C application server framework (tutorial)

Contents of cApps-get.pdf

The online distribution can be found at https://github.com/nelucozac/cApps

You may download it from https://github.com/nelucozac/cApps/archive/master.zip

This distribution kit will be soon updated and will document:
– POST method with Content-type encoding: *application/x-www-urlencoded* only;
– LOAD ad-hoc method to upload files;
– protection against slowloris post attack;
– session support;
– https support.

Any remark is welcome, if it can improve the quality of this framework and / or documentation. I am also waiting for your questions / opinions about any complex application you wish to develop.

Beware! This document includes personal opinions

Develop your C application server based on the http protocol

cApps accepts GET http requests; session / https not yet detailed

Mail: nelu dot cozac at gmail dot com

Mottoes: *To C or not to C, this is the question*
Keep it simple, stupid (*KISS principle*)

This online distribution kit includes a framework which allows you to develop your own application servers, as easy as you develop a php based application. This framework is based on C language and Linux operating system.

System requirements:
– Linux operating system with IPV6 features (at least for compilation; they may or may not be enabled)
– gcc compiler and gdb (GNU debugger)
– MySQL, including libraries to build C applications (see `csrvI` for details)

I assume you have some knowledge (at least medium level) about C programming, html, css, javascript and http protocol.

Copy on your Linux directory the online distribution kit, which includes the following directories and files:
– `main`: – `cAppserver.c`: the main module of the application you will build
– `noSession.c`: dummy functions to supply the lack of session support
– `cAppserver.h`: declarations – data types, functions and variables to be used by your programs
– `cAppserver.cfg`: configuration file for your application server
– `Stack.c`: test how the stack of your machine grows (downward or upward)
– `Sortlog.c`: sort (for further analysis) the content of error log file
– `csrvA` to `csrvJ`: examples about how to develop your own application server

Every `csrv?` directory includes the complete set of files needed to build and run the program (C sources, compilation scripts – files with `.sh` extension, configuration files, data files etc).

After creating the executable (your server), start it by typing
`./`*csrv* `--start &`
where *csrv* is the name of your executable, so the server will run in background. The debug version doesn't use multitasking. That is, every request is processed by the same process (*csrv*), one after the other. You may run the server under `gdb` (step by step, examine its variables etc).

The release version uses multitasking, that is, two or more clones may process simultaneous requests. The number of clones (processes) is specified by the configuration file (see Appendix B). The release version uses `clone` function to create parallel processes, so you should know how the stack of your machine grows. Stacks grow downward on all processors which run Linux (except the HP PA processors), so clone stack usually points to the topmost address of the memory space set up for the clone stack (manual of `clone` function). You may compile and run `Stack.c` to determine the stack behavior.

Beware: don't use any `-O`*n* optimization option to compile `Stack.c`

If you know the stack of your system grows upward, drop the following line from `cAppserver.c` file:
`Srvcfg.stkT = Srvcfg.stks;` /* drop this line if stack grows upward */

General remarks:
– The types and functions defined by this framework are all prefixed by `CAS_` (**C** **A**pplication **S**erver)
– The actual version does not include support for dynamically compressed web pages
– I use the Ratliff indentation style for every C source I write
– I use frequently short names for variables and structure members
– On my examples, I don't always check if a function returns an error; you may add such checking if you want

Important! The executable and the configuration file must be in the same directory, must have the same name, the extension of the configuration file must be `.cfg` . You may have a lot of application servers, developed for various problems, so having different configuration files. Each application server must use its own listening port.

The server accepts local messages which may be issued by the administrator:
`./`*csrv* `--`*message*

where *message* may be:

- `cnfg` the server reloads its configuration file
- `data` the server reloads its data files
- `html` the server reloads its html templates
- `show` the server displays some information:
  – how much space is used from `Conn->Bfi`
  – how much space is used from `Conn->Bft`
  – how much memory is used from stack (release version)
  – the list of active clones (release version)
- `stop` the server stops running

Remark about `csrvI`, which is MySQL based

If you wish, you may develop your own application server, based on PostgreSQL (or other free database server). See, for instance, the appropriate documentation for details (the C PostgeSQL API).

Acknowledgements (all my projects – C sources and html templates – are inspired from below references)

https://beej.us/guide/bgnet/ – Beej's Guide to Network Programming

http://www.linuxprogrammingblog.com/all-about-linux-signals?page=show – All about Linux signals

https://uwsgi-docs.readthedocs.io/en/latest/articles/SerializingAccept.html – Serializing accept(), also known as Thundering Herd, also known as the Zeeg Problem

https://developer.yahoo.com/performance/rules.html?guccounter=1 – Best practices for speeding up your web site

https://hackernoon.com/on-the-web-size-matters-e52ac0f5fdbe – On the web, size matters

http://www.kitebird.com/mysql-book/ch06-2ed.pdf – The MySQL C API

You may also read

https://unixism.net/2019/04/linux-applications-performance-introduction/ – Linux applications performance

https://medium.com/@lucperkins/web-development-in-c-crazy-or-crazy-like-a-fox-ff723209f8f5 –
Web development in C: crazy? Or crazy like a fox?

http://www.phatcode.net/res/223/files/html/fwd.html – *About assembly language programming*

https://developer.mozilla.org/en-US/docs/Web/HTTP/Basics_of_HTTP/MIME_types/Complete_list_of_MIME_types

https://medium.com/better-programming/do-not-burn-you-cpu-working-with-css-animations-e4d19069fb0f

https://cr.yp.to/bib/1995/wirth.pdf – Niklaus Wirth, *A plea for lean software*, *Computer*, *February 1995*

The above links were available in 2019. Some (or all) of them could be available today, too.

Directory `csrvA`: My first application server

Source file `csrvA.c`

```
#include "cAppserver.h"

static void processRequest(CAS_srvconn_t *Conn) {
CAS_nPrintf(Conn,"Hello world!");
}

void CAS_registerUserSettings(void) {
CAS_Srvinfo.preq = processRequest;
}
```

Type the first or the second command (compile script), then the third:

`./dbgsA.sh`	to obtain the debug version or
`./cmplsA.sh`	to obtain the release version
`./csrvA -start &`	to launch the server

Open your browser and type one of the following URLs:
`http://`*IpAddressOfYourLinuxMachine*`:5001`
`http://`*DomainNameOfYourLinuxMachine*`:5001`

If a domain name is assigned to your Linux machine, you may use this name instead of its Ip address. The browser may run on your Linux system or on other computer. You will see this page:

Hello world!

To terminate the server, type
`./csrvA -stop`

The `dbgsA.sh` file includes two lines:

```
gcc -g -I../main/ ../main/cAppserver.c ../main/noSession.c csrvA.c -o csrvA
chmod 700 csrvA
```

The `cmplsA.sh` file includes three lines:

```
gcc -O3 -D _Release_application_server -I../main/ ../main/cAppserver.c
     ../main/noSession.c csrvA.c -o csrvA
chmod 700 csrvA
strip csrvA
```

The `csrvA.c` file includes the `#include` directive and two functions: `CAS_registerUserSettings` and `processRequest`.

`CAS_registerUserSettings` is called only once by the main module of the server, at the start time. This function must include at least one statement: recording the most important function which will process every request of the client, which we called `processRequest` (but you may use any name you like).

The main module calls `CAS_Srvinfo.preq` each time a new connection is established. `CAS_nPrintf` sends a formatted string to the client (browser). See Appendix C for details about data structures `CAS_srvinfo_t` and `CAS_srvconn_t`, Appendix B for details about predefined functions and Appendix E for details about format descriptors.

You don't need to write any other special function, nor the `main` one (which is included by the main module, `cAppserver.c`). Every necessary operation is done by the main module of the server. You may create your own `Makefile`, but I prefer the above variants. When I consider the server (debug version) passes all my tests, then I generate the release version, which is a bit faster.

If necessary, modify the attributes of `cmplsA.sh` and `dbgsA.sh` (if you get *Permission denied* error):
`chmod 700 *.sh`

You may also compile in advance the `cAppserver.c` file to obtain binary object files for debug and release version, respectively:

```
gcc -g cAppserver.c -c -o cAppsd.o
```

```
gcc -O3 -D _Release_application_server cAppserver.c -c -o cAppsr.o
```

You should also compile `noSession.c` file; it is enough to specify the `–O3` option only, you don't need distinct binary objects for debug and release versions. These binary object files may later be used to build any application server.

After compiling your sources to obtain binaries for the release version, you no longer need to specify the `–D` option. The compilation phase will be faster.

Directory `csrvB`: How to obtain GET parameters and http headers

Source file `csrvB.c`

```
#include "cAppserver.h"

static void processRequest(CAS_srvconn_t *Conn) {
char *Meth,*Sec,*Pnam,*Pval,Ip[INET6_ADDRSTRLEN];
struct tm Tim;
switch (Conn->Bfi[0]) {
       case 'G': Meth = "GET"; break;
       case 'P': Meth = "POST"; break;
       case 'L': Meth = "LOAD"; break;
       }
Sec = Conn->Bfi[1] ? "secure": "not secure";
CAS_nPrintf(Conn,"Method: %s, %s<br>",Meth,Sec);
CAS_nPrintf(Conn,"Unix time stamp (server): %D<br>\n",(long long)Conn->uts);
CAS_nPrintf(Conn,"Current date (yyyy/mm/dd) and time (hh:mm:ss) is: ");
localtime_r(&Conn->uts,&Tim);
CAS_nPrintf(Conn,"%d/%02d/%02d %02d:%02d:%02d<br><br>\n",Tim.tm_year+1900,
            Tim.tm_mon,Tim.tm_mday,Tim.tm_hour,Tim.tm_min,Tim.tm_sec);
inet_ntop(CAS_Srvinfo.af,Conn->Ipc,Ip,sizeof(Ip));
CAS_nPrintf(Conn,"Ip address of client: %s<br>\nParameters<br>\n",Ip);
for (Pnam=NULL; Pnam=CAS_getParamName(Conn,Pnam); )
    for (Pval=NULL; Pval=CAS_getParamValue(Conn,Pnam,Pval); )
        CAS_nPrintf(Conn,"%s = %s<br>\n",Pnam,Pval);
Pnam = "a";
CAS_nPrintf(Conn,"<br>Last value of %s = %s",Pnam,CAS_getLastParamValue(Conn,Pnam));
CAS_nPrintf(Conn,"\n<br><br>Headers<br>");
for (Pnam=NULL; Pnam=CAS_getHeaderName(Conn,Pnam); )
    CAS_nPrintf(Conn,"%s: %s<br>\n",Pnam,CAS_getHeaderValue(Conn,Pnam));
Pnam = "uSer-aGent";
CAS_nPrintf(Conn,"<br>%s: %s<br><br>\n",Pnam,CAS_getHeaderValue(Conn,Pnam));
CAS_nPrintf(Conn,"Elapsed time : %.3f\n",CAS_getTime(Conn));
}

void CAS_registerUserSettings(void) {
CAS_Srvinfo.preq = processRequest;
}
```

If you simply access this server:
`http://`*AddressOfYourLinuxMachine*`:5002`
the browser will display something like this:

Method: GET, not secure

Unix time stamp (server): 1540453425
Current date (yyyy/mm/dd) and time (hh:mm:ss) is: 2018/10/25 10:43:45

Ip address of client: *nnn.nnn.nnn.nnn*
Parameters

Last value of a =

Headers
Host: *nnn.nnn.nnn.nnn*:5002
Connection: keep-alive
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 AppleWebKit/537.36 (KHTML, like Gecko) Chrome/65.0.3325.162 Safari/537.36
Accept: text/html, application/xhtml+xml, application/xml; q=0.9, image/webp, image/apng, */*; q=0.8

```
Accept-Encoding: gzip, deflate
Accept-Language: en-US,en;q=0.9

uSer-aGent: Mozilla/5.0 AppleWebKit/537.36 (KHTML, like Gecko) Chrome/65.0.3325.162 Safari/537.36

Elapsed time: 0.002
```

Now try this URL (or something similar):
`http://`*AddressOfYourLinuxMachine*`:5002/?a=qaz&b=poiu&c=qwerty&a=plmn&A=tgb`

The browser also displays the parameters and the http headers (information sent by the browser).

```
Method: GET, not secure

Unix time stamp (server): 1540453481
Current date (yyyy/mm/dd) and time (hh:mm:ss) is: 2018/10/25 10:44:41

Ip address of client: nnn.nnn.nnn.nnn
Parameters
a = qaz
a = plmn
b = poiu
c = qwerty
a = qaz
a = plmn
A = tgb

Last value of a = plmn

Headers
Host: nnn.nnn.nnn.nnn:5002
User-Agent: Mozilla/5.0 (Windows NT 6.3; Win64; x64; rv:59.0) Gecko/20100101 Firefox/59.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
DNT: 1
Connection: keep-alive
Upgrade-Insecure-Requests: 1
Cache-Control: max-age=0

uSer-aGent: Mozilla/5.0 (Windows NT 6.3; Win64; x64; rv:59.0) Gecko/20100101 Firefox/59.0

Elapsed time: 0.003
```

The User-agent values are not identical because I used different browsers to interrogate the server.

The Unix time stamp (server) is displayed as `long long` value (`Conn->uts`, descriptor `%D`).

The IP address of the client (`Conn->Ipc`) is converted to a character string using the `inet_ntop` function. `CAS_Srvinfo.af` is the address family, initialized at the running time according to the configuration file.

The first cycle traverses (by calling `CAS_getParamName` function) the list of parameters: `a`, `b`, `c` and `A`. For each parameter, another (inner) cycle discovers (by calling `CAS_getParamValue` function) and displays its values. The `a` parameter appears twice in this list, so all of its values are displayed twice.

If we are only interested about the last value of `a` parameter, we use `CAS_getLastParamValue`. If the parameter name is not found, an empty string is returned. See Appendix D for details, if you want this function to return `NULL` in such cases.

Next, the headers sent by the browser are displayed. There is a difference between the management of parameters and headers. Some parameters may appear twice or more in this list, and the name is case sensitive. Every header is supposed to appear only once, and its name is case insensitive.

Finally, the elapsed time from the point of connection (returned by `CAS_getTime`) is displayed. The format descriptor for `double` values is `%.`*n*`f` (*n* digits after the decimal point).

You may use constructions like `%`*hh* (**%** followed by two hexadecimal digits). Do not use `%00` because it will create a null terminated string. The server does not manage other string types. If you must manage non-standard strings (with one or more null characters), use the Base64 encoding feature of your browser before sending the request. See `csrvJ` for details about how to implement Base64 encoding and decoding in your C application.

Another way to scan the parameter list

The above sequence, which uses two cycles, may retrieve some parameters twice (or more times), if they are multiplied. You may use another sequence which displays every parameter at the time of its finding:

```
for (Pnam=NULL; Pnam=CAS_getParamName(Conn,Pnam); )
    CAS_nPrintf(Conn,"%s = %s<br>\n",Pnam,CAS_getThisParamValue(Conn,Pnam));
```

`CAS_getThisParamValue` must use, as the second argument, a pointer variable whose value was previously returned by `CAS_getParamName`, otherwise you will get unpredictable results. See Appendix D for details about `CAS_getThisParamValue`.

Remarks:

- When `processRequest` is called, the URL (GET parameters) and headers are already parsed, and the connection time (`Conn->uts`, unix time stamp) is set.
- `processRequest` and its subsequent functions should use local variables for read and write operations. Every global variable must be used only for read operations, or protected by mutexes (see `csrvF`). There may be two or more clones running simultaneously, so they must not interfere.
- Every subsequent functions of `processRequest` must be thread safe.

About using `exec` functions

If you want to call one of the `exec` functions, you must create a new process (for instance, using `fork` call) which will call the needed `exec`. Otherwise, the caller clone (process) will be replaced by the new process image (see `man exec` for details). The parent (caller) and the new process will run concurrently. The caller may wait for the termination of the new process, using `wait` or `waitpid` functions call. You don't need to manage any SIGCHLD signal, it is automatically resolved.

Beware!

– Every URL parameter must have a name (non empty string), otherwise some parameters (including every no name parameter) will not be retrieved. The value of any parameter may be empty string.

– Do not alter any parameter or header name / value; see `csrvI` (using `Conn->Usr` as alternative for modifiable GET parameters).

– If the URL is malformed due to some errors (for instance, bad usage of the characters **=**, **&** or **%**), the server will not identify correctly the names and the values of all parameters.

– If the new process takes too much time to be finished, the release version may terminate the caller clone with SIGALRM signal, if this event is not managed by the programmer. See `csrvH` for details about managing the timeout problem.

– There is a limit for the maximum number of processes per user. You must take into account the number of active clones if you run the release version. Type the following command to see the system limits:
`ulimit -a`

Directory `csrvC`: Sending files to the browser
Implementing rewrite rules, managing Content-type information

If you type the following URL (or something similar):
`http://`*AddressOfYourServer*`:5001/csrvA.c`

the previous servers will send `404 Not found` response header and `Not found` message as content body. Our servers don't know what to do when the client requests a particular file. Sending a file to the client means sending the following information (this is the template format):

```
HTTP/1.1 200 Ok
Content-type: %s
Content-length: %u
Content-disposition: inline; filename=%s
Connection: close
```

Content of the requested file

The *Content-type* depends on the file type (extension): `application/pdf` (pdf document file), `image/gif` (gif image file) etc.

We will develop now a module which understands the following URL (or something similar):
`http://`*AddressOfYourServer*`:5003/?File=A.pdf`

But the user will not type such stupid URLs. So we need a mechanism to translate the simple URL to another, a bit more detailed, which may easily be processed by our server. To achieve this goal we use the rewrite rules feature.

Source file `csrvC.c`

```
#include "cAppserver.h"

static void rewriteRules(CAS_srvconn_t *Conn) {
char *P;
P = strchr(Conn->Bfi,'/') + 1;
if ((*P==0) || isspace(*P) || (*P=='?')) return;
memmove(P+6,P,strlen(P)+1);
memcpy(P,"?File=",6);
P = strchr(Conn->Bfi,'=') + 1;
do {
   c = *P;
   if (!c || isspace(c)) break;
   if (c=='?') {
      *P = '&';
      break;
      }
   P++;
   } while (1);
}

static void userConfig(char *Cfg) {
Cfg = CAS_buildMimeTypeList(Cfg);
/* Cfg points now to the next information on this section */
}

static void processRequest(CAS_srvconn_t *Conn) {
char *Fil;
if (Fil=CAS_getParamValue(Conn,"File",NULL)) {
   CAS_sendFileToClient(Conn,Fil,CAS_Srvinfo.Rh[3],NULL);
   return;
   }
```

```
CAS_nPrintf(Conn,"Vous n'avez pas demandé aucun fichier");
}
void CAS_registerUserSettings(void) {
CAS_Srvinfo.preq = processRequest;
CAS_Srvinfo.cnfg = userConfig;
CAS_Srvinfo.rwrl = rewriteRules;
}
```

userConfig()

The `userConfig` function is first called when the server is launched, immediately after reading of *Server configuration* section. This function is also called every time the administrator sends `--cnfg` message to the server. `userConfig` reads a pair of items (extension, content-type). We first need a list of extensions and the associated content types (see the *User specific configuration* section of `csrvC.cfg` file):

```
.pdf application/pdf
.dnd ?
.gif image/gif
...
* application/octet-stream
```

The last entry of this list (*****) must be present and is reserved for extensions which don't match any of the above. If you want to deny sending files with particular extensions, just use the `?` character instead of content type (example: `.dnd` extension, from **d**o **n**ot **d**ownload). The last entry may be of the following type:
`* ?`
which means: any other extension is not allowed for download.

The list is read by `CAS_buildMimeTypeList`, which builds a list of pairs to be used by `CAS_sendFileToClient`. The `Cfg` argument of `userConfig` points to the information stored in *User specific configuration* – in our case, the list of pairs (extension, content type). `CAS_buildMimeTypeList` returns a pointer to the next information of *User specific configuration* (if any).

You may edit the configuration file, then type the following command:
`./csrvC --cnfg`

To process this request, the server waits until every clone finishes its work, then reloads the configuration file. After processing the information from *Server configuration*, `CAS_Srvinfo.cnfg` is called (if defined). After processing the whole configuration information, the server is ready to accept new requests.

rewriteRules()

The `rewriteRules` function checks for the existence of a filename, and in this case inserts a small string inside the input buffer: `?File=` and eventually replaces `?` by `&`. This function is called every time a GET request is sent to the server. `rewriteRules` examines the content of `Conn->Bfi` (buffer for input strings, the message received from client). If the content is something like

`GET / ...` (space after slash)
or
`GET /?...` (question mark after slash)

we have nothing to do. Otherwise we just insert a small string inside the input buffer, so the new content becomes
`GET /?File=`*filename*`...`

If the filename is followed by `?` character, it is replaced by `&`.

processRequest()

The GET request is parsed when `rewriteRules` returns (parameters and header messages). So we can test if `File` parameter is present; in this case, `CAS_sendFileToClient` is called. The arguments of this function are:

– pointer to information about the current connection;
– path to the requested file;
– the format of response header (in this case `CAS_Srvinfo.Rh[3]`) with the following descriptors (in this order): `%s` (content type), `%u` (file size), `%s` (optional: file name, to be used by the client / browser);
– function to check if the requested file can or cannot be sent; if NULL, every file is accepted.

The format of response header is the fourth on the configuration file, so its index is 3. See Appendix C for details about data types and public variables.

`CAS_sendFileToClient` (with these arguments) accepts every file to be downloaded, if it can be open. You should define your own validation function if you want to choose very carefully which files are allowed to be sent to the client. For instance, you should perform the following checks: the path should begin with `Data/` or `Html/` or something similar. See an example below:

```
static int validFileName(char *Fil) {
if (strstr(Fnm,"../") || (*Fnm=='/') return 0;
if (strcmp(Fil,"favicon.ico")==0) return 1;
if (memcmp(Fil,"Data/",5)==0) return 1;
if (memcmp(Fil,"Html/",5)==0) return 1;
return 0;
}
```

You should accept the `favicon.ico` filename if you choosed to use such "ornaments".

Some particular files are usually displayed by browsers (pdf documents, bitmap images, plain text files). Maybe you need sometimes to download such files directly, without being displayed. You may use the following template in your html page:
`http://`*document*`.pdf?download=yes`

In such cases, you may select the response header depending on the values of supplementary parameters.

Important! If you forget to call `CAS_buildMimeTypeList`, then `CAS_sendFileToClient` will not work.

Personal remark. I don't like at all any web page which is full of css, javascript, images (including very tiny images such as arrows or dots) etc. If you desperately want to use a lot of large css and javascript files, you should compress them with gzip (or something similar) and use an appropriate response header, which includes the following line:
`Content-encoding: gzip`

See Appendix C for details about sending files to the client.

Consult the following pages for some details:

https://royal.pingdom.com/can-gzip-compression-really-improve-web-performance/

https://www.itworld.com/article/2693941/why-it-doesn-t-make-sense-to-gzip-all-content-from-your-web-server.html

An advice from *Best practices for speeding up your web site*: reduce the number of http requests in your page. Read also *On the web, size matters*.

A (complete ?) list of MIME types can be found at the following address:

https://developer.mozilla.org/en-US/docs/Web/HTTP/Basics_of_HTTP/MIME_types/Complete_list_of_MIME_types

Directory `csrvD`: Repetitive html fragments, generating xml pages

This application server allows us to generate the list of squares and cubes from 1 to *n*. The user may also ask to generate the result in xml format. If you simply access it
`http://`*AddressOfYourLinuxMachine*`:5004`
the browser displays the following form:

Squares and cubes from 1 to (max *nnn*): [ ]
Xml format

Now enter *n* (natural number, not greater than *nnn*) and press Enter. The URL becomes
`http://`*AddressOfYourLinuxMachine*`:5004/?n=`*n*
and the following page is displayed (suppose *n* = 5):

Squares and cubes from 1 to (max *nnn*): [5]
Xml format

Number	Square	Cube
1	1	1
2	4	8
and so on		

Source file `csrvD.c`

```
#include "cAppserver.h"

static int Limit;
static char *Htm, *Xml;

static void userConfig(char *Cfg) {
Limit = atoi(Cfg);
}

static void manageUserHtml(char op) {
if (op=='L') {
   Htm = CAS_loadTextFile("Numbers.htm");
   Xml = CAS_loadTextFile("Numbers.xml");
   }
else {
   free(Htm);
   free(Xml);
   }
}

static void processRequest(CAS_srvconn_t *Conn) {
char *Fmt;
int n,i,s;
long long c;
n = atoi(CAS_getLastParamValue(Conn,"n"));
if (n<0) n = 0;
if (n>Limit) n = Limit;
if (CAS_getParamValue(Conn,"Xml",NULL)) {
   CAS_resetOutputBuffer(Conn);             /* because Xml get parameter is present */
   CAS_nPrintf(Conn,CAS_Srvinfo.Rh[3]);  /* this sequence (three lines) prepares */
   Fmt = Xml;                               /* Bfo for xml format */
   }
else Fmt = Htm;                      /* Xml get parameter is absent, use html format (default) */
CAS_nPrintf(Conn,Fmt,Limit,n,n);     /* the first part of html / xml template */
Fmt = CAS_endOfString(Fmt,1);        /* now Fmt points to the second part of html / xml template */
for (i=1; i<=n; i++) {
```

```
    s = i * i;
    c = (long long)s * i;
    CAS_nPrintf(Conn,Fmt,i,s,c);  /* the second part of html / xml template */
    }
Fmt = CAS_endOfString(Fmt,1);
CAS_nPrintf(Conn,Fmt);            /* the third part of html / xml template */
}

void CAS_registerUserSettings(void) {
CAS_Srvinfo.preq = processRequest;
CAS_Srvinfo.cnfg = userConfig;
CAS_Srvinfo.html = manageUserHtml;
}
```

`userConfig()`

Reads `Limit` from the configuration file (*User specific configuration*), the maximum limit for the number of rows of the response table.

`manageUserHtml('L')`

Reads two templates used to display the server response:

- `Numbers.htm` for html format
- `Numbers.xml` for xml format

The two templates are quite similar, so we can use the same program sequence to generate the response. See below the content of each template file and Appendix D for details about `CAS_loadTextFile` function.

`manageUserHtml('R')`

Frees the memory previously allocated for html and xml templates.

`Numbers.htm`

```
<!Doctype html public "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
  <title>Numbers, squares, cubes</title>
  <meta charset="UTF-8">
</head>
<body><form action="/">
  Squares and cubes from 1 to (max %d)
  <input type=text name=n size=4 maxlength=4 value="%.0d">
  <br><br><a href="/?Xml=on&n=%.0d" target=_blank>
  Xml format</a><br><br>
  <table frame=box rules=all>
    <tr align=center>
      <td>Number</td>
      <td>Square</td>
      <td>Cube</td>
    </tr>
    <!-- Break -->
    <tr align=center>
      <td>%d</td>
      <td>%d</td>
      <td>%D</td>
    </tr>
    <!-- Break -->
  </table>
```

```
</form></body>
</html>
```

We observe two comments with *Break* text. These special comments split the file content into three parts. If the file extension is `.htm*` or `.xml*`, and the file contains *Break* comments, the content is split such that each special comment is replaced by one null octet.

Now let us suppose the `Xml` parameter is absent. The first part of this template (`Numbers.htm`) is displayed once, when `processRequest` identifies the parameters. The second part is displayed for every *i* value ($i = 1, \ldots, n$) but only if $n > 0$. The third part is displayed at the end of `processRequest`.

Numbers.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<Numbers>
<Inf L="%d" N="%d">
<!-- %.0d -->
<!-- Break -->
<Row n="%d" s="%d" c="%D"/>
<!-- Break -->
</Numbers>
```

The structures of `Numbers.htm` and `Numbers.xml` are quite similar:
– three parts separated by *Break* comments;
– the first part includes two descriptors `%d`, `%d` (but html template includes three descriptors, see remark below);
– the second part includes descriptors `%d`, `%d`, `%D` (`%D` descriptor is used for `long long` values);
– the third part does not include any descriptor.

Remark. The first part of the html template uses three descriptors, but the corresponding part of the xml template uses only two descriptors. The corresponding call of `CAS_nPrintf` uses three integer values but, if the xml template is used, only the first two values are used.

`Numbers.xml` was designed to have the same structure as `Numbers.htm`. This way we may use the same function to generate both html and xml pages.

Important. The descriptors used by the html / xml templates must be pairwise (type) compatibles with the arguments of corresponding `CAS_nPrintf` calls.

processRequest()

Expects two parameters:
- `n`: number of lines to be displayed (0 ≤ `n` ≤ `Limit`)
- `Xml`: if present, the server response is displayed in xml format, otherwise in html format

`processRequest` takes the value of `n` parameter and convert it to an integer value between 0 and `Limit`. Next, if `Xml` parameter is present, some preparations are necessary to generate an xml response page (in fact, to announce the browser about the content type) and `Fmt` is set to `Xml`. If `Xml` is absent, `Fmt` is set to `Htm` (the response format will be html).

To generate an xml page we need a special response header (see the beginning of the configuration file). This is the fourth header of the configuration file (`csrvD.cfg`) and is identified by `CAS_Srvinfo.Rh[3]`. Because the output buffer is already initialized with the default html response header, we must reset it and re-initialized with the appropriate response header (xml).

After performing some initializations, `Fmt` points either to `Htm` (html format) or `Xml` (xml format). The Fmt format is used to print three values (html format) or two values (xml format).

Next, `Fmt` is set to the second part of the html / xml template. The `CAS_endOfString(S,b)` macro points `b` octets after the end of `S` string. The square and the cube of `i` are computed and displayed ($i = 1, \ldots n$).

Finally, `Fmt` is set to the third part of the html / xml template, which is displayed.

Xml response header

```
HTTP/1.1 200 Ok
Content-type: text/xml
Expires: 0
Connection: close
```

Modifying html / xml templates

We may modify the html / xml templates without modifying the server program. For instance, we can add supplementary texts, inline images, we can add css styles to change the table look etc.

After modifying the template(s), we must announce the server about changes:

```
./csrvD --html
```

If we want to use a very different way to display some variable values / texts (the order of values in the html template is changed), or to display new variable values / texts etc, then we must follow these steps:
– edit the source program and the html / xml templates
– recompile the server program
– stop and restart the server program:

```
./csrvD –stop
./csrvD –start
```

Important! The `csrvD` example is about how to use html templates, which are kept in separate files. This assertion is also valid for the next examples: `csrvE`, `csrvF`, `csrvG` and `csrvI`.

You may use (if you like) a very different (awkward) programming style:

```
int n = 5;
CAS_nPrintf(Conn,"<html>\n<body>Value: %d\n",n);
CAS_nPrintf(Conn,"</body>\n</html>\n");
```

Do you want to edit, recompile, stop and restart your server program every time you must change the look of the response pages ?

Directory `csrvE`: More about repetitive html fragments

This application server displays, for each $i = 1, ..., n$, a table with the powers i^j ($j = 1, ..., c$); n and c are given by the user. Compile and run this application server, then connect to it:
`http://`*AddressOfYourLinuxMachine*`:5005`

Source file `csrvE.c`

```
#include "cAppserver.h"

static int Nrows, Ncols;
static char *Htm;

static void userConfig(char *Cfg) {
sscanf(Cfg,"%d %d",&Nrows,&Ncols);
}

static void manageUserHtml(char op) {
if (op=='L') Htm = CAS_loadTextFile("Powers.htm");
   else free(Htm);
}

static void processRequest(CAS_srvconn_t *Conn) {
char m;
int n,c,i,j;
long long p;
struct { char *begin_htm, *begin_row, *head, *cell, *end_row, *end_htm; } Format;
n = atoi(CAS_getLastParamValue(Conn,"n"));
c = atoi(CAS_getLastParamValue(Conn,"c"));
if (n<0) n = 0;
if (n>Nrows) n = Nrows;
if (n>0) {
   if (c<2) c = 2;
   if (c>Ncols) c = Ncols;
   }
CAS_explodeHtm(Htm,&Format,sizeof(Format));
CAS_nPrintf(Conn,Format.begin_htm,Nrows,n,Ncols,c);
if (n>0) {
   CAS_nPrintf(Conn,Format.begin_row);
    for (j=1; j<=c; j++)
        CAS_nPrintf(Conn,Format.head,j);
   CAS_nPrintf(Conn,Format.end_row);
   }
for (i=1; i<=n; i++) {
    CAS_nPrintf(Conn,Format.begin_row);
    for (j=p=1; j<=c; j++) {
        p *= i;
        CAS_nPrintf(Conn,Format.cell,p);
        }
    CAS_nPrintf(Conn,Format.end_row);
    }
CAS_nPrintf(Conn,Format.end_htm);
}

void CAS_registerUserSettings(void) {
CAS_Srvinfo.preq = processRequest;
CAS_Srvinfo.cnfg = userConfig;
CAS_Srvinfo.html = manageUserHtml;
}
```

`userConfig()`

Reads (from configuration file) `Nrows` (maximum number of rows) and `Ncols` (maximum number of columns).

`manageUserHtml('L')`

Loads `Powers.htm`, the template used to display the server response.

`manageUserHtml('R')`

Frees the memory previously allocated for html templates.

`Powers.htm`

```
<!Doctype html public "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
  <title>Numbers, squares, cubes</title>
  <meta charset="UTF-8">
</head>
<body><form action="/">
  Powers of numbers from 1 to (max %d)
  <input type=text name=n size=4 maxlength=4 value="%.0d"><br><br>
  Powers from 2 to (max %d)
  <input type=text name=c size=1 maxlength=1 value="%.0d"><br><br>
  <input type=submit name=submit value=Submit><br><br>
  <table frame=box rules=all>
    <!-- Break -->
    <tr align=center>
      <!-- Break -->
      <td><i>n</i><sup>%d</sup></td>
      <!-- Break -->
      <td>%D</td>
      <!-- Break -->
    </tr>
    <!-- Break -->
  </table>
</form></body>
</html>
```

The *Break* comments split the html template into five parts::

- `begin_htm`: the beginning of html template
- `begin_row`: the beginning of a new row
- `head`: a head cell
- `cell`: a cell
- `end_row`: the end of row
- `end_htm`: the end of html template

The previous example (`csrvD`) displayed a table with a variable number of rows, but the number of columns was fixed (three). Now, the number of columns is also variable. This is why we need separate members to build a new row.

`processRequest()`

Expects two parameters:

- `n`: the number of rows ($0 \leq$ `n` $\leq$ `Nrows`)
- `c`: the number of columns (if `n` > 0, $2 \leq$ `c` $\leq$ `Ncols`)

We use now a different technique to generate the response table. The previous example used the `nextString` macro to point, step by step, to every part of the html / xml template. We used a simple technique because the template structure was also simple.

Now, the template structure is more complex:
– `begin_row` and `end_row` must be used once for every new row;
– `head` must be used for every column between `begin_row` and `end_row` to generate the texts n^1, n^2 ..., before displaying the rows with computed values;
– `cell` must be used for every column between `begin_row` and `end_row` to display the computed values.

`CAS_explodeHtm` fulfills the `Format` structure such that each member points to the corresponding part of the html template. The `Format` structure must include only pointers to `char` and the number of members must be big enough to cover the whole template. See Appendix D for details about `CAS_explodeHtm`.

Directory `csrvF`: How to use mutexes (online dictionary)
Updating server data

This application server implements an online dictionary. Compile and run the server, then connect to it:
`http://`*AddressOfYourLinuxMachine*`:5006`

The current distribution kit includes a very simple dictionary with two words and three definitions, one word having two distinct definitions. The application uses two distinct files:
– `Dictio.dat`: the list of definitions, each definition on a single line;
– `Dictio.idx`:
  – the number of definitions, the maximum length of a word, the maximum length of a definition;
  – on the next lines, the word and the offset of the corresponding definition in `Dictio.dat` .

When the server program is launched, it opens the `.dat` file and stores its handler for next usages, then loads the `.idx` file and convert it to a structure which allows us to find quickly every word entered by the user.

You may begin to enter a word; if the entered letters are valid, a list of matching words is displayed (the list size is limited to ten words); otherwise, nothing is displayed. When you press *Enter*, the corresponding definitions are displayed (if the word was found in the dictionary).

Using fast mutexes

Let us examine a special sequence from `csrvF.c` module:

```
CAS_serverMutex(Conn,&Dic_info.mtx,'L');
lseek(Dic_inf.dfh,Pinfw->p,SEEK_SET);
l = read(Dic_inf.dfh,Def,Dic_inf.mld);
CAS_serverMutex(Conn,&Dic_info.mtx,'R');
```

When our server receives a request for the definition of a particular word, it searches this word in `Dic_inf.Wrd` list and, if the word is found, detects the first entry related to the searched word. This entry includes the position of its definition in `Dictio.dat` file and the word itself. Now the server is able to perform an `lseek` operation followed by `read` to obtain the definition.

But wait! What if, after performing `lseek` and before `read`, the current clone is suspended by the operating system and another clone solves a new request related to another word ? The new clone performs another `lseek` operation and reads the corresponding definition. When the previous clone gets the control, it will read a wrong definition.

So we need a mechanism to protect the sequence `lseek . . . read` from being altered by another clone. When `CAS_serverMutex` is called with `'L'` as the last argument, the system checks the state of `mtx` mutex. If it is not locked, the system locks it to be used exclusively by the current clone. From now on, any other clone, which asks to lock the same mutex, must wait until it is released by the current clone.

After performing the `read` operation, the current clone releases the `mtx` mutex. Now, if another clone is waiting for the same mutex, it will be awoken by the system. We can't predict which clone will be awoken if two or more clones are waiting for the same mutex. The clone can't lock twice the same mutex before releasing it.

If the logic of your application requires to use two or more simultaneous mutexes, you must be very carefull to avoid deadlocks. As a rule of thumb, every clone should lock / release two or more mutexes in the same order. Every mutex must be global variable of `int` type and `CAS_registerUserSettings` must set it to 1. You may also initialize it to 1 at the compilation time.

How to create a random dictionary ?

Compile the `Credic.c` source file using the `Cmpaux.sh` script file (compile auxiliary programs); the `Credic` executable file will be created. Run `Credic` and enter the following information: number of words, minimum and maximum length of words, minimum and maximum length of definitions. `Credic` creates two files, `Temp.dat` and `Temp.idx`, a random dictionary based on the entered information. If `csrvF` (the application server) is active, it receives a special message which informs it that new data will be used from now on.

In order to talk to the server, `Credic` performs these steps (see the source fragment below):
a) connects to the server using the local host and port; the information are taken from server configuration file;
b) if the connection is established, sends the `--wait` message to the server;
c) renames the above generated files such that the server will be able to use them;
d) when the files are successfully renamed, sends the `Ok data` message to the server;
e) closes the connection.

The first two steps are performed by `contactServer`, which extracts from server configuration file the needed information. The last three steps are performed by the main module of the server.

`csrvF` processes the `--wait` message (which can not be sent by command line) as follows:
– when this message is received, waits until every previous request is processed(every clone becomes inactive);
– sends an `Ok` . . . message to the client (`Credic`);
– waits again for a new message from `Credic`, which should be the `Ok data` string;
– calls `CAS_Srvinfo.data(`*op*`)` twice, first with `'R'` (release old data), then with `'L'` (load new data).

You may wonder: if `--data` and `--wait` messages are practically processed the same way, isn't the `--wait` message superfluous ? Let us suppose `Credic` only creates the new dictionary and sends the `--data` message, without sending supplementary messages like `Ok data` and so on. Now `Credic` would perform the following steps:
a) renames `Temp.dat` to `Dictio.dat`
b) renames `Temp.idx` to `Dictio.idx`
c) sends `--data` message

What if the operating system suspends the client (`./csrvF --data`) after step a) or b) – before step c), and the server receives an external request to display definitions ? The old list of words (kept in main memory) and the new data file don't match at all. This is the difference between the two messages: when the server receives the `--wait` message, it will wait for a very short time until the `Ok` message is received. This very short time is used by `Credic` to rename the necessary files, such that they will be available to the server.

Sometimes you may update the server using `--data` message. Suppose you edit `Dictio.idx` adding a new word and its associate position in `Dictio.dat` file, which must be the end of this file. You must also update the number of words (beginning of `.idx` file). Then you edit `Dictio.dat` ***appending*** the new definition, and so on. In this case, you may safely send the `--data` message to the server: the list of words (kept in main memory) and `Dictio.dat` file match perfectly any time.

Important! The maximum length of words plus maximum length of definitions (including the null terminator string of each item) and the size of `Conn->Bft` (buffer for temporary strings) must be correlated. The `csrvF.c` module allocates space for the entered word and any of its definition from `Conn->Bft` .

How to inform the server about the fact that new data are available (general framework below)

```
#include "cAppserver.h"

/*  data types and global variables; depending on the problem to be solved  */

static void firstUpdatePhase(void) {
/*  initializations, code depending on the problem to be solved; usually creates new data files  */
}

static void secondUpdatePhase(void) {
/*  finalizations, code depending on the problem to be solved; usually renames new data files  */
}

/*  from this point on, keep the code as is  */
static char *Mssg;
static int contactServer(char *Ncfg) {
int Fc,p,v,fs,t;
char Pswd[2004],Ipa[48],*P;
time_t ti;
```

```
union { struct sockaddr_in v4; struct sockaddr_in6 v6; } Sadr;
Fc = open(Ncfg,O_RDONLY);
if (Fc<0) return 0;
fs = lseek(Fc,0,SEEK_END);
Mssg = calloc(fs+1,1);
lseek(Fc,0,SEEK_SET);
fs = read(Fc,Mssg,fs);
close(Fc);
memcpy(Mssg,"--wait ",7);
P = strstr(Mssg,"- Server ");
P = strchr(P,'\n');
sscanf(P,"%s %d %*s %s %d %*s %*s %*s %*s %*s %*s %d",Pswd,&v,Ipa,&p,&t);
strcpy(Mssg+7,Pswd);
memset(&Sadr,0,sizeof(Sadr));
if (v==4) {
   inet_pton(AF_INET,Ipa,&Sadr.v4.sin_addr.s_addr);
   Sadr.v4.sin_family = AF_INET;
   Sadr.v4.sin_port = htons(p);
   v = AF_INET;
   }
else {
   inet_pton(AF_INET6,Ipa,&Sadr.v6.sin6_addr.s6_addr);
   Sadr.v6.sin6_family = AF_INET6;
   Sadr.v6.sin6_port = htons(p);
   v = AF_INET6;
   }
Fc = socket(v,SOCK_STREAM,IPPROTO_TCP);
ti = time(NULL);
while (connect(Fc,(struct sockaddr *)&Sadr,sizeof(Sadr))<0) {
      if (errno!=ECONNREFUSED)
         errno = time(NULL)-ti > t ? ETIMEDOUT: 0;
      if (errno!=0) return 0;
      sleep(1);
      }
send(Fc,Mssg,strlen(Mssg),0);
p = recv(Fc,Mssg,fs-1,0);
Mssg[p] = 0;
if (memcmp(Mssg,"Ok ",3)!=0) {
   fputs("Connection failed, message other than Ok\n",stderr);
   exit(1);
   }
return Fc;
}

int main(int agc, char **Agv) {
int sck;
if (agc!=2) {
   fputs("Update server, argument: server configuration file\n",stderr);
   return 0;
   }
firstUpdatePhase();
sck = contactServer("csrvF.cfg");
if (sck>0) {
   secondUpdatePhase();
   send(sck,"Ok data",7,0);
   close(sck);
```

```
   fputs(Mssg,stderr);
   }
else perror("Can't open configuration file");
return 0;
}
```

Benchmarking the online dictionary

The `Bmark.c` source file shows you how to develop a client program to benchmark a particular application server. First, this program reads the list of words from `Dictio.idx` (in fact, it reads only the first `NREQS` (10000) words; `NREQS` is the total number of requests). Next, you are asked to specify how to contact the server (Ip version, Ip address, port) and the number of simultaneous threads. The number of threads is adjusted because it must be a divisor of `NREQS`.

You may run the benchmark program from local host or from another computer. In the second case, you should have a copy of `Dictio.idx` file. The program will create `nthrs` threads, each of them will perform the same number of requests.

Every thread sends `nrt` requests (number of requests per task): = `NREQS` (total number of requests) `/ nthrs` . The first thread asks for definitions of the first `nrt` words: `Dictio[0],... Dictio[nrt-1]`. The second thread (if `nthrs` ≥ 2) asks for definitions of next `nrt` words and so on.

Finally, some statistics about the performance of the online dictionary are displayed.

Table F.1. Performances of the online dictionary server

Benchmark client		Server (online dictionary, 100000 definitions)					
where it runs	threads	debug version			release version 4 clones		
		total elapsed time	time per request: average	requests per second	total elapsed time	time per request: average	requests per second
local host	1	5.34782	0.00053	1869.92249	4.59041	0.00046	2178.45438
	4	3.81418	0.00038	2621.79891	4.08442	0.00041	2448.32597
	8	4.24091	0.00042	2357.98686	4.00430	0.00040	2497.31603
intranet	1	9.86263	0.00099	1013.92844	10.68372	0.00107	936.00391
	4	8.37401	0.00084	1194.17101	5.95598	0.00060	1678.98369
	8	10.86788	0.00109	920.14240	5.96112	0.00060	1677.53711

Some characteristics of the processor (`lscpu` command): CPU op-mode 32 bits, CPU Mhz 2392.458, vendor-id Genuine Intel, model name Intel® Xeon™, cache size 512 Ko.

Linux operating system: 4.4.0-174-generic (`uname` command).

In practice, servers will not achieve the best possible performance. Many clients may use slow connections, which increase the response time. You should choose reasonable high number of server clones and reasonable low values for stack size and buffers (input, output and temporary strings).

Directory `csrvG`: Another complex application server
Minimum cost path between two nodes of a graph

I assume you have some knowledge (medium or advanced level) about graph theory and algorithms.

Compile and run the server, then connect to it:
`http://`*AddressOfYourLinuxMachine*`:5007`

Now you may enter the departure and the arrival nodes, then click the button to see the result (*Cherchez la route* or *Search route*, depending on the language you selected). Before clicking the button, you may check the checkbox field (*Affichez la route* or *Display the path*). If this field is checked, the server will display both the cost of the path and the path itself (the list of nodes). If not, only the cost will be displayed.

When the server is launched, it reads the graph description from two files:
– `Graph.nod`: the number of nodes (`Graph.nn`) and their names; each node is internally identified by a number between 1 and `Graph.nn` ;
– `Graph.lnk`: the links (arcs) between nodes;
  – the first value is the total number of arcs;
  – the next `Graph.nn` lines includes, for each node x ($1 \le x \le$ `Graph.nn`), the number (k) of its successors, and a list of k couples (y,c), where y is the successor and c is the cost of (x,y) arc.

The list of nodes is stored twice, the first structure keeps the nodes ordered by their identifiers (from 1 to `Graph.nn`), and the second structure keeps the nodes ordered alphabetically.

When the user enters departure and arrival nodes, the server determines their internal identifiers, let us denote them by *dp* (departure) and *ar* (arrival). Next, the server uses the algorithm of Dijkstra to determine the minimum cost path from *dp* to *ar*. Finally, the results are displayed according to the checkbox field.

Generating a new graph

The `Cregraph.c` source program is an example about how to generate a graph. Compile it using the `CmpGr.sh` script file and run `Cregraph`. You will be asked to enter the parameters of the graph you want to generate: the number of squares per edge (*ns*) and the number of nodes per inner square edge (*nn*). Let us suppose *nn* is 3 and *ns* is 2.The program will generate the following graph (figure G.1):

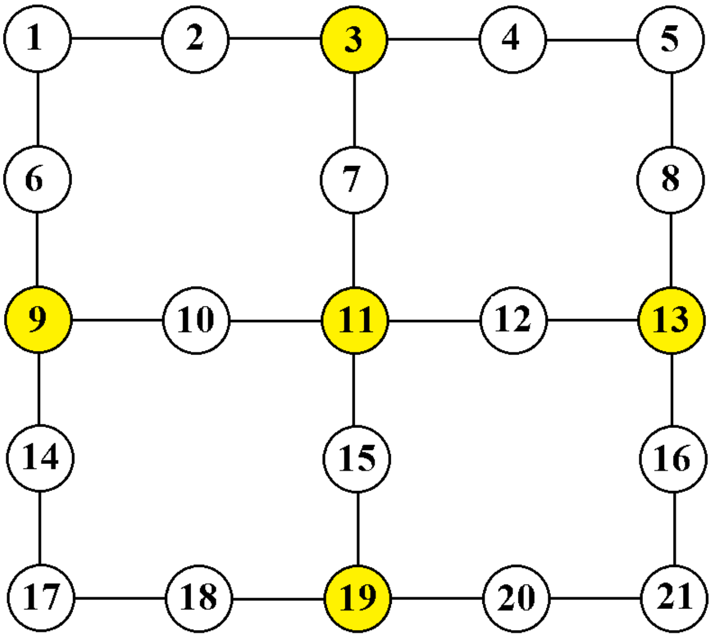


Figure G.1. Graph with *nn* = 3 and *ns* = 2. The main nodes have three or more neighbours and are marked with yellow color. When the response path is displayed, the main nodes are displayed using bold style.

You will also be asked to enter the maximum length of names. When the graph is generated and two temporary files are created (`Temp.nod` and `Temp.lnk`), the server is contacted. If the connection is established, `Cregraph` renames the temporary files to `Graph.nod` and `Graph.lnk`, and the server is able to load the new data.

The maximum number of nodes is limited to 456976. We must generate unique node names, so we use four reserved positions inside the name of node to guarantee the uniqueness. If we would use five positions, the maximum number of nodes would be limited to 11881376.

If you want to generate a graph with maximum number of nodes, you have two possibilities:
– *ns* = 3 and *nn* = 19042, the number of nodes is 456976 and the number of arcs is 913968;
– *ns* = 25 and *nn* = 353, the number of nodes is 456976 and the number of arcs is 915200.

We used similar update programs for `csrvF` and `csrvG`, but there is a subtle difference between the two servers. `csrvF` should be updated using `--wait` message (with some exceptions). `csrvG` may be updated using either `--data` or `--wait` message: it keeps all its data in main memory, so nothing is affected by any reading operation.

You may use graph algorithms to solve, for instance, the problem of determining an optimum journey using the public transport system (buses, trains etc). This kind of application should be developed such that every user could contact it either by desktop computer or mobile device.

How to develop applications for mobile devices ?

From my personal pointy of view, a reasonable solution is to develop separate html templates for desktop and mobile devices. You may simply insert a short sequence in your desktop html template (if the width of screen is less than, say, 720 / *value* pixels, the device is considered to be of mobile type):

```
<script type="text/javascript">
  if (screen.width<value) window.location = '/?Dev=Mbl&...'
  /* javascript code for desktop version */
</script>
```

Your application server should check the value of `Dev` parameter and choose the appropriate html template to generate the result. See `csrvF` (online dictionary) for an html template, which is designed both for desktop and mobile devices.

If you know in advance the size of generated result (displayed information) is reasonable, you may use the same program sequence to generate both the desktop and the mobile variant (see `csrvF` html template). If the page size is too big, you should use different program sequences to generate the desktop and the mobile variant. An example is given below: the list of trains which arrive to / depart from a given station.

The desktop variant may display detailed information about every train. The mobile variant displays only the very important information, otherwise it would be very difficult to consult the whole page. See two examples below.

Station: Bucuresti Nord
Valid from Su,9-Dec-2018 to Sa,14-Dec-2019

No	Train	Services	Arr	Sta	Dep	Train route	Remarks
1	**IR 1003**	Cl1 Cl2	1:24	0:18	1:42	16:41 Vadu Siretului (Ukr) -- 17:49 Dornesti -- 18:12 Darmanesti -- 18:22 Suceava Nord -- 18:28 Suceava -- 19:27 Pascani -- 20:30 Bacau -- 21:18 Adjud -- 23:17 Buzau -- 0:24 Ploiesti Sud -- **1:24 Bucuresti Nord** -- 2:34 Videle -- 3:49 Giurgiu Nord -- 4:50 Ruse (Bul)	Restrictions *SNCF 'CFR Calatori' SA*
2	**IR 1923**	Cl1 Cl2	3:11	0:19	3:30	18:36 Sibiu -- 19:06 Podu Olt -- 19:57 Ciineni -- 20:28 Lotru -- 20:41 Pausa H -- 21:06 Rimnicu Vilcea -- 21:31 Babeni -- 22:40 Piatra Olt -- 23:26 Slatina -- 0:39 Costesti -- 1:05 Pitesti -- 1:41 Golesti -- 2:29 Titu -- **3:11 Bucuresti Nord** -- 5:02 Fetesti -- 5:37 Medgidia -- 6:10 Constanta -- 8:00 Mangalia	Restrictions *SNCF 'CFR Calatori' SA*

260	**IR 1664**	Cl1 Cl2	23:23	-	-	16:40 Iasi -- 16:46 Nicolina -- 17:42 Vaslui -- 18:28 Birlad -- 19:12 Tecuci -- 21:22 Buzau -- 22:31 Ploiesti Sud -- **23:23 Bucuresti Nord**	Restrictions *SNCF 'CFR Calatori' SA*
261	**IR 1774**	Cl1 Cl2	23:32	-	-	19:40 Galati -- 19:59 Barbosi -- 20:17 Braila -- 21:07 Faurei -- 22:24 Urziceni -- **23:32 Bucuresti Nord**	*SNCF 'CFR Calatori' SA*
262	**IR 1821**	Cl1 Cl2	-	-	23:40	**23:40 Bucuresti Nord** -- 0:30 Videle -- 1:16 Rosiori Nord -- 2:12 Caracal -- 2:55 Craiova -- 3:31 Filiasi -- 4:45 Tirgu Jiu -- 5:56 Petrosani -- 7:13 Subcetate - - 7:50 Simeria -- 8:16 Deva -- 8:52 Ilia -- 9:50 Savirsin -- 10:53 Radna -- 11:31 Arad	Restrictions *SNCF 'CFR Calatori' SA*

Home Info		Station: Bucuresti Nord Valid from Su,9-Dec-2018 to Sa,14-Dec-2019	
No	Train	Arr / Dep	Train route
1	**IR 1003** Cl1 Cl2	1:24 1:42	16:41 Vadu Siretului (Ukr) -- **1:24 Bucuresti Nord** -- 4:50 Ruse (Bul) Restrict *SNCF 'CFR Calatori' SA*
2	**IR 1923** Cl1 Cl2	3:11 3:30	18:36 Sibiu -- **3:11 Bucuresti Nord** -- 8:00 Mangalia Restrict *SNCF 'CFR Calatori' SA*
260	**IR 1664** Cl1 Cl2	23:23 -	16:40 Iasi -- **23:23 Bucuresti Nord** Restrict *SNCF 'CFR Calatori' SA*
261	**IR 1774** Cl1 Cl2	23:32 -	19:40 Galati -- **23:32 Bucuresti Nord** *SNCF 'CFR Calatori' SA*
262	**IR 1821** Cl1 Cl2	- 23:40	**23:40 Bucuresti Nord** -- 11:31 Arad Restrict *SNCF 'CFR Calatori' SA*

Important remark about memory management (`calloc`, `malloc`, `free`, `strdup` and similar)

If memory management is performed inside `processRequest`, this process must be protected by mutexes (**`CAS_Srvinfo.mm`**), otherwise the server may crash when simultaneous requests are processed. The memory management functions use some global (but not public) information about the heap, which must be protected against possible corruption.

Directory `csrvH`: Miscellaneous short problems
Logging requests, timeout management

Source file `csrvH.c`

```
#include "cAppserver.h"

static struct {
       unsigned char Ipn[16];
       int fLog, lAdr;
       } Config;

static void recordRequest(CAS_srvconn_t *Conn) {
char *Bf,*Pb,*P,*Q;
int l;
Bf = Conn->Pct;
inet_ntop(CAS_Srvinfo.af,Conn->Ipc,Bf,INET6_ADDRSTRLEN);
Pb = CAS_endOfString(Bf,0);
*Pb++ = ' ';
for (P=Conn->Bfi+5; *P; P++)
    if (isspace(*P)) break;
l = P - Conn->Bfi;
memcpy(Pb,Conn->Bfi,l);
Pb += l;
*Pb++ = '\n';
if (P=strcasestr(P,"User-agent:")) {
   Q = strpbrk(P,"\r\n");
   if (Q==NULL) Q = CAS_endOfString(P,0);
   l = Q - P;
   memcpy(Pb,P,l);
   Pb += l;
   }
*Pb++ = '\n';
*Pb++ = '\n';                     /* The size of buffer for temporary strings (Conn->Bft) must */
l = write(Config.fLog,Bf,Pb-Bf);  /* be at least the size of buffer for input strings (Conn->Bfi) */
}

static int acceptConnection(unsigned char *Ipc) {
return memcmp(Config.Ipn,Ipc,Config.lAdr) != 0;
}

static void userConfig(char *Cfg) {
char *P;
Config.lAdr = CAS_Srvinfo.af == AF_INET ? 4: 16;
for (P=Cfg; *P; P++)
    if (isspace(*P)) break;
*P++ = 0;
inet_pton(CAS_Srvinfo.af,Cfg,Config.Ipn);
while (isspace(*P)) P++;
Cfg = P;   /* other user specific configuration data */
}

static char *concat(CAS_srvconn_t *Conn, ...) {
va_list Prms;
char *Pch,*Out;
int l;
va_start(Prms,Conn);
Out = Conn->Pct;
```

```
do {
   Pch = va_arg(Prms,char *);
   if (Pch==NULL) break;
   l = strlen(Pch);
   if (l==0) continue;
   if (Out+l>=Conn->Pet) return NULL;
   strcpy(Out,Pch);
   Out += l;
   } while (1);
Pch = Conn->Pct;
Conn->Pct = Out + 1;
return Pch;
}

static char *stringToUpper(char *Str) {
char *P,c;
P = Str;
while (c=*P) {
      if (islower(c)) *P = toupper(c);
      P++;
      }
return Str;
}

static void manageTimeout(CAS_srvconn_t *Conn) {
double s,w;
long long p,n;
n = atoll(CAS_getLastParamValue(Conn,"N"));
if (n==0) n = 9000000000;
CAS_nPrintf(Conn,"<br>Heavy program sequence begins, n = %D<br>",n);
for (s=0,w=0.25,p=1; p<=n; p++,w=-w) {
    s += w * p;
    if (p%10000==0) if (Conn->tmo) break;
    }
CAS_nPrintf(Conn,"%s s = %.2f, p = %D<br>",Conn->tmo?"Partial":"Completed",s,p);
Conn->tmo = 0;
CAS_nPrintf(Conn,"Elapsed time: %.3f",CAS_getTime(Conn));
}

static void processRequest(CAS_srvconn_t *Conn) {
char *A,*B,*C,*S;
A = "string A";
B = "string B";
C = "string C";
S = " - ";
CAS_nPrintf(Conn,"First concatenation: %s<br>",concat(Conn,A,S,B,S,C,NULL));
CAS_nPrintf(Conn,"Second concatenation: %s<br>",concat(Conn,B,S,C,A,NULL));
CAS_nPrintf(Conn,"%s<br>",stringToUpper(CAS_sPrintf(Conn,"%x",0xabcdef)));
manageTimeout(Conn);
}

void CAS_registerUserSettings(void) {
Config.fLog = open("csrvH.log",O_APPEND|O_CREAT|O_WRONLY|O_DSYNC,0600);
CAS_Srvinfo.rwrl = recordRequest;
CAS_Srvinfo.preq = processRequest;
CAS_Srvinfo.cnfg = userConfig;
CAS_Srvinfo.acco = acceptConnection;
}
```

Compile and run, then access the server:
`http://`*AddressOfYourLinuxMachine*`:5008`

You will see the following page:

First concatenation: string A - string B - string C
Second concatenation: string B - string Cstring A

What does it happen behind the scene ?

A particular file (`csrvH.log`) is used to record every GET request. The operation is performed by `recordRequest` function, which is registered as rewrite rules function, although it will not alter the request. `recordRequest` records only the GET line and the User-agent information (if present).

The `userConfig` function takes an Ip address found in the user's configuration section of the configuration file. The server will reject every request coming from this Ip address; see the `acceptConncetion` function, which is registered for this purpose.

You are invited to modify this procedure to manage both the fourth and the sixth Ip versions.

This example shows how to reject any request coming from a (single) given Ip address. If you need to manage a variable number of Ip addresses (either for accept or reject), you should include this list in a text file. Use `CAS_Srvinfo.data` to allocate memory / to free the allocated memory.

The `concat` function shows how to develop a concatenation function which uses the buffer for temporary strings (`Conn->Bft`). Following this example you may create other functions which use the `Conn->Bft` buffer. The advantage of using this buffer is that you won't bother allocating (`malloc` or `calloc`) or de-allocating memory (`free`).

`Pch` points to the first character of the resulting string, and is the actual value of `Conn->Pct` (pointer to current temporary). `Out` pointer advances in this buffer and finally will points to the new current temporary. You must be sure its value does not exceed the end of buffer (`Conn->Pet`, pointer to end of temporary buffer strings). In fact, you must keep reserved one octet for the end of string (the null terminator).

You may use the end of `Conn->Bft` to reserve space for some strings (or other data structures, see *srvI* example for details). This way you save stack space and avoid memory allocation / de-allocation. See `csrvF` and `csrvG` for details.

If you need this space reservation for long time (until the request is completely processed), you may reduce the size of `Conn->Bft` by altering the value of `Conn->Pet` (pointer to the end of temporary buffer strings):
`Conn->Pet -=` *value*`;`

Depending on the type of data structure you want to reserve, *value* should be properly adjusted, rounded up to the nearest multiple of `sizeof(int)` or something similar.

See `csrvF` and `csrvI` sections for examples about how to reduce the length of this buffer.

The `stringToUpper` converts every letter of the input string to its corresponding uppercase. The output string is allocated in the buffer for temporary strings.

The `manageTimeout` function simulates a crunching numbers activity. From time to time, the `Conn->tmo` indicator is checked; when its value is not zero, the activity is stopped and `Partial` message is sent to the browser. If the value of `N` parameter is reasonable low, the activity finishes without being stopped and `Complete` message is sent to the browser.

`Conn->Bft` is also used by `recordRequest` to build the string to be recorded.

Beware: the size of buffer for temporary strings must be at least the size of buffer for input strings (`Conn->Bfi`).

If you simply contact the server as above, the default value of `N` parameter is enormous. Now try the following URL to see the difference:
`http://`*AddressOfYourLinuxMachine*`:5008/?N=100000`

Directory `csrvI`: How to use Mysql databases

Before running this example, you must have an account on a Mysql server. Create a table `Dictio` using the following template (see also the `insert.sql` script):

Primary key		***def varchar***
wrd varchar	***wno int***	
program	*1*	*[Late Latin programma, from Greek]: a public notice*
program	*2*	*a plan or system under which action may be taken toward a goal*
system	*1*	*a regularly interacting or interdependent group of items forming a unified whole*
system	*2*	*an organized set of doctrines, ideas, or principles usually intended to explain the arrangement or working of a systematic whole*
system	*3*	*harmonious arrangement or pattern* **;** *order*

Edit the configuration file (*User specific configuration*) including your actual connection information.

Compile the source file, run the program and connect the browser to the server:
`http://`*AddressOfYourLinuxMachine*`:5009`

Source file `csrvI.c`

```
#include "cAppserver.h"
#include <my_global.h>
#include <mysql.h>

typedef struct { char *W; MYSQL *M; } T_usrinf;

static struct { char *H, *D, *U, *P; } MySqlIni;

static char *Fhtm;

static char *myGetString(char *Inp, char **Out) {
char *P,*Q;
P = Inp;
while (isspace(*P)) P++;
Q = P;
while (isspace(*Q)==0)
      if (*Q) Q++; else break;
*Q++ = 0;
*Out = P;
return Q;
}

static void userConfig(char *Cfg) {
Cfg = myGetString(Cfg,&MySqlIni.H);
Cfg = myGetString(Cfg,&MySqlIni.D);
Cfg = myGetString(Cfg,&MySqlIni.U);
myGetString(Cfg,&MySqlIni.P);
}

static void errorFromMysql(CAS_srvconn_t *Conn, char *Msg, char *Sql) {
T_usrinf *Prms;
Prms = Conn->Usr;
if (Msg) CAS_nPrintf(Conn,"Mysql %s failed: error %u (%s)\n%s\n",Msg,
        (unsigned)mysql_errno(Prms->M),mysql_error(Prms->M),Sql);
   else CAS_nPrintf(Conn,"Mysql init failed (probably out of memory)\n");
}

static void manageUserHtml(char op) {
if (op=='L') Fhtm = CAS_loadTextFile("Dictio.htm");
```

```
   else free(Fhtm);
}
static MYSQL_RES *mysqlQueryResult(CAS_srvconn_t *Conn, char *Sql) {
T_usrinf *Prms;
MYSQL_RES *Res;
Prms = Conn->Usr;
if (mysql_query(Prms->M,Sql)==0) {
   Res = mysql_store_result(Prms->M);
   if (Res==NULL) errorFromMysql(Conn,"store","");
   }
else {
   Res = NULL;
   errorFromMysql(Conn,"query",Sql);
   }
return Res;
}
static void displayHelp(CAS_srvconn_t *Conn) {
static char *sqlF = "select distinct wrd from Dictio where wrd like '%s%%' order
                     by 1 limit 10";
char *sqlQ;
T_usrinf *Prms;
MYSQL_RES *Res;
MYSQL_ROW Row;
int l;
Prms = Conn->Usr;
if (*Prms->W) do {
   l = strlen(sqlF) + strlen(Prms->W) + 1;
   sqlQ = Conn->Pet - l;
   sprintf(sqlQ,sqlF,Prms->W);
   Res = mysqlQueryResult(Conn,sqlQ);
   if (Res==NULL) break;
   while (Row=mysql_fetch_row(Res))
         CAS_nPrintf(Conn,"%s\n",Row[0]);
   mysql_free_result(Res);
   } while (0);
}
static void displayDefs(CAS_srvconn_t *Conn) {
static char *sqlF = "select def from Dictio where wrd='%s'";
char *Fmt,*sqlQ;
T_usrinf *Prms;
MYSQL_RES *Res;
MYSQL_ROW Row;
int l;
Prms = Conn->Usr;
CAS_nPrintf(Conn,Fhtm,Prms->W);
Fmt = CAS_endOfString(Fhtm,1);
if (*Prms->W) do {
   l = strlen(sqlF) + strlen(Prms->W) + 1;
   sqlQ = Conn->Pet - l;
   sprintf(sqlQ,sqlF,Prms->W);
   Res = mysqlQueryResult(Conn,sqlQ);
   if (Res==NULL) break;
   while (Row=mysql_fetch_row(Res))
         CAS_nPrintf(Conn,Fmt,Prms->W,Row[0]);
```

```
   mysql_free_result(Res);
   } while (0);
Fmt = CAS_endOfString(Fmt,1);
CAS_nPrintf(Conn,Fmt,CAS_getTime(Conn));
}

static void strCopy(char *Ds, char *So, int ls) {
char c;
int k;
k = 0;
while (c=So[k]) if (!isspace(c) || (k>0)) {
      *Ds = isalpha(c) ? tolower(c): c;
      Ds++;
      if (++k==ls) break;
      }
*Ds = 0;
}

static void processRequest(CAS_srvconn_t *Conn) {
T_usrinf Prms;
char *P;
int l;
Conn->Usr = &Prms;
Prms.M = mysql_init(NULL);
if (Prms.M == NULL) {
   errorFromMysql(Conn,NULL,NULL);
   return;
   }
if (mysql_real_connect(Prms.M,MySqlIni.H,MySqlIni.U,MySqlIni.P,MySqlIni.D,0,
    NULL,0)==NULL) {
   errorFromMysql(Conn,"connect","");
   return;
   }
P = CAS_getLastParamValue(Conn,"W");
l = strlen(P);
if (l>255) l = 255;
Prms.W = Conn->Pet - l - 2;
strCopy(Prms.W,P,l);
l += sizeof(int);
Conn->Pet -= l - l % sizeof(int);
P = CAS_getLastParamValue(Conn,"O");
if (*P=='H') displayHelp(Conn);
   else displayDefs(Conn);
mysql_close(Prms.M);
}

void CAS_registerUserSettings(void) {
CAS_Srvinfo.cnfg = userConfig;
CAS_Srvinfo.preq = processRequest;
CAS_Srvinfo.html = manageUserHtml;
}
```

First of all, you must include the two Mysql header files. The above module uses a global, static structure (`MySqlIni`) which holds: `H` (Mysql server host), `D` (database name), `U` (user name), `P` (password). If necessary, you may extend this structure to include supplementary information (port number, socket name etc). See *The C Mysql API* for details. The `userConfig` function gets the above information from the configuration file (*User specific configuration*).

The processRequest function uses Prms (user defined T_usrinf structure) which includes W (pointer to a null terminated string, the word to be processed) and M (pointer to a MYSQL structure). Conn->Usr points to Prms, so it can be used by any subsequent function without passing supplementary arguments. Next, mysql_init is called to obtain a connection handler, and mysql_real_connect to connect to the server.

Now suppose the browser is connected to this server. When you enter a sequence of letters, a request is sent to the server to display a list of matching words; the request is solved by displayHelp. When you hit the *Enter* key, a request is sent to the server to display a list of definitions for the entered word; the request is solved by displayDefs.

The GET parameter W (the entered word) is copied at the end of buffer for temporary strings (see the figure below).

Buffer for temporary strings (Conn->Bft) . . . *Pointer to end of temporary strings* (Conn->Pet)

The length of the word is limited to 255 characters and only letters are copied, to avoid any injection attempt (strCopy function). After copying, the length of this buffer is reduced, so the parameter can be safely used by other subsequent functions without the risk of alteration.

I copied the GET parameter to the end of this buffer to avoid as much as possible memory allocation and de-allocation operations.

displayDefs and displayHelp follow the same logic. First, Conn->Usr is copied to Prms (pointer to T_usrinf structure), so the functions will be able to use both the GET parameter and the Mysql connection handler. Other preparations may be performed if necessary.

If the entered word (Prms->W) is not the empty string, some Mysql specific operations are performed. In these cases, the sql request is first prepared; the corresponding string is generated at the end of buffer for temporary strings.

The mysqlQueryResult performs the query given by the Sql string and, if the query is successfully finished, the result set is retrieved and returned to the calling function. On error, a message is sent to the browser and NULL is returned.

The above program is designed under the following assumption: the text definition does not include any special character (**<** – less than, **>** – greater than etc). If you want to manage more complex definitions, you should use the following statement:

```
CAS_nPrintf(Conn,Fmt,Prms->W,CAS_convertString(Row[0],'H'));
```

CAS_convertString with H (html conversion) as the second argument returns a new string where special characters are replaced by the corresponding escape sequences (**<** by `<`, **>** by `>` and so on).

The result set is processed row by row and the used memory is finally de-allocated. When processRequest finishes its work, the connection to the Mysql server is closed (mysql_close).

Remark. You may adapt this program to be used in conjunction with PostgreSQL or other free database management system. See the specific documentation for details.

Using Conn->Usr as alternative for modifiable GET parameters

A php script may easily use the following statement: $_GET['W'] = *new value*;

Our C program must use the following template:

Prms.W = CAS_getLastParamValue(Conn,"W"); /* *first initialisation* */

Prms.W = *new string value*;

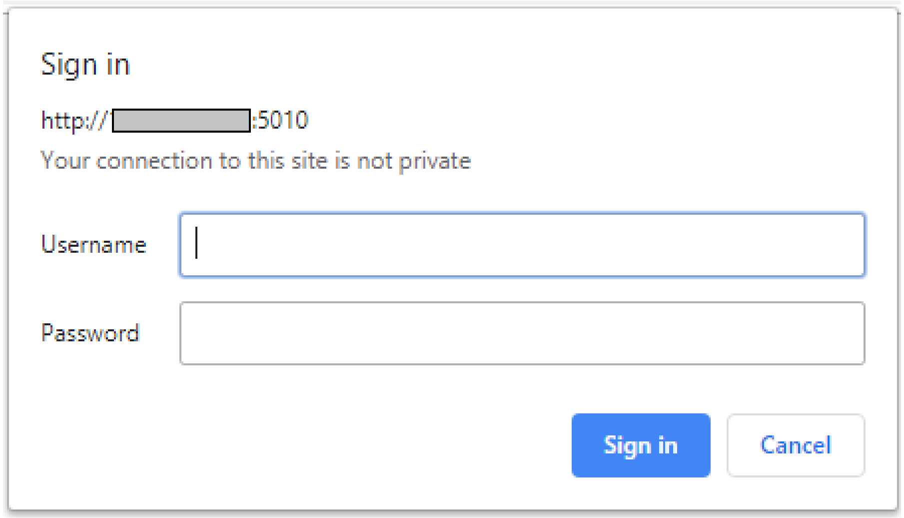
The C program must use Prms.W or Prms->W, depending on the definition of Prms.

Directory `csrvJ`: Basic authentication

This application server shows how to manage the Basic authentication paradigm. If you enter for the first time the following URL

`http://`*AddressOfYourLinuxMachine*`:5010`

the browser displays the following page (or something similar):



Source `csrvJ.c`

```
#include "cAppserver.h"

static struct { char *Usr, *Pwd; } Auth;

static struct { signed char D[256], E[65]; } Base64;

static void userConfig(char *Cfg) {
char *P;
for (P=Cfg; isspace(*P)==0; P++) ;
Auth.Usr = P;
while (isspace(*P)==0) P++;
*P++ = 0;
for (P=Cfg; isspace(*P); P++) ;
Auth.Pwd = P;
while (isspace(*P)==0)
      if (*P) P++; else break;
*P++ = 0;
Cfg = P;
while (isspace(*Cfg)) Cfg++;
/* Cfg points to the next information on this section */
}

static char *encodeBase64(CAS_srvconn_t *Conn, Usrsigned char *Bin, int lbi) {
int k,l,b,c;
char *Out;
l = (lbi * 4 + 2) / 3;
if (c=l%4) l += 4 - c;
Out = Conn->Pct;
if (Out+l-1>=Conn->Pet) return NULL;
memset(Out,'=',l);
Out[l] = k = l = 0;
while (k<lbi) {
      Out[l++] = Base64.E[Bin[k]>>2];
```

```
      b = Bin[k++] % 4 << 4;
      c = k < lbi ? Bin[k] >> 4: 0;
      Out[l++] = Base64.E[b+c];
      if (k>=lbi) break;
      b = Bin[k++] % 16 << 2;
      c = k < lbi ? Bin[k] >> 6: 0;
      Out[l++] = Base64.E[b+c];
      if (k>=lbi) break;
      Out[l++] = Base64.E[Bin[k++]%64];
      }
Conn->Pct = Out + l;
return Out;
}

static char *decodeBase64(CAS_srvconn_t *Conn, char *Str) {
char *Out,*Dst,C[4];
unsigned char d;
int j;
if (Str=strchr(Str,' ')) Str++;
   else return NULL;
for (Dst=Str; d=(unsigned char)*Dst; Dst++)
    if (Base64.D[d]<0) break;
Out = Conn->Pct;
if (Out+(Dst-Str)>=Conn->Pet) return NULL;
memset(Out,0,Conn->Pet-Out);
Dst = Out;
while (Str) {
      C[0] = C[1] = C[2] = C[3] = 0;
      for (j=0; j<4; j++) {
          C[j] = d = *Str++;
          if (Base64.D[d]<0) {
             C[j] = 0;
             break;
             }
          }
      if (C[0]==0) break;
      d = Base64.D[C[0]];
      *Dst = d << 2;
      if (C[1]==0) break;
      d = Base64.D[C[1]];
      *Dst++ |= (d & 0x30) >> 4;
      *Dst = d << 4;
      if (C[2]==0) break;
      d = Base64.D[C[2]];
      *Dst++ |= d >> 2;
      *Dst = (d & 3) << 6;
      if (C[3]==0) break;
      d = Base64.D[C[3]];
      *Dst++ |= d;
      }
Conn->Pct = CAS_endOfString(Out,1);
return Out;
}

static void sendUnauthorized(CAS_srvconn_t *Conn) {
CAS_resetOutputBuffer(Conn);
CAS_nPrintf(Conn,CAS_Srvinfo.Rh[3]);
```

```
CAS_nPrintf(Conn,"You must enter User name and Password");
}

static void processRequest(CAS_srvconn_t *Conn) {
char *Hau,*Pwd,*Str;
Hau = CAS_getHeaderValue(Conn,"Authorization");
if (Hau==NULL) {
   sendUnauthorized(Conn);
   return;
   }
CAS_nPrintf(Conn,"Header value: %s<br>",Hau);
Str = decodeBase64(Conn,Hau);
if (Str==NULL) {
   CAS_nPrintf(Conn,"Header malformed or not enough temporary buffer space");
   return;
   }
if (Pwd=strchr(Str,':')) {
   *Pwd++ = 0;
   if (!strcmp(Auth.Usr,Str) && !strcmp(Auth.Pwd,Pwd))
      CAS_nPrintf(Conn,"Ok, user name and password match");
   else
      CAS_nPrintf(Conn,"User name or password don't match");
   }
else
   CAS_nPrintf(Conn,"Can't obtain user name and password from header value");
}

void CAS_registerUserSettings(void) {
char *P;
int k;
CAS_Srvinfo.preq = processRequest;
CAS_Srvinfo.cnfg = userConfig;
memset(Base64.D,-1,sizeof(Base64.D));
for (k=0,P=Base64.E; k<64; k++,P++) {
    if (k<26) *P = k + 'A';
    else
    if (k<52) *P = k + 'a' - 26;
    else
    if (k<62) *P = k + '0' - 52;
    else *P = k == 62 ? '+': '/';
    Base64.D[*P] = k;
    }
}
```

401 Unauthorized header

```
HTTP/1.1 401 Unauthorized
Connection: close
WWW-authenticate: Basic realm="Realm"
Content-type: text/html; charset=UTF-8
Connection: close
```

`CAS_registerUserSettings`

After registering the `processRequest` function, we set the two member of `Base64` structure with the appropriate values, such that we will be able to decode / encode strings according to Base64 rules.

processRequest

If we enter for the first time the URL of this application, the *Authorization* header is missing, so the server must send a particular response to inform the browser about the fact that this URL is protected. We must use the 401 Unauthorized response header, which is identified by `CAS_Srvinfo.Rh[3]`, so the output buffer must be reset and the new header is sent to the client / browser.

The default header can now be replaced by another one, which is user defined. We already used this procedure, see `csrvD`. After sending the response header, the server must also send a message to be displayed by the browser if the user clicks the *Cancel* button.

If the browser sends the *Authorization* header, we suppose its value matches the following template:
`Basic` *base64-string*

`decodeBase64` decodes the base64 string and allocates the output on `Conn->Bft` (buffer for temporary strings). The output string must match the following template: *username*`:`*password*

decodeBase64

This function is designed to be used in the Authorization context. This is why it checks for the presence of ' ' character before decoding. If this character is not present in the value of the *Authorization* header, maybe a malicious user sent an incorrect header, and NULL is returned. This function returns also NULL if there is not enough space to decode the input string. In the second case, you should increase the size of `Conn->Bft` (buffer for temporary strings).

encodeBase64

I also provided the `encodeBase64` function, which can be used, for instance, to include some inline images in the output html response. The output string is reserved from `Conn->Bft`:

```
Out = Conn->Pct;
if (Out+l-1>=Conn->Pet) return NULL;
```

If you want to allocate memory from heap and not from buffer for temporary strings, then use:

```
Out = malloc(l+1);
if (Out==NULL) return NULL;
```

In this last case, do not forget to release the allocated memory before `processRequest` finishes its process.

Appendix A: Why C language and Linux operating system ?

Advantages of Linux C programming (beware, this is my humble opinion, I am computer programmer)

– generally speaking, network programming on Linux is very easy
– this framework is developed to run under Linux, the release version uses the `clone` system call to implement multithreading; using this function gives us a lot of advantages that cannot be achieved otherwise

Using clones allows you to specify a lot of details about the new process: how to control the file descriptors, the memory space, ***where the stack is located***, ***the stack size*** etc (in fact the main module specifies all these details).

The caller (in our case the main module) allocates space for the stack before calling `clone`. The stack is initially set to zero. After starting the new process, the stack may be examined by the main module to determine how much amount is used: you simply type the following command (*csrv* is the name of your server, release version):
`./`*csrv* `--show`

Both the debug and the release versions display how much space is used from `Bfi` (buffer for input strings) and `Bft` (buffer for temporary strings). Only the release version displays the maximum amount of stack usage. This information gives you an idea about how to choose these values (see Appendix B, configuration file) such that the server will run safe and reduce the memory consumption.

The server allocates memory for buffers (input, output and temporary) at the start point, immediately after reading the configuration file. When the administrator issues a `--cnfg` request (local command), the server frees the previously allocated memory and allocates new memory space, according to the new settings (which may or may not be changed). The release version allocates memory for the stacks of its clones at the start point. This memory will remain allocated during the whole life of the server

I don't know other simpler solution to estimate how much memory is needed for the stack. If you succeed to determine a reasonable value for the stack size and other parameters, you may configure the server to manage more simultaneous requests with reasonable memory consumption.

Another advantage is that, if a clone crashes due to programming errors (such as segmentation fault), other clones are not affected (excepting some very bad cases: the problematic clone corrupts global variables or memory used by another clone). Using POSIX threads doesn't give you this possibility. You may object that crashing will compromise running other clones. This is not necessarily true (but see exceptions above). The developer must follow some few very simple rules. `processRequest`, `rewriteRules` and all of their subsequent functions should use local variables for read and write operations. All these subsequent functions must be thread safe. The global variables must be used only for read operations, or must be protected by mutexes.

Last but not least, every clone must use its own alarm timer, which is a process resource.

Advantages of C based application against script (language) based application

If you want to implement an application server to solve very complex problems, *which involve high volume of computation*, the C based application is much faster than classic one (which may be built using a script language, for instance php). After starting the C based server, all the important data may be read into main memory (see examples `csrvF` and `csrvG`), so the server will not read them from disk when a new connection is established.

You may object that classic applications are also fast: the operating system caches the requested files and serves them from main memory. Classic applications are not able to manage some global information, as ours do (see examples `csrvF` and `csrvG`). The C based application server, which is already compiled, may take the content directly from main memory.

Another advantage is revealed when you have to debug a complex application. You simply run your server (debug version) under `gdb`, step by step, examine its variables etc. How easy / difficult is to debug php scripts ?

You are invited to compare the performances of `csrvG` application (minimum cost path in a graph) against your favorite language / technology. What about running time and memory consumption ?

Now suppose two or more simultaneous requests must be processed by the server. Two requests are supposed to be simultaneous if the second request arrives while the first one is not yet finished. The C based application uses only one copy of the global data (`csrvG` case: the list of nodes and the whole graph, previously loaded by `CAS_Srvinfo.data` call). Classic applications use separate copies of the data to be processed, or you must use very complex programming techniques to avoid memory duplicates.

It would be interesting to benchmark implementations of `csrvF` or `csrvG` problems using PHP accelerators; a list of them can be found at

https://en.wikipedia.org/wiki/List_of_PHP_accelerators

***"Software is getting slower more rapidly than hardware becomes faster"* (Niklaus Wirth, *A plea for lean software*).**

This is a very good reason to develop C based application servers, which are not bloated (depending on the skills of programmer).

Disadvantages of C based application (yes, nobody is perfect)

C programs are verbose: you must define every variable you use, perform explicit conversions from ASCII to binary or vice versa etc. You must manage very carefully every pointer used for memory allocation. Do not forget to release the allocated memory, otherwise the server may run more and more slowly due to memory leaks, or some clones may simply crash.

Allocation and release operations, *if peformed inside* `processRequest`, must be protected by `CAS_Srvinfo.mm`.

If you develop a script application, you don't need to worry about any of the above problems. You may also use anywhere the `exit` function (or similar). Your C modules must not use this function, otherwise the final part of (or the whole) response page will not be sent to the browser if `processRequest` does not return properly.

Tips for C developers (`processRequest` and, if needed, `rewriteRules`)

First of all, every modifiable variable should be local, not global. Global variable should only be read. If you must modify a global variable, protect it using mutexes (see `csrvF` for details). If you want, for instance, to count how much times a particular page is accessed, you must use a global variable for this purpose.

Some very difficult cases may arise when a pointer variable points to a particular memory block, which is allocated by the main process of the server, or by another clone. If this memory is altered, the main process or the altered clone may crash. If the other clone crashes, it is very difficult to identify the cause of error. This case may happen if two clones simultaneously process two distinct requests. If you simply retry later the 'problematic' request (recorded on the error log file), it may be processed without errors.

Conclusion: when you are using pointers, any pointer variable must point only inside the allocated memory, which is associated to that pointer.

How to develop C++ applications ?

If you desperately want to develop C++ applications, just follow these steps:

– compile the main module to obtain binary object files:

```
gcc -O3 -D _Release_application_server -c cAppserver.c -o cAppsr.o
gcc -g -c cAppserver.c -o cAppsd.o
```

– edit `cAppserver.h` adding the `extern "C"` attribute:

```
extern "C" double CAS_getTime(CAS_srvconn_t *Conn);
```

Now you are ready to develop your C++ application, using `cAppsr.o` to obtain release version, or `cAppsd.o` to obtain debug version.

Appendix B: The configuration file

Response headers

```
HTTP/1.1 200 Ok
Content-type: text/html
Expires: 0
Connection: close
```

```
HTTP/1.1 404 Not found
Expires: 0
Connection: close
```

```
HTTP/1.1 301 Moved
Location: https://%s
Connection: close
```

The first three response headers must always appear at the beginning of the configuration file. You may add supplementary headers, depending on your application. You may personalize any header (including the above mandatory) if you wish; you may add the following line (or anything else):
`Server:` *My application server, version n.n*

The main module of the server uses the second response header (404 Not found) in case of missing the rewrite rules procedure (defined by the programmer), if the request doesn't match any format below:
`GET / ...` (space after slash)
`GET /?...` (question mark after slash)

If the browser asks for `favicon.ico` or any other file, the 404 Not found response header is received. You may use your own rewrite rules procedure to change this behavior (see `csrvC` for details).

Every header must be followed by exactly two empty lines. I don't use the content-length header in case of html content (first response header) because I don't know in advance the output size. To determine the content length, I should write first the output to a temporary file, and finally send the header and the content (or, I should use chunked transfer encoding). For the sake of simplicity and to gain speed, I preferred to send directly to the client a very simple header followed by the content. Another breach of the http protocol is that every line is terminated by LF (Linux style) not by CR LF.

Usually, `processRequest` is called to generate html content depending on the application logic and the GET parameters. This is why the output buffer (`Conn->Bfo`) is prefilled with 200 Ok response header. Every `CAS_nPrintf` call appends new strings to this buffer. When this buffer is full, or `processRequest` ends, the main module sends the content to the client.

We saw two examples using supplementary headers:
– `csrvD` (generating xml pages);
– `csrvJ` (basic authentication).

If a particular request requires another content type (instead of html), you should use the following sequence before the first `CAS_nPrintf` call:

```
CAS_resetOutputBuffer(Conn);
CAS_nPrintf(Conn,CAS_Srvinfo.Rh[n],...);
```
/* $n \geq 2$ is the index of your special header */

Server configuration

`Abcdefgh-ijklmnopq-rstuvwxy`	first line
`4 0.0.0.0 127.0.0.1 5000 0`	second line
`2 3 4000000`	third line
`4000 60000 1000000`	fourth line
`15 0`	fifth line

The first line of this section stores the server password, which is used for local commands (messages). You may choose any sequence which doesn't include white spaces; its length should not exceed 2000 characters. When you send a message to the server (`--cnfg`, `--data`, `--html`, `--stop`), the client sends also this password for security reasons: any other user having access to the Linux machine is able to send such messages, so the password is necessary.
The second line (red marked) stores the following information:
– the internet protocol version (4 or 6)
– bind address (`0.0.0.0` means any IPV4 address of your machine); if your machine uses two or more Ip addresses, you may choose as bind address any available value
– local host address (`127.0.0.1`)
– listening port (5000); if you have supervisor rights on your Linux system, you may use any number as listening port, including 80 (http) if this port is free
– the last value is not used (listening port for secure connections)

If the IPV6 features are enabled on your Linux system (use `ifconfig` or `lsmod`), you may use the following values (or something similar):

```
6 :: ::1 5000 0
```

The third line stores the following values:
– first value: maximum simultaneous clones (in this example: 2 clones)
– second value: request processing timeout (in this example: 3 seconds)
– third value: stack size (in this example: 4000000, automatically rounded up to the nearest multiple of 1024)

The fourth line stores the following values (the server will round every value up to the nearest multiple of 1024):
– first value: size of input buffer (`Conn->Bfi`, in this example: 4000)
– second value: size of output buffer (`Conn->Bfo`, in this example: 60000)
– third value: size of temporary strings buffer (`Conn->Bft`, in this example: 1000000)

If we generate the release version of our server, we may display the used amount of stack, buffer for input strings (`Bfi`) and buffer for temporary strings (`Bft`):
`./`*csrv* `-show`

Of course, we must first force the server to cover (if it is possible) every function of our module. We must imagine as many difficult cases as possible when we test the behavior of the server, typing some URLs and browsing.

If the input buffer is too small, you may get *Input buffer overflow* error. The message tells you how much space is needed. You should increase the size of the input buffer.

The fifth line stores the following values:
– first value: receive timeout (in this example: 15 seconds)
– second value: minimum number of octets to be received by POST / LOAD method (if zero, these methods are not allowed); see Appendix G for details

Receive timeout (seconds)

The number of seconds the client is allowed to stay connected; if this limit is reached, the server closes the connection and declares receive timeout error. If the client doesn't send the entire request, the server doesn't wait for the empty line and parses only the received message. This is another breach of the http protocol, as a precautionary measure to reduce the impact of some possible slowloris attacks. In case of receiving timeout error, the release version registers the Ip address of client in the error log file, so you may later examine the error list and take an appropriate decision (for instance, deny any further access from these Ip addresses).

Request processing timeout (seconds; see `csrvH` for details)

The running (real) time is measured from the point of `processRequest` call. If the process doesn't finish the work before reaching the time limit, a `SIGALRM` signal is delivered and `Conn->tmo` is set to one. If `processRequest` (or any subsequent function) doesn't check the above variable and doesn't set it to zero, the release version will terminate the process (next time, when `SIGALRM` will be delivered again) and the corresponding message is sent to the browser. The release version also registers the event in the error log file, so you may later examine it and improve the implementation.

Beware! If you don't manage correctly the timeout problem, the server (release version) may block indefinitely. Let us suppose the first clone processes its assigned request and loops indefinitely. The second clone processes the next request, and so on. When every clone is busy, and a new request is coming, there is no clone to accept this request, so the server will hang.

The error log file has the same name as the server, is located in the same directory, and the extension is `.err` .

Do not change password, internet protocol version, local host and listening port, otherwise local commands will not work. If you send `--cnfg` message, the server will update the headers, the size of buffers and the timeout values. If you want to modify the number of clones or the stack size, you must stop and restart the server.

Appendix C: Data types and public variables

`CAS_srvconn_t` data structure

```
typedef struct {                 // do not alter members which are red marked below
        unsigned char Ipc[16];   // Ip address of client
        time_t uts;              // unix time stamp (seconds elapsed from Epoch)
        char *Bfi;               // buffer for input strings
        char *Bfo;               // buffer for output response (sent to the client)
        char *Bft;               // buffer for temporary strings
        char *Pct;               // pointer in Bft zone (current position)
        char *Pet;               // pointer to the end of Bft zone
        volatile char tmo;       // process timeout status (set to 1 when process time limit exceeds)
        void *Usr;               // pointer to the user's local zone
        } CAS_srvconn_t;
```

`CAS_srvconn_t`is part of another, bigger structure which includes more information about the connection between server and client, the most important one being the socket returned by `accept` system call.

`CAS_srvinfo_t` data structure

```
typedef struct {        // do not alter members which are red marked below
        int af;         // address family, set to AF_INET or AF_INET6
        char **Rh;      // response headers
        int32_t mm;     // mutex, used for memory management (see csrvG)
                        // mm is automatically set to 1 when the server is started
        char *Nv;       // defines the behavior of CAS_getLastParamValue
        void (*data)(char op);              // manage user data (load and release)
        void (*html)(char op);              // manage user html (load and release)
        void (*cnfg)(char *);               // user specific configuration
        void (*preq)(CAS_srvconn_t*);       // user process request
        void (*rwrl)(CAS_srvconn_t*);       // user rewrite rules
        int (*acco)(unsigned char *);       // check if accept connection
        } CAS_srvinfo_t;
CAS_srvinfo_t CAS_Srvinfo;                  // defined by the main module, cAppserver.c
```

The response headers (taken from the configuration file) are identified as follows:

– `CAS_Srvinfo.Rh[0]`: 200 Ok
– `CAS_Srvinfo.Rh[1]`: 404 Not found; `strlen(CAS_Srvinfo.Rh[1])` <= `250`
– `CAS_Srvinfo.Rh[2]`: 301 Moved; redirect to https, reserved for secure version
– `CAS_Srvinfo.Rh[3]` and the rest (if present): defined by developer

`CAS_Srvinfo.acco`

If defined, this function is called immediately after establishing a new connection, and before receiving any data from client. This function is called directly by the main process, so global variables may be used for read and write operations.

Beware! These global variables must only be read by `CAS_Srvinfo.preq`, `CAS_Srvinfo.rwrl` and their subsequent functions. If `CAS_Srvinfo.acco` uses pointers and changes their values due to frequent (re)allocations, and they are used by `CAS_Srvinfo.preq` and `CAS_Srvinfo.rwrl`, they must be protected by `CAS_Srvinfo.mm`:
– `acco` must protect the pointer when (re)alloc it
– `preq` and `rwrl` must always protect the pointer

As a rule of thumb, this function should be very short and fast.

If the function returns `1` (true), the connection is accepted and data will be received from client.
If the function returns `0` (false), the connection is closed without receiving any data from client.

Your server should be able to manage both the fourth and the sixth Ip versions.

CAS_Srvinfo.data, **CAS_Srvinfo.html**, **CAS_Srvinfo.cnfg**

If defined, any function (from the above list) is called by the main process immediately after the start moment, and after any administrator request (`--data`, `--html` or `--cnfg`, respectively). When an administrator request is received, the server waits until every previous request is processed. Any further connection is temporarily suspended and the request is solved. Finally, the server is ready to receive new connections.

Any function of the above list (`data`, `html`, `cnfg`) may safely read and write any global variable.

Beware! These functions are called by the main process, so if they crash for any reason, the main process terminates. The release version does not terminate the rest of threads (clones).

CAS_Srvinfo.preq and **CAS_Srvinfo.rwrl**

These functions must use only local variables for read and write operations. Global variables must only be read, or protected by mutexes. If you want to use global variables which are managed by `CAS_Srvinfo.acco`, see the above remark.

cApps-post.pdf will document the rest of information.

Appendix D: Functions defined by this framework

void CAS_resetOutputBuffer(CAS_srvconn_t *Conn);

Resets the output buffer, to use a specific response header (see `csrvD` and `csrvJ` for examples). You must call it before the first `CAS_nPrintf` call, which must send special response headers.

void CAS_nPrintf(CAS_srvconn_t *Conn, char *Fmt, ...);

Sends a string which is formatted according to `Fmt` format. See Appendix E for details about format descriptors. When `CAS_nPrintf` finishes, `Conn->Bft` (buffer for temporary strings) is reset. The pointer to the current position in this buffer, `Conn->Pct`, is set to the beginning of the buffer.

If we previously called any function using the temporary buffer (`CAS_sPrintf`, `CAS_convertString`, functions defined by the programmer – see `csrvH`), and we want to preserve the returned values, we must duplicate them (`strdup` from standard C library) before calling `CAS_nPrintf`.

char *CAS_sPrintf(CAS_srvconn_t *Conn, char *Fmt, ...);

Returns a temporary string which is formatted according to `Fmt` format. See Appendix E for details about format descriptors. If there is not enough memory in the buffer for temporary strings (`Bft`), `NULL` is returned. This function is intended to be similar to PHP `sprintf` function, which returns a string.

#define CAS_endOfString(S,k) (S) + (strlen(S) + k)

This macro points `k` octets after the end of `S` string.

double CAS_getTime(CAS_srvconn_t *Conn);

Returns the number of seconds (real time) elapsed from the point of connection, and before receiving any data.

Beware! This function is based on the `gettimeofday()` system call, so the elapsed (computed) time may be affected by discontinuous jumps in the system time (for instance, if the system administrator manually changes the system time in the middle of request processing).

char *CAS_getParamName(CAS_srvconn_t *Conn, char *From);

If `From` is `NULL`, returns the first parameter of GET list. Otherwise, returns the parameter which appears after `From` (previously returned). If the returned value is `NULL`, the end of GET list was detected.

char *CAS_getParamValue(CAS_srvconn_t *Conn, char *Name, char *From);

If `From` is `NULL`, returns the first value of `Name` parameter. Otherwise, returns the value of this parameter which appears after `From` (previously returned). If the returned value is `NULL`, the end of GET list was detected.

char *CAS_getLastParamValue(CAS_srvconn_t *Conn, char *Name);

Returns the last value of `Name` parameter. If this parameter is not found in the GET list, returns an empty string `""`. You may change the behavior of this function; in your function `CAS_registerUserSettings` insert the following statement:

```
CAS_Srvinfo.Nv = NULL;
```

#define CAS_getThisParamValue(Conn, Pnam) (Pnam + strlen(Pnam) + 1)

For the sake of consistency, `CAS_getThisParamValue` and `CAS_getLastParamValue` use the same arguments, although `CAS_getThisParamValue` ignores the first argument. The second argument must be a pointer variable whose value was previously returned by `CAS_getParamName`, otherwise you will get unpredictable results. This macro returns the value associated to `Pnam` argument.

char *CAS_getHeaderName(CAS_srvconn_t *Conn, char *From);

If From is NULL, returns the first header of headers list. Otherwise, returns the header which appears after From (previously returned). If the returned value is NULL, the end of list was detected.

char *CAS_getHeaderValue(CAS_srvconn_t *Conn, char *Name);

Returns the value of Name header (case insensitive). Each header is supposed to appear only once.

char *CAS_convertString(CAS_srvconn_t *Conn, char *Str, char op);

Str is an input string to be converted. The function return a temporary string (from Conn->Bft) converted as follows.
If op is H (html conversion), the function returns the converted string with the following characters replaced:
< ⇒ `<` **>** ⇒ `>` **&** ⇒ `&` " ⇒ `"`

If op is U (url conversion), the function returns the converted string with letters including _ (underscore) and decimal digits unchanged, any other character is replaced by %hh sequence where hh is the ASCII code of the character.

If there is not enough memory in the buffer for temporary strings (Bft), NULL is returned.

If op is not accepted, an empty string is returned.

char *CAS_loadTextFile(char *Nft);

Returns the content of Nft file. If the file cannot be accessed (not found, permission denied etc), the application stops with an error message. This function must be used to load text files which are compulsory to run the server. If the file extension is .htm* or .xml*, and the file contains <!—Break --> comments, the content is split such that each special comment is replaced by one null octet.

int CAS_explodeHtm(char *Htmi, void *Htmo, int siz);

Htmi is a chain of null terminated strings, the result of CAS_loadTextFile call with .htm or .xml file as argument. The end of chain is marked by two consecutive null octet characters.
Htmo is the address of a structure whose members are of char* type.
siz is the size of Htmo structure (octets).

CAS_explodeHtm copies each string of the chain to a member of Htmo structure. If every string was successfully copied, the function returns 1 (true). If Htmi stores *n* strings obtained by CAS_loadTextFile ($n \geq 2$), Htmo must have at least *n* members.

If Htmo has not enough members to take all strings, the structure members are set to NULL and the function returns 0 (false). You should check the returned value, otherwise the application may crash (segmentation fault) if the output structure has not enough members.

int CAS_serverMutex(CAS_srvconn_t *Conn, int32_t *mtx, char op);

If op is 'L', CAS_serverMutex tries to lock the variable where mtx points to. If the variable is not locked by another clone, this variable is immediately locked; otherwise, CAS_serverMutex waits until it is released. Before locking for the first time a mutex variable, it must be set to 1.
If op is 'R', CAS_serverMutex releases the lock over the variable where mtx points to.

On success, the function returns 1 (true).

On error, it returns 0 (false): op must be 'L' or 'R'.

char *CAS_buildMimeTypeList(char *Cfg);

Builds the MIME type list according to the input string provided by Cfg. This function must be called by CAS_Srvinfo.cnfg and every line of the input string must match the following format:

.*ext* *Content-type or* **?** examples: `.pdf   application/pdf`
`.html.gz   text/html`

Beware! If you use an extension of the second type, your response header must include the following line:
`Content-encoding: gzip`

If an extension is associated to **?** (question mark), any file with this extension will not be allowed for download. The last line of the list must be
***** *Content-type or* **?**

This entry is reserved for any extension not found in the given list. *Content-type* of the last line may be **?**, which means: any extension not found in the given list will not be allowed for download.

`void CAS_sendFileToClient(CAS_srvconn_t *Conn, char *Nft, char *Rhf, int (*valid)(char *));`

Sends the `Nft` (**N**ame of **f**ile to be **t**ransfered) file to the client, according to `Rhf` (Response header format) and `valid(Nft)` validation response. `Rhf` should be one of the user defined response headers, `CAS_Srvinfo.Rh[`*n*`]` where $n \geq 2$. The format of response header must use the following descriptors (in this order):

`%s` (content type), `%u` (file size), `%s` (optional: file name, to be used by the client / browser)

If `valid` is NULL, every file is allowed for download. If defined, `valid(Nft)` should return 1 if `Nft` is allowed for download and 0 otherwise. The validation function is called first (before scanning the MIME type list) and, if returns 1, the server tries to send the requested file.

If the requested file can be open and is accepted for download (the validation function returns 1 and the extension is accepted), the client will receive its content. Otherwise, an explanatory error message will be received and displayed.

If you send a compressed (gzip) text file (html, xml, css, javascript etc), the filename is not mandatory.

`void CAS_sendContentToClient(CAS_srvconn_t *Conn, char *Nft, char *Rhf, void *Buf, int siz);`

Sends `siz` octets from `Buf` (binary content) to the client, according to `Rhf` (Response header format). `Nft` (**N**ame of **f**ile to be **t**ransfered) is used:
– to identify the Content-type associated to the information from (`Buf`, `siz`);
– by the client, which must know the file name to be saved (if save operation is required by Response header).

The extension of `Nft` should not be associated to `?`, otherwise the client will receive *Permission denied* message.

To gain speed and save memory, you should use this function to send binary content which has previously been loaded by `CAS_Srvinfo.data('L')` or `CAS_Srvinfo.html('L')`.

Important:
– if you forget to call `CAS_buildMimeTypeList`, then send functions (`CAS_sendFileToClient` and `CAS_sendContentToClient`) will not work;
– `CAS_nPrintf`, `CAS_sendFileToClient` and `CAS_sendContentToClient` must not be mixed, that is, a program sequence must use either: one or more `CAS_nPrintf` calls / one `CAS_sendFileToClient` call / one `CAS_sendContentToClient` call.

cApps-post.pdf will document the rest of information.

Appendix E: Format descriptors accepted by CAS_nPrintf and CAS_sPrintf

`%%`	or `%` followed by space: write `%` character
`%c`	the corresponding argument is of `char` type
`%s`	the corresponding argument is a pointer to null terminated string; write the whole string
`%.ns`	(*n* > 0 is a decimal number) the corresponding argument is a pointer to null terminated string; write the first *n* characters (or the whole string if its length is less than *n*)
`%.nf`	(*n* ≥ 0 is a decimal number) the corresponding argument is a value of `double` type; write the value with *n* digits after the decimal point; if *n* = 0, write the value without decimal point and digits after; if the value is zero, write `0`
`%.nF`	if the value of `double` type is zero, write nothing, otherwise `%.nf` format is used
`%.ne`	the corresponding argument is of `long double` (*extended*) type
`%.nE`	
`%d`	the corresponding argument is of `int` type
`%.0d`	if the corresponding `int` value is 0 (zero), generate nothing
`%nd`	(*n* > 0 is a decimal number) write the value of `int` type using at least *n* characters if the first digit of *n* is `0` (zero), the value is zero padded
`%u`	the corresponding argument is of `unsigned` type
`%.0u`	
`%nu`	
`%x`	the corresponding argument of `unsigned` type is printed in hexadecimal
`%.0x`	
`%nx`	
`%D`	the corresponding argument is of `long long` type
`%.0D`	
`%nD`	
`%U`	the corresponding argument is of `unsigned long long` type
`%.0U`	
`%nU`	
`%X`	the corresponding argument of `unsigned long long` type is printed in hexadecimal
`%.0X`	
`%nX`	

The special sequence `%%` generates the `%` character. Sometimes this sequence cannot be used, for instance in html template pages used to generate responses, because the template page will not be correctly displayed by the browser. In these cases we may use `%` followed by space, so we may see how the page will look.

Some special formats have been designed to be used in forms, see `csrvD` and `csrvE`. When the server is accessed, the *n* value is displayed using `%.0d` format, so the first access will display an empty string instead of `0` (zero) value. If your program requires to use (`long`) `double` values, use `%.nF` or `%.nE` format.

If the format descriptor is not in the accepted set, it will be ignored. The format descriptor will end at the first letter detected. For every numeric format, the maximum width is limited to 31 positions.

If you want to use some special formatted string (not documented here), you must first use `sprintf` or `snprintf` (standard library), then pass the result as an argument with `%s` format descriptor.

The set of accepted format descriptors is limited because of the special behavior of `CAS_sPrintf` function (see Appendix D). If there is not enough memory in the buffer for temporary strings (`Bft`), `NULL` is returned. I can't use `snprintf` function (standard library) because it doesn't signal any error if the length of the output string exceeds the imposed limit.

Appendix F: The management of errors

Release version

When the release version is started, it tries to:
– open the configuration file and (eventually) the files specified by the developer;
– allocate memory for its important data;
– create a listening socket according to the configuration.

If the server succeeds to perform the initializations, an error log file is open / created, having the same name as the program file and the `.err` extension (see the directory content). The following errors may be recorded:

– *Connection refused*	the requests coming from some particular Ip addresses are rejected by your own `CAS_Srvinfo.acco` function
– *Request timeout*	the server timed out waiting for the request; maybe you should increase the corresponding value (configuration file) if you are absolutely sure the requests are not slowloris attacks
– *Post method not allowed* – *Load method not allowed*	POST and LOAD methods are not allowed (see Appendix G for details)
– *Input buffer overflow* (*nnn octets needed*)	the input buffer can't store the parameters and the headers sent by client you should increase the size of the input buffer

If any unknown method is detected (including HEAD, PUT, UPDATE, DELETE etc), the server response is 404 *Not found*. Run-time errors are also recorded by the release version:
– *Segmentation fault*
– *Floating point exception*
– *Alarm clock*

Every recorded run-time error is followed by the list of GET parameters and their values at the moment of event generation. After recording the run-time error, the clone exits. You may later run the debug version of your application and debug it to correct the errors.

If the request processing takes too much time, a SIGALRM signal is generated. If the signal is generated for the first time, `Conn->tmo` is set to 1. If this signal is generated next time, and `Conn->tmo` is not zero, the clone sends an error message to the browser, records the error and exits.

If you know a particular request may need too much time to be processed, you should check (from time to time) the value of `Conn->tmo` and set it to zero. You may also include any other necessary / useful action; see `csrvH` for details.

You may analyse the error log content using the `Sortlog.c` program.

Debug version

I suppose the debug version runs under the direct control of the developer, that is, after launching the application, you fulfill the form displayed by the browser and watch over the application server at the same time. The first three messages (*Connection refused*, *Request timeout*, *Post / Load rejected*) are only displayed on the standard error (console).

The run-time errors are not captured by special functions (as the release version does). If a run-time error terminates the debug version, you should run it again using a debugger (gdb) to identify the cause of the error.

Only the SIGALRM signal is processed by the debug version: it only sets `Conn->tmo` to 1, so you may check it from time to time if necessary.

Do not forget: the debug version should be used only to thoroughly test the server application.

Remark. The server doesn't use the standard 40*x* error codes / messages; the error messages are displayed / recorded to be read by developer / administrator.

Appendix G: Using POST method (introduction)

Create (on your workstation) the following file and open it using your favorite browser.

Testpost.htm

```
<html><body>
<form action="http://AddressOfYourLinuxMachine:5002" method=post>
<input type=text name=a value=abc size=3 readonly>
<input type=submit name=S value=Send>
</form></body></html>
```

Now launch the csrvB server on your Linux machine, then click ***Send*** button. The following page will be displayed:

- POST method not allowed

The recognition of POST method is disabled. Let us examine the *Server configuration* section of csrvB.cfg file. The fifth line is reserved for receive parameters: receive timeout (seconds), number of octets to be received by POST / LOAD method and maximum allowed file size (octets) to be uploaded. If the second value is zero (or absent), POST and LOAD methods are not allowed.

Now you may edit the csrvB.cfg file, replacing this value by 4000, for instance, then reload the configuration:
./csrvB --cnfg

If you reload Testpost.htm file on your browser and click again the *Send* button, the request will now be accepted.

Important! The only difference between GET and POST methods (from programmer's point of view) is the first character of Conn->Bfi (buffer for input strings). We use the same functions / macros to obtain the names and values of parameters: CAS_getParamName, CAS_getParamValue, CAS_getLastParamValue and CAS_getThisParamValue.

Understanding the POST method

Let us suppose the client / browser sends the following request:

```
POST / HTTP/1.1
Content-length: nnn
Content-encoding: application/x-www-form-urlencoded
...
```

*param1***=***value1***&***param2***=***value2* . . .

The server expects to receive after the first recv system call at least the list of all headers, including the empty line. Otherwise, the request is rejected (protection against slowloris attack). Now, let us suppose the content length is 8000 octets and the POST / LOAD parameter (from configuration) is 4096 (the value is automatically rounded). The server expects to receive from client at least 4096 octets before receiving the whole request, otherwise the request is rejected.

If your parameter list is big enough (for instance, size > 32 Ko), you should choose very carefully an appropriate value as minimum octets to be received, checking different clients / browsers (desktop and mobile).

Beware! If the first line of your request is something like POST /?*param***=***value* you will lose these parameters. The POST method uses only the parameters from the request body.

cApps-post.pdf will detail the POST and LOAD methods.