Introduction to computer networking

Second part of the assignment

Academic year 2014-2015

Abstract

In this assignment, students will have to implement a server application using Java Sockets.
The server uses the HTTP protocol to provide a chat service to its clients. It will use the following additional concepts: pagination, page redirection, session cookies, POST methods, chunked encoding and gzip compression.
Students attending the INFO-0010-4 course will work in teams of 2 students. Other students (attending the INFO-0010-2 course) are exempted from this part.
The deadline for this projet is the 13th of May.

Sections 1 and 2 define your assignment objectives. Section 3 and further are an executive summary of the technologies we use.

1 HTTP chat server

As stated in the abstract, you will implement a HTTP chat server using Java Sockets. The server waits for TCP connections on a given port, and can handle HTTP GET and HTTP POST requests through that connection when established.

A HTTP GET method is used to retrieve the (paginated (see Section 1.4) and possibly compressed (see Section 7)) content of the chat server, as a HTML page[1][2].
A HTTP POST (see Section 3) method is used to:

a) Send the credentials (login/password pair) of the user[3], or

b) Post a new message[4].

The server will be up for new connections (as long as there still are available threads) and be able to handle concurrency.

You don't have to write a corresponding client code: the browser is the client.

[1] this page will be named `viewPosts.html`

[2] Every HTML page returned by your server does not correspond to a file on disk, but is dynamically generated in your java source code.

[3] to a page named `identification.html`

[4] to a page named `postMessage.html`

1.1 Launch

The software is invoked on the command line with one additional argument:
`java WebServer maxThreads`
where *maxThreads* is the maximum number of java Threads that can be run in a concurrent way to handle the requests. One can also see this argument as the maximum number of requests that can be treated "simultaneously" (see section 8).

The server listens on port $80xx$ – where xx is your group ID (if you don't have one, please submit your group composition to Samuel Hiard).

1.2 Incorrect or incomplete commands

When dealing with network connections, you can never assume that the other side will behave as you expect.

It is thus your responsibility to check the validity of the client's request (and respond with the correct status code) and to ensure that a malevolent person won't make your server freeze by initiating a TCP connection and keep it open for an indefinite period of time (see Section 8 for a counter-measure).

1.3 Authentication and Sessions

User authentication is performed by the use of a login/password pair and session cookies (see Section 5).

When a user connects to the server for the first time, or after a disconnection, it has to input its credentials. If this information is valid, the server will generate a random value, which will be used as session ID. This ID is transferred to the client as a cookie, so that each time the client connects to the server using the same browser, his/her authentication remains valid.

The user has the possibility to close its session, which will invalidate the session cookie.

Credentials are hard-coded values in the server memory.
These values are: {"Leduc","mypass1"}, {"Hiard","itagpw?"} and {"Kurose","&Ross"}
No extra protection on the credentials are required (encryption, storing only the hash values, ...).

1.4 Pagination

When the content of the chat is requested, all messages are not necessarily returned.

The URL of your server contains two optional parameters: `page` (default: 1), and `maxPosts` (default: 10), where *maxPosts* is the maximum number of posts on a page, and *page* is the page number.

For instance:

- `/viewPosts.html` will return the newest 10 messages.
- `/viewPosts.html?page=2` will return messages 11 to 20 (considering that message 1 is the newest message).
- `/viewPosts.html?maxPosts=20` will return the newest 20 messages.
- `/viewPosts.html?maxPosts=5&page=3` will return messages 11 to 15 .

You should include `<A HREF=...>` tags to ease the navigation in the different message pages.

1.5 Page Redirection

On a successful authentication on `identification.html`, or after posting a message (on `postMessage.html`) the server will send a redirect order (see Section 4) to `viewPosts.html`.
A call to the root page (`"/"`) will redirect to `viewPosts.html`
A call to `viewPosts.html` without proper authentication will redirect to `identification.html`
As long as the authentication failed, the user remains on the `identification.html` web page.

1.6 Chunked encoding

The content of `viewPosts.html` will be transferred to the client by using *Chunked encoding* (See Section 6). Chunks will have a maximum size of 100 bytes.

1.7 Compression*

The content of `viewPosts.html` can be compressed using Gzip compression (see Section 7) for bonus points.

2 Guidelines

- You will implement the programs using Java 1.7 or Java 1.8, with packages `java.lang`, `java.io`, `java.net` and `java.util`,
- You will ensure that your program can be terminated at any time simply using CTRL+C, and avoid the use of ShutdownHooks
- You will not manipulate any file on the local file system.

- You will ensure your main class is named WebServer, located in WebServer.java at the root of the archive, and does not contain any `package` instruction.
- You will ensure that your program is fully functional (i.e. compilation and execution) on the student's machines in the lab (ms8xx.montefiore.ulg.ac.be).

Submissions that do not observe these guidelines could be ignored during evaluation.

Your commented source code will be delivered no later than the 13th of May to *S.Hiard@ulg.ac.be* as a .zip package.

Your program will be completed with a .pdf report (out of the zip package) addressing the following points:

Software architecture: How have you broken down the problem to come to the solution? Name the major classes responsible for requests processing.

Multi-thread coordination: How have you synchronized the activity of the different threads?

Limits: Describe the limits of your program, esp. in terms of robustness.

Possible Improvements: This is a place where you're welcome to describe missing features or revisions of these specifications that you think would make the server richer or more user-friendly.

3 The HTTP Protocol

HTTP is a client/server applicative protocol that enables *resources* to be shared between machines based on their symbolic names (the *URL*s). "Resource" is a generic term that can encompass both a file or content generated dynamically from a database (e.g. using CGI or PHP).

Main methods (i.e. actions that a client can invoke on an HTTP server) retrieve content or meta-data (i.e. content type, last modification timestamp) for a specific resource. The `GET` method retrieves both content and meta-data, while `HEAD` only retrieves meta-data.

The protocol also defines additional methods to upload content (`POST`) as well as ways to `PUT` and `DELETE` resources on a server. In this work, only `GET` and `POST` methods will be used.

As many applicative protocols defined by IETF, HTTP is a *text-oriented* protocol, meaning that all exchanges are intended to be human-readable (as opposed to "binary" protocols such as IP, TCP and BitTorrent that feature custom information packing). Each request and reply is made of multiple lines of text, usually with only one chunk of information per line.

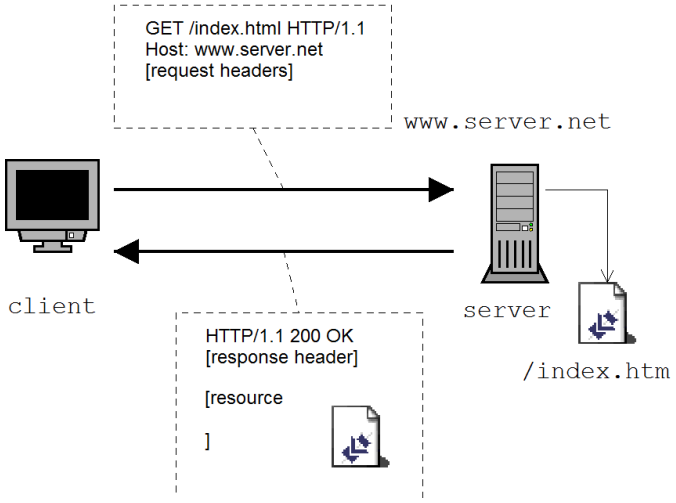

Figure 1: *A typical HTTP Request/Response cycle*

<http-request> ::= *<method-name>* *<url>* `HTTP/`*<http-version>* `CRLF`
<http-headers> `CRLF`
`CRLF`

HTTP Request Overall Syntax

`<method-name>` names the operation to take place

`<url>` indicates which resource is to be manipulated. Note that it usually only provides the *path* and *query* part of the resource location[5].

`<http-version>` is the protocol's version. We will exclusively work with version 1.1, as described in RFC2616.

`<CRLF>` are two control characters used as line terminators (\r\n in Java). Watch out: software exists that do not obey the standards and solely use the `LF` character as line terminator.

`<http-headers>` Any number of optional header lines to define preferences over content type, required freshness, etc. They follow the generic format `<option-name>: <option-value>`.

Providing extra information "one line at a time" allows fairly simple extensions of the protocol, as options that are not recognized by an entity (either client or server) can easily be discarded. The precise syntax and semantic of the HTTP headers are defined

[5]unless a Web proxy is involved

in section 14 of the RFC 2616 describing the HTTP/1.1 protocol[6].

Overall HTTP reply syntax

<http-response> ::= `HTTP/`*<http-version>* *<ret-code>* *<status>* CRLF
<http-headers> CRLF
CRLF
[response-body]

`<ret-code>` is a numerical value defining whether the request could be handled properly, that is intended to be used by the client software. The first digit indicates the success level (1=ongoing, 2=successful, 3=reiterate, 4=client-side error, 5=server-side error) while the last two digits further refine the type of error or to what extent the request could be fulfilled.

`<status>` is a human-readable message matching the `ret-code` that could be displayed to the end-user.

`<http-headers>` provide additional information on the content type, size, the encoding used to deliver it over the channel, etc.

`<response-body>` is the actual resource content (when found). Note that the body may be empty for some methods.

POST methods

POST methods differ from GET methods as some additional content is provided after the request header. As for HTTP server responses, it provides a `Content-Length` field whose value is the number of bytes to be read after the end of the header.

POST methods are usually triggered after the user clicked on a submit button of a HTML form.[7]

4 Redirection Mechanism

The redirection mechanism allows the server to request the browser to generate a new GET request without the need for the user to do anything.

[6] available at http://www.ietf.org/rfc/rfc2616.txt

[7] You can find information about HTML forms on the internet, for example: http://www.w3schools.com/html/html_forms.asp

There are at least two ways to perform a redirection:

1. Return with the code `303 See Other` instead of `200 OK` and provide a field `Location` with the location of the new webpage.
   e.g.:
   `HTTP/1.1 303 See Other`
   `Location: http://localhost:8088/viewPosts.html`

2. Return with the code `200 OK`, but replace the content with a Javascript command to change the location.
   e.g.:
   `HTTP/1.1 200 OK`
   `Content-Length: 52`
   other header fields

   `<script>document.location="viewPosts.html";</script>`

5 (Session) Cookies

Cookies are a good way of keeping track of a session. They are small text elements that the browser will store, then submit on each new request to the same server.

5.1 Setting a cookie

When the server wants a cookie to be set, it uses the `Set-Cookie` field in the response header. For example:

`Set-Cookie: SESSID=rk64vvmhlbt6rsdfv4f02kc5g0; path=/`

will create (or replace) a cookie whose key is `SESSID` and whose value is `rk64vvmhlbt6rsdfv4f02kc5g0`, at the root and without any expiration time. Thus, on each new request, the browser will add
`Cookie: SESSID=rk64vvmhlbt6rsdfv4f02kc5g0`
in its request header.

5.2 Deleting a cookie

To delete a cookie, the server just has to update it with an expired expiration date. Note that some browsers discard the expiration date. Therefore a good practice is to also update the content of the cookie with an invalid value. E.g.:
`Set-Cookie: SESSID=deleted; path=/; expires=Thu, 01 Jan 1970 00:00:00 GMT`

6 Chunked encoding

Chunked Transfer Encoding is a data transfer mechanism, since HTTP/1.1, that allows the server to start the transmission of some blocks without having to know in advance the size of the whole content.

Instead of using the `Content-Length` field, the server will provide a `Transfer-Encoding` field, with the value `chunked` for that field. Then, after the response header, the server will send data blocks, starting with the size of each block (in Hexadecimal). The transfer is finished when the server sends a block of size 0[8].

For instance, the server could respond with:

```
HTTP/1.1 200 OK
Content-Type: text/plain
Transfer-Encoding: chunked

18
This is a first chunk.

1A
And this is another one.

1C
But this one is cut in half
17
without carriage return
0
```

which will be reconstructed as

```
This is a first chunk.
And this is another one.
But this one is cut in half without carriage return
```

7 Gzip compression

Webpages that tend to have a large content are often compressed, if the client accepts that (using the `Accept-Encoding` field).

This can be achieved simply by sending the webpage content through a `java.util.zip.GZIPOutputStream`, and by sending that compressed content to the

[8] "0" followed by two carriage return line feeds

`OutputStream` of the socket.

However, you should consider the following implications:

1. As gzip-encoded data are binary data and not text, particular attention should be given to the charset encoding. The selected charset should be explicitly specified at the creation of the Gzip output stream, provided to the client using the `charset` parameter of the `Content-Type` field, and, of course, given that this charset is accepted by the browser (in the `Accept-Charset` field of the request header).

2. When Gzip compression is combined with chunked encoding on server side (to manipulate the response before transmitting it to the client), compression is always performed first, then chunked encoding is performed on the compressed data, and not the other way around.

8 Thread Pool

When a server accepts a connection, it usually invokes a new *thread* that will handle that connection, such that the server can go back to listening to the port. This is very convenient to guarantee a certain level of accessibility but also has a flaw.

The *(Distributed) Denial of Service* (or(D)Dos) is an attack that targets servers with this kind of behaviour. In this attack, one (for DoS) or several (for DDoS) machines initiate many bogus connections. If the server launches a new thread for each of these connections, it will soon encounter performance problems or even crash.

To circumvent this problem, one can use a *Thread Pool* that limits the number of threads that can be executed concurrently, while keeping the other jobs on hold until new threads become available. This thread pool can be implemented through the use of `java.util.concurrent.Executors` by calling the `newFixedThreadPool(int maxThreads)` method to create a fixed-size pool of $maxThreads$ threads and calling the `execute(Runnable worker)` method to assign the work represented by *worker* to one of a thread in the pool, when available.

You can (and are encouraged to) use a thread pool in your assignment.

Good programming...