RCOMP - Redes de Computadores (Computer Networks)

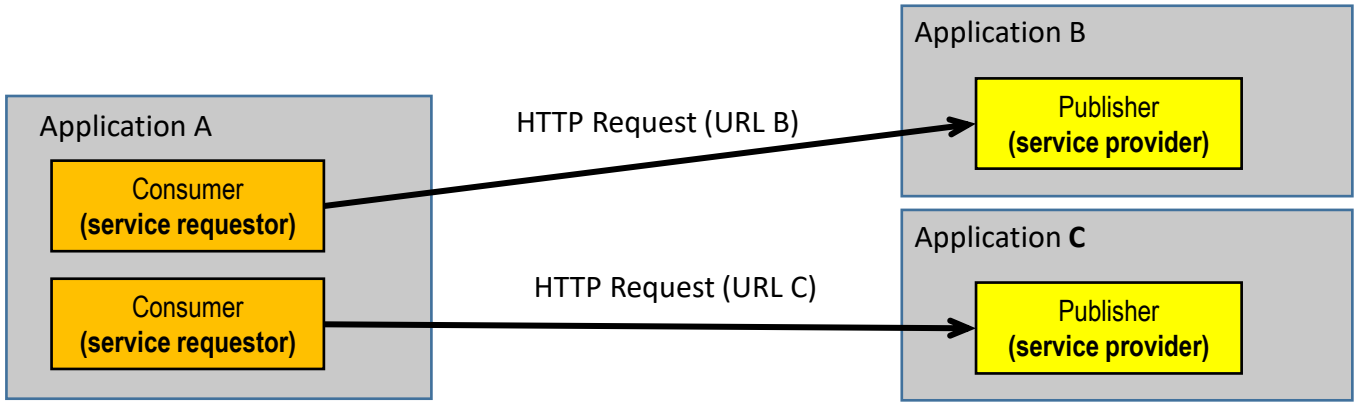
2023/2024

Theoretical-practical lesson 10

- HTTP, Web services, REST, AJAX and Web UI.
- Analysing a simple HTTP server with AJAX support in C language.

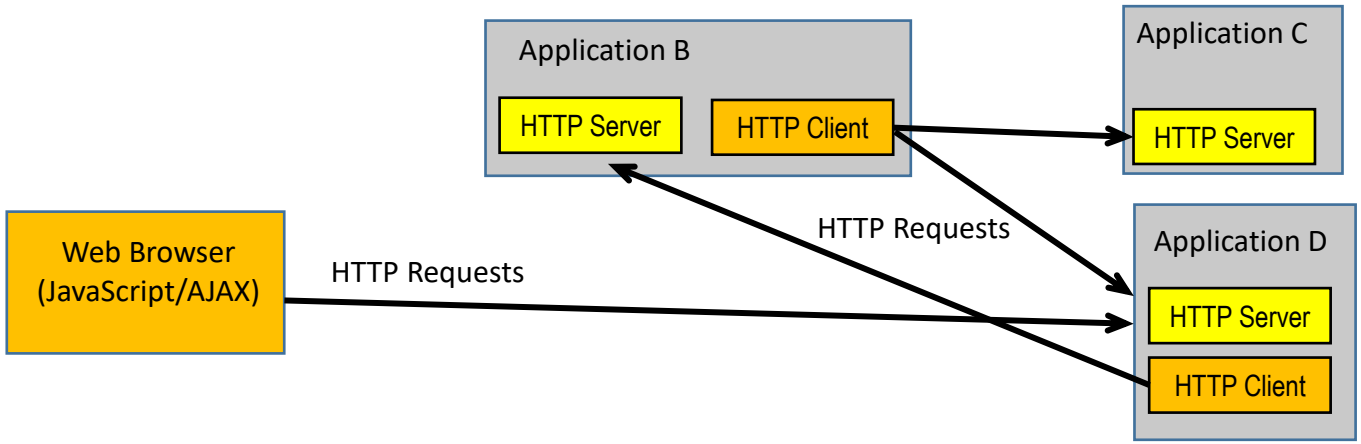
Web Services

The central concept on web services is the use of HTTP for application-to-application communications without direct human intervention.
One application assumes the **HTTP client role** (**service requestor or consumer**) and the other application the **HTTP server role** (**service provider or publisher**). So, the web service is made available to service requestor applications by the service provider application. Of course, the same application can be both a consumer and a provider.



This is a very wide-ranging concept, and allows programmers to implement it with high freedom, for instance regarding what HTTP methods are going to be used, how resources on the provider side are named, and which content types are going to be used on data transfers between requestors and providers. As far as the HTTP protocol is respected, everything is possible.

Web Services - constraints



There's no obstacle for one application being both a consumer and a provider.

Standard Web Browsers are also encompassed as they may become web services consumers by using **XMLHttpRequest** JavaScript objects. Therefore, quite complex distributed environments can be established through web services.

The general freedom in implementing web services tends to turn things somewhat chaotic, thus some efforts have been done on instituting some **rules and principles**, often referred to as **constraints**.

One of these sets of constraints for web services architecture is known as **RESTful**, standing for REST compliant.

RESTful web services' architectural constraints

REST stands for Representational State Transfer, it's a constrained, resource-based design model to implement web services. Main principles (constraints) are:

- Clients request operations over server-side resources (each identified by an URI), operations over server-side resources are: Create, Read, Update and Delete (CRUD), each corresponds to a specific HTTP request method.
- Resources contents should be transferred in XML or JSON representations. Nevertheless, HTML and others might be used if appropriate.
- Servers are stateless in the sense they don't store information about each client's dialogue context. Therefore, on every request clients must provide all required context data for the operation.
- If the server has a state, then that state context must be represented by an addressable resource (URI), clients may then refer that state context on requests.

RESTful web services consumer applications can request the following four operations over a resource (URI):

Operation	HTTP methods
Create a resource	POST; PUT
Read/retrieve a resource	GET
Update/Modify a resource	PUT
Delete/remove a resource	DELETE

RESTful – resources and collections

The only **safe method** is GET, meaning it doesn't change the resource on the server side. Methods PUT, GET, and DELETE are regarded as **idempotent methods**; this means making more than one successive identical request over the same resource has no additional effects beyond the effect of the first request.

A URI may refer to a **single resource** or a **collection** of resources, **singular names** are to be used for single resources, **plural names** for resources collections. Depending on being a single resource or a resources collection, different HTTP methods will represent different actions over the target resource (URI):

HTTP method	Single resource (singular name URI)	Resources collection (plural name URI)
GET	Retrieve the resource.	List the of resources items in the collection. Retrieved data is a list of resources' URIs and optionally other resources' data.
PUT	Replace the resource, if it does not exist, create it.	Replace the whole collection with another collection.
POST	**Not used** because the URI would be regarded as a collection and a new collection item would be created within it.	Create a new resource item within the collection. The new resource URI is automatically assigned.
DELETE	Delete the resource.	Delete the entire collection.

RESTful - URI naming (guidelines and best practices)

- A singular name for a single resource or a collection's item/element.
- A plural name for a collection of resources.
- Verbs for controllers and functions.
- Notice that, excluding the origin (i.e., the server's DNS name), the URI is case sensitive.
- Use either camel casing or, preferably, lowercase with words separated with hyphens (spinal case), instead of underscores (snake case).
- Avoid CRUD names (Create/Read/Update/Delete) for a URI.
- URI path elements should represent resources' hierarchical structure.
- A URI path component can be used to represent a variable's value, in REST that's the recommendation, nevertheless, a query string can also be appended to a URI.

RESTful – Contents transfer

Resources' contents must be transferred between providers and consumers (in both directions) in an implementation independent representation.

Text (ASCII characters organized in lines) is a universally supported concept and, within some limits, it's also acceptable for human reading. For those reasons it's widely used to represent data, nevertheless, rules must be established so that data represented in text format can be analysed by applications.

We already are aware about the HTML specification that uses text, and yet HTML is more focused on data presentation on not so much in data representation.

A somewhat similar, but more generic specification is Extensible Mark-up Language (XML), RESTful constraints don't impose the use of XML, but they clearly point out to the use of either XML or JSON to represent generic data.

When web services resources are transferred between applications in XML format, the **Content-type: application/xml** HTTP header line should be added. When contents are in JSON format, the **Content-type: application/json** HTTP header line should be added.

Extensible Mark-up Language (XML)

XML is a data representation format through text, it's designed to be both human-readable and also easy to be processed by applications. The use of XML is one alternative for contents transfer in REST web services.

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. Another difference is, HTML specifies a way to present data to end-users, XML specifies only the data representation.

An 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 opened tag must be closed by an end tag. As with HTML, if a tag doesn't have any data (content) it may be closed immediately by ending it with /> instead of >.

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

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 the data element's properties. Properties should be specified by adding 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.

AJAX - Web browsers as web services consumers

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, namely by using JavaScript.

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

In such objects, the **open()** method is used to create a request (not actually send it), any HTTP method can be used over a specified URL, and HTTP header lines can be set, one by one, with the **setRequestHeader()** method. Then the request may be sent by calling the **send()** method, request are, by default, asynchronous.

Asynchronous means when the **send()** method is called, the application will not be blocked waiting for the response, this is most important for a web browser. Before sending the request, call-back functions must be settled, they will be automatically called when a response arrives.

This technique is known as AJAX (Asynchronous JavaScript and XML), and it allows a dramatic improvement in web pages usability and interaction.

If data is 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.

AJAX - Web browsers as web services consumers

Before sending an asynchronous request, the object's property **onload** should be assigned with a call-back function, it will be called asynchronously (in background) when the response arrives. Once the response arrives, within the **onload** call-back function, the **status** property may be checked for the HTTP response´s status code, of course, value 200 stands for ok.

By default, the **XMLHttpRequest** object has no timeout associated, meaning it will wait forever for a response, however, the **timeout** property can be assigned with a value in milliseconds. 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 a 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's 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 subset of HTTP features, and not the whole HTTP protocol specification, isn't such an extensive task.

This HTTP server project covers most basic static contents retrieval, 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.

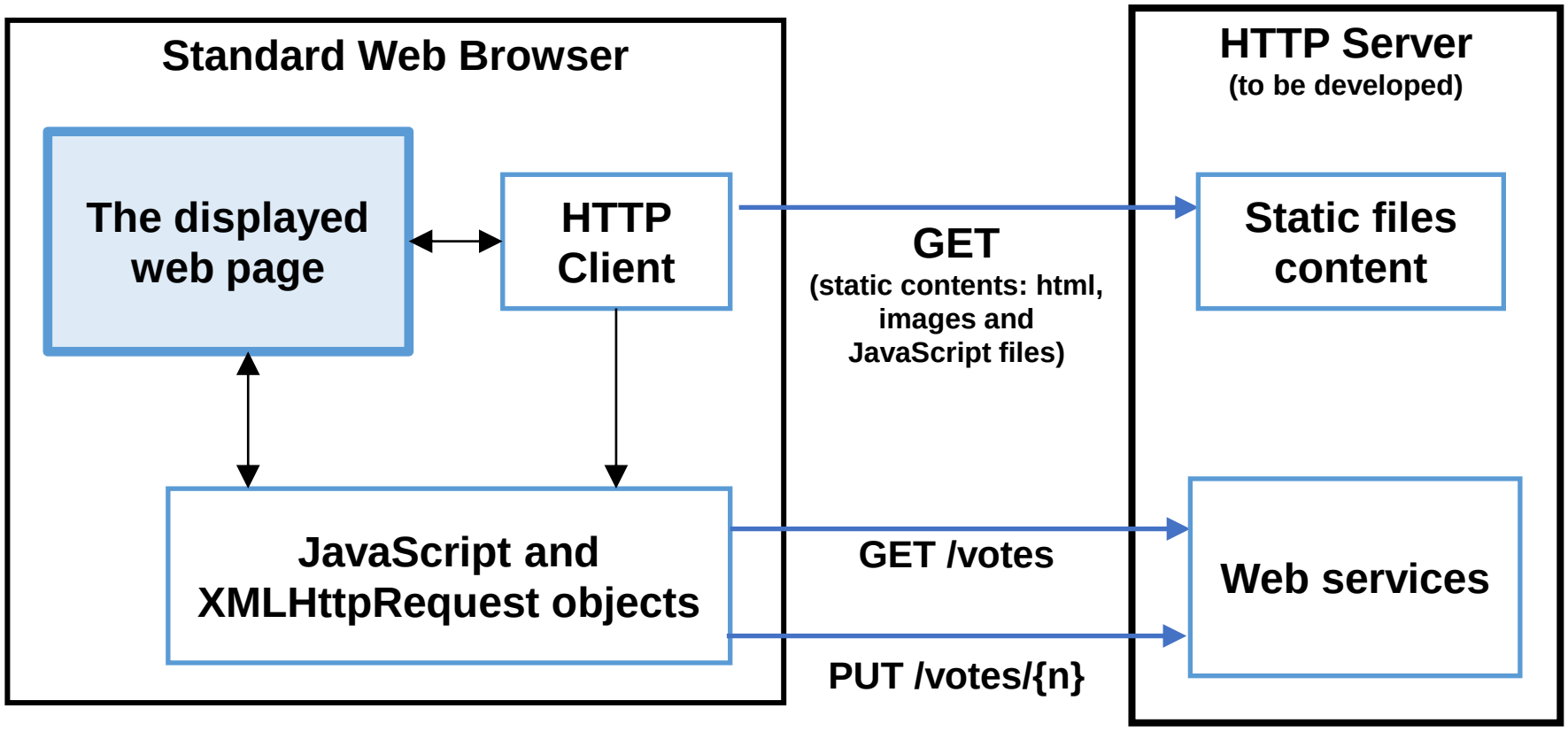
Specific design requirements

The HTTP server is going to have the following characteristics and significant limitations:

- No persistent connections support, thus, the server will always send the **Connection: close** header line to clients.
- All header field lines in client's requests are ignored.
- 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 an established folder. By analyzing the file's name, some relevant content types should be inferred and 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 actually 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 an XML content and it would be up to the consumer (JavaScript) creating the HTML content from it to be presented.

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 header of text lines, 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 such a 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 has to 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 writing a header line, we already know its length, so another possible solution is:

```
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 the content's length. The header will always include the **Connection: close** line, and of course it's ended by an empty line. The sendHttpStringResponse() function, 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 functions as a null terminated string, therefore the content size has to 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 is returned.

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

The sendHttpFileResponse() function handles with 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 the filename's extension is analysed 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 (the content's length) the fseek() and ftell() functions are used. If the caller hasn't provided a status, **200 Ok** is used.
All data required to send the HTTP response message's header is now settled, so sendHttpResponseHeader() is called.

Then, we can start reading data from the beginning of the file (rewind) and send it to the HTTP client as it's read. When there's no more data to read from 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 attain static file contents as requested by clients.

This is just a demo voting system, for this demo only four alternatives (candidates) are established, each candidate's 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, it's mostly for debugging purposes, and 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 AF_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 and then the processHttpRequest() function 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 Inter Process Communication (IPC) would be required to update vote counters on the parent process.

IPC is being avoided 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 it's 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 the **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 an HTML content holding the current votes counting, and necessary HTML tags for vote casting.
Otherwise, it's assumed it must be a reference to a static file, so the URI is appended (strcat) to the BASE_FOLDER and **sendHttpFileResponse()** is called.

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, its 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 included in the content.

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's 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 receive no PUT content because that's the way this service was designed. Because this function is called within the main process, and not in a 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** (www/rcomp-ajax.js), containing some required 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");
          request.timeout = 5000;
          request.send();
          }
```

It’s called once the HTML page is loaded, creates the XMLHttpRequest object and settles call-back functions. For a success the upDate() function is called, it 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 received, the function is called again in 1.5 seconds. Call-back functions are also settled for error events. Finally, the web service to consume 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);
          request.send();
          }
```

It's called by user interaction (clicking the voting button), 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 without a 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 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.

Results - the web page

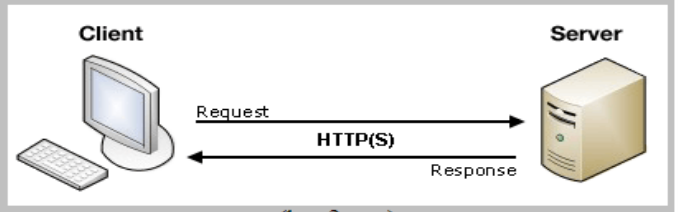
HTTP server demo - Voting with AJAX

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.

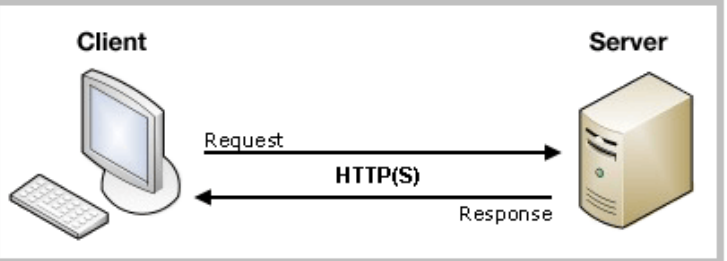
Results - the web page when the server becomes unavailable

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, established for **timeout** and **error** events, will keep trying until the service is available again.