SWitCH – Computing systems and networks (SCOMRED)

2018/2019

Laboratory Script – Week 06

IMPERIOUS: After each practical activity (in yellow), students are supposed to reflect on results, and wait for the teacher's acknowledgment before progressing to the next practical activity in this script.

1. HTTP server and Web UI

The HTTP protocol aims at transferring contents between applications, it's based on the client-server model and operates over a TCP connection established by the HTTP client towards the HTTP server.

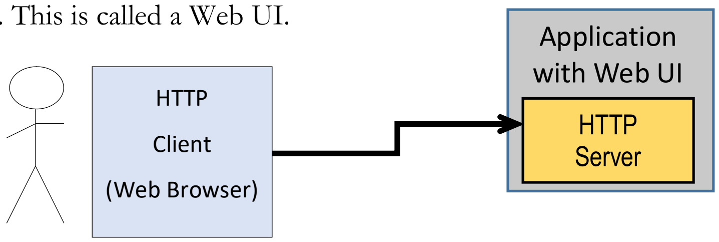
Once the TCP connection is established the HTTP client sends a request (HTTP request) using one method (HTTP request type, e.g. GET, POST, PUT) directed to a server side resource identified by a URI, in response the server sends back the resource's content.

Server side resources may be static or dynamic, static resources are most often stored in files, so in that case the URI referrers to a file and the server simply replies back with the file contents. Dynamic contents are generated by the server on runtime and unlike static resources they will be different for different requests over the same URI.

Contents transfer is not limited to server to client direction, clients may also send a content to the server. Requests with the GET method have no body, so in this case client to server data transfer is rather limited, yet some data may be embedded in the URI, usually called the query string.

Requests with POST and PUT methods may have a body, so by using them, any sort of content can be transferred from the client to the server.

The HTTP protocol and all kinds of contents that can be transferred by using it (HTML, images, etc.) can be used to offer a Graphical User Interface (GUI) by an application, to achieve that a small **HTTP server** must be included in the application. This is called a Web UI.



Here, small **HTTP server** stands for an HTTP server that does not fully implement the HTTP protocol in all its features, but only features required for the specific use of that application.

HTTP messages processing

All HTTP messages (requests and responses) have the same general format, the first part (header) is made of a variable number of variable length text lines, each line ending with CR/LF. An empty line informs the header has ended, and thus, if it exists, next it's the message's body. If a body exists, then, by now, whoever is reading the messages (client or server) is aware of that because has received header lines about the body, in special the **Content-length** header field.

2. A simple HTTP server implementation in Java (HTTPmessage.java)

We are now going to add a small HTTP server to our **TCP Chat server** project so that users are able to interact with it by using a standard Web Browser. For that purpose we are going to create a new class named **HTTPmessage**.

Starting by the most basic features, we should first address the issue of reading and writing variable length text lines in HTTP format (CR/LF terminated). Concerning writing, there are no big issues, we known the text line length so it's just a matter of writing the line through the TCP connection (Output stream):

```java
public class HTTPmessage {

    private static final int CR=13;
    private static final int LF=10;

    private static final String VERSION="HTTP/1.1";

    private static final String CONTENT_TYPE="Content-type:";
    private static final String CONTENT_LENGTH="Content-length:";
    private static final String CONNECTION="Connection:";

    private static final String[][] knownFileExt = {
            { ".pdf" , "application/pdf" } ,
            { ".js" , "application/javascript" } ,
            { ".txt" , "text/plain" } ,
            { ".gif" , "image/gif" } ,
            { ".png" , "image/png" }
    };

    static private void writeHeaderLine(DataOutputStream out, String line)
        throws IOException {
        out.write(line.getBytes(), 0, line.length());
        out.write(CR); out.write(LF);
    }
```

The **writeHeaderLine()** method receives as arguments the **DataOutputStream** where to write the line and the text line to be written as a **String**.

Reading is not so straightforward because we don't know in advance how long the text line to be received is. The only possible solution is reading bytes one by one until CR/LF is received:

```java
static private String readHeaderLine(DataInputStream in) throws IOException {
    String ret="";
    int val;
    do {
        val=in.read();
        if(val==-1) throw new IOException();
        if(val!=CR) ret=ret+(char)val;
    }
    while(val!=CR);
    val=in.read(); // read LF
    if(val==-1) throw new IOException();
    return ret;
}
```

The **readHeaderLine()** method receives as argument the **DataInputStream** from where the text line is to be read and returns the received text line as a **String**. The **read()** method is used to read one byte at a time and add it to the string until CR is received. If the **read()** method returns **-1**, that stands for an error an thus, an **IOException** in created an sent to the calling method.

Both these methods are defined as static in our **HTTPmessage** class and they are to be used by the non-static part of the class.

The non-static part has several attributes representing the message, they may represent the properties of an HTTP message that has just been received or of a message to be sent.

```
//// NON-STATIC (INSTANCE) ELEMENTS

private boolean isRequest;
private String method;
private String uri;
private String status;

private String contentType;
private byte[] content;
```

Two constructors are defined, the **HTTPmessage(DataInputStream in)** receives as argument a **DataInputStream** from where to read the message to be store in the new object.

```
public HTTPmessage(DataInputStream in) throws IOException {
    String firstLine=readHeaderLine(in);
    isRequest= !firstLine.startsWith("HTTP/");
    method=null;
    uri=null;
    content=null;
    status=null;
    contentType=null;

    String[] firstLineComp=firstLine.split(" ");
    if(isRequest) {
        method=firstLineComp[0];
        uri=firstLineComp[1];
    }
    else {  // response
        status=firstLineComp[1] + " " + firstLineComp[2];
    }

    String headerLine;

    do {
        headerLine=readHeaderLine(in);
        if(headerLine.toUpperCase().startsWith(CONTENT_TYPE.toUpperCase())) {
            contentType=headerLine.substring(CONTENT_TYPE.length()).trim();
        }
        else
        if(headerLine.toUpperCase().startsWith(CONTENT_LENGTH.toUpperCase())) {
            String cLen=headerLine.substring(CONTENT_LENGTH.length()).trim();
            int len;
            try { len=Integer.parseInt(cLen); }
            catch(NumberFormatException ne) { throw new IOException(); }
            content = new byte[len];
        }
    }
    while(!headerLine.isEmpty());

    // READ CONTENT
    if(content!=null) in.readFully(content,0,content.length);
}
```

This method can be used to read both HTTP requests and responses, it starts by reading the first line (request line in the case of a request or status line in the case off a response). If this line doesn't start by "HTTP/" we assume it's a request, otherwise a response.

Further processing of the first line depends on being a request or a response, for a request the method and URI are extracted and stored in the new object, for a response, the only the status code is extracted and stored in the new object.

Next we step into the header lines processing, lines are read until an empty line is received, only the **Content-type** and **Content-length** header lines are processed, other lines are silently ignored.

Once the header has finished, if there was a **Content-length** header line, then the body is read and stored in the new object.

The other constructor, **HTTPmessage()**, creates a new empty **HTTPmessage** to be later defined:

```
public HTTPmessage() {
    isRequest=true;
    method=null;
    uri=null;
    content=null;
    status=null;
    contentType=null;
}
```

Beyond several get and set interface methods to access properties, a method to send an **HTTPmessage** is also defined, if the message properties are inconsistent, then it returns **false**.

```
public boolean send(DataOutputStream out) throws IOException {
    if(isRequest) {
        if(method==null||uri==null) return false;
        writeHeaderLine(out, method + " " + uri + " " + VERSION);
    }
    else {
        if(status==null) return false;
        writeHeaderLine(out,VERSION + " " + status);
    }

    if(content!=null) {
        if(contentType!=null) writeHeaderLine(out,CONTENT_TYPE + " " + contentType);
        writeHeaderLine(out,CONTENT_LENGTH + " " + content.length);
    }
    writeHeaderLine(out,CONNECTION + " close");
    writeHeaderLine(out,"");
    if(content!=null) {
        out.write(content,0,content.length);
    }
    return true;
}
```

Depending on being a request or a response, the proper request line or status line is written through the provided **DataOutputStream**.

If the message has a content, then the corresponding **Content-type** and **Content-length** header lines are also written. To finish the header **Conection: close** and an empty line are then written.

If there's a content, then the content is sent after the header.

A message content can be set from either a string or a file (for static resources):

```
public void setContentFromString(String c, String ct) {
    content=c.getBytes(); contentType=ct;
}

public boolean setContentFromFile(String fname) {
    File f=new File(fname);
    contentType=null;
    if(!f.exists()) {
        content=null;
        return false;
    }
    for (String[] k : knownFileExt) {
        if (fname.endsWith(k[0]))
            contentType = k[1];
    }
    if(contentType==null) contentType="text/html";

    int cLen = (int) f.length();
    if(cLen==0) {
        content=null;
        contentType=null;
        return false;
    }

    content = new byte[cLen];

    DataInputStream fr;
    try {
        fr = new DataInputStream(new FileInputStream(f));
        try { fr.readFully(content,0,cLen); fr.close(); }
        catch(IOException ex) {
            System.out.println("Error reading file");
            content=null;
            contentType=null;
            return false;
        }
    }
    catch(FileNotFoundException ex)  {
        System.out.println("File Not Found");
        content=null;
        contentType=null;
        return false;
    }
    return true;
}
```

If setting the content from a string, the caller must also specify the content-type.

For files only a small number of file types (filename's extension) are supported. The default content-type in **text/html**. If the file doesn't exist or another error occurs when reading it this method returns false.

3. Adding the HTTP server to the TCP Chat project

Now that we have HTTP messages processing (very basic implementation) you can add the **HTTPmessage** class to the TCP server project and implement the server itself.

In essence an HTTP server is a TCP server, so if we have previously implemented the TCP server through classes TcpServer and TcpServerConn we are now going to add two similar classes, HttpServer and HttpServerConn, working pretty the same way.

Starting with the **main()** class (**TcpChatSrv**), we are now going to start an additional thread for HTTP connections:

```
public class TcpChatSrv {
    private static final int TCP_SERVER_PORT=9999;
    private static final int HTTP_SERVER_PORT=8080;
    private static final String WEB_ROOT_FOLDER="www";

    public static void main(String args[]) throws Exception {
        Thread tcpServer=new TcpServer(TCP_SERVER_PORT);
        tcpServer.start();
        Thread httpServer=new HttpServer(HTTP_SERVER_PORT, WEB_ROOT_FOLDER);
        httpServer.start();

        tcpServer.join();
    }
}
```

The **HttpServer** class constructor receives the port number and also the folder from where to fetch files for static contents.

The **HttpServer** class is identical to the **TcpServer** class:

```
public class HttpServer extends Thread {

    private int serverPort;
    private String webRootFolder;

    public HttpServer(int port, String root) {serverPort=port; webRootFolder=root;}

    @Override
    public void run() {
        ServerSocket sock;
        Socket cliSock;
        try { sock = new ServerSocket(serverPort); }
        catch(IOException ex) {
            System.out.println("HTTP server: local port number " + serverPort +
                         " not available - server aborted");
            return;
        }
        System.out.println("HTTP server: ready, listening on local port number " +
                                 serverPort);
        while(true) {

            try { cliSock=sock.accept(); }
            catch(IOException ex) {
                System.out.println("HTTP server: failed to accept client connection");
                cliSock=null;
            }
            if(cliSock!=null) {
                Thread cliThread = new HttpServerConn(cliSock,webRootFolder);
                cliThread.start();
                System.out.println("HTTP server: new client connection from " +
                         cliSock.getInetAddress().getHostAddress() +
                                    ", port number " + cliSock.getPort());
            }
        }
    }
}
```

The **HttpServerConn** class constructor receives a socket, connected to the client and is supposed to read an HTTP message from it and then send back an HTTP response:

```
public class HttpServerConn extends Thread {
    private Socket cli;
    private String webRootFolder;
    private DataInputStream sIn;
    private DataOutputStream sOut;

    public HttpServerConn(Socket s, String root) { cli=s; webRootFolder=root;}

    @Override
    public void run() {
        try {
            sIn = new DataInputStream(cli.getInputStream());
            sOut = new DataOutputStream(cli.getOutputStream());
            HTTPmessage request = new HTTPmessage(sIn);
            HTTPmessage response = new HTTPmessage();
            response.setResponseStatus("200 OK");

            if(request.getMethod().equals("GET")) {
                if (request.getURI().equals("/messages")) {
                    String content = "<html><body><h1>TCP Chat Messages:</h1><hr>";
                    content += ChatMessages.getLastHTML();
                    content += "<hr /></body></html>";
                    response.setContent(content, "text/html");

                }
                else { // NOT GET /messages , THEN IT MUST BE A FILE
                    String fullname = webRootFolder + "/";
                    if (request.getURI().equals("/")) fullname = fullname +
                         "index.html";
                    else fullname = fullname + request.getURI();
                    if (!response.setContentFromFile(fullname)) {
                        response.setContentFromString(
                          "<html><body><h1>404 File not found</h1></body></html>",
                                "text/html");
                        response.setResponseStatus("404 Not Found");
                    }
                }
            }
            else {
                    response.setContentFromString(
                  "<html><body><h1>ERROR: 405 Method Not Allowed</h1></body></html>",
                            "text/html");
                    response.setResponseStatus("405 Method Not Allowed");
            }
            response.send(sOut);
        }
        catch(Exception ex) { System.out.println("HTTP I/O error"); }
        try { cli.close(); } catch(Exception ex) { System.out.println(
                                 "HTTP I/O error closing client connection"); }
    }
}
```

The **HttpServerConn run()** starts by getting the streams from the socket, then receives an HTTP message by calling the **TCPmessage** constructor. An empty response HTTP message is created and the response status is set to **200 OK**, this will make this message to be a response.

For now we will only handle GET requests, so if that's not the case, a **405 Method Not Allowed** response is sent.

In the case of a GET, we will handle dynamically requests for URI **/messages**, and provide for that an HTML page with the last chat messages. With that goal, a new method **getLastHTML()** was added to class **ChatMessages**:

```java
public static synchronized String getLastHTML() {
    String res="";
    for(int i=0;i<numMessages;i++) {
        res=res+"<p>" + lastMessages[i] + "</p>";
    }
    return res;
}
```

This method simply returns a string with last chat messages, each within a <p> HTML tag.

If the GET is not for the **/messages** URI, then we assume it must be a file stored in the provided Web Root Folder.

PRACTICE:

Implement what has been described so far. You may download all files from Moodle.

In summary:

1. Pickup your TcpChatSrv project
2. Add new classes: TCPmessage; HttpServer; HttpServerConn
3. Add features to old classes: TcpChatSrv; ChatMessages

4. Setup web root folder and test

Before checking the **/messages** URI processing through the GET request, we'll check if our HTTP server is properly processing GET requests for static files.

PRACTICE:

On your project's root folder create a new directory named **www**. Download into it HTML and image files provided on Moodle. They include an **index.html** file and two image files for testing. As usually happens with HTTP servers, a GET request for the web root folder (GET /) will be replied by our HTTP server with the content of a file named **index.html**.

Start the server application (**TcpChatSrv**), it now includes an HTTP server listening on port number 8080.

Next, use your personal Web Browser to open URL **http://localhost:8080**, you should be able to see the **index.html** file content and the two referred images.

The **index.html** page provides a link to the URL **http://localhost:8080/messages**, it will open in a separate browser window/tab the list of last exchanged chat messages.

Use the old **TcpChatCli** application to post some chat messages, to see them you must then reload the **http://localhost:8080/messages** page.

For now this is all our Web UI is capable of, but that's a start. All background is settled for much more.