# Socket programming project

**General information**

The goal of this project is to implement a simple URL-checking service for authenticated users.

Network traffic has two parts: authentication and URL-queries. Authentication part is done in TCP-based protocol and queries using UDP-packets.

Implementation language for your code should be in C++ and the network communication part should use the BSD/POSIX Socket programming interface. Your code should compile and run in the Lintula Linux environment. If you don't have account for these machines, you should apply for one right away. **Each project group has to test their code in Lintula environment before returning the code.**

The *DEBUG*-mode must be implemented by all groups. Other (actual authenticated) implementations will get you extra points from the work.

**Important detail:**

**Every code** (process) in your work has to have a timer which will stop the program after a fixed time. This requirement is in place to reduce the amount of runaway processes when everyone is developing their work...  
**Implement this timer first thing in every program you are writing for this assignment**.

An example of a timer of this kind is belwo:

#ifdef \_\_cplusplus

#define CPP 1

#endif

#if CPP

#include <csignal>

#include <unistd.h>

#include <cstdlib>

#else

#include <signal.h>

#include <unistd.h>

#include <stdlib.h>

#endif

#define MAX\_RUNTIME 60\*60 /\* one hour in seconds \*/

static void

watchdog(int signro)

{

exit(signro); /\* process will exit when the timer signal arrives \*/

}

int main()

{

if( signal(SIGALRM, watchdog) == SIG\_ERR ) {

exit(2); /\* something went wrong in setting signal \*/

}

alarm( MAX\_RUNTIME ); /\* after this time the program will always exit \*/

/\* This location would contain the rest of the program.

\* In this example we'll just pause the program (to wait for the alarm signal)

\*/

pause();

return 0;

}

## Authentication

The client and the server programs will authenticate the client user using these actions:

|  |  |
| --- | --- |
| **Client** | **Server** |
| * Will initialize a TCP connection to the server and provide a username U (DEBUG, AUTH1) |  |
|  | * Queries user database and gets password P1 of username U (if user is unknown the connection is terminated [1]) (AUTH1) * Creates a random number R, which is sent to the client (AUTH1) |
| * Asks an password P2 from the user * Concatenates the password and the random number (P2+R) and calculates from this data [an SHA256-hash](http://en.wikipedia.org/wiki/SHA_hash_functions) value H1. * Value H1 is sent to the server (AUTH1) |  |
|  | * Calculates SHA256-hash( P1+R ) value H2. * compares values H1 ja H2. * IF values are the same, then password is correct and connection is accepted. ELSE the connection is terminated. (AUTH1, in DEBUG-mode connection is always accepted) |

[1] A secure implementation should not terminate connection at this point, but just continue the handshake with random values. Otherwise an attacker could too easily to check what users the server knows about...

The previous data exchange will be done using the distrib-protocol.doc . When the authentication is finished, both endpoints know the same password (named P1 and P2), this password is referred as P in next parts.

When [Concatenating](http://en.wikipedia.org/wiki/Concatenation) binary numbers (R in P1+R and P2+R), the number is represented in network byte order.

The implementation should use HASH-function: [SHA256-hash](http://en.wikipedia.org/wiki/SHA_hash_functions), whose implementation can be found in [Nettle-library](http://www.lysator.liu.se/~nisse/nettle/) (which is already installed in Lintula system). An example using the nettle library can be found in your project GIT.

After a successful authentication, the TCP-connection is terminated and the client can continue to send URL-checking queries using UDP-part of the distrib-protocol.doc

## URL-checking

The server receives UDP query packets as defined in the distrib-protocol.doc and replies to them also using an UDP packet. The actual meaning of URL-checking is not important for our exercise so the server can reply whatever status it likes (random, always OK, ...).

You can get the 32 bit [UNIX-timestamp](http://en.wikipedia.org/wiki/Unix_time) used in the reply packet in Linux:

#include <stdint.h>

#include <time.h>

...

time\_t t = time(0);

uint32\_t it = (uint32\_t) t;

Every UDP-packet will be [HMAC-SHA256-signed](http://en.wikipedia.org/wiki/HMAC) using the common password P, which verifies that only authenticated clients can send the queries (AUTH1).   
An example of this signature calculation is available in the assignment GIT.

## Server

The name of your server executable must be: server and the program should recognize at least these command line parameters:

* --port T where number T defines the port number where the server listens to connections (both TCP and UDP)
* ONE of these parametes must be specified:
  + --debug server is running in debug mode (only the DEBUG parts of the protocol are implemented)
  + --normal server is running in normal operating mode (AUTH1 parts implemented)
  + --authN server implements extra functionality, where number N is chosen by the project team
* Server can implement other command line arguments if needed by some extra functionality you are implementing.

You code submission must contain a text file named server.txt which contains the username and password with which your server can be used.

**Client**

The name of the client executable must be: client and the program should recognize at least these command line arguments:

* --server N where 'N' is the server to connect to, either in "dotted decimal" format ("130.230.4.2") or in nameserver resolvable text string (e.g. "mustavaris.cs.tut.fi")
* --port T port number of the service in the server, where the connection attempt is made.
* --user U where 'U' is the username string (1-64 characters)
* --pwd P where 'P' is the password for the user (1-20 characters)
* --query URL where 'URL' is the request string for the server
* ONE of these parameters must be specified:
  + --debug make the connection in debug mode (only the DEBUG parts of the protocol are implemented)
  + --normal make the connection in normal operating mode (AUTH1 parts implemented)
  + --authN connection with extra functionality, where number N is chosen by the project team
* Client can implement other command line arguments if needed by some extra functionality you are implementing.

The client program will always first complete the authentication phase using a TCP connection, after which it will send ONE query packet using UDP. Server will respond to the query and the client will print out the query results (time and OK/NOTOK) to the user.

## Grading

**The minimal implementation for this project work is a working server and client programs which implement the TCP handshake in DEBUG mode; client sends one UDP query packet, server sends a reply packet and the reply is printed out to the client user. All communications are implemented using IPv4 traffic.**

You can earn extra points by implementing one or more of extra features. **If you implement any of the extras, it must be documented in your retuned code in a file named README.txt**. Only those features mentioned in the file can get extra points. Remember that the basic functionality (DEBUG) MUST be working in your client and server even if you implement any of the extra parts (extras must be activated using command line flags in executable programs).

The points available for these features can still change.

|  |  |
| --- | --- |
| Feature | Points |
| Concurrency The server is using some mechanism (UNIX fork(), threads, select/poll) which allows it to serve more than one client at the same time (both TCP and UDP handling can be done at the same time). | +2p |
| Authentication is implemented AUTH1-parts of the protocol are implemented. | +1p |
| SRP authentication is implemented Fatal fault in AUTH1 is that the server must store the user's password in plaintext format. Create a new authentication scheme (AUTH2) which uses [SRP authentication](https://tools.ietf.org/html/rfc2945)to correct the issue. You'll need to define and document a version of the distib-protocol to add the extra exchange of SRP-parameters between the server and client. | +3p |
| UDP-traffic is verified UDP-packet traffic is implemented in such a way, that most UDP-traffic problems are addressed (packets can be lost or duplicated in transit). Your new protocol must still use UDP, so converting to TCP is not an option. | +2p |
| IPv6 All communication (TCP ja UDP) are implemented using IPv6 (add an extra commandline flag --ipv6 for this). | +1p |
| SSL TCP-communication is implemented using SSL-secured transport (add and extra commandline flag --ssl for this). | +1p |

## Notes and extra information

* Library documentation:
  + [Nettle](http://www.lysator.liu.se/~nisse/nettle/)
* In minimal implementation, the server can operate iteratively (serve one client at the time). If you are implementing a concurrent server, then remember to take care of mutual exclusion in all the necessary places in your code (this assignmnet is not about concurrent programming, but if you use it we'll require you to use it correctly).
* The requirement "server queries user database" does NOT mean that you should use an actual database engine in this project work. For our purposes this "database" can be a file or an environment variable or an internal data structure in your program.
* If you are using C++ as your implementation language, remember to include C-library headers (e.g. sockets) using extern "C" { -structure.
* Internet is full of examples on how to use socket-interface, and you are free to use them (remember to check the course teachings also, so you won't use the interface in wrong way). We also provide our own example code: socket-example.zip
* **Remember to take into account the partial read and write operations when using a TCP-connection.** (readn-example code can be found in the lecture slides.)
* Handling a packet data in C/C++ can be difficult at times and we have en example code of that also. Memory.cc file
* Parsing a command line is not the main focus of our project work, so you probably should use an exsisting library for that. Linux has e.g. [getopt library available](http://linux.die.net/man/3/getopt).