

STester

Apostol Horia-Andrei

anul II, grupa A4

1 Introducere

Proiectul **STester** constă în dezvoltarea unei aplicații de tip server-client, în vederea testării semi-automate a unor servere specificate de către utilizator. Aplicația va fi dezvoltată în considerarea testării serverelor de tip *HTTP*, *FTP* sau *DNS*. Fiecare utilizator va dispune de o interfață proprie de unde se va putea conecta la server și configura (*CRUD*) diverse cazuri de test specifice pentru unul sau mai multe servere alese, fără a se suprapune cu testele altor utilizatori. De asemenea, cazurile de test și rezultatele acestora, împreună cu data la care au fost rulate, vor fi salvate într-un fișier care va putea fi accesat ulterior de utilizatorul căruia îi aparține. Astfel, utilizatorul poate face mai apoi *Regression Testing* sau *Retest*, după caz. Numele utilizatorilor va fi salvat într-un fișier de intrare, iar logarea se va face fără parolă.

2 Tehnologii utilizate

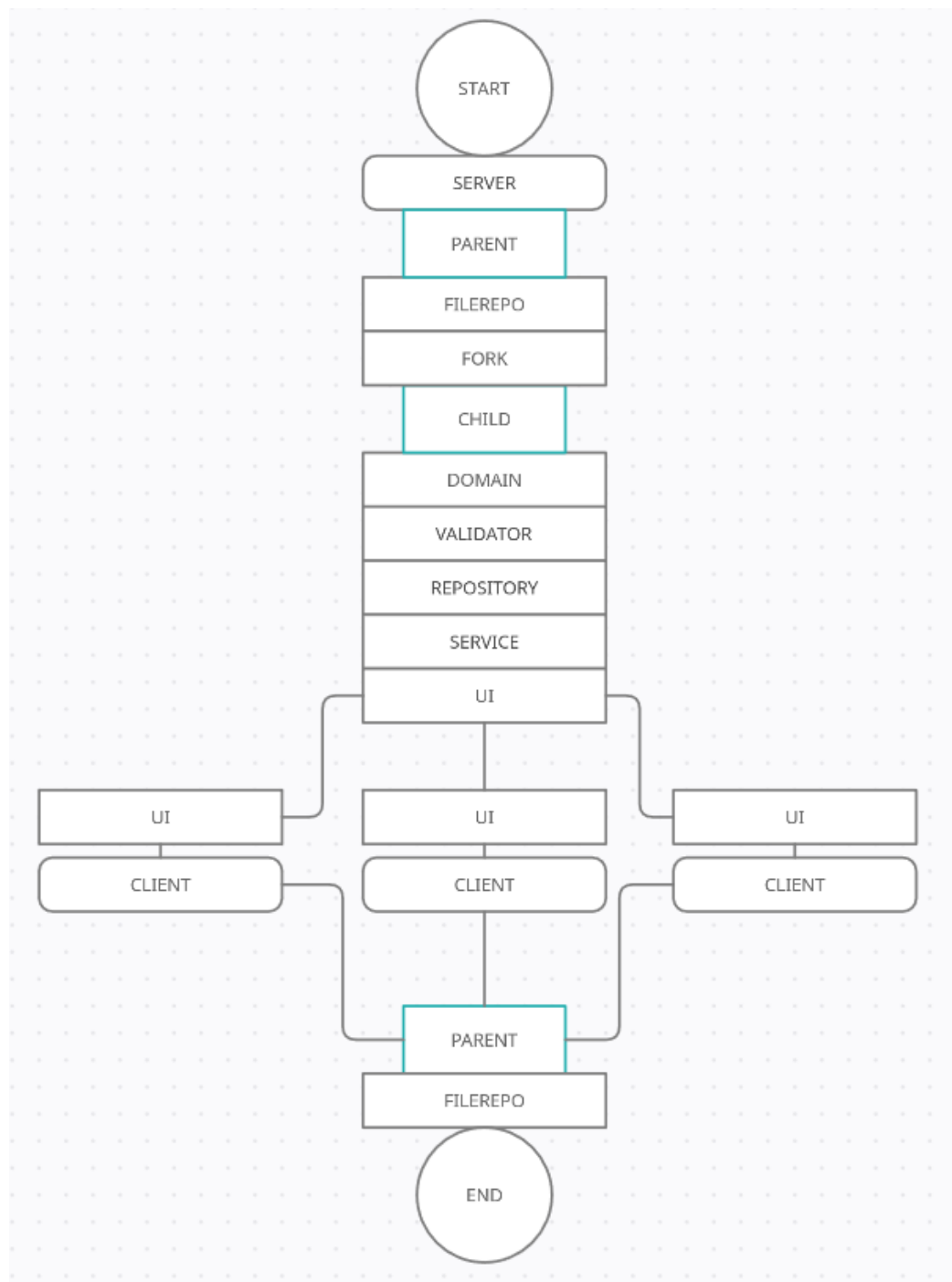
Limbajul de programare utilizat va fi *C++*, iar comunicarea va fi realizată cu ajutorul funcțiilor *C* studiate la laborator, printr-un server *TCP* concurent, care permite conectarea mai multor clienți deodată. Modul prin care se face concurența este cu ajutorul proceselor copil (cu instrucțiunea *fork()*), care vor lucra în simultan fără a se intercala. Alegem această metodă deoarece nu vrem ca un utilizator să poată schimba, șterge sau adăuga teste în mediul de lucru al altui utilizator. De asemenea, alegem acest tip de dezvoltare pentru că procesele copil vor fi proiectate pentru a face operații complexe, nu vor exista variabile globale și fiecare proces va fi izolat, pentru o securitate

sporită, în cazul în care vom avea teste de securitate (*conceptul de sandboxing*). În fine, comunicarea prin *TCP* ne asigură că nu vor exista pierderi sau trunchieri de date, acest lucru fiind esențial în cadrul unei aplicații de testare. Pentru testarea serverelor *HTTP* și *DNS* voi folosi servere publice (ex. *www.google.com* iar pentru testarea serverelor *FTP* voi crea un server pe o mașină *Windows* la care mă voi conecta de pe mașina *Linux*. Instrumentul folosit pentru această sarcină va fi probabil *FileZilla*, deoarece vizualizarea grafică a operațiilor oferă o acuratețe sporită la testare. Pentru testarea manuală a cazurilor de test pe care le voi implementa voi folosi instrumentul *TestRail* și voi controla rezultatele și versiunile cu ajutorul *GitHub*. Voi lucra și testa pe o mașină virtuală cu o distribuție *Debian*, *Kali Linux*, iar unde este necesar voi folosi o mașină cu sistem de operare *Windows 10*.

3 Arhitectura aplicației

Abordarea problemei va fi de tip *Bottom-Up* deoarece în vederea îmbunătățirii aplicației se pot adăuga mult mai ușor alte cazuri de test sau funcționalități. Aplicația va fi construită pe modelul arhitecturii stratificate, unde vom avea în primul rând stratul *Domain*. Acolo vom avea informații despre entitățile pe care le vom folosi în aplicație, împreună cu validatoare. De exemplu, vom avea *Server Domain* și *Test Case Domain* unde vom salva caracteristicile și stările de bază ale unor entități de acest tip. Cu ajutorul claselor din *C++* vom crea mai multe obiecte de tip *DTO* care vor sta la baza aplicației. Următorul strat va fi *Repository*, care va conține mai multe mulțimi de entități, cum ar fi o mulțime cazuri de test sau servere. Apoi, vom face stratul *Service* care se va ocupa de validarea și comportamentul entităților. Toate acestea vor fi combinate în procesele copil ale serverului. În procesul părinte va exista doar un *File Repository* unde vom memora tot ce ține de fișiere text. În cele din urmă vom face stratul *Ui*, localizat în aplicație și server și în client, proiectat pentru a primi și transmite informații. Toate informațiile care prin intermediul *Ui* ajung la server din client vor fi preluate imediat de *Service* (care are rolul de *GRASP Controller*) și validate pentru a nu exista posibilitatea ca datele transmise ilegal de la un client să împiedice rularea corectă a serverului. Toate operațiile pe care le va face un utilizator vor fi executate în procesul copil specific acestuia. În procesul părinte se va face bineînțeles conexiunea și management-ul proceselor, dar se vor face și operații referitoare la login și controlarea istoricului de teste. În principiu,

pentru comunicarea între server și clienți voi utiliza modelul *TCP* concurent cu *fork()* studiat la laborator.



4 Detalii de implementare

Comunicarea între procese va fi realizată cu ajutorul unui *socket()*. Voi folosi structuri precum *sockaddr_in* pentru comunicarea între servere, dar și structuri precum *hostent* sau *protoent* de unde voi extrage date necesare comunicării, respecti testării. De exemplu, un *DTO* server, care este descris prin *nume* și *adresa IP* va fi implementat astfel:

```
class Server {
private:
    const char *host_name, *ip_address;
public:
    Server(const char *host_name, const char* ip_address)
        : host_name{ host_name }, ip_address{ ip_address } {};

    void set_host_name(const char *host_name){
        this->host_name = host_name;
    }
    void set_ip_address(const char *ip_address){
        this->ip_address = ip_address;
    }
    const char* get_host_name(){
        return this->host_name;
    }
    const char* get_ip_address(){
        return this->ip_address;
    }
};

int main() {
    Server my_server("www.google.com", "142.250.187.132");
    return 0;
}
```

Însă în cazul în care vreau să aflu adresa IP a unui host am nevoie de o structură *hostent*.

```
typedef struct hostent {
    char *h_name;
    char **h_aliases;
    short h_addrtype;
    short h_length;
    char **h_addr_list;
};
```

Problema cu aceasta este că *char **h_addr_list* este de fapt un tablou de structuri *in_addr*, care reprezintă adrese de internet. Creând un obiect de tip server putem păstrăm doar câmpurile de care avem nevoie și să le modificăm unde este cazul. Putem face cast direct pentru a nu trebui să facem mereu conversii în aplicație. Un exemplu de implementare trunchiată (în aplicație se vor face toate separat):

```
#include "iostream"
#include "netinet/in.h"
#include "netdb.h"
#include "arpa/inet.h"
#include "unistd.h"

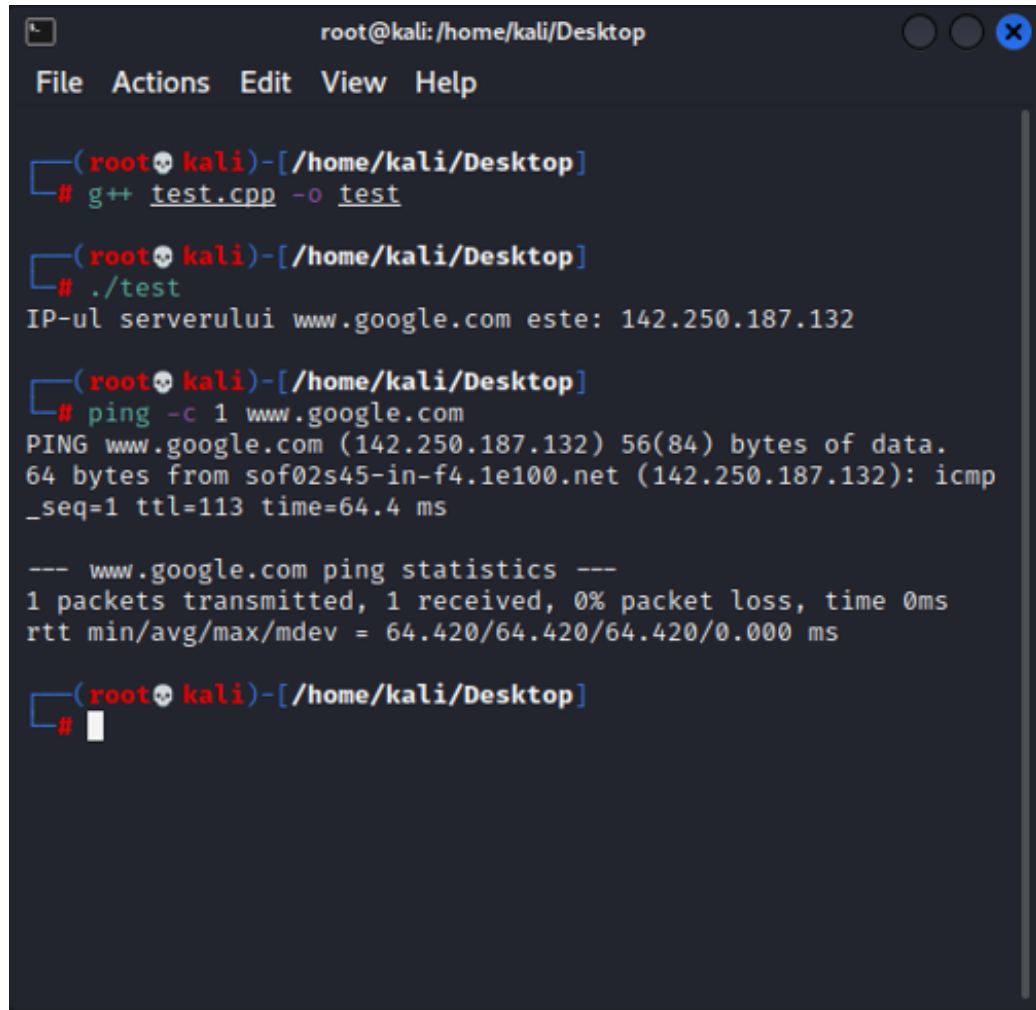
class CreateServer {
private:
    const char* host_name;
    in_addr *host_address;
public:
    CreateServer(hostent *host){
        this->host_name = host->h_name;
        this->host_address = reinterpret_cast<in_addr*>(host->h_addr);
    }
    const char* get_host_name(){
        return this->host_name;
    }
    const char* get_ip_address(){
        return inet_ntoa(host_address[0]);
    }
};

int main() {
    const char* host_name = "www.google.com";
    CreateServer my_server(gethostbyname(host_name));

    std::cout << "IP-ul serverului " << host_name << " este: " << my_server.get_ip_address();
    return 0;
}
```

În cazul de mai sus *h_addr* este un macro pentru *h_addr_list[0]*

Rezultatul acestei clase de test este corect:

A terminal window titled 'root@kali: /home/kali/Desktop' with a menu bar (File, Actions, Edit, View, Help). The terminal shows the following commands and output:

```
(root@kali)-[/home/kali/Desktop]
# g++ test.cpp -o test

(root@kali)-[/home/kali/Desktop]
# ./test
IP-ul serverului www.google.com este: 142.250.187.132

(root@kali)-[/home/kali/Desktop]
# ping -c 1 www.google.com
PING www.google.com (142.250.187.132) 56(84) bytes of data.
64 bytes from sof02s45-in-f4.1e100.net (142.250.187.132): icmp
_seq=1 ttl=113 time=64.4 ms

--- www.google.com ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 64.420/64.420/64.420/0.000 ms

(root@kali)-[/home/kali/Desktop]
#
```

Din moment ce putem afla date despre *host-uri*, putem implementa funcționalitatea de măsurare a latenței, prin trimiterea și primirea de pachete către server și înapoi la server-ul nostru și folosirea structurii *timeval*. De altfel, putem verifica și dacă funcționalitatea de *DNS-Lookup* este făcută corect. În cazul unui server *HTTP*, încercăm să ne conectăm la server și trimitem un mesaj oarecare. Asta, cred eu, ar trebui să ne confirme posibilitatea de download. În caz contrar, voi încerca implementarea unei funcționalități de tip *wget* pe cât posibil. Pentru serverele *FTP* voi implementa comenzile uzuale în cod și voi testa funcționalități de genul *mkdir*, *get*, *put*.

Un scenariu de utilizare: *Utilizatorul 1* se conectează la server și se loghează în aplicație. În același timp, *Utilizatorul 2* se conectează la server și se loghează în aplicație. *Utilizatorul 1* vrea să afle dacă site-ul *Emag* funcționează și creează un caz de test, alegând din meniu tipul de test și introducând numele host-ului de la tastatura. *Utilizatorul 2* e pasionat de rețele de calculatoare și abia ce și-a făcut propriul server *FTP*. Ambii utilizatori rulează testele după ce le creează și mai apoi citesc fișierul de jurnalizare, apăsând tasta corespunzătoare din meniul precedent. Din fericire, ambele teste au funcționat. *Utilizatorul 1* închide aplicația pentru că se grăbește să prindă reducerile de pe *Emag*, iar *Utilizatorul 2* începe să testeze și alte funcționalități.

5 Concluzii

În concluzie, soluția propusă constă în modelarea cât mai precisă a entităților aplicației, ca mai apoi să se poată adăuga diverse tipuri de teste. De exemplu, aplicația s-ar putea îmbunătăți prin adăugarea de teste de tipul *Performance Testing*, cum ar fi *Ce s-ar întâmpla dacă 100.000 de utilizatori s-ar conecta simultan la server*, sau teste de tipul *Security Testing/Penetration Testing* cum ar fi găsirea de vulnerabilități, și încercări de *XSS*. De asemenea, în cazul acesta, o idee suplimentară ar fi implementarea unei logări cu parolă. S-ar putea face în primă fază cu un *std::map* din *C++* iar mai apoi, maparea cheie valoare cu într-o bază de date *SQLite*, cu parolă criptată.

6 Bibliografie

<https://profs.info.uaic.ro/~computernetworks/cursullaboratorul.php>
<https://profs.info.uaic.ro/~matei.madalina/retele/#laboratory-6>
<https://www.tenouk.com/cnlinuxsockettutorials.html>
<https://linux.die.net/>