SPL - PS11

Multiple Client Server and Java New-IO (nio) classes

The Client-Server Model

A s**ocket connection** based on top of a **TCP-connection** is **symmetric** between the two ends (except for the connection establishment stage)

To establish a connection, the model determines that:

1. The **server waits** for connection requests from clients.

2. The **server** only **reacts** to the **initiative** of the **client.**

3. To **remain available** for further connections, the server does not open the connection directly with the client on the incoming socket. Instead, it creates a **private socket** which the client can keep connected for the whole period of the session.

Multiple-Client Server

A server which can **handle only one client** at a time is not such a good server (last week example).

We would like to **handle multiple clients** at the same time.

We do it by starting a **new thread** for each incoming connection.

We use **ConnectionHandler thread** to handle the connections and leave the **server's thread** to **keep listening** for incoming connections.

Multiple-Client Server .Vs. one-Client Server

EchoServer

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

public static void main(String[] args) throws IOException {
    // Get port
    int port = Integer.decode(args[0]).intValue();

    EchoServer echoServer = new EchoServer(port);

    // Listen on port
    try {
        echoServer.initialize();
    } catch (IOException e) {
        System.out.println("Failed to initialize on port " + port);
        System.exit(1);
    }

    // Process messages from client
    try {
        echoServer.process();
    } catch (IOException e) {
        System.out.println("Exception in processing");
        echoServer.close();
        System.exit(1);
    }

    System.out.println("Client disconnected - bye bye...");

    echoServer.close();
}
```

Multiple-Client Server

MultipleClientProtocolServer

```java
public static void main(String[] args) throws IOException {
    // Get port
    int port = Integer.decode(args[0]).intValue();

    // Create server object
    MultipleClientProtocolServer server = new MultipleClientProtocolServer(port, new
                                                                        EchoProtocolFactory());
    // Run server in independent thread
    Thread serverThread = new Thread(server);
    serverThread.start();

    try {
         serverThread.join();
    } catch (InterruptedException e) {
         System.out.println("Server stopped");
    }
}
```

Multiple-Client Server
EchoServer

```
class EchoServer {
   private BufferedReader in;
   private PrintWriter out;
   ServerSocket echoServerSocket;
   Socket clientSocket;
   int listenPort;

   public EchoServer(int port) {
     in = null;
     out = null;
     echoServerSocket = null;
     clientSocket = null;
     listenPort = port;
   }

  // Starts listening
  public void initialize() throws IOException {
    // Listen
    echoServerSocket = new ServerSocket(listenPort);
    System.out.println("Listening...");

    // Accept connection
    clientSocket = echoServerSocket.accept();
    System.out.println("Accepted connection from client!");
    System.out.println("The client is from: " + clientSocket.getInetAddress() + ":" + clientSocket.getPort());

    // Initialize I/O
    in = new BufferedReader(new InputStreamReader(clientSocket.getInputStream(),"UTF-8"));
    out = new PrintWriter(clientSocket.getOutputStream(), true);
    System.out.println("I/O initialized");
  }
```

Multiple-Client Server
MultipleClientProtocolServer

```
Public class MultipleClientProtocolServer implements Runnable {
  ServerSocket serverSocket;
  ProtocolFactory protocolFactory;
  int listenPort;

  public MultipleClientProtocolServer(int port, ProtocolFactory pf) {
    serverSocket = null;
    listenPort = port;
    protocolFactory = pf;
  }
  public void run() {
    try {
        serverSocket = new ServerSocket(listenPort);
        System.out.println("Listening...");
    } catch (IOException e) {
        System.out.println("Cannot listen on port " + listenPort);
    }
    while (true) {
      Try {       // Accept connection (serverSocket.accept() returns new object)
         ConnectionHandler newConnection = new ConnectionHandler(serverSocket.accept()
                                                                , protocolFactory.create());

         // run newConnection in new thread
         new Thread(newConnection).start();

      }catch (IOException e) {
         System.out.println("Failed to accept on port " + listenPort);
      }
    }
  }
  // Closes the connection
  public void close() throws IOException { serverSocket.close(); }
}
```

Multiple-Client Server
MultipleClientProtocolServer

```
public class ConnectionHandler implements Runnable {
   private BufferedReader in;
   private PrintWriter out;
   Socket clientSocket;
   ServerProtocol protocol;

   public ConnectionHandler(Socket acceptedSocket, ServerProtocol p) {
      in = null;
      out = null;
      clientSocket = acceptedSocket;
      protocol = p;
      System.out.println("Accepted connection from client!");
      System.out.println("The client is from: " + acceptedSocket.getInetAddress() + ":" + acceptedSocket.getPort());
   }

   public void run() {
        Try {
             initialize();
        } catch (IOException e) {
             System.out.println("Error in initializing I/O");  }
        Try {
             process();
        } catch (IOException e) {
             System.out.println("Error in I/O");
        }
        System.out.println("Connection closed - bye bye...");
        close();
   }

   public void initialize() throws IOException { // Initialize I/O
      in = new BufferedReader(new InputStreamReader(clientSocket.getInputStream()));
      out = new PrintWriter(clientSocket.getOutputStream(), true);
      System.out.println("I/O initialized");
   }
```

Multiple-Client Server

EchoServer

```java
public void process() throws IOException {
    String msg;
    while ((msg = in.readLine()) != null) { //in = new BufferedReader(new
                                            //      InputStreamReader(clientSocket.getInputStream(), "UTF-8"));
        System.out.println("Received \"" + msg + "\" from client");
        if (msg.equals("bye")) {
            out.println("Ok, bye bye..."); //out = new PrintWriter(clientSocket.getOutputStream(), true);
            break;
        } else {
             out.println("You sent me \"" + msg + "\". I really appreciate it.");
        }
    }
}
```

MultipleClientProtocolServer

```java
public void process() throws IOException {
    String msg;
    while ((msg = in.readLine()) != null) {
        System.out.println("Received \"" + msg + "\" from client");
        String response = protocol.processMessage(msg);
        if (response != null) {
            out.println(response);
        }
        if (protocol.isEnd(msg)) {
            break;
        }
    }
}
```

Multiple-Client Server
EchoServer

```
public void process() throws IOException {
    String msg;
    while ((msg = in.readLine()) != null) { //in = new BufferedReader(new
                                            //      InputStreamReader(clientSocket.getInputStream(), "UTF-8"));
        System.out.println("Received \"" + msg + "\" from client");
        if (msg.equals("bye")) {
            out.println("Ok, bye bye..."); //out = new PrintWriter(clientSocket.getOutputStream(), true);
            break;
        } else {
             out.println("You sent me \"" + msg + "\". I really appreciate it.");
        }
    }
}
```

MultipleClientProtocolServer

```
class EchoProtocol implements ServerProtocol {
  private int counter;
  public EchoProtocol() {
    counter = 0;
  }
  public String processMessage(String msg) {
    counter++;
    if (isEnd(msg)) {
      return new String("Ok, bye bye...");
    } else {
      return new String(counter + ". Received \"" + msg + "\" from client");
    }
  }
  public boolean isEnd(String msg) {
    return msg.equals("bye");
  }
}
```

Non-blocking IO

If we examine our **client**, once we reach the **following code:**

```
msg = userIn.readLine()
```

the program is **blocked** until the **user press enter**. **No data** can be **received through** the **network buffer**.

We want a **non-blocking i/o** handling so that the **network buffer** will keep **working**, even when we get to the above code.

Also, we would like to do so **without** using more than **one thread**.

Why?

Non-blocking IO

If we examine our **client**, once we reach the **following code:**

```
msg = userIn.readLine()
```

the program is **blocked** until the **user press enter**. **No data** can be **received through** the **network buffer**.

We want a **non-blocking i/o** handling so that the **network buffer** will keep **working**, even when we get to the above code.

Also, we would like to do so **without** using more than **one thread**.

Why?

chat application
The client is expected to keep receiving messages, even when the user is typing a new message.

Another example could be a **client** that needs to get **updates from several servers**: it should try reading from all of them without blocking.

Java NIO API

Java's **java.nio** package is a **new**, efficient IO package. It also support **Non-blocking IO.**

The key players we need to know are:

- Channels
- Buffers
- Selector

Java NIO API

Channels, a new primitive I/O abstraction

java.nio.channels.Channel is an **interface**

Channels (classes implementing the interface) are designed to provide for bulk data transfers **to** and **from** **NIO buffers** (**A Channel is something you can read from and write to**).

Channels can be either **blocking** (by default) **or** **non-blocking**.

socket channels

allow for **data transfer** between **sockets** and **NIO buffers**.

java.nio.channels.SocketChannel **(similar to Socket)**
java.nio.channels.ServerSocketChannel **(similar to ServerSocket)**

- Here **read()**, **write()** and **accept()** methods can be **non-blocking.**

- The ServerSocketChannel's **accpet()** method returns a SocketChannel.

- The other side of the connection can be either **blocking** **or** **non-blocking** (it doesn't matter).

Java NIO API

socket channels

Setting up a non-blocking ServerSocketChannel listening on a specific port:

```
int port = 9999;
ServerSocketChannel ssChannel = ServerSocketChannel.open();
ssChannel.configureBlocking(false);  // false for non-blocking
ssChannel.socket().bind( new InetSocketAddress(port) );
```

Setting up a non-blocking SocketChannel and connecting to a server:

```
SocketChannel sChannel = SocketChannel.open();
sChannel.connect( new InetSocketAddress("localhost", 1234) );
sChannel.configureBlocking(false); // false for non-blocking
```

Java NIO API

NIO buffers

NIO data transfer is based on **buffers** (**java.nio.Buffer** and related classes).

A **buffer** is a **container** that can hold a **finite and contiguous sequence of primitive** data types. It's essentially an **object wrapper** around an **array of bytes** with imposed limits.

Using the right implementation, allows the **buffer contents** to **occupy** the same **physical memory** used by the **operating system** for its native I/O operations, thus **allowing the most direct transfer mechanism**, and eliminating the need for any additional copying.

In most operating systems (provided some properties of the particular area of memory) **transfer** can take place **without using the CPU** at all.
The **NIO buffer** is intentionally **limited in features** in order to support these goals.

In simple words – allows faster and more efficient implementations

NIO buffers maintain several **pointers** that dictate the function of its accessor methods.
The NIO buffer implementation contains a rich set of methods for **modifying these pointers:**

flip()
get()
put()
mark()
reset()

Java NIO API

NIO buffers

Channels know how to read and write into Buffers, and buffers can read and write into other buffers.

We'll be using **ByteBuffer** (These are buffers that hold bytes).

Creating a new ByteBuffer:

```
final int NUM_OF_BYTES = 1024;
ByteBuffer buf = ByteBuffer.allocate(NUM_OF_BYTES);
```

Creating a ByteBuffer from a an array of bytes:

```
byte[] bytes = new byte[10];
ByteBuffer buf = ByteBuffer.wrap(bytes);
```

Java NIO API

Capacity pointer - is the maximum number of items a buffer can hold
position pointer – points to "start"
limit pointer – value that ranges from zero to capacity, representing an arbitrary limitation for the buffer

```
ByteBuffer buffer = ByteBuffer.allocate(512); //creation of a nondirect ByteBuffer
String str = "Hello";
byte[] data = str.getBytes();
buffer.put(data); //add the string "Hello" to the byte buffer
```

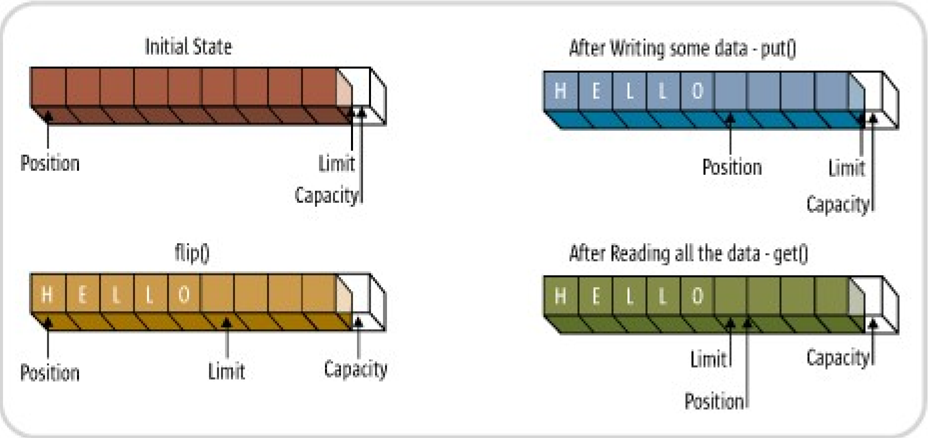

Figure 1 Buffer properties

The **position pointer** now points to the **next empty cell** after the data.

So if we are to use this **buffer to read** the data, we need to **flip the position** of the position pointer.

```
buffer.flip(); //readies the buffer for draining by resetting the position and limit pointer
int limit = buffer.limit();
byte[] data = new byte[limit];
buffer.get(data);
System.out.println(new String(data));
```

After the **flip() is called**, the **position pointer** points to the **first cell**, and the **limit pointer** points to the cell where the **position pointer used to point** before the flip() method was called.

Java NIO API

NIO buffers

The flip() method

position pointer – points to “start"
limit pointer – points to “size" or “end"

The **flip()** method, **moves** the **position pointer** to the **origin** of the underlying array (if any) and the **limit pointer** to the **former position** of the **position pointer**.

Each **buffer** has a size and a position marker.
A **read** operation **reads** a specified number of **bytes** from the **current position**, and **updates** the **position marker** to point to the yet **unread bytes**.

Similarly, a write operation writes some bytes from the current position, and then advances the position.

You can't read or write more than the size of the buffer.

This means that if someone wrote some bytes into a buffer, and then you want to read them, you need to set the position marker to the start of the buffer, and the size of the buffer to the former position.

Java NIO API

Character sets

In Java, a character set is a mapping between Unicode characters (or a subset of them) and bytes.

The **java.nio.charset** package of NIO provides facilities for identifying character sets and providing encoding and decoding algorithms for new mappings.

```
Charset charset = Charset.forName("UTF-8");
_decoder = charset.newDecoder();
_encoder = charset.newEncoder();
```

Java NIO API

From Channel to Buffer and back

Reading from a channel to a buffer:

```
numBytesRead = _socketChannel.read(buf); //returning number of bytes read
```

Writing from a buffer to a channel:

```
numBytesWritten = _socketChannel.write(buf); //returning number of bytes written
```

If read or write **returns -1**, it means that the **channel is closed**.

Read and write operations on Buffers **update the position marker** accordingly.

Echo nio client -
notice that in this example we still use blocking io, but we're getting really close to non blocking io

```
import tokenizer.StringMessageTokenizer;
import tokenizer.FixedSeparatorMessageTokenizer;

public class NIOEchoClient {

  public static void main(String[] args) throws IOException {
      final int NUM_OF_BYTES = 1024;
      SocketChannel sChannel = null; // the connection socket

      // used for reading
      ByteBuffer inbuf = ByteBuffer.allocate(NUM_OF_BYTES);
      // the message tokenizer accumulate bytes until it has a complete message.
      StringMessageTokenizer tokenizer = new
                          FixedSeparatorMessageTokenizer("\n",Charset.forName("UTF-8"));

      // Get host and port
      String host = args[0];
      int port = Integer.decode(args[1]).intValue();
      System.out.println("Connecting to " + host + ":" + port);

      // connect to the server
      Schannel = SocketChannel.open();
      Try {
          sChannel.connect( new InetSocketAddress(host, port) );
          sChannel.configureBlocking(true); // we still use blocking io here
          System.out.println("Connected to - "+host);
      } catch(IOException e){
          System.out.println("Failed to connected to - "+host);
          sChannel.close();
          System.exit(1);
      }
      System.out.println("Connected to server!");
```

Echo nio client - continue

```
    // initialize user input, and start main loop:
    String msg;
    boolean b = true;
    BufferedReader userIn = new BufferedReader( new InputStreamReader(System.in) );

    while (b && (msg = userIn.readLine())!= null) {
        msg += "\n"; //make sure to add the end of line
        // write the line to the server
        ByteBuffer outbuf = ByteBuffer.wrap(msg.getBytes("UTF-8"));
        while (outbuf.remaining() > 0) {
            sChannel.write(outbuf); //Writing from outbuf to sChannel
        }
        // read a line from the server and print it
        while (!tokenizer.hasMessage()) {
            inbuf.clear(); //The position is set to zero, the limit is set to the capacity

            sChannel.read(inbuf);//An attempt is made to read up to r bytes from the channel,
                                 // (where r is the number of bytes remaining in the buffer)

            inbuf.flip();//The limit is set to the current position and then the position
                         //  is set to zero
            tokenizer.addBytes(inbuf);
        }

        System.out.println(tokenizer.nextMessage()); // write the line to the screen
        if (msg.equals("bye\n")){
            System.out.println("Client exit...");
            b = false;
        }
     }
     System.out.println("Exiting...");
     // Close all I/O
     sChannel.close();
     userIn.close();
  }
}
```

Tokenizer

```
public class FixedSeparatorMessageTokenizer implements StringMessageTokenizer {
    private final String _messageSeparator;
    private final StringBuffer _stringBuf = new StringBuffer();
    private final CharsetDecoder _decoder;
    private final CharsetEncoder _encoder;

    public FixedSeparatorMessageTokenizer(String separator, Charset charset) {
        this._messageSeparator = separator;   // we used "\n"
        this._decoder = charset.newDecoder(); // we used Charset.forName("UTF-8")
        this._encoder = charset.newEncoder();
    }

    /**
     * Add some bytes to the message.
     * Bytes are converted to chars, and appended to the internal StringBuffer.
     * Complete messages can be retrieved using the nextMessage() method.
     *
     * @param bytes an array of bytes to be appended to the message.
     */
    public synchronized void addBytes(ByteBuffer bytes) {
        //remaining(): Returns the number of elements between the current position and the limit
        CharBuffer chars = CharBuffer.allocate(bytes.remaining());
        this._decoder.decode(bytes, chars, false); // false: more bytes may follow. Any
                                                   unused bytes are kept in the decoder.

        chars.flip(); //The limit is set to the current position and then the position
                                                                  is set to zero
        this._stringBuf.append(chars);
    }

    public synchronized boolean hasMessage() { //Is there a complete message ready?
        return this._stringBuf.indexOf(this._messageSeparator) > -1;
    }
```

Tokenizer - continue

```java
    /**
     * Get the next complete message if it exists, advancing the tokenizer to the next message.
     * @return the next complete message, and null if no complete message exist.
     */
    public synchronized String nextMessage() {
        String message = null;
        int messageEnd = this._stringBuf.indexOf(this._messageSeparator);
        if (messageEnd > -1) {
            message = this._stringBuf.substring(0, messageEnd);
            this._stringBuf.delete(0, messageEnd+this._messageSeparator.length());
        }
        return message;
    }


    /**
     * Convert the String message into bytes representation, taking care of encoding and framing.
     *
     * IGNORE THIS ONE IN TIRGUL 11
     *
     * @return a ByteBuffer with the message content converted to bytes, after framing information has been added.
     */
    public ByteBuffer getBytesForMessage(String msg) throws CharacterCodingException {
        StringBuilder sb = new StringBuilder(msg);
        sb.append(this._messageSeparator);
        ByteBuffer bb = this._encoder.encode(CharBuffer.wrap(sb));
        return bb;
    }

}
```

Java NIO API

Selectors (we will see it next week)

A selector (**java.nio.channels.Selector** and subclasses) provides a mechanism for **waiting on channels** and **recognizing** when one or more **become available for data transfer.**

When a number of channels are registered with the selector, it enables blocking of the program flow until at least one channel is ready for use, or until an interruption condition occurs.

Although this multiplexing behavior could be implemented with Java threads, the selector can provide a significantly more efficient implementation using native platform threads or, more likely, even lower-level operating system constructs. A POSIX-compliant operating system, for example, would have direct representations of these concepts, select(). A notable application of this design would be the common paradigm in server software which involves simultaneously waiting for responses on a number of sessions.