public class Server {
  public static void main(String args[]) {

    ServerSocket echoServer = null;
    String line;
    DataInputStream is;
    PrintStream os;
    Socket clientSocket = null;

    /*
     * Open a server socket on port 2222. Note that we can't choose a port less
     * than 1023 if we are not privileged users (root).
     */
    try {
      echoServer = new ServerSocket(2222);
    } catch (IOException e) {
      System.out.println(e);
    }

    /*
     * Create a socket object from the ServerSocket to listen to and accept
     * connections. Open input and output streams.
     */
    System.out.println("The server started. To stop it press <CTRL><C>.");
    try {
      clientSocket = echoServer.accept();
      is = new DataInputStream(clientSocket.getInputStream());
      os = new PrintStream(clientSocket.getOutputStream());

      /* As long as we receive data, echo that data back to the client. */
      while (true) {
        line = is.readLine();
        os.println("From server: " + line);
      }
    } catch (IOException e) {
      System.out.println(e);
    }
  }
}
```

Compiling and running the application

To try this application you have to compile the two programs: **Example 23** and **Example 24**.

Save these programs on your computer. Name the files **Client.java** and **Server.java**. Open a shell window on your computer and change the current directory to the directory where you saved these files.

Type the following two commands in the shell window.

```
javac Server.java
javac Client.java
```

If java compiler is installed on your computer and the PATH variable is configured for the shell to find **javac** compiler, then these two command lines will create two new files in the current directory : the files **Server.class** and **Client.class**

Start the server in the shell window using the command:

```
java Server
```

You will see the following message in this window

```
The server started. To stop it press <CTRL><C>.
```

telling you that the server is started.

Open a new shell window and change the current directory to the directory where you saved the application files.

Start the client in the shell window using the command:

```
java Client
```

You will see the following message in this window

```
The client started. Type any text. To quit it type 'Ok'.
```

telling you that the client is started.

Type, for example, the text **Hello** in this window. You will see the following output.

```
hello
From server: hello
```

telling you that the message **Hello** was sent to the server and the echo was received by the client from the server.

A multi-threaded Client/Server application

The next example is a chat application. A chat application consists of a chat server and a chat client. The server accepts connections from the clients and delivers all messages from each client to other clients. This is a tool to communicate with other people over Internet in real time.

The client is implemented using two threads - one thread to interact with the server and the other with the standard input. Two threads are needed because a client must communicate with the server and, simultaneously, it must be ready to read messages from the standard input to be sent to the server.

The server is implemented using threads also. It uses a separate thread for each connection. It spawns a new client thread every time a new connection from a client is accepted. This simplifies a lot the design of the server. Multi-threading, however, creates synchronization issues. We will present two implementations of the chat server. An implementation that focus on multi-threading without considering the synchronization issues will be presented first. Then we will focus on the synchronization issues that a multi-threaded implementation creates. Finally, an updated version of the multi-threaded chat server that fixes the synchronization issues is presented.

The chat Client

The code below is the multi-threaded chat client. It uses two threads: one to read the data from the standard input and to sent it to the server, the other to read the data from the server and to print it on the standard output.

```
//Example 25

import java.io.DataInputStream;
import java.io.PrintStream;
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.IOException;
import java.net.Socket;
import java.net.UnknownHostException;

public class MultiThreadChatClient implements Runnable {

  // The client socket
  private static Socket clientSocket = null;
  // The output stream
```

```
    private static PrintStream os = null;
    // The input stream
    private static DataInputStream is = null;

    private static BufferedReader inputLine = null;
    private static boolean closed = false;

    public static void main(String[] args) {

      // The default port.
      int portNumber = 2222;
      // The default host.
      String host = "localhost";

      if (args.length < 2) {
        System.out
            .println("Usage: java MultiThreadChatClient <host> <portNumber>\n"
                + "Now using host=" + host + ", portNumber=" + portNumber);
      } else {
        host = args[0];
        portNumber = Integer.valueOf(args[1]).intValue();
      }

      /*
       * Open a socket on a given host and port. Open input and output streams.
       */
      try {
        clientSocket = new Socket(host, portNumber);
        inputLine = new BufferedReader(new InputStreamReader(System.in));
        os = new PrintStream(clientSocket.getOutputStream());
        is = new DataInputStream(clientSocket.getInputStream());
      } catch (UnknownHostException e) {
        System.err.println("Don't know about host " + host);
      } catch (IOException e) {
        System.err.println("Couldn't get I/O for the connection to the host "
            + host);
      }

      /*
       * If everything has been initialized then we want to write some data to the
       * socket we have opened a connection to on the port portNumber.
       */
      if (clientSocket != null && os != null && is != null) {
        try {

          /* Create a thread to read from the server. */
          new Thread(new MultiThreadChatClient()).start();
          while (!closed) {
            os.println(inputLine.readLine().trim());
          }
          /*
           * Close the output stream, close the input stream, close the socket.
           */
          os.close();
          is.close();
          clientSocket.close();
        } catch (IOException e) {
          System.err.println("IOException:  " + e);
        }
      }
    }

    /*
     * Create a thread to read from the server. (non-Javadoc)
     *
     * @see java.lang.Runnable#run()
     */
    public void run() {
      /*
       * Keep on reading from the socket till we receive "Bye" from the
       * server. Once we received that then we want to break.
       */
      String responseLine;
      try {
        while ((responseLine = is.readLine()) != null) {
          System.out.println(responseLine);
          if (responseLine.indexOf("*** Bye") != -1)
            break;
        }
        closed = true;
      } catch (IOException e) {
        System.err.println("IOException:  " + e);
      }
    }
  }
```

The chat server

We continue with the multi-threaded chat server. It uses a separate thread for each client. It spawns a new client thread every time a new connection from a client is accepted. This thread opens the input and the output streams for a particular client, it ask the client's name, it informs all clients about the fact that a new client has joined the chat room

and, as long as it receive data, echos that data back to all other clients. When the client leaves the chat room, this thread informs also the clients about that and terminates.

```
//Example 26

import java.io.DataInputStream;
import java.io.PrintStream;
import java.io.IOException;
import java.net.Socket;
import java.net.ServerSocket;

/*
 * A chat server that delivers public and private messages.
 */
public class MultiThreadChatServer {

  // The server socket.
  private static ServerSocket serverSocket = null;
  // The client socket.
  private static Socket clientSocket = null;

  // This chat server can accept up to maxClientsCount clients' connections.
  private static final int maxClientsCount = 10;
  private static final clientThread[] threads = new clientThread[maxClientsCount];

  public static void main(String args[]) {

    // The default port number.
    int portNumber = 2222;
    if (args.length < 1) {
      System.out
          .println("Usage: java MultiThreadChatServer <portNumber>\n"
              + "Now using port number=" + portNumber);
    } else {
      portNumber = Integer.valueOf(args[0]).intValue();
    }

    /*
     * Open a server socket on the portNumber (default 2222). Note that we can
     * not choose a port less than 1023 if we are not privileged users (root).
     */
    try {
      serverSocket = new ServerSocket(portNumber);
    } catch (IOException e) {
      System.out.println(e);
    }

    /*
     * Create a client socket for each connection and pass it to a new client
     * thread.
     */
    while (true) {
      try {
        clientSocket = serverSocket.accept();
        int i = 0;
        for (i = 0; i < maxClientsCount; i++) {
          if (threads[i] == null) {
            (threads[i] = new clientThread(clientSocket, threads)).start();
            break;
          }
        }
        if (i == maxClientsCount) {
          PrintStream os = new PrintStream(clientSocket.getOutputStream());
          os.println("Server too busy. Try later.");
          os.close();
          clientSocket.close();
        }
      } catch (IOException e) {
        System.out.println(e);
      }
    }
  }
}

/*
 * The chat client thread. This client thread opens the input and the output
 * streams for a particular client, ask the client's name, informs all the
 * clients connected to the server about the fact that a new client has joined
 * the chat room, and as long as it receive data, echos that data back to all
 * other clients. When a client leaves the chat room this thread informs also
 * all the clients about that and terminates.
 */
class clientThread extends Thread {

  private DataInputStream is = null;
  private PrintStream os = null;
  private Socket clientSocket = null;
  private final clientThread[] threads;
  private int maxClientsCount;

  public clientThread(Socket clientSocket, clientThread[] threads) {
    this.clientSocket = clientSocket;
```

```
      this.threads = threads;
      maxClientsCount = threads.length;
    }

    public void run() {
      int maxClientsCount = this.maxClientsCount;
      clientThread[] threads = this.threads;

      try {
        /*
         * Create input and output streams for this client.
         */
        is = new DataInputStream(clientSocket.getInputStream());
        os = new PrintStream(clientSocket.getOutputStream());
        os.println("Enter your name.");
        String name = is.readLine().trim();
        os.println("Hello " + name
            + " to our chat room.\nTo leave enter /quit in a new line");
        for (int i = 0; i < maxClientsCount; i++) {
          if (threads[i] != null && threads[i] != this) {
            threads[i].os.println("*** A new user " + name
                + " entered the chat room !!! ***");
          }
        }
        while (true) {
          String line = is.readLine();
          if (line.startsWith("/quit")) {
            break;
          }
          for (int i = 0; i < maxClientsCount; i++) {
            if (threads[i] != null) {
              threads[i].os.println("<" + name + "&gr; " + line);
            }
          }
        }
        for (int i = 0; i < maxClientsCount; i++) {
          if (threads[i] != null && threads[i] != this) {
            threads[i].os.println("*** The user " + name
                + " is leaving the chat room !!! ***");
          }
        }
        os.println("*** Bye " + name + " ***");

        /*
         * Clean up. Set the current thread variable to null so that a new client
         * could be accepted by the server.
         */
        for (int i = 0; i < maxClientsCount; i++) {
          if (threads[i] == this) {
            threads[i] = null;
          }
        }

        /*
         * Close the output stream, close the input stream, close the socket.
         */
        is.close();
        os.close();
        clientSocket.close();
      } catch (IOException e) {
      }
    }
  }
```

The synchronization issues of the multi-threaded chat server implementation

Consider now the synchronization issues such an implementation creates. To simplify our task let us split the chat server code as follows, see the partitioned code below.

```
1   //Example 26

    import java.io.DataInputStream;
    import java.io.PrintStream;
    import java.io.IOException;
    import java.net.Socket;
    import java.net.ServerSocket;

    /*
     * A chat server that delivers public and private messages.
     */
    public class MultiThreadChatServer {

      // The server socket.
      private static ServerSocket serverSocket = null;
      // The client socket.
      private static Socket clientSocket = null;

      // This chat server can accept up to maxClientsCount clients' connections.
      private static final int maxClientsCount = 10;
      private static final clientThread[] threads = new clientThread[maxClientsCount];
```

```
  public static void main(String args[]) {

    // The default port number.
    int portNumber = 2222;
    if (args.length < 1) {
      System.out
          .println("Usage: java MultiThreadChatServer <portNumber>\n"
              + "Now using port number=" + portNumber);
    } else {
      portNumber = Integer.valueOf(args[0]).intValue();
    }

    /*
     * Open a server socket on the portNumber (default 2222). Note that we can
     * not choose a port less than 1023 if we are not privileged users (root).
     */
    try {
      serverSocket = new ServerSocket(portNumber);
    } catch (IOException e) {
      System.out.println(e);
    }

    /*
     * Create a client socket for each connection and pass it to a new client
     * thread.
     */
    while (true) {
      try {
        clientSocket = serverSocket.accept();
        int i = 0;
```

2

```
        for (i = 0; i < maxClientsCount; i++) {
          if (threads[i] == null) {
            (threads[i] = new clientThread(clientSocket, threads)).start();
            break;
          }
        }
```

3

```
        if (i == maxClientsCount) {
          PrintStream os = new PrintStream(clientSocket.getOutputStream());
          os.println("Server too busy. Try later.");
          os.close();
          clientSocket.close();
        }
      } catch (IOException e) {
        System.out.println(e);
      }
    }
  }
}

/*
 * The chat client thread. This client thread opens the input and the output
 * streams for a particular client, ask the client's name, informs all the
 * clients connected to the server about the fact that a new client has joined
 * the chat room, and as long as it receive data, echos that data back to all
 * other clients. When a client leaves the chat room this thread informs also
 * all the clients about that and terminates.
 */
class clientThread extends Thread {

  private DataInputStream is = null;
  private PrintStream os = null;
  private Socket clientSocket = null;
  private final clientThread[] threads;
  private int maxClientsCount;

  public clientThread(Socket clientSocket, clientThread[] threads) {
    this.clientSocket = clientSocket;
    this.threads = threads;
    maxClientsCount = threads.length;
  }

  public void run() {
    int maxClientsCount = this.maxClientsCount;
    clientThread[] threads = this.threads;

    try {
      /*
       * Create input and output streams for this client.
       */
      is = new DataInputStream(clientSocket.getInputStream());
      os = new PrintStream(clientSocket.getOutputStream());
      os.println("Enter your name.");
      String name = is.readLine().trim();
      os.println("Hello " + name
          + " to our chat room.\nTo leave enter /quit in a new line");
```

4

```
      for (int i = 0; i < maxClientsCount; i++) {
        if (threads[i] != null && threads[i] != this) {
          threads[i].os.println("*** A new user " + name
              + " entered the chat room !!! ***");
        }
      }
```

```
5         while (true) {
            String line = is.readLine();
            if (line.startsWith("/quit")) {
              break;
            }
```

```
6           for (int i = 0; i < maxClientsCount; i++) {
              if (threads[i] != null) {
                threads[i].os.println("<" + name + "> " + line);
              }
            }
```

```
7         }
```

```
8         for (int i = 0; i < maxClientsCount; i++) {
            if (threads[i] != null && threads[i] != this) {
              threads[i].os.println("*** The user " + name
                  + " is leaving the chat room !!! ***");
            }
          }
```

```
9         os.println("*** Bye " + name + " ***");
          /*
           * Clean up. Set the current thread variable to null so that a new client
           * could be accepted by the server.
           */
```

```
10        for (int i = 0; i < maxClientsCount; i++) {
            if (threads[i] == this) {
              threads[i] = null;
            }
          }
```

```
11        /*
           * Close the output stream, close the input stream, close the socket.
           */
          is.close();
          os.close();
          clientSocket.close();
        } catch (IOException e) {
        }
      }
    }
```

Consider the portions of code colored in green, that is, the portions 2,4,6,8,10 (see the code above). All these portions use the array **threads[]**. This array, however, is shared by all threads of the server. The array is passed by reference to the constructor of the thread every time a new thread is created. The modification of this array by a thread is visible by all other threads. These portions of code are called critical sections because, if used uncontrolled, they can cause unexpected behaviour end even exceptions as explained below.

Since all threads run concurrently, the access to this array is also concurrent. Suppose now that a thread (Thread 1) enters the portion 4 while another thread (Thread 2) enters the section 10 of the code. The section 4 uses the array **threads[]** to inform the clients about a new client. The section 10, however, removes from this array the thread references of the client that leaves the chat room. It can happen, that a **threads[i]** reference, while being used in section 4, is set to null in section 10 by another thread - by the thread of the client leaving the chat room. The table below shows this scenario. In this scenario the Thread 1 executes the **if** statement in portion 4. Suppose **threads[i]** is not null at this instant. Suppose, also, Thread 1 is interrupted by the operating system immediately after evaluating the **if** statement condition. It means, Thread 1 is put in a waiting queue, while the Thread 2 starts executing portion 10. Such kind of execution is called inter-living. Suppose, Thread 2 sets **threads[i]** to null when executing portion 10. Finally, Thread 1 is resumed and executes **threads[i].os.println()** statement. But **threads[i]** is null at this instant. This will cause a null pointer exception.

This exception will close abnormally the connection with a client. And all that because of another client that decided to leave the chat room. The same situation can arise if we consider the concurrency of any of the section 2,6,8 with the section 10. Such situations are not acceptable and must be resolved correctly in a concurrent multi-threaded application.

	Thread 1	Thread 2
4	`for (int i = 0; i < maxClientsCount; i++) {` `  if (threads[i] != null && threads[i] != this) {`	

10		`for (int i = 0; i < maxClientsCount; i++) {`
		`  if (threads[i] == this) {`
		`  threads[i] = null;`
		`  }`
		`  }`

4	`threads[i].os.println("*** A new user " + name`
	`  + " entered the chat room !!! ***");`
	`  }`
	`  }`	

To avoid such kind of exception, the threads must be synchronized so that they execute the critical portions of code (in green) sequentially, and thus, without inter-living. For example, in the table below, the execution of the two portions of code is sequential - the critical sections executes without interruption. We call such execution synchronized. To archive this synchronization we have to use the **synchronized(this){}** statement, like below.

	Thread 1	Thread 2
4	`synchronized (this){` `  for (int i = 0; i < maxClientsCount; i++) {`	

	`if (threads[i] != null && threads[i] != this) {`
	`  threads[i].os.println("*** A new user " + name`
	`  + " entered the chat room !!! ***");`
	`  }`
	`  }`
	`  }`	

10		`synchronized(this) {`
		`  for (int i = 0; i < maxClientsCount; i++) {`
		`  if (threads[i] == this) {`
		`  threads[i] = null;`
		`  }`
		`  }`
		`  }`

All **synchronized(this){}** statements exclude mutually each other. It means, that when a thread enters the **synchronized(this){}** statement it verifies first that any other **synchronized(this){}** statement is not being executed by another thread. If a such a statement is being executed by a thread, then this thread, as well as all other threads trying to execute a **synchronized(this){}** statement, are forced to wait until the thread executing the **synchronized(this){}** terminates this statement. When the thread executing a **synchronized(this){}** statement leaves the critical section, that is, when it terminates the **synchronized(this){}** statement, a thread waiting for critical section enters its **synchronized(this){}**. When a thread enters **synchronized(this){}** statement it blocks all other threads from entering their **synchronized(this){}** statements. Thus, putting all critical sections in **synchronized(this){}** statements we are guarantied that the chat server will execute correctly without rising null pointer exceptions caused by concurrent execution of other critical sections.

The **synchronized(this){}** statement is a powerful tool. However, using it requires a good understanding of the synchronization issue. The incorrect use of **synchronized(this){}** statement can cause deadlocks of the program. A deadlock is a scenario when one thread waits for another thread to leave its critical section forever. To explain this scenario, suppose we extended the critical section 6 like below. This is, suppose the **synchronized(this){}** statement includes a loop that potentially can execute forever.

6	`synchronized(this) {`
	`  while (true) {`
	`  String line = is.readLine();`
	`  if (line.startsWith("/quit")) {`
	`  break;`
	`  }`
	`  for (int i = 0; i < maxClientsCount; i++) {`
	`  if (threads[i] != null) {`
	`  threads[i].os.println("<" + name + "> " + line);`
	`  }`
	`  }`
	`  }`
	`  }`

The **while (true)** loop will execute until it receives **"/quit"** command from the input stream. Suppose the **"/quit"** command never arrives or it arrives after a very long time. The thread executing this loop inside the **synchronized(this){}** statement will block all other threads from executing their synchronized code because they will wait at their **synchronized(this){}** statements. For example, the portion of code in red (see below) will be never executed by Thread 2, if Thread 1 entered the **while (true)** loop and stays in forever.

	Thread 1	Thread 2

6	`synchronized(this) {`
	`  while (true) {`
	`  String line = is.readLine();`
	`  if (line.startsWith("/quit")) {`
	`  break;`
	`  }`
	`  for (int i = 0; i < maxClientsCount; i++) {`
	`  if (threads[i] != null) {`
	`  threads[i].os.println("<" + name + "> " + line);`
	`  }`
	`  }`
	`  }`
	`  }`	

10		`synchronized(this) {`
		`  for (int i = 0; i < maxClientsCount; i++)`
		`  if (threads[i] == this) {`
		`  threads[i] = null;`
		`  }`
		`  }`
		`  }`

Thus, when synchronizing programs, an appropriate solution must be implemented to solve such issues, otherwise the **synchronized(this){}** statement can cause very long delays and even deadlocks.

And certainly, you have to avoid putting unnecessary **synchronized(this){}** statements in the program. For example, it is not necessary to synchronize the portion 2 of code (see the table of the partitioned code). Even if this code modifies **threads[]** array, a better inspection of the code discovers that there is no risk this modification will create null pointer exceptions or other problems to the program.

The synchronized version of the chat server

In this section we present the updated version of the chat server that fixes the synchronization issues described in the previous section. The **synchronized(this){}** statement is used to solve the synchronization issues.

Also, this version of chat server is improved to deliver private messages to clients.

```
//Example 26 (updated)

import java.io.DataInputStream;
import java.io.PrintStream;
import java.io.IOException;
import java.net.Socket;
import java.net.ServerSocket;

/*
 * A chat server that delivers public and private messages.
 */
public class MultiThreadChatServerSync {

  // The server socket.
  private static ServerSocket serverSocket = null;
  // The client socket.
  private static Socket clientSocket = null;

  // This chat server can accept up to maxClientsCount clients' connections.
  private static final int maxClientsCount = 10;
  private static final clientThread[] threads = new clientThread[maxClientsCount];

  public static void main(String args[]) {

    // The default port number.
    int portNumber = 2222;
    if (args.length < 1) {
      System.out.println("Usage: java MultiThreadChatServerSync <portNumber>\n"
          + "Now using port number=" + portNumber);
    } else {
      portNumber = Integer.valueOf(args[0]).intValue();
    }

    /*
     * Open a server socket on the portNumber (default 2222). Note that we can
     * not choose a port less than 1023 if we are not privileged users (root).
     */
    try {
      serverSocket = new ServerSocket(portNumber);
    } catch (IOException e) {
      System.out.println(e);
    }

    /*
     * Create a client socket for each connection and pass it to a new client
     * thread.
     */
    while (true) {
      try {
        clientSocket = serverSocket.accept();
        int i = 0;
        for (i = 0; i < maxClientsCount; i++) {
          if (threads[i] == null) {
            (threads[i] = new clientThread(clientSocket, threads)).start();
            break;
          }
        }
        if (i == maxClientsCount) {
          PrintStream os = new PrintStream(clientSocket.getOutputStream());
          os.println("Server too busy. Try later.");
          os.close();
          clientSocket.close();
        }
      } catch (IOException e) {
        System.out.println(e);
      }
    }
  }
}

/*
 * The chat client thread. This client thread opens the input and the output
 * streams for a particular client, ask the client's name, informs all the
 * clients connected to the server about the fact that a new client has joined
 * the chat room, and as long as it receive data, echos that data back to all
 * other clients. The thread broadcast the incoming messages to all clients and
 * routes the private message to the particular client. When a client leaves the
 * chat room this thread informs also all the clients about that and terminates.
 */
class clientThread extends Thread {

  private String clientName = null;
  private DataInputStream is = null;
  private PrintStream os = null;
  private Socket clientSocket = null;
  private final clientThread[] threads;
  private int maxClientsCount;
```

```
public clientThread(Socket clientSocket, clientThread[] threads) {
  this.clientSocket = clientSocket;
  this.threads = threads;
  maxClientsCount = threads.length;
}

public void run() {
  int maxClientsCount = this.maxClientsCount;
  clientThread[] threads = this.threads;

  try {
    /*
     * Create input and output streams for this client.
     */
    is = new DataInputStream(clientSocket.getInputStream());
    os = new PrintStream(clientSocket.getOutputStream());
    String name;
    while (true) {
      os.println("Enter your name.");
      name = is.readLine().trim();
      if (name.indexOf('@') == -1) {
        break;
      } else {
        os.println("The name should not contain '@' character.");
      }
    }

    /* Welcome the new the client. */
    os.println("Welcome " + name
        + " to our chat room.\nTo leave enter /quit in a new line.");
    synchronized (this) {
      for (int i = 0; i < maxClientsCount; i++) {
        if (threads[i] != null && threads[i] == this) {
          clientName = "@" + name;
          break;
        }
      }
      for (int i = 0; i < maxClientsCount; i++) {
        if (threads[i] != null && threads[i] != this) {
          threads[i].os.println("*** A new user " + name
              + " entered the chat room !!! ***");
        }
      }
    }
    /* Start the conversation. */
    while (true) {
      String line = is.readLine();
      if (line.startsWith("/quit")) {
        break;
      }
      /* If the message is private sent it to the given client. */
      if (line.startsWith("@")) {
        String[] words = line.split("\\s", 2);
        if (words.length > 1 && words[1] != null) {
          words[1] = words[1].trim();
          if (!words[1].isEmpty()) {
            synchronized (this) {
              for (int i = 0; i < maxClientsCount; i++) {
                if (threads[i] != null && threads[i] != this
                    && threads[i].clientName != null
                    && threads[i].clientName.equals(words[0])) {
                  threads[i].os.println("<" + name + "> " + words[1]);
                  /*
                   * Echo this message to let the client know the private
                   * message was sent.
                   */
                  this.os.println(">" + name + "> " + words[1]);
                  break;
                }
              }
            }
          }
        }
      } else {
        /* The message is public, broadcast it to all other clients. */
        synchronized (this) {
          for (int i = 0; i < maxClientsCount; i++) {
            if (threads[i] != null && threads[i].clientName != null) {
              threads[i].os.println("<" + name + "> " + line);
            }
          }
        }
      }
    }
    synchronized (this) {
      for (int i = 0; i < maxClientsCount; i++) {
        if (threads[i] != null && threads[i] != this
            && threads[i].clientName != null) {
          threads[i].os.println("*** The user " + name
              + " is leaving the chat room !!! ***");
        }
      }
```

```
        }
        os.println("*** Bye " + name + " ***");

        /*
         * Clean up. Set the current thread variable to null so that a new client
         * could be accepted by the server.
         */
        synchronized (this) {
          for (int i = 0; i < maxClientsCount; i++) {
            if (threads[i] == this) {
              threads[i] = null;
            }
          }
        }
        /*
         * Close the output stream, close the input stream, close the socket.
         */
        is.close();
        os.close();
        clientSocket.close();
      } catch (IOException e) {
      }
    }
  }
```

Compiling and running the application

To try this application you have to compile the two programs: **Example 25** and **Example 26 (updated)**.

Save these programs on your computer. Name the files **MultiThreadChatClient.java** and **MultiThreadChatServerSync.java**. Open a shell window on your computer and change the current directory to the directory where you saved these files.

Type the following two commands in the shell window.

```
javac MultiThreadChatServerSync.java
javac MultiThreadChatClient.java
```

If java compiler is installed on your computer and the PATH variable is configured for the shell to find **javac** compiler, then these two command lines will create two new files in the current directory : the files **MultiThreadChatServerSync.class** and **MultiThreadChatClient.class**

Start the server in the shell window using the command:

```
java MultiThreadChatServerSync
```

You will see the following message in this window

```
Usage: java MultiThreadChatServerSync <portNumber>
Now using port number=2222
```

telling you that the chat server is started and that it is listening for connections on port number 2222. The phrase **Usage: java MultiThreadChatServerSync <portNumber>** tells you that you can start the server specifying a parameter - the port number. By default, however, the port 2222 is used.

Open a new shell window and change the current directory to the directory where you saved the application files.

Start the client in the shell window using the command:

```
java MultiThreadChatClient
```

You will see the following message in this window

```
Usage: java MultiThreadChatClient <host> <portNumber>
Now using host=localhost, portNumber=2222
Enter your name.
```

telling you that the client is started.

Type, for example, the name **Anonymous1** in this window. You will see the following output.

```
Hello Anonymous1 to our chat room.
To leave enter /quit in a new line
```

telling you that the client **Anonymous1** entered the chat room. It tells you also that to quit the chat room the client has to enter **/quit** command.

Open one more shell window and change the current directory to the directory where you saved the application files.

Start a new client client in the shell window using the command:

```
java MultiThreadChatClient
```

You will see the following message in this window

```
Usage: java MultiThreadChatClient <host> <portNumber>
Now using host=localhost, portNumber=2222
Enter your name.
```

telling you that the client is started. Now you have two clients connected to the server.

Type, for example, the text **Anonymous2** in this window. You will see the following output.

```
Hello Anonymous2 to our chat room.
To leave enter /quit in a new line
```

telling you that the client **Anonymous2** entered the chat room. It tells you also that to quit the chat room the client has to enter **/quit** command.

In the window of the client **Anonymous1** the following message will be printed.

```
*** A new user Anonymous2 entered the chat room !!! ***
```

If we enter now a message in any of the client window the message will be printed also in the window of the other client. This kind of message exchange is a chat session.

A chat session example

Below we show a possible scenario of a chat session between the two clients.

Chat client Anonymous1	Chat client Anonymous2

`$ java MultiThreadChatClient`
`Usage: java MultiThreadChatClient`
`Now using host=localhost, portNumber=2222`
`Enter your name.`
`Anonymous1`
`Hello Anonymous1 to our chat room.`
`To leave enter /quit in a new line`	

	`$ java MultiThreadChatClient`
	`Usage: java MultiThreadChatClient`
	`Now using host=localhost, portNumber=2222`
	`Enter your name.`
	`Anonymous2`
	`Hello Anonymous2 to our chat room.`
	`To leave enter /quit in a new line`

`*** A new user Anonymous2 entered the chat room !!! ***`	
`Hello Anonymous2` `<Anonymous1> Hello Anonymous2`	
	`<Anonymous1> Hello Anonymous2`
	`Hi Anonymous1` `<Anonymous2> Hi Anonymous1`
`<Anonymous2> Hi Anonymous1`	
`How are you ?` `<Anonymous1> How are you ?`	
	`<Anonymous1> How are you ?`

	I am well <Anonymous2> I am well
<Anonymous2> I am well	
	And you ? <Anonymous2> And you ?
<Anonymous2> And you ?	
I am fine. <Anonymous1> I am fine.	
	<Anonymous1> I am fine.
	Bye <Anonymous2> Bye
<Anonymous2> Bye	
Bye Anonymous2 <Anonymous1> Bye Anonymous2	
	<Anonymous1> Bye Anonymous2
	/quit *** Bye Anonymous2 ***
*** The user Anonymous2 is leaving the chat room !!! ***	
/quit *** Bye Anonymous1 ***	

Conclusions

Java sockets API (Socket and ServerSocket classes) is a powerful and flexible interface for network programming of client/server applications.

On the other hand, Java threads is another powerful programming framework for client/server applications. Multi-threading simplifies the implementation of complex client/server applications. However, it introduces synchronization issues. These issues are caused by the concurrent execution of critical sections of the program by different threads.

The **synchronized(this){}** statement allows us to synchronize the execution of the critical sections. Using this statement, however, requires a good understanding of the synchronization issues. The incorrect use of **synchronized(this){}** statement can cause other problems, such as deadlocks and/or performance degradation of the program.

Exercises

1. Try the examples presented in this document.
2. In the Example 26 (the updated version), the portion 2 of the server code is not synchronized. It uses, however, the **threads[]** array that causes synchronization issues. Why the portion 2 is not a critical section ?
3. Modify the multi-threaded chat server program to use a **ArrayList** or a **LinkedHashSet** instead of the **threads[]** array. Should the portion 2 of the server code be synchronized or not in this modification of the program ? Why ?
4. Try to do the same modification of the program but using **Vector** or **HashMap** instead of the **threads[]** array. Should the portion 2 of the server code be synchronized or not in this modification of the program ? Why ?

Useful links and credits:

1. Sockets programming in Java: A tutorial

Last Revised:

Back to Concurrent Programming Page