Redes de Computadores (RCOMP)

Theoretical-Practical (TP) Lesson 10

2017/2018

- Analysing a sample HTTP server in C language.

(to be implemented and testes on laboratory classes)

RESTful web services

A web service is an HTTP/HTTPS service made available by an HTTP server application to other applications acting as HTTP clients, but excluding web browsers. Web browsers are excluded because transferred contents are not meant to be directly presented by browsers to end-users. The idea is using HTTP as a general purpose application-to-application protocol.

Representational State Transfer (REST) is a design model for web services focused in resources and following a set of principles (REST constrains).

Resources on the server side (provider/publisher) are identified by URIs and available to clients (consumers/requestors) by using HTTP methods over that URIs. Several rules apply to resources naming through a URI.

In REST, operation over resources are CRUD oriented (Create, Read, Update, Delete). Nevertheless, a URI may also refer to a function or controller, in this case the URI name should be a verb.

Resources' contents must be transferred between providers and consumers in an implementation independent representation, usually Extensible Mark-up Language (XML).

Extensible Mark-up Language (XML)

XML is a data representation format through text, designed to be both human-readable and also easy to be processed by applications. REST web services resources should be transferred between applications in XML format by specifying the **Content-type: application/xml** HTTP header line.

As with HTML, XML encapsulates data within tags represented between symbols **<** and **>**, but unlike with HTML where tag names have special meanings, in XML they do not. In XML tags may be freely established by applications conforming their needs. Also, HTML specifies a way to present data to end-users, XML specifies only the data representation.

A XML content may optionally start by a special line called **XML prolog**:

<?xml version="1.0" encoding="UTF-8"?>

The XML prolog line is optional, but every XML content must have a **root tag** embracing the whole content. Tag names are case sensitive and every started tag must be closed by an end tag. If a tag doesn't have any data it may be ended immediately when started by ending it with **/>** instead of **>**.

If a tag's content includes the **<** symbol or the **&** symbol, they must be represented, correspondingly by **<** and **&**.

XML – tag's attributes

XML tags may have attributes, attributes are pairs **name="value"** declared within the start tag, attribute names are also case sensitive and the attribute value must always be quoted.

Tag's attributes should be used to identify the data element and not data's properties, properties ought to be specified by sub tags.

Example:

```
<?xml version="1.0" encoding="UTF-8"?>
<users>
  <user id="100" />
  <user id="101"></user>
  <user id="102"><name>ABC</name></user>
  <user id="103">
        <name>ABC</name>
        <phone>9999909</phone>
  </user>
</users>
```

In this example, <users> is the root tag. It contains four tags named <user>, the first two are empty, although specified in different ways.

Web services testing – Postman

In standard web browsers, when a URL is manually typed, the browser always assumes the method to use is GET. So a web browser is not a suitable tool to impersonating consumer applications and test web services.

Applications generally called **postman** do the trick, they are able to generate HTTP requests with any method, also setting HTTP headers and the message's content as required. In addition, they also provide extensive information about the response received from the provider.

Postman is an essential tool when developing web services, by testing them, the developer assures himself they are working properly before they are actually used by real consumer applications.

Postman is often used to manually perform **functional tests**, but it may also be programmed through scripts to automatically perform sets of **unity tests**, thus ensuring web services are kept in conformity during development.

Several more or less sophisticated versions of postman are freely available, some even run on standard web browsers as plugins or extensions.

Web browsers as consumers - JavaScript

The standard use of web browsers: retrieve contents and display them to end-users, has no place in the web services model.

Having said that, the fact is, modern web browsers are themselves platforms where applications can be run, for instance using JavaScript.

The **XMLHttpRequest** object is an HTTP client available in JavaScript, by using it, JavaScript applications/functions may become web services' consumers.

In this object, the **open()** method is used to create a request (not actually send it), any HTTP method can be used over a specified URL, by default the request is asynchronous. HTTP header lines can be settled one by one with the **setRequestHeader()** method before finally sending the request to the provider by calling the **send()** method.

Asynchronous request means when calling the **send()** method the application will not be blocked waiting for the response, this is most important for a web browser.

If data it to be sent (PUT or POST), it can be specified as argument of the send() method, data can also be sent with GET, but in that case it will be part of the URI provided to the open() method.

Web browsers as consumers - JavaScript

Before sending an asynchronous request, the object's property **onload** must be assigned with a call-back function to be called asynchronously when the response arrives. Once the response arrives, within the **onload** call-back function, the status property contains the HTTP status code, 200 for ok.

By default the **XMLHttpRequest** object has no timeout associated, it will wait forever for a response, however, the **timeout** property can be assigned with a value in milliseconds to change this default behaviour. If **timeout** is settled, then the **ontimeout** property should be assigned with a call-back function to handle that scenario.

Event property	Standing for …
onreadystatechange	The state has changed, the state property will contain one of the following values: 0 (request not initialized); 1 (server connection established); 2 (request received); 3 (processing request); 4: (request finished and response is ready)
onabort	The request was aborted by calling the abort() method.
onerror	The request has failed.
onload and **onloadend**	The request was successful (load). The request processing has finished successfully or not.
ontimeout	The request failed due to timeout (as defined by the timeout property value greater than zero).

Implementing demo HTTP server in C language

Fully implementing a network client or server application can be a very simple or a rather extensive activity, it all depends on the application protocol complexity and features.

HTTP basic concepts are pretty simple to implement. One TCP connection, a request is sent, a reply is returned. Both the request and the reply use the same message format: a text header possibly followed by a body. A limited number of possible request types (methods) and an also limited number of possible responses.

So, implementing an HTTP server to support a limited set of HTTP features, and not the whole HTTP protocol specification, isn't such an extensive task.

This HTTP server project was designed to cover basic static contents fetching through the GET method, web services and AJAX.

It's a voting system, the current voting results must be displayed and kept updated to all users, any user may vote any number of times, the results being shown to all users must be always up-to-date.

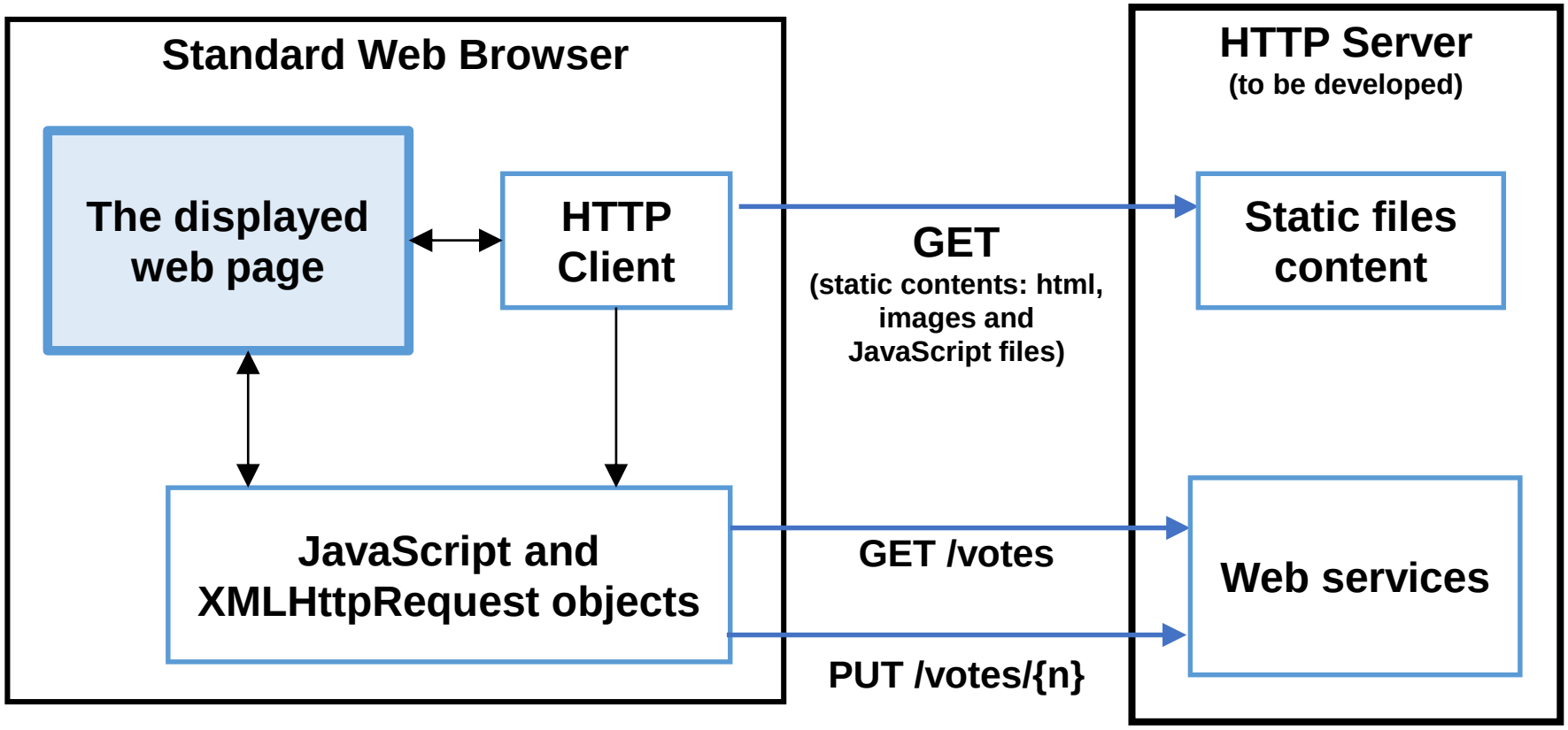
Specific design requirements

The HTTP server will have the following characteristics and limitations:

- No persistent connections support, thus, the server will always send the **Connection: close** header line to clients .
- Ignores all header fields in client's requests.
- GET **/votes** returns the voting standings as an HTML list. This list will also include JavaScript linked buttons to cast votes.
- Other GET requests as assumed to refer to static content files, stored within a established folder. By analyzing the file's name, some relevant content types should be supported.
- PUT is supported for the **/votes/{n}** URI, standing for a vote casting on candidate number {n}, on this demo, candidates are numbered from 1 to 4. PUT requests will not carry any body content.

When designing web services and consumers, data processing can be implemented on both sides. In this project the GET /votes provides a ready to use server generated HTML content, but it could be otherwise, for instance the server could provide a XML content and it would be up to the consumer (JavaScript) creating the HTML content from it.

Architecture



Next we will analyze the provided C implementation
(C/http-server-ajax-voting/http_srv_ajax_voting.c)

Reading and writing HTTP headers (http.h and http.c)

Every HTTP message starts by a text lines header, each header line is CR+LF terminated. The first thing we need to implement an HTTP client or server is a pair of functions to read and write this text lines in this format, in C language the most convenient way to represent text lines is by null terminated strings:

```
void readLineCRLF(int sock, char *line)
{
char *aux=line;
for(;;) {
            read(sock,aux,1);
            if(*aux=='\n')
                        {
                        *aux=0;return;
                        }
            else
            if(*aux!='\r') aux++;
            }
}

void writeLineCRLF(int sock, char *line)
{
char *aux=line;
while(*aux) {write(sock,aux,1); aux++;}
write(sock,"\r\n",2);
}
```

Reading a header line must be done byte by byte, this is because we do not know the line's length until we hit CR+LF.

Unlike with reading, when we are going to write a header line, we already know it length:

```
void writeLineCRLF(int sock, char *line)
{
write(sock,line,strlen(line));
write(sock,"\r\n",2);
}
```

Sending an HTTP response header (http.h and http.c)

An HTTP server receives an HTTP request message and then replies with an HTTP response message. To send HTTP response messages a simple function was defined:

```
void sendHttpResponseHeader(int sock, char *status, char *contentType, int contentLength) {
        char aux[200];
        sprintf(aux,"%s %s",HTTP_VERSION,status);
        writeLineCRLF(sock,aux);
        sprintf(aux,"Content-type: %s",contentType);
        writeLineCRLF(sock,aux);
        sprintf(aux,"Content-length: %d",contentLength);
        writeLineCRLF(sock,aux);
        writeLineCRLF(sock,HTTP_CONNECTION_CLOSE);
        writeLineCRLF(sock,"");
        }
```

The first argument it the socket through which the response is to be sent (written), next the status code and text, the content type and content length. The header will always include the Connection: close line, and of course is ended by an empty line.

The sendHttpStringResponse() function is calls the previous function to send a response with a text content (body) stored in a string (C null terminated string):

```
void sendHttpStringResponse(int sock, char *status, char *contentType, char *content) {
       sendHttpResponse(sock,status,contentType,content,strlen(content));
       }
```

HTTP responses with other contents (http.h and http.c)

For cases where the content is not text:

```
int sendHttpResponse(int sock, char *status, char *contentType, char *content, int
contentLength) {
        int done, todo;
        char *aux;
        sendHttpResponseHeader(sock, status, contentType, contentLength);
        aux=content; todo=contentLength;
        while(todo) {
                done=write(sock,aux,todo);
                if(done<1) return(0);
                todo=todo-done;
                aux=aux+done;
                }
        return(1);
        }
```

Because the content may not be text it's not passed to the functions as a C null terminated string, thus the content size must be provided by the caller (contentLength).
Also, because the content's size may be rather large, content writing operations may be incomplete (when writing, **done** may be less than **todo**), so we ensure the whole content is effectively written. If the whole content writing is successful the 1 value is returned, otherwise 0.

HTTP responses for file contents (http.h and http.c) 1/3

The sendHttpFileResponse() function was implemented for static files contents:

```
void sendHttpFileResponse(int sock, char *status, char *filename) {
        FILE *f;
        char *aux;
        char line[200];
        int done;
        long len;
        char *contentType="text/html";

        f=fopen(filename,"r");
        if(!f) {
                sendHttpStringResponse(sock, "404 Not Found", contentType,
                                "<html><body><h1>404 File not found</h1></body></html>");
                return;
                }
        aux=filename+strlen(filename)-1;
        while(*aux!='.' && aux!=filename) aux--;

(…)
```

It receives a filename whose content is to be sent in the body of the HTTP response message, if opening the requested file fails, the **404 Not Found** status is sent with a simple HTML content. The content type defaults to text/html, but next we are going analyse the filename's extension to settle a more appropriate content type. The aux pointer will be pointing to the last dot in the filename, or to the filename itself if there's no dot.

HTTP responses for file contents (http.h and http.c) 2/3

```
(…)
if(*aux=='.')
                {
                if(!strcmp(aux,".pdf")) contentType="application/pdf";
                else
                if(!strcmp(aux,".js")) contentType="application/javascript";
                else
                if(!strcmp(aux,".txt")) contentType="text/plain";
                else
                if(!strcmp(aux,".gif")) contentType="image/gif";
                else
                if(!strcmp(aux,".png")) contentType="image/png";
                }
        else
                contentType="application/x-binary";
(…)
```

Conforming to the filename's extension the content type value is settled, by default the **text/html** is used for filename with unhandled extensions. If there's no dot in the filename, the content type is going to be be **application/x-binary**.

HTTP responses for file contents (http.h and http.c) 3/3

```
(…)
        fseek(f,0,SEEK_END);
        len=ftell(f);
        if(!status) status="200 Ok";
        sendHttpResponseHeader(sock, status, contentType, len);
        rewind(f);
        do {
                done=fread(line,1,200,f);
                if(done>0) write(sock,line,done);
                }
        while(done>=0);
        fclose(f);
        }
```

To know the file's size (content length) the fseek() and ftell() are used. If the caller hasn't provided a status, **200 Ok** is used.
We have now all data required to send the HTTP response message's header (sendHttpResponseHeader).

Then we can start reading data from the file's start (rewind), and send it to the HTTP client as as it's read from the file. When there's no more data to read from the the file, fread() returns zero or -1 in the case of error.

HTTP server (http_srv_ajax_voting.c)

```
(…)
#include "http.h"
#define BASE_FOLDER "www"

void processHttpRequest(int sock, int conSock);    // implemented ahead
void processGET(int sock, char *requestLine);      // implemented ahead
void processPUT(int sock, char *requestLine);      // implemented ahead

#define NUM_CANDIDATES 4
char *candidateName[] = { "Candidate A", "Candidate B", "Candidate C" , "Candidate D" };
int candidateVotes[NUM_CANDIDATES];
unsigned int httpAccessesCounter=0;
(…)
```

Beyond other required header files, the already implemented functions defined in **http.h** are included, the defined **BASE_FOLDER** represents the folder from where to fetch static file contents as requested by clients.

This is just a demo voting system, for this purpose only four alternatives (candidates) are established, each candidate current number of votes is stored in candidateVotes[NUM_CANDIDATES], so the first candidate will have index zero. An HTTP requests counter is also established and started, mostly for debugging purposes, it will also be shown in the server's web page.

HTTP server's main loop (http_srv_ajax_voting.c)

```
(…)
int main(int argc, char **argv) {
(…)
        for(i=0; i<NUM_CANDIDATES; i++) candidateVotes[i]=0;
        signal(SIGCHLD, SIG_IGN); // AVOID LEAVING TERMINATED CHILD PROCESSES AS ZOMBIES

        while(1) {
                newSock=accept(sock,(struct sockaddr *)&from,&adl);
                httpAccessesCounter++;
                processHttpRequest(sock,newSock);
                }
        close(sock);
        return(0);
        }
```

The **main()** server function implements a basic TCP server by preparing an IF_INET6 socket for accepting TCP connections as usual. It then initializes voting counters and starts the usual TCP infinite loop of client connections acceptance. For each client connection, the accesses counter is updated an then the processHttpRequest() is called.

Notice that so far no child process has been created. The point is, when a vote is casted through an HTTP request the vote counters must be updated, if that was handled in a child process, then IPC would be required to update vote counters on the parent process.

We are going to avoid IPC by implementing vote casting processing in the main process and not in a child process.

processHttpRequest() function (http_srv_ajax_voting.c)

```
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>");
                }
        close(conSock);
        }
```

Once the request line is read, if it's a GET method request, then a child process is created to handle it through the **processGET()** function. If it's a vote casting (PUT /votes/…) no child process is created and its handled through the **processPUT()** function in the main process.
If the request is neither a GET nor a PUT for a URI started by /votes/, the server replies with an **405 Method Not Allowed** status response.

processGET() function (http_srv_ajax_voting.c)

```
void processGET(int sock, char *requestLine) {
        char *aux, line[200], filePath[100], uri[100];

        do {    // READ AND IGNORE HEADER LINES
                readLineCRLF(sock,line);
                }
        while(*line);

        strcpy(uri,requestLine+4);
        aux=uri; while(*aux!=32) aux++; *aux=0;
        if(!strncmp(uri,"/votes",8)) {
                sendVotes(sock); return;
                }
        if(!strcmp(uri,"/")) strcpy(uri,"/index.html"); // BASE URI
        strcpy(filePath,BASE_FOLDER);
        strcat(filePath,uri);
        sendHttpFileResponse(sock, NULL, filePath);
        }
```

After reading and ignoring all request's header lines, the URI is analysed, if it's **/votes** the **sendVotes()** function is called to send a response with a HTML content with the current votes counting and necessary HTML tags for vote casting.
Otherwise we assume it must be a reference to a static file so we append (strcat) the URI to the BASE_FOLDER and call **sendHttpFileResponse()**.

sendVotes () function (http_srv_ajax_voting.c)

```
void sendVotes(int sock) {
          char buffer[1024], line[200];;
          strcpy(buffer,"<hr><ul>");
          for(int i=0; i<NUM_CANDIDATES; i++) {
                    sprintf(line, "<li><button type=\"button\" onclick=\"voteFor(%i)\">Vote
for %s</button> %s - %d votes </li>", i+1, candidateName[i], candidateName[i],
candidateVotes[i] );
                    strcat(buffer,line);
                    }
          sprintf(line, "</ul><hr><p>HTTP server accesses counter: %u</p><hr>",
                    httpAccessesCounter);
          strcat(buffer,line);
          sendHttpStringResponse(sock, "200 Ok", "text/html", buffer);
          }
```

This function creates an HTML content and sends it as content of an HTTP response message, it's sole purpose is being called by processGet() in response to a GET /votes HTTP request.

The created HTML content is an unnumbered list tag (
) with buttons calling JavaScript voteFor() function to cast votes (by calling web services) and the current votes for each candidate. The JavaScript voteFor() function receives the candidate number as argument, first candidate is number one.
In addition, the current HTTP accesses counter value is also provided.

processPUT() function (http_srv_ajax_voting.c)

```
void processPUT(int sock, char *requestLine) {
          char *aux, line[200], uri[100];
          int candidate;
                              // READ AND IGNORE HEADER LINES
          do { readLineCRLF(sock,line);  } while(*line);

          strcpy(uri,requestLine+4);
          aux=uri; while(*aux!=32) aux++; *aux=0;
          aux=uri+strlen(uri)-1; while(*aux!='/') aux--; // FIND LAST SLASH
          aux++;
          candidate=atoi(aux); candidate--; // CONVERT TO INDEX VALUE
          if(candidate<0||candidate>NUM_CANDIDATES) {        // BAD CANDIDATE INDEX
                    sendHttpStringResponse(sock, "405 Method Not Allowed", "text/html",
                              "<html><body>HTTP method not supported</body></html>");
                    return;
                    }
          candidateVotes[candidate]++;
          sendHttpStringResponse(sock, "200 Ok", "text/plain","");
          }
```

After reading and ignoring all request's header lines, the URI is analysed to isolate the last URI path element, it should be a number (1..4). It's converted to an integer (atoi), if not within range, a **405 Method Not Allowed** response is sent. Otherwise, the number of votes is updated. This function doesn't receive any PUT content because that's the service was designed that way. Because this function is called within the main process an not any child process, the new voting status is effective for all following client requests.

Main HTML page (www/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>Linux/C version</h3>
<hr><center>
<table width=60% border=1 cellpadding=20 cellspacing=20><tr>
<td align=left><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>
```

The page loads the JavaScript file **rcomp-ajax.js**, containing some functions when the HTML body is loaded the browser will automatically call (*onload*) the **refreshVotes()** JavaScript function. This function will use the **XMLHttpRequest** object to consume web services and update the page area identified as votes ( ).
Additionally this page also loads some images just for the sake of checking the server is handling appropriately GET requests for images.

JavaScript function refreshVotes() (www/rcomp-ajax.js)

```
function refreshVotes() {
          var request = new XMLHttpRequest();
          request.onload = function upDate() {
                    document.getElementById("votes").innerHTML = this.responseText;
                    setTimeout(refreshVotes, 1500);
                    };
          request.ontimeout = function timeoutCase() {
                    document.getElementById("votes").innerHTML = “Still trying ...";
                    setTimeout(refreshVotes, 1000);
                    };
          request.onerror = function errorCase() {
                    document.getElementById("votes").innerHTML = “Still trying ...";
                    setTimeout(refreshVotes, 1000);
                    };
          request.open("GET", "/votes", true);
          request.timeout = 5000;
          request.send();
          }
```

It’s called on the HTML page load, creates the XMLHttpRequest object and settles call-back functions, for a success it calls the upDate() function that replaces the **votes** area in the HTML page with the received response (responseText). The update() function also schedules an automatic call to **refreshVotes()** in 1.5 seconds. This means once a response is receive, the function is called again in 1.5 seconds. Call-back functions are also settled for error events. Finally the web service to called is defined (GET /votes), a timeout is settled (5 sec.) and the request is started (send).

JavaScript function voteFor() (www/rcomp-ajax.js)

```
function voteFor(option) {
          var request = new XMLHttpRequest();
          request.open("PUT", "/votes/" + option , true);
          request.send();
          }
```

It's called by user interaction (voting button clicking), it sends a PUT request to the server with the URI **/votes/{n}** , as defined by the server for the candidate.

This is a PUT request with no body, the only required data is the URI itself. Under REST point of view, votes is a resource collection and {n} a resource (candidate number). Because the only use case for PUT (update) over a candidate is casting a vote, there is no real need to provide any data on PUT requests.

Also bear in mind that, in this server implementation, PUT requests processing assumes there's no body, if a PUT request with a body is sent, the server will crash. The server is designed to provide web services strictly for this consumers.

Result - the web page

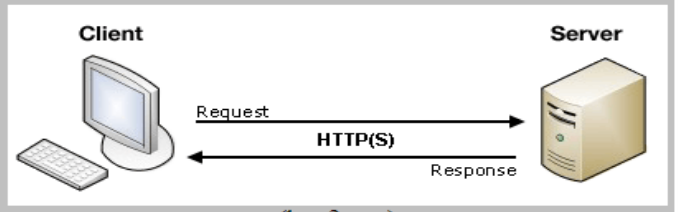
HTTP server demo - Voting with AJAX

Linux/C version

- Vote for Candidate A — Candidate A - 1 votes
- Vote for Candidate B — Candidate B - 0 votes
- Vote for Candidate C — Candidate C - 9 votes
- Vote for Candidate D — Candidate D - 1 votes

HTTP server accesses counter: 109

Image contents are supported:



(http2.png)

(http.gif)

The **HTTP server accesses counter** should be always increasing because the refreshVotes() JavaScript function is cyclically being called. The voting board is update every 1.5 seconds, plus the time it takes to complete the GET /votes request.

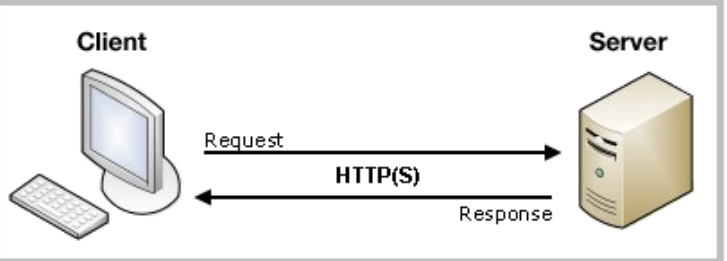
Result - the web page when the server becomes unavailable

HTTP server demo - Voting with AJAX

Linux/C version

No response from server, still trying ...

Image contents are supported:



(http2.png)

(http.gif)

JavaScript call-back functions defined for **timeout** and **error** events will keep trying until the service is available again.