Computer Networks *(Redes de Computadores - RCOMP)* – 2025/2026

Laboratory Class 13 (PL13 – 2.5 hours)

Implementing an HTTP server with AJAX support (C and Java).

1. Implementing an HTTP server – AJAX voting

As presented in a prior Theoretical-Practical class, develop a small HTTP server application with AJAX support, capable of processing some specific client's requests regarding a voting server application:

- No support for persistent connections, thus, it will always send to clients the **Connection: close** header line.

- Ignore client's header requests about type, language and encoding, etc.

- **GET /votes** returns a ready to use HTML content with current voting standings and buttons linked to JavaScript functions to cast votes.

- **GET** requests to other URIs are regarded as requests for static files contents stored in the **www/** folder. Some common content types (Content-type) should be supported.

- Vote casting is implemented through **PUT** requests to URI **/votes/{N}**, where {N} stands for the candidate's number (1..4). PUT requests are not required to actually carry any data.

- Provided web services (GET /votes and PUT /votes/{N}) are to be consumed by JavaScript running on the browser through **XMLHttpRequest** objects (AJAX).

- The GET /votes request is intended to be used by JavaScript to keep the displayed web page information updated and should be periodically called.

- The PUT /votes/{N} request is used to cast a vote on candidate number {N}.

1.1. The suggested HTTP server implementation in C language

This C implementation is split into three source files, the **http.h** file contains functions' interfaces (prototypes), and constant definitions, related to the HTTP protocol, and it's included both by **http.c** and **http_srv_ajax_voting.c**. The **http.c** file contains the corresponding implementations, it's compiled to an object file **http.o** that is later linked together with the main application source file **http_srv_ajax_voting.c**, at compilation time, resulting in the final executable application file **http_srv_ajax_voting**.

1.2. Base definitions for HTTP (http.h and http.c)

The header file (**http.h**) contains definitions to be included (#include) in the main application (**http_srv_ajax_voting.c**). Functions' interfaces defined in **http.h** are implemented in **http.c**.

These files have already been studied, so we will not analyse them thoroughly here. One fundamental pair of functions, required to implement any HTTP client or server, are those meant to provide reading and writing of variable length CR+LF terminated text lines.

```
void readLineCRLF(int sock, char *line);
void writeLineCRLF(int sock, char *line);
```

This is required to handle HTTP message's header. The reading operation is especially delicate because the reader doesn't know the line's length in advance, so it must read one byte at a time.

The **CR** byte (Carriage Return) has decimal value **13** and may be represented in source files by the '\r' character. The **LF** byte (Line Feed) has decimal value **10** and can be represented in source files by the '\n' character.

Beyond these two fundamental functions, **http.h** is focused only on sending HTTP response messages, receiving HTTP messages, and sending requests is not implemented here.

1.2.1. The HTTP server - http_srv_ajax_voting.c

To start with, an HTTP server must accept TCP connections from clients, so in essence, it's a TCP server.

One design issue, to be addressed in this implementation, is a client's request may be a vote casting, and that changes the current voting standings. So, the up-to-date current vote standings must be accessible when a request is processed and during a request's processing it may be necessary to change that shared information (if it's a vote cast).

With a standard multi-process TCP server implementation, this would require IPC (inter-process communication), for instance using shared memory, also because several processes would be using the same shared memory (current voting standings) a mutual exclusion access mechanisms would be required, typically a semaphore.

To avoid using IPC, the purposed implementation is not entirely multi-process. The processHttpRequest() function is called for each received connection, however, before creating a child process the function checks if it's a vote casting (PUT /votes/{N}). If so, no child process is created and the request is processed within the main process, thus the current voting standings are not actually shared among several processes, it's only updated in the main process. But ultimately it does the trick.

```
void processHttpRequest(int sock, int conSock) {
  char requestLine[200];

  readLineCRLF(conSock,requestLine);
  if(!strncmp(requestLine,"GET /",5)) {
         if(!fork()) {                // GET requests are processed in background
                close(sock);
                processGET(conSock,requestLine);
                close(conSock);
                exit(0);
                }
         close(conSock);
         return;
         }
  if(!strncmp(requestLine,"PUT /votes/",11)) processPUT(conSock,requestLine);
  else {
         sendHttpStringResponse(conSock, "405 Method Not Allowed", "text/html",
                "<html><body>HTTP method not supported</body></html>");
         puts("Oops, the method is not supported by this server");
         }

  close(conSock);
  }
```

For other requests (GET), a child process is created (fork), when doing so the current voting standings will also be duplicated to the child process (remember the fork() system-call creates an exact copy of the current process).

With this approach, PUT requests from clients are handled one by one by, in a single process, by the order they arrive, and never more than one at the same time. This avoids the need for shared memory (where current voting standings would be stored) and at the same time eliminates any concurrency issue.

From the efficiency point of view, this is not ideal, yet for our purpose it makes the implementation and code considerably simpler. One other option, of course, would be the use of threads, that's what we will do in the Java version.

1.3. The proposed HTTP server implementation in Java language

Unlike the C version, this is a plain typical multi-thread TCP server architecture, capable of handling several requests of any kind at the same time.

Because threads are used, no IPC is required, yet because all threads will be accessing the same data (current voting standings) mutual exclusion is required.

1.3.1. HTTPmessage.java

In HTTP servers and clients, one key concept is undoubtedly the HTTP message, thus, in an object-oriented implementation it should be represented by a class.

The provided implementation (**HTTPmessage.java**) is extremely incomplete, nevertheless, it's able to send and receive both HTTP requests and responses, either with or without a content (body).

The class's static elements are private, they include several HTTP constants, methods to read and write HTTP header lines, and some associations between content types and filenames extensions. Methods, readHeaderLine() and writeHeaderLine() implementations are very similar to those used in C language for functions readLineCRLF() and writeLineCRLF().

Each class instance has a few fields to store the HTTP message itself:

```
    private boolean isRequest;
    private String method;
    private String uri;
    private String status;
    private String contentType;
    private byte[] content;
```

The **HTTPmessage(DataInputStream in**) constructor is used to receive an HTTP message from a provided socket's input stream and store it in the newly created instance. The other defined constructor takes no arguments and creates an undefined request message.

The first constructor is the only method for receiving an HTTP message. After reading the first line, it checks if it's a request or a response, and stores the relevant information for each case. Next, header fields are processed, and mostly ignored, only content length and content type are stored. If there's content after the HTTP header, the content is read.

The **send(DataOutputStream out)** public method is used to send the message through a provided socket's output stream.

```
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;
       }
```

If there's a content, then, the content-type and content-length header fields are included, and the content itself is sent after the HTTP header.

The **setContentFromString(String c, String ct)** public method settles the HTTP message's content-type (ct), and the content itself from a provided string (c).

The **setContentFromFile(String fname)** public method settles the HTTP message's content by reading it from a provided filename. Returns false if it fails to read the file. On success, this method also settles the content type for a few well-known file extensions.

Most other methods are defined to provide access to object's private elements (encapsulation).

1.3.2. HttpServerAjaxVoting.java

This totally static class implements the server loop in the **main()** method, it's an already familiar multi-thread TCP server implementation in Java. For each client´s request, a new instance of the **HttpAjaxVotingRequest** class is created and then launched in background as a thread by calling the **start()** method.

Also implemented in this class, the current vote standings, and an HTTP requests counter. These elements are private, they must be accessed by calling a set of synchronized static methods. Thus, these methods ensure mutual exclusion. While a thread is running one of these methods, other threads calling any of these methods will be blocked and waiting for their turn.

1.3.3. HttpAjaxVotingRequest.java

The purpose of this class is handling HTTP requests from clients, for each accepted TCP connection one instance is created by calling the HttpAjaxVotingRequest() constructor and then executed as a thread. The constructor receives the connected socket, and the base folder from where to fetch files for static contents (GET requests).

The thread's execution is enforced by calling the start() method which in turn calls the run() method. Once the socket's input (inS) and output (outS) streams are obtained, an HTTP message is received (request) and a response is created.

```
HTTPmessage request = new HTTPmessage(inS);
HTTPmessage response = new HTTPmessage();
if(request.getMethod().equals("GET")) {
     if(request.getURI().equals("/votes")) {
            response.setContentFromString(
                   HttpServerAjaxVoting.getVotesStandingInHTML(), "text/html");
            response.setResponseStatus("200 Ok");
            }
     else {
            String fullname=baseFolder + "/";
            if(request.getURI().equals("/")) fullname=fullname+"index.html";
            else fullname=fullname+request.getURI();
            if(response.setContentFromFile(fullname)) {
                   response.setResponseStatus("200 Ok");
                   }
            else {
                   response.setContentFromString(
                          "<html><body><h1>404 File not found</h1></body></html>",
                          "text/html");
                   response.setResponseStatus("404 Not Found");
                   }
            }
     response.send(outS);
     }
else { // NOT GET
     if(request.getMethod().equals("PUT")
            && request.getURI().startsWith("/votes/")) {
            HttpServerAjaxVoting.castVote(request.getURI().substring(7));
            response.setResponseStatus("200 Ok");
            }
     else {
            response.setContentFromString(
                   "<html><body><h1>ERROR: 405 Method Not Allowed</h1></body></html>",
                   "text/html");
            response.setResponseStatus("405 Method Not Allowed");
            }
     response.send(outS);
```

```
    }
```

The GET /votes request has a special treatment, as it returns the HTML content produced by the getVotesStandingInHTML() method of the HttpServerAjaxVoting class, containing the current voting standings, and buttons attached to JavaScript functions to cast votes (HTTP request PUT /votes/{N}).

1.4. The HTML root document (www/index.html file)

This document's content is provided by the server for GET requests to URI / or URI /index.html.

```
<html><head><title>HTTP demo</title>
<script src="rcomp-ajax.js"></script>
</head>
<body bgcolor=#C0C0C0 onload="refreshVotes()"><h1>HTTP server demo - Voting with AJAX</h1>
<h3>Java version</h3>
<hr>
<center><table width=60% border=1 cellpadding=20 cellspacing=20><tr>
<td height="300" align=left width=50% valign="top">
<big><div id="votes">Please wait, loading voting results ...</div></big>
</td></tr></table></center>
<hr>
<center><table border=0><tr><td align=center>Image contents are supported:<br><br><img src=http2.png><br>(http2.png)</td>
<td align=center><img src=http.gif><br>(http.gif)</td></tr></table></center>
</body></html>
```

It retrieves JavaScript code defined in file **www/rcomp-ajax.js**, and once loaded runs the **refreshVotes()** JavaScript function for the first time. The document's area identified by name votes ( ) is updated by this function by making an HTTP request "GET /votes".

1.5. JavaScript functions and AJAX (www/rcomp-ajax.js file)

The **refreshVotes()** JavaScript function is called once the main HTML document is loaded.

```
function refreshVotes() {
   var request = new XMLHttpRequest();
        var vBoard=document.getElementById("votes");
        request.onload = function() {
            vBoard.innerHTML = this.responseText;
            setTimeout(refreshVotes, 2000);
            };
        request.ontimeout = function() {
            vBoard.innerHTML = "Server timeout, still trying ...";
            setTimeout(refreshVotes, 100);
        };
        request.onerror = function() {
            vBoard.innerHTML = "No server reply, still trying ...";
            setTimeout(refreshVotes, 5000);
        };
   request.open("GET", "/votes", true);
   request.timeout = 5000;
   request.send();
   }
function voteFor(option) {
   var request = new XMLHttpRequest();
   request.open("PUT", "/votes/" + option , true);
   request.send();
   var vBoard=document.getElementById("votes");
   vBoard.innerHTML = vBoard.innerHTML + "<p>Casting your vote ...";
   }
```

The **refreshVotes()** function creates an HTTP "GET /votes" request, establishes call-back functions to handle the corresponding results, and schedules a subsequent call to the same function. This means the content is constantly updated with a periodicity equal to the time it takes to get a response, plus 2 seconds after a successful response arrives. On success, the **votes** area of the document is replaced with the retrieved plain text content, if fact it's an HTML content.

The **timeout** property of the **XMLHttpRequest** is settled to 5 seconds. In the event of timeout (and error as well), the function is scheduled to be executed again within some time, so if the server fails to respond, the client keeps trying.

This **voteFor()** function is attached to buttons to cast votes by issuing an HTTP "**PUT /votes/{N}"** request.

1.6. Compiling and testing

Select a host to run the server, say **ssh4.dei.isep.ipp.pt** (a CNAME for **vsrv27.dei.isep.ipp.pt**). You could use any other host, including your workstation.

This host's address in the laboratories network is 10.8.0.83 (IPv4) and fd1e:2bae:c6fd:1008::83 (IPv6), but it may also be referred to by the DNS names **labs-vsrv27.dei.isep.ipp.pt** or **labs-ssh4.dei.isep.ipp.pt**.

Download the provided source files, the **Makefile** and the **www** folder. Ask the class's teacher for a unique TCP port number to be used by your server (**MY-PORT-NUMBER**).

As usual, to build run the **make** command.

Now, start the C version of the HTTP server:

```
./http_srv_ajax_voting MY-PORT-NUMBER
```

a) In a workstation connected to the laboratories network, start a standard web browser, and open the URL:

   http://labs-vsrv27.dei.isep.ipp.pt:MY-PORT-NUMBER

   The main HTML page should be displayed, check that the **HTTP accesses counter** in the page is increasing every two seconds, if so, it means JavaScript is refreshing the content as expected. You may ask nearby colleagues to also open your URL in their browsers, then, of course, the counter will increase much faster.

b) Test vote casting, ask colleagues using your server to do the same. You may also open another browser's window to access your URL.

c) Use a postman application to cast votes on the second candidate (**PUT /votes/2**).

d) The Java version includes a DemoConsumer casting 200 votes on the first candidate, compile it and run it on another host:

```
java DemoConsumer labs-vsrv27.dei.isep.ipp.pt MY-PORT-NUMBER
```

   It simply performs two hundred HTTP "PUT /votes/1 requests" to the server, check the results on the web page.

e) Force a server crash (press CTRL+C on the server's console). Check the browser's web page again.

f) Start the server again. Due to the server's forced crash, it may take some time before the local port number is available again, be patient.

g) Once the server finally starts, check the browser's web page again. It should be recovered, of course all counters are back to the starting point.

Stop the server's C version, and then start the Java version:

```
java HttpServerAjaxVoting MY-PORT-NUMBER
```

Repeat the same tests as before, now with the Java version, it should behave the same way.