# SZAKDOLGOZAT

Kránitz Gábor

2015

Pannon Egyetem

Villamosmérnöki és Információs Rendszerek Tanszék

Programtervező Informatikus BSc szak

# SZAKDOLGOZAT

# Tűzfalak tesztelése hálózati forgalom visszajátszással

Kránitz Gábor

Témavezető: Dulai Tibor

Külső konzulens: Tollas Ferenc

2015

Ide jön az eredeti vagy a fénymásolt feladatkiírás.

# Nyilatkozat

Alulírott Kránitz Gábor diplomázó hallgató kijelentem, hogy a szakdolgozatot a Pannon Egyetem Villamosmérnöki és Információs Rendszerek tanszéken készítettem Programtervező informatikus BSc szak (BSc in Computer Science ) megszerzése érdekében.

Kijelentem, hogy a szakdolgozatban lévő érdemi rész saját munkám eredménye, az érdemi részen kívül csak a hivatkozott forrásokat (szakirodalom, eszközök, stb.) használtam fel.

Tudomásul veszem, hogy a szakdolgozatban foglalt eredményeket a Pannon Egyetem, valamint a feladatot kiíró szervezeti egység saját céljaira szabadon felhasználhatja.

Veszprém, 2015. december 02.

Aláírás

Alulírott Dulai Tibor témavezető kijelentem, hogy a szakdolgozatot Kránitz Gábor a Pannon Egyetem Villamosmérnöki és Információs Rendszerek tanszéken készítette Programtervező informatikus BSc szak (BSc in Computer Science ) megszerzése érdekében.

Kijelentem, hogy a szakdolgozat védésre bocsátását engedélyezem.

Veszprém, 2015. december 02.

Aláírás

# Köszönetnyilvánítás

Köszönöm a családomnak a sok türelmet és segítséget, amit kaptam, nélkülük ez a szakdolgozat nem készült volna el.

Köszönöm témavezetőmnek, Dulai Tibornak az elmúlt egy év során adott iránymutatását.

Végül, de nem utolsó sorban, szeretném megköszönni munkatársaimnak a sok segítséget, szaktársaimnak a bíztatást.

# TARTALMI ÖSSZEFOGLALÓ

E szakdolgozat témája egy olyan rendszer fejlesztése, amelynek segítségével előre rögzített hálózati forgalom visszajátszásával a tűzfalak egyszerűen tesztelhetőek.

A keretrendszer fejlesztése első sorban Python-ban történt, viszont a rendszer egyes komponenseinek átalakításához szükség volt C++ kódok módosítására amihez Qt-t használtam.

**Kulcsszavak:** *Zorp, TcpForwarder, rögzítés, visszajátszás, teljesítmény teszt, keretrendszer*

# ABSTRACT

The topic of this thesis paper is the development of a system, that allows simple testing of firewalls using playback of pre-recorded network traffic.

The develelopment of the framework was done mainly in Python, however the conversion of some components of the system required C++ code that was modified by me using Qt.

**Keywords:** *Zorp, TcpForwarder, record, replay, performance test, framework*

# Tartalomjegyzék

# 1. A feladat összefoglalása

Szakdolgozatom témáját egy informatikai biztonsággal foglalkozó cégnél a BalaBit IT Biztonságtechnikai Kft-nél választottam, ahol Junior Szoftverfejlesztő pozícióban dolgozom.

A tűzfalak és egyéb hálózati kommunikációt folytató programok automatikus teszteléséhez nagy hardveres és emberi erőforrásigény társulhat, például kliens és szerver gépek egyidejű üzemeltetése illetve azok felügyelete.

A feladat célja olyan szoftver tervezése és implementálása, amely képes előre rögzített hálózati forgalmat (TCP) visszajátszani, mely segítségével szimulálható egy valós teszt forgatókönyv.

A szoftvernek képesnek kell lennie validálni az átmenő forgalmakat. A szoftver készítésénél figyelembe kell venni az igényt a könnyű konfigurálhatóságra, telepítésre, könnyű bővíthetőségre. Felépítés tekintetében egy központosított moduláris rendszer megvalósítása a cél.

# 2. Első lépések

## 2.1. A környezet megismerése

Első lépésként meg kellett ismerjem a BalaBit[1] által fejlesztett Zorp tűzfalat, ami alapjául szolgál a cég termékeinek. A hozzá kapcsolódó TcpForwarder nevű eszközt, ami a tesztelés segítségét szolgálja, illetve ezen eszközök működését és egymáshoz illesztésének menetét.

## 2.2. Már meglévő komponensek

### 2.2.1. Zorp GPL tűzfal bemutatása

[2] Szakdolgozatomban a Zorp tűzfal ingyenes változatát használtam, amit egyes Linux disztribúciókban csomagkezelő segítségével is telepíthetünk saját környezetünkben.

Ennek a változatnak természetes korlátozottabbak a képességei, mint a kereskedelmi változatnak, de számomra ez is elegendő volt jelenleg.

A Zorp GPL egy robusztus határvédelmi eszköz, kiterjedt informatikai hálózattal rendelkező nagyvállalatok és más magas biztonsági igényű intézmények számára készült. A Zorp teljes ellenőrzést biztosít a normál és titkosított hálózati forgalom felett, és képes a forgalom tartalmának szűrésére és módosítására is.

A hálózati forgalom aprólékos feldolgozásával nyert információk alapján a szkriptekkel kiegészíthető grafikus konfiguráció lehetőséget ad az adminisztrátor számára, hogy kompromisszumok nélkül implementálja a vállalat hálózatokra vonatkozó biztonságtechnikai előírásait.

A termék rugalmas autentikációs képességeire olyan fejlett szolgáltatások épülnek, mint például a Single Sign On autentikáció (a felhasználónak elég csupán egyszer azonosítania magát és minden további autentikációt a rendszer automatikusan elvégez) vagy a felhasználói szintű QoS ( a különböző felhasználók, csoportok eltérő minőségű kapcsolatokat vegyenek igénybe, például a használt kliensalkalmazás, célszerver címe, stb. alapján lehet korlátozni a kapcsolat sávszélességét és más paramétereit ).

A Zorp alkalmazásszintű határvédelmi technológia által nyújtott védelem alkalmas a legmagasabb szintű biztonsági igények kielégítésére is. A Zorp tipikus felhasználói az államigazgatási, a pénzügyi és a távközlési szektorból, illetve az iparvállalatok közül kerülnek ki.A széleskörű szolgáltatáspalettának és a részletes testre szabási lehetőségeknek köszönhetően a Zorp technológia több, egymástól jól elkülöníthető területen is hatékonyan alkalmazható .

A rendszer moduláris jellegéből adódóan Python nyelven saját fejlesztésű egységekkel bővíthető.

Főbb jellemzők:

- Egyedi, különleges IT biztonságtechnikai probléma megoldása
- Titkosított csatornák (pl. HTTPS, POP3S, IMAPS, SMTPS, stb.) szűrése
- Központi tartalomszűrés (vírus és spam) akár titkosított csatornákban is
- Speciális protokollok (pl. Radius, SIP, MS RPC stb.) szűrése
- Single Sign On (Kerberos) autentikáció megvalósítása
- Felhasználói szintű QoS megvalósítása

1. ábra: Példa egy hálózat felépítésének modelljére

### 2.2.2. Pcap bemutatása

A számítógép hálózatok területén a pcap (packet capture) biztosít egy alkalmazás-programozási felületet (API) a hálózati forgalom rögzítésére. Unixon alapuló rendszerekben a libvap könyvtár, Windows-os rendszerekben pedig a WinPcap implementálja.

A pcap API C nyelven készült, más nyelvek wrappert használnak hozzá,de maga a libvap vagy a WinPcap sajnos nem biztosítanak saját wrappert. A libcap és a WinPcap rengeteg nyílt forráskódú és kereskedelmi forgalomban kapható hálózati szoftverhez biztosítja a csomagrögzítő és szűrő motort, például hálózat monitorzó eszközök, forgalom generátorok, hálózati behatolást detektáló rendszerek.

Az elkapott csomagok fájlokba rögzíthetőek és a rögzített csomagok fájlból kis is olvashatóak.Libcap vagy WinPcap használatával olyan programok készíthetőek melyek a hálózi forgalomból elkapott csomagokat analizálni képesek, vagy már az előre rögzített csomagokat fájlból kiolvasba tudják azokat analizálni.

Libcap és WinPcap által használt formátumban elmentett fájlokat például tcp-dump, Wireshark vagy CA NetMaster nevű programokkal tudjuk felhasználni. A jellemző fájl kiterjesztés a .pcap, de használatban van a .cap és a .dmp kiterjesztés is.

A .pcap fájl használata egyszerű és sok program képest ezt kezelni. Előnyként említhető, hogy a ki és bemenő adatcsomagokat is képes kezelni.

Programok melyek a libcap-et/WinPcap-et használnak:

- tcpdump
- ngrep
- Wireshark
- Snort
- Nmap
- Kismet
- McAffe
- Scapy

Wrapper könyvtárak libcap-hez és WinPcap-hez:

- Perl: Net::Pcap
- Python: python-libcap, Pcapy
- Ruby: PacketFu
- Tcl: tclpcap, tcap, pktsrc
- Java: jpcap, jNetPcap, Jpcap,Pcap4j
- .Net: WinPcapNet, SharpPcap, Pcap.Net

A fájl rendelkezik egy globális header-rel, amely globális információkat tartalmaz és az azokat követő nulla vagy több rögzített rekordot minden egyes rögzített csomaghoz:

| Global Header | Packet Header | Packet Data | Packet Header | Packet Data | Packet Header | Packet Data | ... |
|---|---|---|---|---|---|---|---|

Global Header felépítése:

```
typedef struct pcap_hdr_s {
                guint32 magic_number;
                guint16 version_major;
                guint16 version_minor;
                gint32  thiszone;
                guint32 sigfigs;
                guint32 snaplen;
                guint32 network;
                } pcap_hdrq_t;
```

- magic_number: fájl formátum és a byte rendezés detektálásra használt
- version_major, version_minor: fájlformátum verziószáma
- thiszone: a következő csomag header időbélyegében a helyi időzóna és a GMT (UTC) közötti idő korrekciója másodpercekben
- the correction time in seconds between GMT (UTC) and the local timezone of the following packet header timestamps
- sigfigs: időbélyeg pontossága
- snaplen: „snapshot length”
- network: kapcsolati réted header típusa, csomag elején lévő header típusát specializáljaRecord (Packet)

Header felépítése:

```
typedef struct pcaprec_hdr_s {
        guint32 ts_sec;
        guint32 ts_usec;
        guint32 incl_len;
        guint32 orig_len;
        } pcaprec_hdr_t;
```

- ts_sec: dátum és idő amikor a csomagot elkaptuk
- ts_usec: a szokásos pcap fájlokban a csomag elkapásának ideje ezredmásodpecben

- incl_len: az elkapott csomagnak a fájlba mentett adat byte-jainak a száma
- orig_len: az elkapott csomag hossza a hálózaton

### 2.2.3. TcpForwarder bemutatása

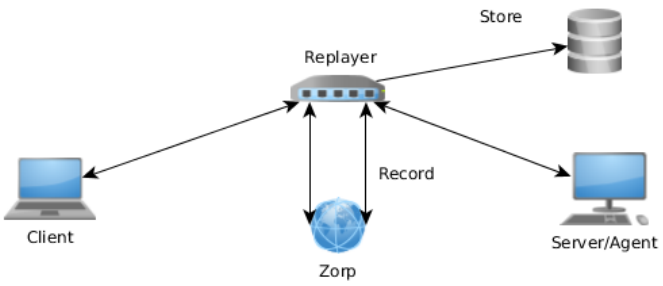
Jelenleg a cég rendelkezik egy TcpForwarder nevű eszközzel ami képes adott átmenő kapcsolatok átirányítására a megadott ip címek és portok alapján, illetve ezen kapcsolatokat a Zorp-pal együttműködve rögzíteni illetve visszajátszani.

Azonban a TcpForwarder nem nyílt forráskódú! A cég nem adott ki belőle publikus verziót mint mondjuk a Zorp-ból.

Ez a megvalósítás C++ nyelven készült, de nem minden esetben használható a megfelelően ezért szükség lehet ennek a rendszernek az áttervezésére és javítására.

Az eszköz az adattároláshoz négy darab fájlt használ. Külön külön elmenti a szerver és kliens oldal adatait kimenő és bemenő irányban, illetve egy kontrolfájlban az egész folyamatot, hogy mi milyen sorrendben történt amit később csak visszaolvas.

A rendszer felépítése két féle képpen ábrázolható:




2. ábra: Rögzítés TcpForwarderrel

A második ábrán (2.) egy olyan hálózatnak a topológiája látható, amelyre a forgalmak rögzítése közben szükség van. Ilyen módban szükség van egy kliens és egy

szerver gépre, egy adattároló eszközre, a Zorp tűzfalra és egy hálózati eszközre ami megteremti a kapcsolatot a többi eszköz között.

A forgalom oda-vissza működik, a kliens géptől elindulva áthaladva a TcpForwarderen keresztül megérkezik a kapcsolat a tűzfalba, ez után ismét a TcpForwarderen át tovább halad a szerver gép felé.

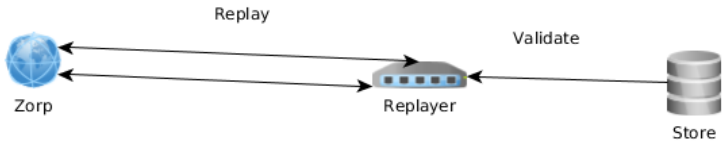
Az aktuális beállításainknak megfelelően pedig megtörténik a kapcsolat rögzítése.

(Kliens) <-> (TcpForwarder) <-> (Zorp) <-> (TcpForwarder) <-> (Szerver) Ez a két TcpForwarder ugyanazt az eszközt jelenti.

Ez a felépítés egy valós tesztkörnyezet alapja lehet, ahol már ténylegesen tesztelhetjük a Zorp-ra épülő eszközt, viszont a szakdolgozatomban elkészült rendszer bemutatására ennél egyszerűbb struktúrát alkalmaztam.

A TcpForwarder másik üzemeltetési módja a már előre rögzített kapcsolat visszajátszására szolgál.

Ennek szemléltetésére az alábbi ábra szolgál: (3.)




3. ábra: Kapcsolat visszajátszása TcpForwarder-rel

A visszajátszás esetében már nincs feltétlen szükségünk a kliens és szerver gépekre. Minimális hálózat felépítéséhez elegendő egy a tűzfalat futtató gép és az azon tárolt kapcsolat. Ilyenkor ezen a gépen indítva a TcpForwardert a megfelelő konfigurációs beállításokkal vissza tudjuk játszani a Zorp-on keresztül a meglévő kapcsolatot. (TcpForwarder) <-> (Zorp)

A TcpForwarder konfigurálásához, egy .ini kiterjesztésű konfigurációs fájl szolgál. Ebben a fájlban van definiálva a TcpForwarder számára, hogy milyen ip címekről milyen porton figyeljen a bejövő kapcsolatokra, illetve ezeket milyen címre és porton keresztül továbbítsa. Ez megtörténik a kliens és a szerver felöli irányból is.

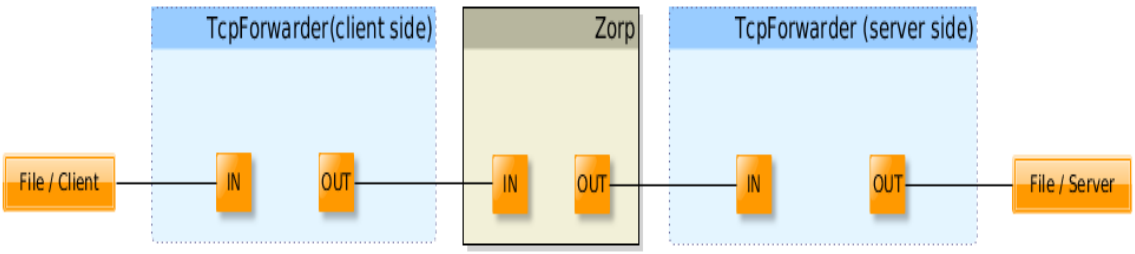
A zorp/szerver/kliensek címei és portjai egy ini fájlban vannak rögzítve. Az „in" rész jelenti az elfogadott bejövő kapcsolat, az „out" rész pedig a TcpForwarder cél címei és portjai.

Példa fájl:

```
[client]
in_address = "any"
in_port = 2209

out_address = "1.1.1.1"
out_port = 2222

[server]
in_address = "any"
in_port = 2210

out_address = "1.1.1.1"
out_port = 23
```

A "client" in_address és in_port jelenti, hogy a TcpForwarder milyen ip címekről és milyen porton várja, hogy kliensek csatlakozzanak hozzá. A csatlakozott klienseket a "client" out_address és out_port-ján továbbítja a Zorp-ot futtató gép felé, ahol a Zorp a "server" in_port-ját figyeli. A TcpForwarder a Zorp felöl érkező kapcsolatokat a "server" out_address és out_portja felé továbbítja.

Az alábbi ábrán (4.) látható, hogy a fent leírt részegységek miként kapcsolódnak össze:




4. ábra: A kliens, TcpForwarder, Zorp és szerver kapcsolata

## 3. Hasonló célú rendszerek

Következő lépésként megvizsgáltam, hogy milyen hasonló célú szoftverek, illetve szoftvercsomagok érhetők el mind nyílt forrású formában, mind pedig kereskedelmi forgalomban.

Erre azért volt szükség, hogy pontosabb képet kapjak a jelenleg fellelhető megoldásokról, és munkám során az így szerzett pozitív és negatív tapasztalatokat eredményesen hasznosíthassam. Kereséseim során számos kész megoldást találtam, viszont ezek között nem akadtam olyan eszközre amely megfelelne a cégem által támasztott igényeknek.

Vannak olyan eszközök melyek arra specializálódtak, hogy a hálózati forgalmat megfigyelhessük az adatcsomagok szintjén. Vagy akár ezek módosítására is képesek legyünk.

Például az alábbi két eszköz is ilyen:

- TCPDUMP [3]

  A tcpdump egy általános, ingyenes (BSD license) csomag ellenőrző eszköz. Karakteres felületen lehetővé teszi, hogy láthatóvá váljanak a TCP/IP és más csomagok is, melyeket a hálózaton keresztül küldünk illetve fogadunk.

  Előnyei közé tartozik, hogy a legtöbb Unix alapú operációs rendszeren működik, illetve létezik Windows-os megvalósítása is WinDump néven.

- Wireshark [4]

  Másik hasonló, de már fejlettebb eszköz a Wireshark nevű ingyenes program.

  Ez a program szintén a hálózaton továbbított csomagok megfigyelésére való, akárcsak a tcpdump, viszont ez már rendelkezik egy grafikus felülettel és

néhány egyéb rendezési illetve szűrési beállítással.

A Wireshark mind Unix mind Windows-os környezetben használható. A tcpdump-pal szembeni előnyei ellenére, sajnos ez az eszköz is csak a csomagok illetve a hálózat ellenőrzésére használható.

Az általam fellelt eszközök másik csoportja a hálózatok tesztelésére szakosodott, oly módon, hogy az azokon átmenő kapcsolatok számát próbálják felmérni vagy az adatáteresztő képességüket.

Ilyen eszköz például:

- Iperf [**?**]

  Ez szintén egy ingyenes hálózat tesztelő eszköz, amely képes TCP illetve UDP adatfolyamokat generálni a hálózaton, amivel mérhetővé válik annak teljesítménye.

Kereséseim során fellelt eszközök, csak részben fednék le a cég által felállított követelményeket. A jelenlegi helyzetben nem elég csak rögzíteni a hálózaton továbbított adatcsomagokat, vagy épp csak a hálózat teljesítményét tesztelni. Képesnek kell lenni arra, hogy a Zorp tűzfalra épített rendszereken meg tudjuk mondani, hogy valós nagyvállalati környezetben adott hálózati protokollok esetében a rendszer mennyire képes ellátni a feladatát és ezt milyen minőségben teszi.

Az általam fejlesztett megoldás viszont a TcpForwarder segítségével konkrétan erre a kérdésre próbál választ adni.

# 4. A rendszer tervezése

## 4.1. Követelmények

Elsődleges szempont volt, hogy az időnként szükséges teljesítménytesztek lebonyolításánál a lehető legtöbb feladatot automatizálhassuk és a lehető legkönnyebben elvégezhetőek legyenek a műveletek.

A teszt rendszer tervezésekor az alábbi követelmények lettek megállapítva:

- Könnyű konfigurálhatóság
- Központosított moduláris rendszer

- Automatikus logolás
- Könnyű bővíthetőség

## 4.2. Modulok

Maga az egész tesztrendszer magábafoglalja a TcpForwarder-t is ugye, de annak részletes bemutatására nem térnék ki, mert az már egy előzőleg elkészült rész.

A szakdolgozat keretein belül fejlesztett tesztrendszer magját alkotó rhino fantázia névre keresztelt eszköz az alábbi főbb modulokból áll:

1. Configurations
2. Agents, Clients
3. Network
4. Pool(Thread Pool)

### 4.2.1. Configurations

A rendszer configurations mappájában olyan .config kiterjesztésű fájlokat tárulunk, melyekben előre definiálhatóak az eszköz által létrehozott Clients és Agents tulajdonságai, illetve a Clients ál végrehajtott műveletek paraméterezhetőek.

### 4.2.2. Agents, Clients

Az Agents és Clients modulok felelősek a hálózati réteg elindítására, a konfigurációk inicializálására, illletve a kapcsolat fenntartására a kontroller modullal.

A kliensek tekintetében a rendszer rendelkezik egy BaseClient elnevezésű interfaceszel, amit minden egyes kliensnek implementálnia kell amit a későbbiekben létrehoznánk.

### 4.2.3. Network

A Network modul felelős a hálózat fenttartásáért, a kapcsolatok kezelésére a csatlakozott kliensekkel.

#### 4.2.4. Pool(Thread Pool)

A rendszer ezen modulja felelős a szálkezelésért. Az elindított és beregisztrált kliensek vezérlését végzi, a kliensek feladatait indítja, ütemezi.

### 4.3. Választott technológiák

A feladat megtervezése után a következő lépés a technológia kiválasztása volt, melyben a megvalósítást véghezvihetem.

Munkahelyi feladataim illetve a Zorp Python-os háttere miatt hamar erre a programozási nyelvre esett a választás.

A fejlesztés illetve a dolgozat elkészítése során verziókövetésre Git-et, Latex-et Gummi-val, illetve az ábrák szerkesztéséhez Yed-et használtam. A TcpForwarder esetleges módosításához Qt-t használtam. A leadott rendszert egy virtuális gépben helyeztem el, amit Oracle VM VirtualBox-ban készítettem és használtam. [6]

A virtuális gépen egy Linux Mint 17(qiana) fut. [7]

Magát a keretrendszert elindító komponsensek pedig Bash script-ekben készültek.

#### 4.3.1. Python

[8]

A Python egy magas szintű, általános célú programozási nyelv, melynek elsődleges filozófiája a fejlesztési feladatok megkönnyítése és produktivitásának növelése, még ha ez a sebesség rovására is megy.

A Python egy script nyelv, úgynevezett interpreteres nyelv, mely nem igényel például a C/C++ kódokhoz hasonlóan fordítót, a megírt kódok azonnal futtathatóak, nincsenek elválasztva egymástól a forrás és tárgykódok.

## 5. A fejlesztés részletei

### 5.1. Architektúrális terv

Az alábbi ábrán tekinthető meg a keretrendszer architekturális terve: (16.)

## 5.2. Fejlesztési napló

1. Környezet kiépítése

   Első feladatom az volt, hogy a saját virtuális gépemen egy megfelelő környezetet építsek ki a Zorp installálásával. A TcpForwarder lefordítása után a saját környezetemben be kellett konfiguráljam a hálózatnak megfelelően, hogy együtt tudjon működni a Zorp-pal.

2. A keretrendszer alapjai

   Megvalósításra került a rendszer alapja, elkészültek a Clients, Agents modulok.

3. Bővítés

   A parancssori utasítások bővültek, a Network és ThreadPool modulok implementálása megtörtént.

4. TcpForwarder hívása

   A TcpForwarder bekötése a rendszerbe és ezen keresztül történő irányítása.

5. Tesztelés

   Telnet kliens implementálása a rendszerbe, valódi környezetben és terméken történő tesztelés.

## 5.3. Modulok részletes bemutatása

A továbbiakban szeretném részletesen bemutatni a keretrendszer moduljait.

### 5.3.1. Configurations

A konfigurációkat egy egyszer json fájlban tudjuk definiálni, amit később a klienseknél tetszőlegesen felhasználhatunk.

Ennek a konfigurációs fájlnak mindenképp tartalmaznia kell a következőket:

- Szimulált kliensek számát
- Kliens nevét a betöltéshez és inicializáláshoz
- A teszt típusát

- Tetszőleges paramétereket a teszt metódusokhoz
- A teszt metódus nevét amit a kliens végrehajt majd

Az alábbi kódrészleten egy példát láthatunk mely a kliensek viselkedését és tulajdonságait hivatott tartalmazni:

```
{
 "command": "set_test_config",
 "simulate_client_number": 4,
 "client_name": "rhino.clients.exampleclient.ExampleClient",
 "test_type": "INFINITE",
 "time": 1234,
 "parameters": {
        "server_address":"1.1.1.1",
        "another_parameter":"value"
 },
 "test_methods": [
     {
         "method_name": "my_test_method_with_one_parameter",
         "method_execution_number": 1,
         "method_params": [
             "param1"
         ]
     },
     {
         "method_name": "my_test_method",
         "method_execution_number": 1,
         "method_params": []
     },
     {
         "method_name": "my_test_method_with_two_parameters",
         "method_execution_number": 1,
         "method_params": [
             "param1",
             "param2"
         ]
     }
 ]
}
```

- "simulate_client_number": Itt adhatjuk meg, hogy párhuzamosan hány darab klienst hozzunk létre

- "client_name": Ez a paraméter a már előre definiált kliens fajtáját jelenti, ezt igényeinknek megfelelően a saját kliensünkkel kiválthatjuk.
- "test_type": Itt a teszt futásának fajtáját adhatjuk meg, ami jelen esetben három féle lehet: SEQUENCE, INFINITE, TIME_LIMITED
- "parameters": Az itt megadott paraméterek átadódnak majd az elkészült klienseknek.
- "test_methods": Ebben a paraméterben egy listát adunk át a rendszernek, ahol tetszóleges mennyiségű teszt metódust adhatunk meg.

  A lista elemeinek a következő felépítésnek kell megfelelniük:

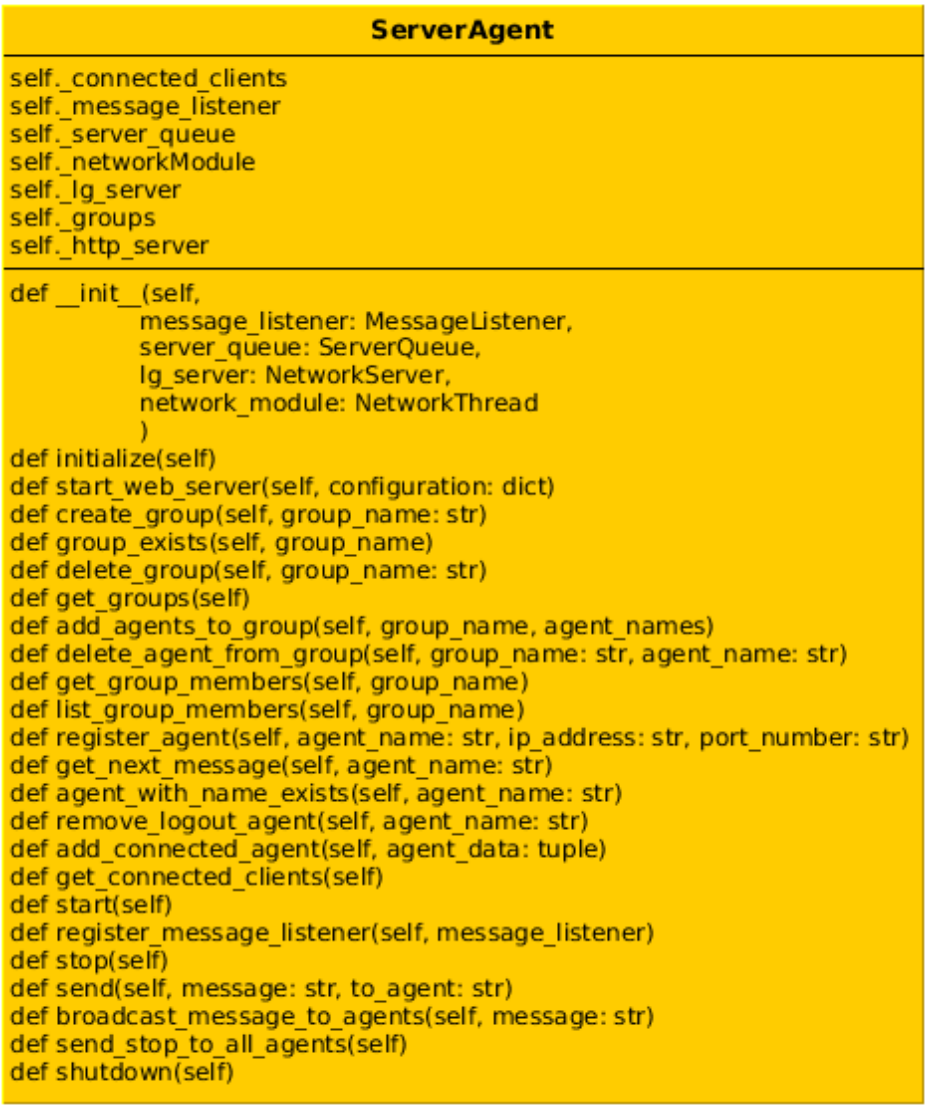
  - "method_name": Ez a paraméter a kliensben definiált tesztmetódus nevét tartalmazza ami majd le fog futni.
  - "method_execution_number": A tesztmetódus hány alkalommal fusson le.
  - "method_params": Itt szintén egy listát kell megadnunk, ami a meghívott teszt metódusnak megfelelően megegyező mennyiségű paramétert tartalmaz string-ek formájában.

### 5.3.2. Agents - agents.py

Ebben a modulban két féle osztály van implementálva.

A rendszer indításakor parancssori aromentumként kapja meg a kontroller, hogy az aktuális indítás kliens, vagy szerver módban történt, és ennek megfelelően vagy ServerAgent vagy ClientAgent ként fog futni.

A ServerAgent álal megvalósított metódusok, az alábbi osztálydiagramon láthatóak: (5.)




5. ábra: ServerAgent

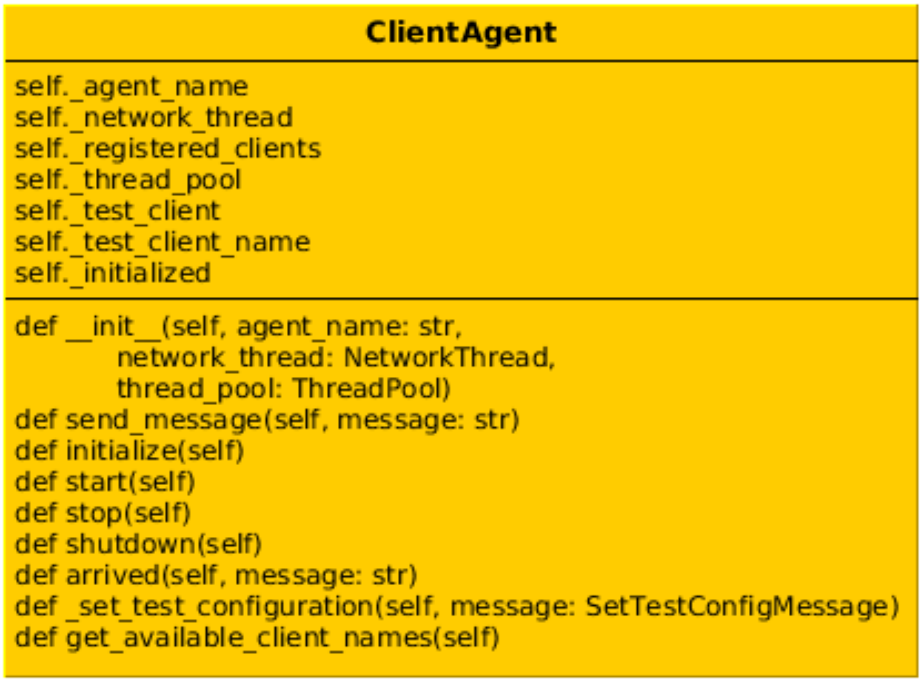
Ebből az osztályból létrejött objektum az alábbi adattagokat tartalmazza:

- Rendelkezik a self._connected_clients listával, ami a csatlakozott kliensekről tárol (név , IP cím) párokat.
- Rendelkezik egy self._message_listener adattaggal ami egy MessageListener objektumot tárol.
- A self._server_queue adattagba egy ServerQueue objektumot tárol, ami olyan üzeneteket tartalmaz amiket mindenképp ki kell küldeni a csatlakozott klienseknek.
- A self._networkModule adattag egy NetworkThread-et vesz át.
- A self._lg_server adattag egy NetworkServer objektumot vesz át.
- A self._groups egy lista, ami a futás közben készített csoportokat tartalmazza.
- A self._http_server pedig egy a későbbiekben esetleges webes felülethez szükséges szerver objektumot tartalmazza

A ServerAgent osztály felelős azért, hogy biztosítsa a kliensek koordinálását, azokról információt szolgáltasson ha szükség van rá. Képes csoportokba szervezni a klienseket, a csoportokat módosítani/törölni.

A másik osztály amely itt kapott helyet az a ClientAgent, aminek az osztálydiagramja az alábbi ábrán látható (6.)

- Rendelkezik egy saját névvel, ez egy egyszerű string típus
- Rendelkezik egy self._network_thread adattaggal ami egy NetworkThread-et tárol
- Egy listában tárolja a regisztrált klienseket self._registered_clients = []
- A self._thread_pool adattagban egy ThreadPool objektumot tárol
- A self._test_client és a self._test_client_name adattagok egy kliens példányról tárolnak információt
- A self._initialized adattag jelzi, hogy a teszt konfiguráció be lett-e állítva Ez 1 értéket vesz fel ha a kliens elindult, különben 0.

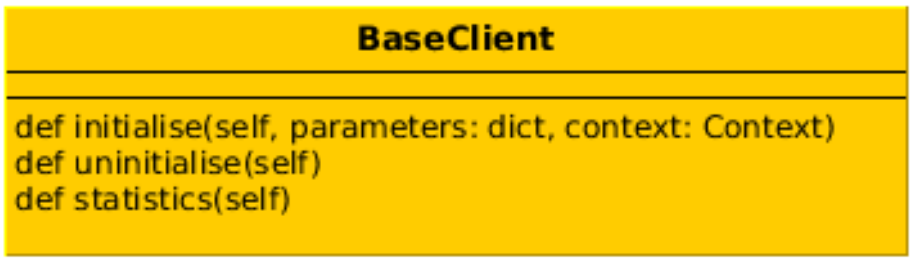



6. ábra: ClientAgent

### 5.3.3. Clients - /clients

A rendszerben lehetőségünk van definiálni számtalan különböző klienst a rendszer clients nevű mappája alá.

Az itt létrehozott kliensek mindegyikének implementálnia kell az alábbi diagramon látható interfészt: (7.)




7. ábra: BaseClient

A kliensek felépítésének egyszerű féldáját, egy az alább látható példa kód segítségével szeretném bemutatni:

```
@implements(BaseClient)
class ExampleClient:
    def __init__(self):
        pass

    @test_method
    def my_test_method(self):
        print("my_test_method␣has␣been␣called")

    @test_method
    def my_test_method_with_one_parameter(self, str_param: str):
        print("my_test_method_with_one_parameter␣has␣been␣called↩
            ")

    @test_method
    def my_test_method_with_two_parameters(self, str_param: str,↩
        str_param2: str):
        print("my_test_method_with_two_parameters␣has␣been␣↩
            called")

    def initialise(self, parameters: dict, context: Context):
        print('Initialized␣with␣parameters:', parameters)

    def uninitialise(self):
        pass

    def statistics(self) -> dict:
        return {'stat_param1': randint(0, 234),
                'stat_param2': randint(0, 3000),
                'stat_param3': randint(0, 300),
                'stat_param4': randint(0, 35),
                'stat_param5': randint(0, 799)
               }
```

Láthatjuk, hogy a kódrészlet elején a @implement(BaseClient) sorral jelüljük, hogy a BaseClient interfészt fogjuk most mevalósítani.

Az interfész initialise, uninitialise és statistic metódusának mindenképp szerepelnie kell saját kliens osztályunkban is, viszont azok tartalma és képességeik csak attól függenek, hogy mi mit teszünk beléjük.

Jelenleg a statisztikát szolgáltató metódus is csak véletlenszerű adatokból felépített eredménnyel tér vissza.

Ahogy ebben az ExampleClient osztályban is látható, a @test_method jelzéssel felvihetünk további metódusokat, amiket az osztályhoz tartozó már korábban részletezett konfigurációs fájlban tetszés szerint paraméterezhetünk és indíthatunk akárhány alkalommal.

Az itt definiált klienseket a keretrendszer konfigurációs fájljába is be kell kötnünk, hogy elérhetővé váljanak futás közben.

Ez a konfigurációs fájl a virtuális gép /home/gabor/load_generator/rhino/ elérési út alatt található rhino.config fájl.

Egy lehetséges példa a rhino.config fájl tartalmára:

```
{
    "log_level" : 2,
    "log_file": "rhino.log",
    "web_server": "True",
    "web_server_listen_port":8080,
    "web_directory":"www",
    "clients": [
        "rhino.clients.exampleclient.ExampleClient",
    ]
}
```

Jelen esetben az alábbi attributumok lényegesek a mi szempontunkból:

- "log_level": Ez a paraméter felelős a rendszer futás közbeni logolás mértékének beállításáért
- "log_file": Ezzel a paraméterrel egy string segítségével adhatjuk meg, hogy milyen elérési úton illetve milyen nevű fájlba mentődjenek a log üzenetek.

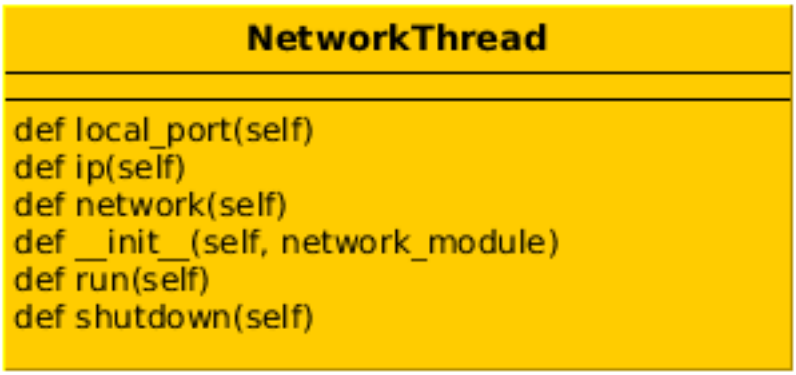
- "cliens": Az itt látható lista az, amelybe szükséges az általunk definiált kliensek bekötése.

  Példánk alapján ez a bejegyzés azt jelenti, hogy a rhino/clients/ mappában az exampleclient.py kiterjesztésű fájlban az ExampleClient osztályt szeretnénk elérni.

### 5.3.4. Network

A network modul alatt az alábbi osztályok lettek implementálva, melyek működése asszinkron módon történik, a NetworkThread osztály kivételével.

- NetworkThread (8.)



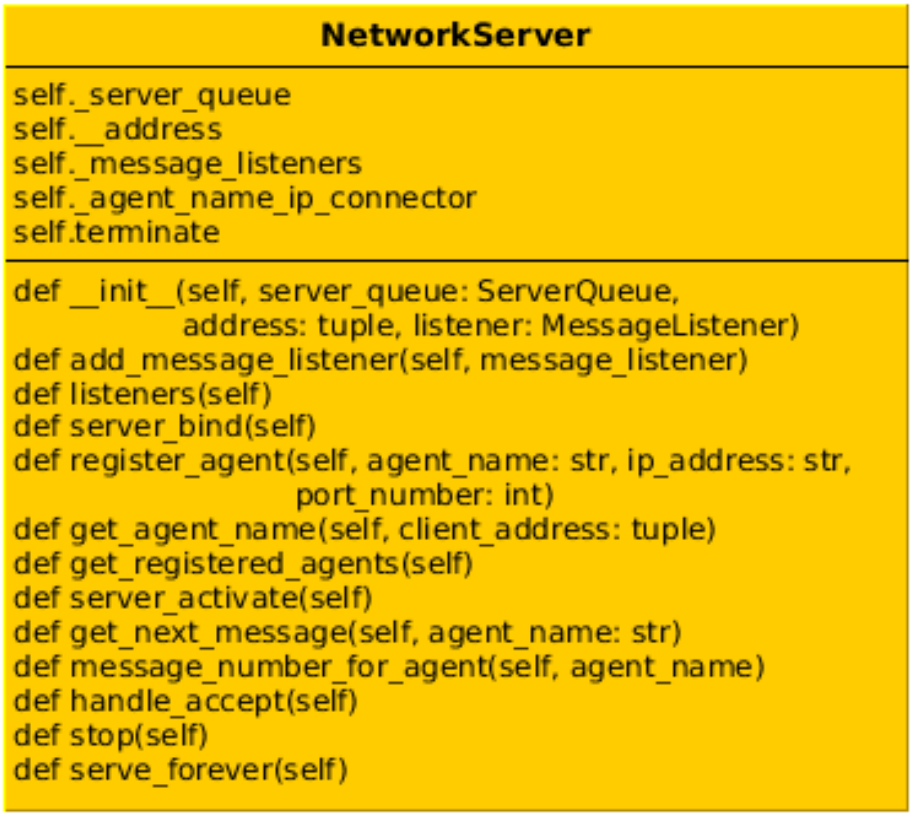

8. ábra: NetworkThread

A NetworkThread osztály felelős a hálózati szálak kezeléséért.

Indíthat, megállíthat forgalmakat.

Igény esetén képes visszatérni ip címmel és port számmal.

- NetworkServer (9.)



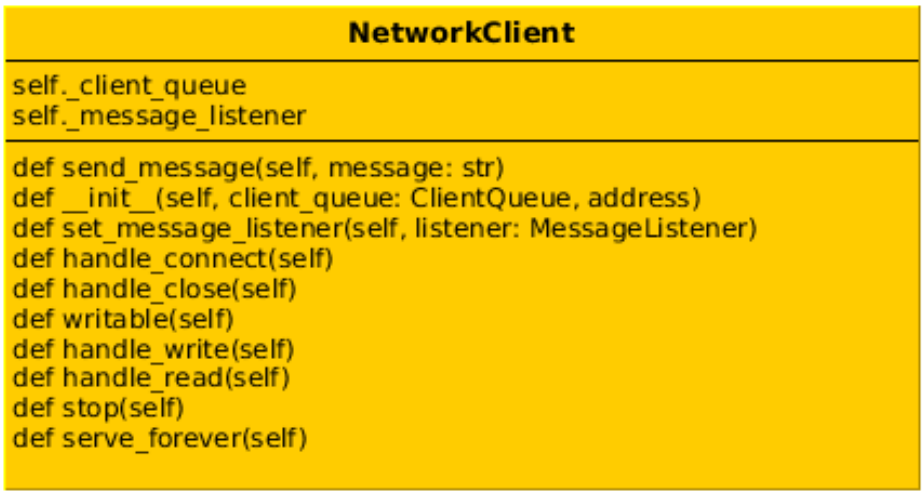

9. ábra: NetworkServer

Az osztály adattagjai:

- self._server_queue adattag egy ServerQueue objektumot vesz át.
- self.__address ez az adattag (ip, port) párokat tartalmaz.
- self._message_listeners adattag egy lista, amelybe MessageListener objektumokat tárolunk.
- self._agent_name_ip_connector egy dict adattag, amiben kulcsként (ip_address, port_number) párokat és a hozzá tartozó agentek nevét tároljuk.
- self.terminate logikai változó.

A NetworkServer osztály felelős hálózati kommunikáció lebonyolításáért. A listenerek kezeléséért, a szerver bind-olásáért, képes klienseket regisztrálni, róluk információt kérni. Üzeneteket kezel, szerverek indítását/megállítását végzi. Szerver

- NetworkClient (10.)



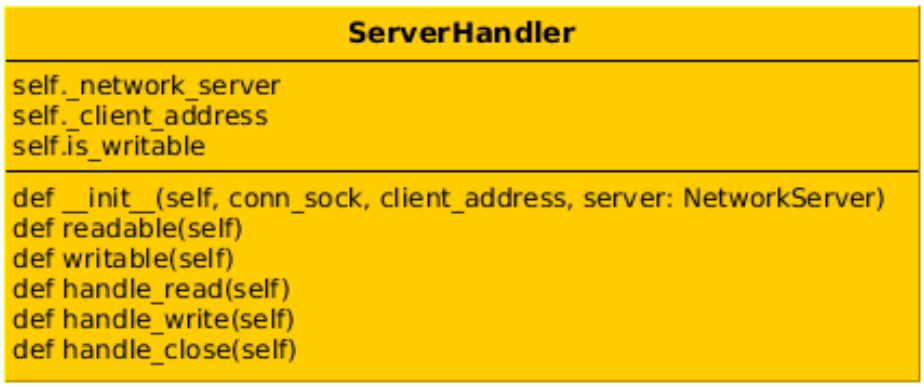

10. ábra: NetworkClient

Az osztály adattagjai:

  - self._client_queue adattag egy ClientQueue objektumot kap, ami az üzenetek kezelését végzi.
  - self._message_listener egy MessageListener objektumot tartalmaz.

A NetworkClient osztály feladata az üzenetek kezelése, fogadása a kliensek felöl illetve továbbítása azok felé.

- ServerHandler (11.)




11. ábra: ServerHandler

Az osztály adattagjai:

- self._network_server ez az adattag egy NetworkServer objektumot tartalmaz.
- self._client_address kliens címét tároló addattag.
- self.is_writeable logikai változó.

A ServerHandler osztály felelős a csatlakozott kliensek kapcsolatának kezeléséért.

### 5.3.5. Pool

Ebben a modulban az alábbi osztályok lettek implementálva:

- ThreadPoolInterface

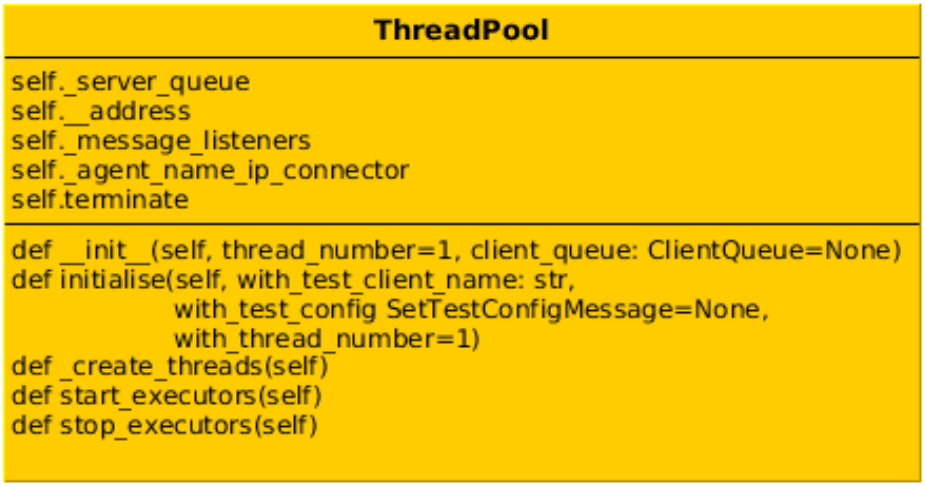
```
@interface
class ThreadPoolInterface:

    def initialise(self, with_test_client_name: str, ↩
        with_test_config: SetTestConfigMessage=None, ↩
        with_thread_number=1):
         pass

    def start_executors(self):
         pass

    def stop_executors(self):
         pass
```

Ez az osztály egy alap interfészt biztosít, amit a későbbiekben a ThreadPool osztálony belül implementálásra került.

- ThreadPool (12.)

Az osztály adattagjai:

- self._thread_number ez az adattag jelenti, hogy hány szálat fog létrehozni a rendszer.
- self._threads az elkészült szálakat ebben a listában tároljuk.




12. ábra: ThreadPool

  - self._running logikai változó, a futást jelzi.
  - self._test_config ez az adattag tartalmazza az inicializálás során átvett konfigurációt.
  - self._test_client_name ez az adattag tartalmazza az inicializálás során átvett teszt kliens nevét.
  - self._context adattag egy Context(client_queue) típusú objektumot tartalmaz ami az üzenetküldésért és objektumok regisztrálásáért felel.

  Ez az osztály felelős a szálak kezeléséért.

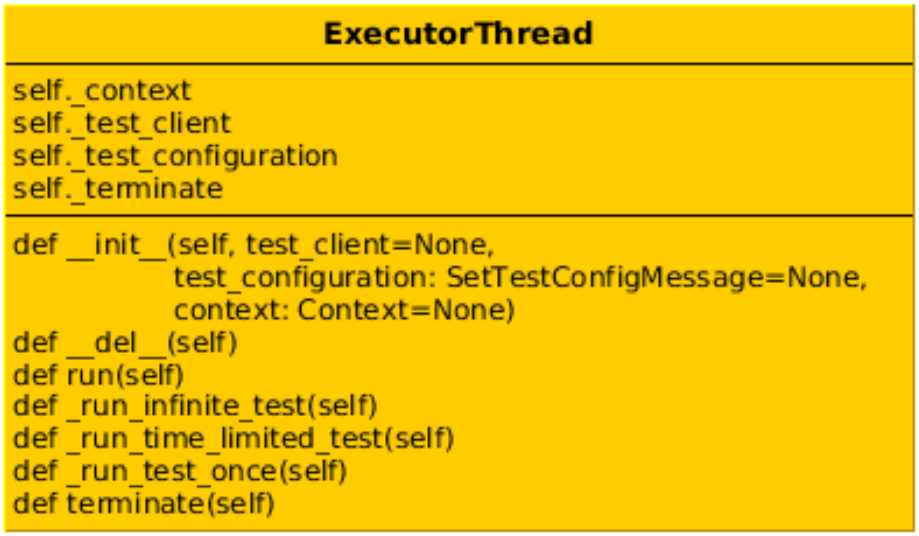
  A konfigurációnak megfelelően létrehozza az ott megadott mennyiségű szálat, amit majd az ExecutorThread osztálynak átad.

  Ezen osztályon keresztül történik a szálakhoz tartozó kliensek feladatainak indítása és megállítása.

- ExecutorThread

  Az osztály adattagjai:

  - self._context Context típusú objektumot tartalmazó adattag.
  - self._test_client a konfigurációból átvett kliens nevét tartalmazza.
  - self._test_configuration a beállított teszt konfigurációt tartalmazza.
  - self._terminate logikai változó.




13. ábra: ExecutorThread

Az ExecutorThread objektumok feladata, hogy a konfigurációban megadott módon és a paramétereknek megfelelően futtassa a teszt kliensek metódusait.

A létrehozott szálak indítása nem szekvenciális módon történik. Ha legalább 2 szálat létrehoztunk és kiadjuk a parancsot ezek elindítására akkor az összes szál párhuzamos módon egyszerre fog elindulni.

# 6. Felhasználói dokumentáció

A következő fejezetben szeretném bemutatni az elkészült keretrendszer és a már meglévő komponensek együttes működését és használatát egy leegyszerüsített rendszeren.

A dolgozatom mellékleteként csatolt virtuális gép tartalma:

- A rendszer alapja egy már korábban említett Linux Mint 17-es operációs rendszer
- Telepített publikus verziójú Zorp tűzfal, bekonfigurálva
- A fejlesztett keretrendszer az aktuális állapotában
- Lefordított TcpForwarder, elkészített konfigurációs fájllal

## 6.1. Rendszer követelmények

A rendszer indításához az alábbi programot szükséges feltelepíteni egy számítógépre : [6]

A telepítés után a .vbox kiterjesztésű fájl megnyitásával elindíthatjuk a a virtuális gépet. Arra figyeljünk, hogy a virtuális gépnek szüksége van egy processzor magra, 2GB memóriára illetve XGB háttértárra.

Lehetőleg próbáljuk meg olyan számítógépen elindítani a rendszert ami rendelkezik legalább 2 fizikai maggal rendelkező CPU-val, illetve mindenképp több mint 4GB memóriával rendelkezik(6-8GB ajánlott, a számítógép operációs rendszerének függvényében) a zökkenőmentes futtatás érdekében.

A rendszerben használatos felhasználó név "gabor", a jelszó pedig egy darab "a" betű. Ezek segítségével tudunk majd root jogot szerezni, illetve belépni a rendszerbe a tesztek során.

## 6.2. Használat és a működés bemutatása

A következőkben már egy valósnak mondható forgatókönyv alapján szeretném bemutatni a rendszer működését.

Ahogy a keretrendszer fejlődött, elkészült hozzá egy telnet kapcsolaton alapuló kliens típus, ami az előre elkészített konfigurációs fájlokkal képes automatikus módon a megadott célgépen egy telnet kapcsolatot létesíteni, ezen a kapcsolaton keresztül néhány alap műveletet végrehajtani és ezt közben a TcpForwarder segítségével rogzíteni, illetve visszajátszani.

Ezen konfigurációs fájlok a configurations mappában találhatóak.

Név szerint:

- telnet_parallel.config: Ez a konfiguráció a "test_parallel_telnet" nevű teszt metódust fogja használni, 10 klienst fog elindítani, illetve a tesztmetódust minden egyes kliens 5 alkalommal fogja lefuttatni.
- telnet_record.config: Ez a konfiguráció 1 telnet klienst fog létrehozni illetve egy alkalommal fogja lefuttatni a "test_record" metódust.
- telnet_replay.config: Az utolsó konfiguráció pedig a már előre rögzített "example_connection_for_replay" mappa alatt tárolt adatokat fogja 30 alkalommal

visszajátszani és validálni.

Példámban a "telnet_parallel.config"-ot fogom használni, mert jelenleg ezzekkel a beállításokkal képes a rendszer a legnagyobb terhelést kifejteni, amit ha egy rendszermonitort(System Monitor) elindítunk miközben futtatjuk a klienseket ez jól láthatóvá is válik a processzorhasználaton.

1. Miután a rendszer elindult és betöltött indítsunk el egy "Terminator"-t. Baloldalt felül található piros négyzetekkel kirakott ikon, vagy az asztalon található indítóikon segítségével.

   Ez egy egyszerű karakteres felületű terminál, mint bármelyik Linux operációs rendszeren, csak van néhány plussz hasznos funkciója ami segítségünkre lehet.

   A könnyebb kezelhetőség érdekében osszuk fel 4 részre a Terminátor ablakát. Ezt megtehetjük az alábbi gyorsbillentyűkkel:

   - CTRL + SHIFT + E : függőlegesen osztja ketté az ablakot
   - CTRL + SHIFT + O : vízszintesen osztja ketté az ablakot

   Vagy a Terminátor ablakán jobb egérgombot megnyomva kiválasztjuk a "Split Horizontally/Vertically" opciókat. Ez után ha szeretnénk a könnyebb áttekinthetőség érdekében az egyes kis ablakokat át is tudjuk nevezni.

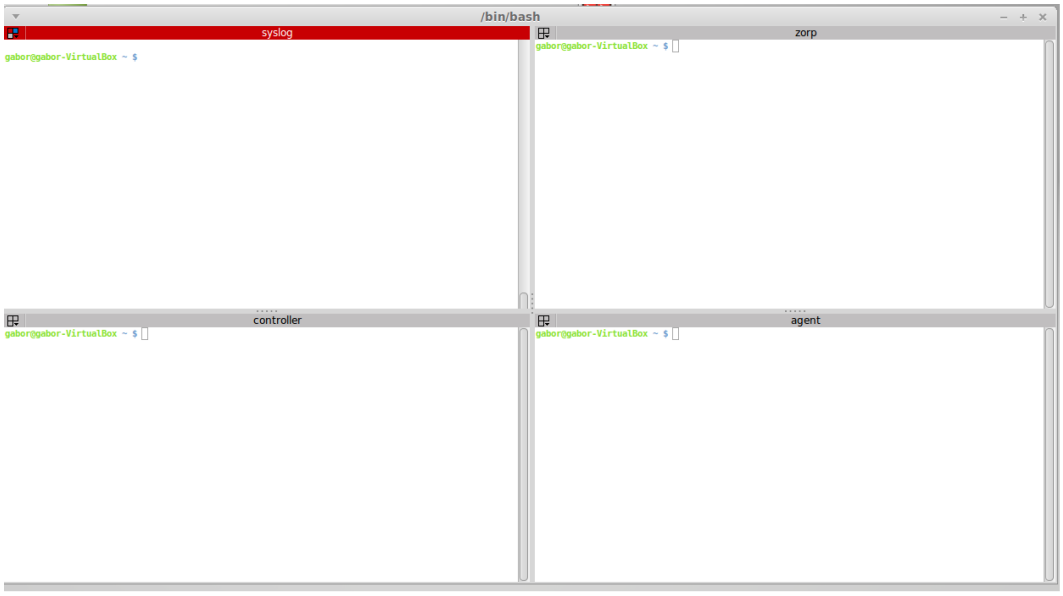
   Ezek után valami hasonlót kell lássunk:

   Ez a nézet a későbbiekben segítségünkre lesz, hogy egyszerűbben átlássuk, az éppen futó folyamatokat és azok eredményeit.

2. Következő lépésként ellenőrízzük, hogy az előre beállított hálózati interfész megfelelő ip címmel rendelkezik-e.

   Ezt a következő parancs kiadásával tudjuk ellenőrizni, aminek hatására ehhez hasonló kimenetet kell kapnunk:

```
gabor@gabor-VirtualBox ~ $ ifconfig
eth0      Link encap:Ethernet  HWaddr 08:00:27:48:66:19
          inet addr:192.168.0.100  Bcast:192.168.0.255 ↩
             Mask:255.255.255.0
```




14. ábra: Terminator

```
          inet6 addr: fe80::a00:27ff:fe48:6619/64 Scope:↩
              Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric↩
              :1
          RX packets:76401 errors:0 dropped:0 overruns:0 ↩
              frame:0
          TX packets:25636 errors:0 dropped:0 overruns:0 ↩
              carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:89970010 (89.9 MB)  TX bytes:2145449 ↩
              (2.1 MB)

eth1      Link encap:Ethernet  HWaddr 08:00:27:a0:43:ee
          inet addr:1.1.1.1  Bcast:1.255.255.255  Mask↩
              :255.0.0.0
          inet6 addr: fe80::a00:27ff:fea0:43ee/64 Scope:↩
              Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric↩
              :1
          RX packets:0 errors:0 dropped:0 overruns:0 frame↩
              :0
          TX packets:248 errors:0 dropped:0 overruns:0 ↩
              carrier:0
          collisions:0 txqueuelen:1000
```

```
          RX bytes:0 (0.0 B)  TX bytes:51681 (51.6 KB)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:621 errors:0 dropped:0 overruns:0 ↩
              frame:0
          TX packets:621 errors:0 dropped:0 overruns:0 ↩
              carrier:0
          collisions:0 txqueuelen:0
          RX bytes:59730 (59.7 KB)  TX bytes:59730 (59.7 KB↩
              )
```

Számunkra a kérdéses csatlakozó az "eth1" jelölésű. Ennek az "inet addr" címének 1.1.1.1-nek kell lennie.

3. Következő lépésként az egyik terminál ablakban adjuk ki a következő parancsot:

```
gabor@gabor-VirtualBox ~ $ tail -f /var/log/syslog
```

Ennek eredménye ként ebben az ablakban folyamatosan látni foglyuk, hogy a rendszer milyen üzeneteket logol a különbözé programoktól, futó szolgáltatásoktól.

4. Kattintsunk át egy másik ablakba, az előzőt hagyjuk folyamatosan futni.

Következő lépésünkhöz szükség lesz root jogosultságokat szereznünk. Ezt a következő parancs használatával érhetjük el, aminek kiadása után a rendszer kérni fogja a jelszavunkat ami ahogy már korábban említettem egy darab "a" betű.

Az alábbi kimenethez hasonlónak kell látszania:

```
gabor@gabor-VirtualBox ~ $ su
Password:
gabor-VirtualBox gabor #
```

A következő paranccsal a telepített Zorp mappájába ugrunk és ellenőrizzük a mappa tartalmát:

```
gabor-VirtualBox gabor # cd /usr/local/etc/zorp/
gabor-VirtualBox zorp # ls -l
total 24
-rw-r--r-- 1 root root  618 nov   14 16:45 instances.conf
-rw-r--r-- 1 root root  547 márc  26  2015 instances.conf.↩
    sample
-rw-r--r-- 1 root root 3710 márc  26  2015 policy.py.sample
-rw-r--r-- 1 root root  987 dec    1 11:10 policy-telnet.py
drwxr-xr-x 2 root root 4096 márc  26  2015 urlfilter
-rw-r--r-- 1 root root 1425 márc  26  2015 zorpctl.conf.↩
    sample
```

Amire nekünk most szükségünk van az az "instances.conf" és a "policy-telnet.py" fájlok.

Az "instances.conf" fájl tartalma:

```
################################################
##
## Copyright (c) 2000-2001 BalaBit IT Ltd, Budapest, ↩
    Hungary
## All rights reserved.
##
################################################
#
# This file lists the Zorp instances you want to run.
#
# The instance name and arguments _must_ be separated by ↩
    spaces instead
# of tabs! Otherwise zorpctl will stop working.

#instance  arguments

zorp_telnet --verbose=10 --policy /usr/local/etc/zorp/↩
    policy-telnet.py
```

Ebben a fájlban tudjuk paraméterekkel ellátni a Zorp futó service-ét.

Jelen esetben a logolást a maximális 10-es értékre állítottuk a "–verbose=" kapcsolóval. A "–policy" kapcsoló után meg kell adnunk egy elérési utat, ami

a saját policy fájlunkhoz vezet, amit a Zorp majd használni fog.

A policy-telnet.py fájl tartalma a következő:

```
# policy.py showing how to use audit policies with ↩
    startAudit

## Run
# $ telnet localhost -p 2222

from Zorp.Core import *
from Zorp.Telnet import *

config.options.kzorp_enabled = False

InetZone("intnet", "0.0.0.0/0")

def zorp_telnet():
    Service("telnet", TelnetProxy, router=DirectedRouter(↩
        SockAddrInet("1.1.1.1", 2210), forge_addr=FALSE))
    Dispatcher(transparent=FALSE, bindto=DBIface(iface="↩
        eth1", port=2222, protocol=ZD_PROTO_TCP), service="↩
        telnet")
```

Ebben a fájlban definiáljuk a saját "zorp_telnet()" metódusunkat, ami alapján a Zorp működni fog.

Jelen helyzetben a rendszer az 1.1.1.1 ip címen és a 2210-es porton fogja továbbítani az adatokat. Illetve az eth1-es hálózati interfészen a 2222-es porton pedig várja a csomagokat.

Ha ezek a fájlok rendben vannak, akkor a következő parancs kiadásával elindíthatjuk a Zorp-ot:

```
gabor-VirtualBox zorp # zorpctl start
Starting Zorp Firewall Suite: zorp_telnet#0
gabor-VirtualBox zorp #
```

Közben láthattuk a másik ablakban ahol a log változásait figyeljük, hogy a Zorp service elindult.

Ha esetleg módosítani szeretnénk a policy vagy az instance fájlon, akkor a módosítások után az alábbi paranccsal újraindíthatjuk a szolgáltatást, hogy életbe léphessenek a módosításaink:

```
gabor-VirtualBox zorp # zorpctl restart
```

Leállítása pedig a stop parancs segítségével történik:

```
gabor-VirtualBox zorp # zorpctl stop
```

5. A következő lépés a keretrendszer elindítása lesz.

   Ehhez a harmadik terminál ablakban menjünk bele a rhino mappába.

   Ez az alábbi parancs kiadásával egyszerűen meg tudjuk tenni, és ha listázzuk a mappa tartalmát akkor ezt a kimenetet kell lássuk:

```
gabor@gabor-VirtualBox ~ $ cd Desktop/load_generator/rhino/
gabor@gabor-VirtualBox ~/Desktop/load_generator/rhino $ ls -l
total 2248
-rwxr-xr-x  1 gabor gabor     117 nov   15 16:00 agent.bash
-rw-r--r--  1 gabor gabor     150 nov   15 15:47 command.test
drwxr-xr-x  2 gabor gabor    4096 dec    1 14:53 configurations
-rwxr-xr-x  1 gabor gabor      77 nov   15 15:55 controller.bash
drwxr-xr-x  2 gabor gabor    4096 nov   15 15:47 doc
drwxr-xr-x  2 gabor gabor    4096 dec    1 14:13 example_connection_for_replay
-rw-r--r--  1 gabor gabor     488 nov   15 15:47 MANIFEST
-rw-r--r--  1 gabor gabor      70 nov   15 15:47 MANIFEST.in
-rw-r--r--  1 gabor gabor 1525093 nov   18 11:52 out.ogv
-rw-r--r--  1 gabor gabor     767 dec    1 14:52 README.txt
drwxr-xr-x 12 gabor gabor    4096 dec    7 19:12 rhino
-rw-r--r--  1 gabor gabor     483 nov   16 13:29 rhino.config
-rw-r--r--  1 gabor gabor  709073 dec    1 14:54 rhino.log
-rw-r--r--  1 gabor gabor    1258 dec    1 14:52 setup.py
```

```
drwxr-xr-x  2 gabor gabor     4096 nov    15 15:47 ↩
    specification
drwxr-xr-x  5 gabor gabor     4096 dec     1 14:53 www
gabor@gabor-VirtualBox ~/Desktop/load_generator/rhino $
```

Ugyanezt a műveletet ismételjük meg az utolsó terminál ablakban is.

Az egyik ablakban el tudjuk majd indítani a keretrendszerhez tartozó controller scriptet, a másik ablakon pedig az agent scriptet.

Néhány mondatban ismertetném ezen scriptek tartalmát, és feladatukat.

controller.bash script tartalma:

```
#!/bin/bash
python3.4 -m rhino  --mode server -lp 8888 --config rhino.↩
    config
```

agent.bash:

```
#!/bin/bash
python3.4 -m rhino --name testagent --mode client -lp 8889 ↩
    -ci 127.0.0.1  -cp 8888 --config rhino.config
```

A "controller.bash" script feladata egy szerver indítása, amin keresztül irányítani tudjuk a klienseket.

Az "agent.bash" script felelős egy teszt kliens elindításáért.

Magyarázat a scriptekben használt kapcsolókhoz:

- –config: A betöltött konfigurációs fájl elérési útja.
- –mode: Itt kell megadnunk, hogy a rendszert szerver vagy kliens módban akarjuk elindítani [server|client].
- –name: Ezzel a kapcsolóval adhatjuk meg az elindított kliensünk nevét. Ha a rendszer kliens módban van elindítva akkor ezt a paramétert mindenképp meg kell adnunk.
- -lp: Ez a kapcsoló jelenti, hogy milyen helyi portot használjon a rendszer.
- -ci: Ezt a kapcsolót kliens módban kell megadnunk, azt jelenti, hogy a kontroller milyen ip-címen dolgozik.

- -cp: Ez a kapcsoló szintén a klienshez tartozik. Azt a portot kell itt megadnunk ahol a kontroller figyel.

6. Miután megismertük a két script feladatát és tartalmát, az egyik üres terminál ablakban elindíthatjuk a "controller.bash", a másik ablakban pedig az "agent.bash" scripteket.

   Az "agent.bash" script elidítása után nem jelenik meg semmi a kimeneten, csak ha a kontrollerrel valamilyen feladatot hajtunk végre a kliensekkel.

   A kontroller indításához adjuk ki az alábbi parancsot, majd ennek a képernyőnek kell megjelenni a terminálunkban:

```
gabor@gabor-VirtualBox ~/Desktop/load_generator/rhino $ ./↩
    controller.bash
(Lg) >
```

   Ez a felület jelenti azt, hogy a program elindult.

   Itt kiadva a "help" parancsot, a rendszer segítség képp kilistázza, hogy milyen parancsokkal rendelkezik:

```
(Lg) > help

Documented commands (type help <topic>):
========================================
add_agents_to_group     help                    start
create_group            list_group_members      ↩
    start_statistics
delete_group            list_groups             stop
get_available_clients   load_file               ↩
    stop_statistics
get_client_info         set_group_test_config
get_connected_agents    set_test_config

Undocumented commands:
======================
exit   version

(Lg) >
```

Ahogy a kimeneten is látszik a rendszerben vannak dokumentációval rendelkező parancsok, amiket a leírását a "help <parancs>" segítségével megkaphatunk.

Például ha az "add_atents_to_group" parancsra vagyuk kiváncsiak akkor az alábbi parancs kiadása után ezt a kimenetet kapjuk:

```
(Lg) > help add_agents_to_group
Add agents to group. add_agents_to_group <group name> <↩
   agents separated with comma>
(Lg) >
```

A parancsok használatának megkönnyítése érdekében a rendszer rendelkezik parancs kiegészítéssel a TAB billentyű megnyomásával, akárcsak egy hagyományos terminál esetében.

Jelen környezetben egy teszt klienst fogunk indítani, amivel szemléltetni szeretném a keretrendszer működését és az alábbi parancsok használatát:

- get_connected_agents: Ezzel a paranccsal a kontrollerhez csatlakozott agent-eket tudjuk kilistázni
- get_available_clients: Ez a parancs az elérhető előre definiált klienstípusokat tartalmazza amit a csatlakozott agent indítani képes.
- set_test_config: Ennek a parancsnak a segítségével tudjuk megadni egy agent-nek, hogy milyen teszt konfigurációs fájl alapján dolgozzon. 5.3.1
- get_client_info: Ez a parancs az adott teszt agent-höz tartozó kliensekről szolgáltat információt.

A controller és az agent elindítása után a controller terminál ablakjában ellenőrizzük, hogy rendben csatlakozott-e a teszt agent-ünk. Ezt az alábbi paranccsal tudjuk megtenni, illetve ezt a kimenetet kell kapnunk:

```
(Lg) > get_connected_agents
Connected agents
==================
testagent   127.0.0.1
(Lg) >
```

Láthatjuk, hogy rendben elindult a "testagent" nevű agent-ünk, ami a 127.0.0.1-es ip címen dolgozik.

A következő parancs kiadásával láthtjuk, hogy milyen típusú kliensek kezelésére képes a teszt agent-ünk:

```
(Lg) > get_available_clients testagent
(Lg) > Available clients on agent  testagent
================================================
rhino.clients.exampleclient.ExampleClient
rhino.clients.memorycpuclient.MemoryCPUMeasureClient
rhino.clients.sshclient.SSHClient
rhino.clients.telnetclient.TelnetClient


(Lg) >
```

A keretrendszer jelenlegi állapotában az ExampleClient és a TelnetClient van megfelelően implementálva, a többi számunkra most érdektelen.

Ezt követően meg kell adnunk, hogy a "testagent" milyen konfigurációs fájl alapján dolgozzon, amit a következő paranccsal teszünk meg:

```
(Lg) > set_test_config testagent configurations/↩
   telnet_parallel.config
```

Ez után már csak egy dolgot kell tennünk, mégpedig kiadni a start parancsot.

```
(Lg) > start testagent
('testagent', '127.0.0.1')
(Lg) >
```

Ennek eredményeként láthtjuk, hogy abban a terminálablakban ahol a syslog figyelését illetve ahol az "agent.bash" scriptet indítottuk megjelent a kliensek futtatása miatt változó kimenet.

Lefuttatva a get_client_info parancsot az alábbi kimenet fogad minket:

```
(Lg) > get_client_info testagent
(Lg) > <rhino.msg.msgfactory.ClientInfoMessage object at 0↩
   x7f714b484908>
Agent information  testagent
================================================
```

```
Initialized: True
Client name: rhino.clients.telnetclient.TelnetClient
Thread pool size: 10
State: Running
```

Láthatjuk, hogy elkészültek a TelnetClient típusú klienseink, számszerint 10 darab, amik benne vannak a pool-unkban és jelenleg aktívak.

A fenti pontokban vázolt működés ugyanígy használható a többi előre definiált konfigurációs fájlal, illetve igény szerint ezek módisíthatóak is.

A rendszernek még vannak hiányosságai, illetve korlátai amit a következő pontban részletesebben kifejtek.

# 7. Összefoglalás

## 7.1. Az elkészült munka értékelése

Dolgozatom témája egy olyan keretrendszer kialakítása volt, amely a lehető legnagyobb mértékben megkönnyíti a cég által fejlesztett egyes termékek teljesítőképességeinek megismerését és ezekről információt tudjon szolgáltatni.

A fejlesztés során számos kihívással találkoztam, melyek lekűzdése közben rengeteg új ismeretet sajátítottam el.

Jelenleg a keretrendszer rendelkezik hiányosságokkal és ismert hibákkal, amiket még implementálni kell a teljeskörű működés érdekében illetve javítani a zavartalan működésért, de már jól látható, hogy a tervezett funkcionalitásnak képes lesz eleget tenni valós igénybevétel esetén is.

Ezt a már implementált telnet protokolt használó kliens is bizonyítja.

Idő közben a keretrendszer fejlesztéséhez csatlakozott két front-end fejlesztő munkatárs, akik egy webes felületen illetve az azt kiszolgáló webserver létrehozásán dolgoztak.

Ebben a fejlesztésben én nem vettem részt, de munkájuk megtalálható a rendszer mappa struktúrájában.

## 7.2. Tesztek

A keretrendszer egyes komponensei unit tesztek által automatikusan tesztelhetőek, de a fejlesztés során a rendszer működése kézzel folyamatosan end-to-end[9] tesztelve volt.

## 7.3. Ismert hibák

- A kontroller futása közben kiadott egyes parancsok után néha szükség van egy extra "Enter" billentyű leütésére, hogy újabb parancsot adhassunk ki.

  Ilyen parancs például a "get_client_info".

- Javításra szorul a TcpForwarder vezérléséért felelős része a rendszernek az alábbi két ponton:

  - A keretrendszer által végrehajtott rögzítés során nem minden futás alatt kerül rögzítésre megfelelő módon az áthajtott kapcsolat.

    Ez a jelenség a "telnet_record.config" fájl használata közben jelentkezik.

    Ilyenkor helytelen működés esetén a teszt kliens újboli indítására megfelelően végbe megy a folyamat.
  - Visszajátszásnál a több szálú működés jelenleg nem működik megfelelően, mert a TcpForwarder implementációja nem teszi lehetővé ezt a működést.

    E miatt a "telnet_replay.config" fájl használatánál a konfigurációban egyelőre csak 1 kliens szimulálható egyszerre.

    Több kliens esetén a validálás hibásan fog lefutni néhány esetet leszámítva.

  Ezen hibák oka ismert és a továbbiakban a lehető leghamarabb javításra kerülnek, mert jelentős hátrány származik belőlük.

## 7.4. További fejlesztési lehetőségek

A keretrendszer fejlesztése nem ér véget szakdolgozatom befejezésével.

A fejlesztés folytatódni fog, hogy a lehető legnagyobb mértékben ki tudja szolgálni a cég által támasztott végleges követelményeket.

A következő pontokban röviden bemutatom, hogy a rendszer egyes pontjai milyen tovább fejlesztési lehetőségekkel rendelkeznek.

### 7.4.1. Kliensek

A rendszer egyelőre a már említett telnet protokollal dolgozik főként, viszont ezen terület jelentős bővítési lehetőséggel bír.

A cég által fejlesztett termék számos egyéb protokolt támogat (például SSH, RDP, VNC) amikhez szükséges implementálni a megfelelő kliens típusokat, hogy a keretrendszer ezen prokollokon keresztül is képes legyen terhelni a rendszert.

### 7.4.2. TcpForwarder

A már meglévő implementációt a már említett hibák miatt is szükséges átalakítani, hogy a teljesítmény tesztelések során lehetőség nyíljon a párhuzamos futtatásra, így a terhelés nagyságrendekkel növelhető.

A forwarder által használt adatrögzítési metodika áttervezése is elkezdődött, de még nem sikerült teljes mértékben ezt átalakítani.

Ennek eszköze a Google által fejlesztett protocol buffer kiterjesztés lesz. [10]

### 7.4.3. GUI

A későbbiekben a könnyebb használhatóség és felhasználóbarátabb kezelés érdekében szerencsésebb lenne a jelenlegi karakteres felületet elhagyni, és egy megfelelő grafikus felhasználói felületet tervezni és implementálni a keretrendszerhez.

Ennek érdekében kezdődött el a már korábban említett web-es felület fejlesztése is.

### 7.4.4. Statisztikák

Jelenleg a keretrendszer statisztikákat / logolást végző része kezdetleges állapotban van.

Ennek a modulnak a fejlesztése jelentős mennyiségű információt tudna szolgáltatni a terhelt rendszer működésével kapcsolatban.

Például a keretrendszer által kifejtett terhelés milyen mértékben emészti fel az erőforrásokat mondjuk a processzor, memória, háttértár tekintetében.

Ezekhez az információkhoz viszont még szükséges implementálni a megfelelő klienseket.

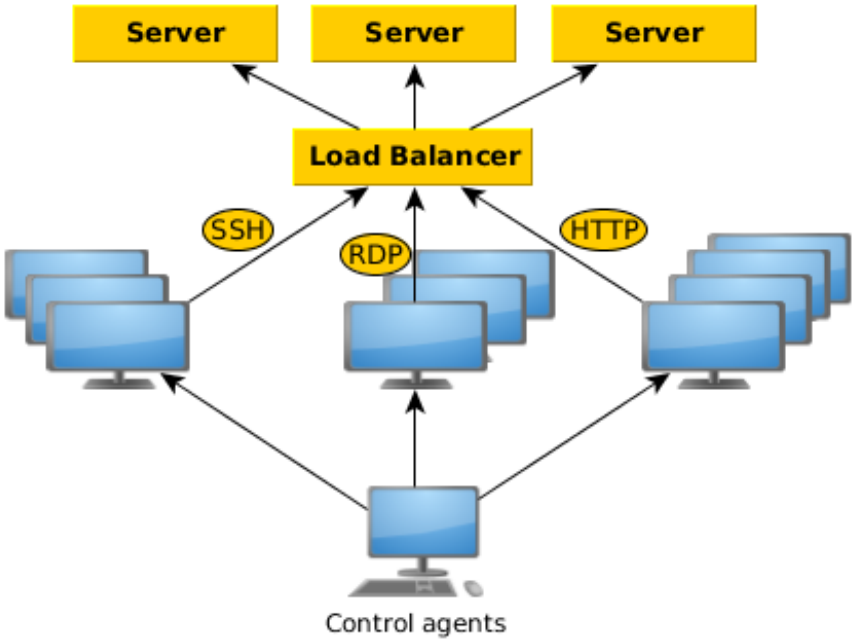
### 7.4.5. Automatikus terjedés

Ezen terület fejlesztésének jelentősége olyan esetekben növekszik meg, mikor a terhelendő szerver(ek) teljesítménye elért egy bizonyos szintet.

Ekkor ha a keretrendszer képes a megadott ip-című host gépekre települni, és azon gépeket felhasználni a terhelés növelésének érdekében akkor ismét egy jelentős feladat végrehajtásán sikerült egyszerűsítenünk.

Ennek egy következő fejlesztési lépcsője lehet, ha a rendszer automatikusan képes lenne feltérképezni az adott hálózatot felhasználható gépek után kutatva.

Az alábbi képen szeretném érzékeltetni, hogy ideális helyzetben a fentebb felsorolt fejlesztések hatására hogyan épülne fel a rendszer egy kiterjettebb hálózat esetén: (15.)




15. ábra: Teljes rendszer

# 8. Irodalomjegyzék

[1] https://www.balabit.com/hu
*BalaBit IT Biztonságtechnikai Kft.*

[2] https://www.balabit.com/hu/network-security/zorp-gpl
*Zorp GPL*

[3] http://www.tcpdump.org/
*TCPDUMP*

[4] https://www.wireshark.org/
*Wireshark hálózati csomagokat megfigyelő szoftver*

[5] https://iperf.fr/
*Iperf*

[6] https://www.virtualbox.org/
*VirtualBox*

[7] http://www.linuxmint.com/release.php?id=22
*Linux Mint 17*

[8] https://www.python.org/
*Python*

[9] http://www.tutorialspoint.com/software_testing_dictionary/end_to_end_testing.htm
*end-to-end teszt leírás*

[10] https://developers.google.com/protocol-buffers/
*Google Protocol Buffer fejlesztői dokumentációja*

# 9. Mellékletek

## 9.1. CD melléklet

A szakdolgozat CD mellékletének könyvtárszerkezete:

/KranitzGabor-TZ906Y-szakdolgozat.pdf

/szakdolgozat-forraskod

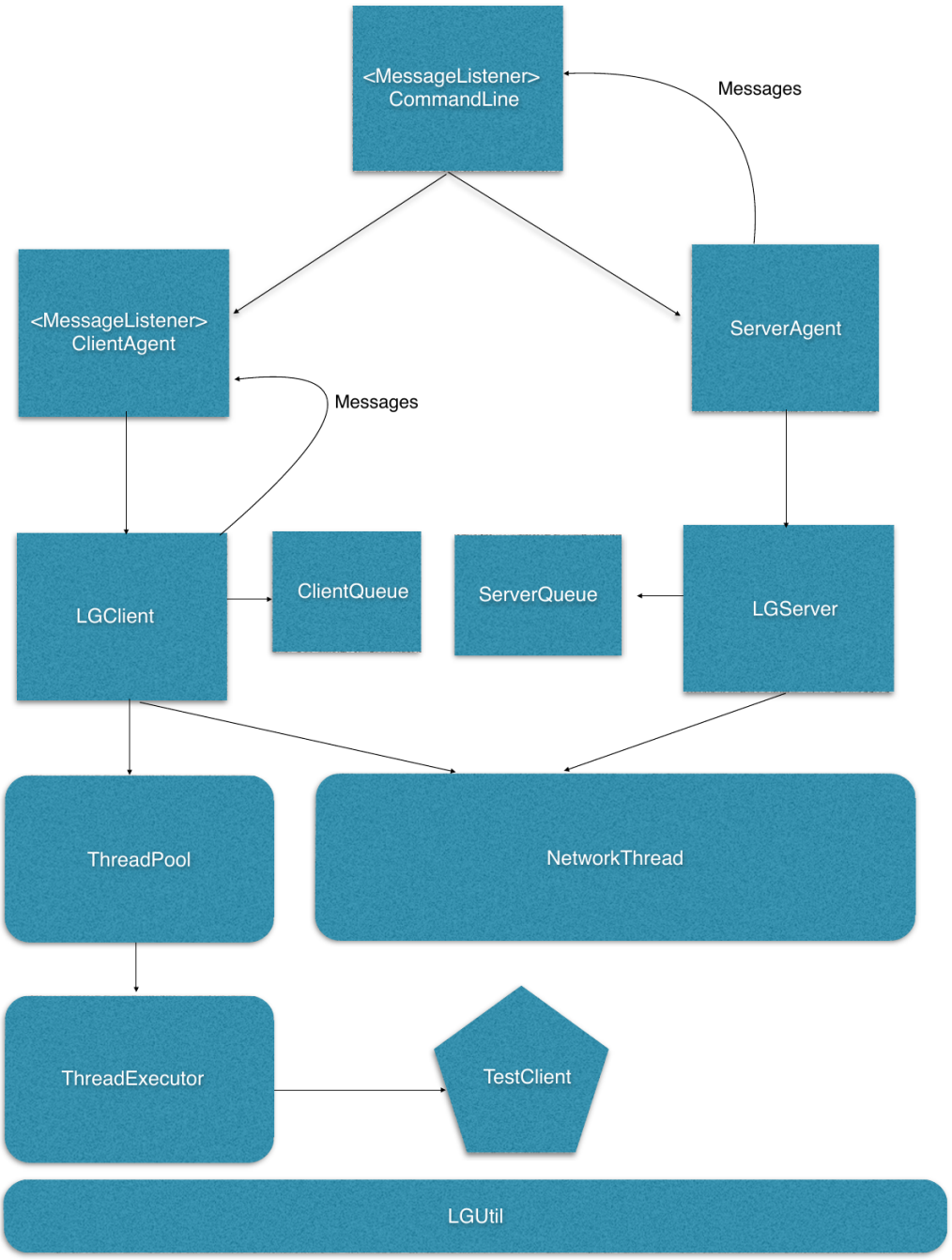
/kepek

/szakdolgozat.tex

/virtualis-gep

/logs

/internetes-hivatkozasok




16. ábra: A keretrendszer felépítése