Eötvös Loránd Tudományegyetem

Informatikai Kar

Média- és Oktatásinformatika Tanszék

PHPVisor

Folyamatkezelő rendszer megvalósítása PHP-ban

Dr. Illés Zoltán
Egyetemi docens
Fischer Zsolt
SLAmetrix Kft.

Mikus Márk István
Programtervező Informatikus BSc

Budapest, 2018

Tartalomjegyzék

1. Bevezetés **3**

1.1. Motiváció 3

1.2. Feladat 4

2. Felhasználói dokumentáció **5**

2.1. Bemutatás 5

2.2. Telepítés 6

2.3. Rendszer követelmények 7

2.3.1. Egyéb függőségek 7

2.4. Használat 7

2.4.1. Server 8

2.4.1.1. Konfiguráció 8

2.4.1.2. Socket konfiguráció 9

2.4.1.3. Folyamat konfiguráció 10

2.4.1.4. Parancssori beállítások 12

2.4.1.5. Futtatás 14

2.4.1.6. Működés 14

2.4.1.7. Hibaüzenetek 16

2.4.2. Client 17

2.4.2.1. Konfiguráció 17

2.4.2.2. Parancssori konfiguráció 18

2.4.2.3. Futtatás 19

2.4.2.4. Működés 19

2.4.2.5. Felület értelmezése 22

2.4.2.6. Hibaüzenetek 24

3. Fejlesztői dokumentáció **27**

3.1. Tervezés 28

3.1.1. Server 29

3.1.1.1. Felhasználói esetek 30
3.1.1.2. Osztályok 31
3.1.1.3. Felhasználói esetek 35
3.1.2. Client 35
3.1.2.1. Osztályok 35
3.1.2.2. Felhasználói esetek 37
3.2. Megvalósítás 38
3.2.1. Server 39
3.2.2. Client 40

4. Tesztelés **41**
4.1. Server 41
4.1.1. Feketedoboz tesztesetek 41
4.1.2. Fehérdoboz tesztesetek 42
4.2. Client 42
4.2.1. Feketedoboz tesztesetek 42
4.2.2. Fehérdoboz tesztesetek 42
4.3. Hatékonyság 43

5. Zárszó **44**

6. Irodalomjegyzék **45**

1. fejezet

Bevezetés

A korszak, amelyben élünk, informatikai rendszerek sokaságával van átszőve. A számítógép feltalálása óta ezek az eszközök egyre inkább létfontosságúvá váltak, mára már nélkülözhetetlenek. Rutinszerű, illetve egyedi folyamatok sokasága jellemzi a ma hálózatait, rendszereit, alrendszereit.

1.1. Motiváció

Korunk szerves részévé vált az internet mind a hétköznapi, mind az ipari életben, így a megfelelő weboldalak, webalkalmazások üzemeltetésére is nagy hangsúly került. Ugyanakkor egy weboldal fenntartása sem feltétlenül merül ki annyiban, hogy egy gépen adott típusú webszervert kell karbantartani. Adatok szolgáltatásához, szinkronizációjához vagy beszerzéséhez szükségesek lehetnek akár olyan folyamatok is, amelyek nem szerves (vagy esetleg publikus) részei az üzemeltetendő alkalmazásnak. A webes világban, ma már újabb és újabb technológiák hódítanak tért maguknak, mint például a Microsoft .NET rendszere vagy az Oracle JAVA-s környezetei. Ugyanakkor a klasszikusnak számító, UNIX operációs rendszeren üzemeltetett szerver, illetve az azon futtatott PHP kombináció még napjainkban is a legjellemzőbb üzemeltetési formák közé tartozik.

A dolgozatomhoz az inspirációt a Supervisor[1] nevű program adta, ami egy Python nyelven implementált folyamatkezelő. Segítségével, előre beállított programok automatikusan újrafuttathatók, bármikor leállíthatók vagy elindíthatók. Ezen felül a rendszer hálózaton keresztül vezérelhető. Továbbá ösztönzött a felismerés, hogy egy PHP-ban implementált hasonló működésű program, nagyban leegyszerűsítené azoknak a helyzetét, akik a fentebb klasszikusnak jelölt környezetben tartják karban az alkalmazásaikat, hiszen nincs szükség újabb függőséget jelentő alkalmazások

beszerzésére, ezzel megkönnyítve az üzemeltetők munkáját.

Dolgozatom tárgya tehát, két PHP-ban megvalósított konzolos alkalmazás, egy szerver-komponens, amely magába foglalja a kommunikációs szerver feladatait, a folyamatok vezérlését, illetve a működés naplózását Továbbá egy kliens-komponens, ami a szerver-komponens felé képes közölni a felhasználó által kívánt, folyamatokat befolyásoló parancsokat.

1.2. Feladat

Egy olyan kliens-szerver architektúrájú rendszer létrehozása a cél, amely a fentebb említett két részre bomlik.

A szerver-komponens egy olyan demonizálható program, amely egy meghatározott konfigurációs állomány feldolgozása után, annak tartalma szerint meghatározott parancsokat futtat, illetve meghatározott kommunikációs csatornákat nyit ki. A parancsokhoz olyan beállítások érhetőek el, mint például: automatikus újraindítás, újraindítási stratégiák megadása, standard output fájlleíró átirányítása naplófájlba (választható naplózási szinttel) vagy standard input-leíróra irányított stream, megadott elérési útvonal alapján. Az adott folyamatokhoz relatív prioritást is megadhatunk, továbbá kategória/csoport cikk-neveket is aggadhatunk rájuk, amelyek segítségével csoport szinten indíthatjuk a folyamatokat vagy állíthatjuk meg azokat. A kommunikációs csatornákért socketek felelnek, így a socket manager funkciók (létrehozás, hallgatózás, lezárás) mellett, meghatározott üzenetek feldolgozását, illetve az ezekre vonatkozó válaszüzenetek előállítását implementáló funkciók is a program részét képezik.

A kliens feldata, hogy szintén socketeken keresztül, csatlakozzon a szerveroldalon megnyitott kommunikációs portra, ahová a saját konfigurációjában meghatározott felhasználónév és jelszó párossal, továbbá a kívánt akcióval (, illetve esetleges paramétereivel) üzenjen a szervernek. A progam lehetőséget ad elkérni a folyamatok státuszát, megállítani adott folyamatokat pid (folyamat azonosító) vagy teljes név alapján, elindítani folyamatokat nevük alapján vagy csoportokat indítani esetleg leállítani csoportnév alapján. A kliens lehetőséget biztosít továbbá tetszőleges (előre konfigurált) szignál küldésére, illetve a folyamat folytatásra késztetésére (szintén szignállal), amennyiben az operációs rendszer a folyamatot STOPPED státuszba kényszeríti.

2. fejezet

Felhasználói dokumentáció

2.1. Bemutatás

A program elsősorban a webalkalmazást üzemeltetőket célozza meg, hiszen ez több olyan szituációval járhat amikor az alkalmazás állapotaitól függetlenül, szeretnénk adatokat teríteni. Esetleg az alkalmazás olyan gépen fut, amelyet hálózati tűzfal véd, és így a megfelelő beavatkozáshoz elegendő a megfelelő, egyeztetett portokat átengedni a tűzfalon.

A program szerver-komponensét kell elhelyezni a szervergépen, ahol szeretnénk a folyamatokat futtatni, majd konfiguráció után elindítani. A kliens-komponens segítségével egy másik gépről rá tudunk csatlakozni a megfelelő portra, illetve a megfelelő információk birtokában (felhasználónév és jelszó páros) vezérlési parancsokat tudunk kommunikálni a szerver-komponens felé, amely a folyamatokat vezérli.

Fő szempont volt az alkalmazás fejlsztése közben a konfigurálhatóság, illetve a kezelhetőség is. Ennek eredményeképpen a két megvalósított alkalmazás egy-egy könnyen kezelhető konzolos program, amelyek parancssor és egyszerű JSON formátumú állományok segítségével is konfigurálhatóak.

2.2. Telepítés

Az alkalmazás különböző könyvtárakra bomlik, amelyek segítik elkülöníteni egymástól a két futtatható komponenst amennyiben szükséges. Mivel a programok PHP-s szkriptek, ezért külön telepítésre nincs szükség amennyiben a rendszer követelmények fejezet alatt taglaltaknak eleget tesz a célkörnyezet. A program futtatásához elegendő a PHP CLI SAPI[2] legalább 7.0-s verziójával, továbbá a megfelelő forrásfájlokkal rendelkezni.

A szerver alkalmazás szükséges forrásai az *app* könyvtár *Server.php* fájlja, illetve az ugyanott található *config* könyvtárban elhelyezett *server.cfg.json* konfigurációs állomány, az *src* könyvtár alatt fellelhető *Autoloader.php*, az *External* könyvtár teljes tartalma, továbbá az *Internal* mappa *Application.php* fájlja, a *Server* alkönyvtár teljes tartalma, valamint az *Options* és *Configuration* almappákban található *AbstractOptions.php* és *AbstractConfiguration.php* fájlok, illetve ezen mappákban található *Server* alkönyvtárak tartalma. (A program mappastruktúrájának első szintjén található *.version* fájl is a szerverhez tartozik, de nem kulcsfontosságú eleme.)

A kliens programhoz szükséges forrás-állományok a fentebb leírtakhoz hasonló módon különíthetőek el, annyi különbséggel, hogy a megfelelő, elkülönülő fájlokat a *Client* nevű alkönyvtárakban találjuk. Így tehát a kliens program forrásai a következőek: az *app* könyvtár *Client.php* fájlja, illetve az ugyanott található *config* könyvtárban elhelyezett *client.cfg.json* konfigurációs állomány, az *src* könyvtár alatt fellelhető *Autoloader.php*, az *External* könyvtár teljes tartalma, továbbá az *Internal* mappa *Application.php* fájlja, a *Client* alkönyvtár teljes tartalma, valamint az *Options* és *Configuration* almappákban található *AbstractOptions.php* és *AbstractConfiguration.php* fájlok, illetve e mappákban található *Client* alkönyvtárak tartalma.

Megjegyzés: A szerver programnak további php konfigurációs függőségei vannak, amelyek Unix környezetben jellemzően alapértelmezetten elérhetőek. Alkalmaz PCNTL[3] valamint POSIX [4] függvényeket is. A kommunikáció socketeken történik, így az ehhez kapcsolódó beállítások is indokolhatnak közbeavatkozást.[5]

2.3. Rendszer követelmények

Mindkét alkalmazás működéséhez a futtató számítógépnek az alábbi követelményeket teljesítenie kell:

Operációs rendszer	Linux, Ubuntu vagy bármely UNIX
Processzor	Az optimális működéshez legalább 2 magos processzor, 3GHz órajelű
Háttértároló	Nem igényel a forrás állományok, illetve a konfigurációs fájloknál nagyobb tárterületet (Szerver esetén, a naplóállományok mérete is beleszámít)
Internet	Internetkapcsolat nem szükséges, ugyanakkor a socketes kommunikáció jellemzően valamely kialakított hálozaton zajlik
Bemenet	Billentyűzet szükséges

2.3.1. Egyéb függőségek

A programok futtatását, a PHP-s konzol-modul segítségével futtatjuk, így szükség van a már fentebb említett PHP CLI 7.0-s verziójára, amely verzió már támogatja a skalár típus-hintek alkalmazását. Az alábbi felsorolás tartalmazza PHP-nak azon beépített moduljait, amelyeket a program használ:

- PHP - POSIX - IEEE 1003.1 (POSIX.1) rendszer interfész. (Windows-ra nem elérhető)
- PHP - PCNTL - Folyamat kezelő kiterjesztés, függősége nincsen csak be kell kapcsolni amennyiben nem elérhető.

2.4. Használat

A programok használata, a konzolos alkalmazásoknál már megszokott módon történik. Lehetőség van bizonyos beállítások, illetve módok ki- és bekapcsolására parancssori argumentumok, kapcsolók segítségével. Amennyiben a rendszerre telepített PHP elérhető a `/usr/bin/php` útvonalon, akkor a programokhoz elegendő az *app* könyvtárban állva kiadni a megelelő parancsot. Szerver esetén: `./Server.php`, kliens esetén: `./Client.php`.
A program működését befolyásoló kapcsolókat, ez után írva, szóközökkel elválasztva adhatjuk meg. A kapcsolók között vannak rövid, illetve hosszabb (nevű) kapcsolók.

Jellemzően a rövid kapcsolók elé - jelet kell írni, míg a hosszúak elé -- jeleket kell megadni. Paraméteres kapcsoló esetén a kapcsoló megadását követően, egy szóköz beírása után adható meg a paraméter értéke. (Például: ./Server.php -n –user testuser) A beállítási lehetőségek hierarchikus viszonyban állnak egymással, ugyanis a parancssori opciók mind megadhatóak a konfigurációs állományon keresztül is. Amennyiben a felhasználó egy olyan parancssori kapcsolót ad meg, amely opció a konfigurációs fájlban is definiált, úgy a kapcsoló beállítása lép életbe. A legtöbb beállítási lehetőség alapértelmezett értékkel is rendelkezik, de vannak olyanok is amelyeket mindenképp fájlból nyer a program. (Pl.: Socket ill. Process konfigurációk specifikus adatai)

2.4.1. Server

2.4.1.1. Konfiguráció

A szerverhez rendelt alapértelmezett konfigurációs állomány a már említett */app/config* könyvtárban található. A *server.cfg.json* fájl egy JSON formátumú állomány, amely tartalmát tekintve egy objecktum, amely kulcs-érték párjai meghatározzák az egyes opciókat. Az opciók sorrendje mindegy, ugyanakkor a példa állomány szintaxisát, és írásmódját (kis- és nagybetűk, szöveges/szám értékek, camelCase, listák, objektumok) egyéni konfigurációs állomány létrehozásakor is szem előtt kell tartani.

Az elérhető opciók listája (pontos írási módja : lehetséges értékei - illetve hatása [alapértelmezett értéke]):

- logChildLogDir : könyvtár elérési útvonala - Az a könyvtár, ahová a megadott folyamatoknak a naplói kerülnek. ["/tmp/PHPVisor/log/child/"]
- logFilePath : fájl elérési útvonala - Az a fájl, amelybe a szerver alkalmazás a naplóbejegyzésit írja. ["/tmp/PHPVisor/log/server.log"]
- logFileMaxBytes : fájlméret - Amennyiben csak szám, úgy byte-okban értendő, különben pedig a megfelelő mértékegységgel ellátott fájlméret, amely a nalófájl maximális méretét jelenti. ["50KB"]
- logMaxNumOfBackups : egész szám - A naplófájlokról készült archív "backup" naplók maximális száma. [5]
- logLevel : debug/notice/warning/error - A naplózási szint. Ha egy rendszer esemény naplózni kívánt bejegyzése nem éri el a naplózási szintet, úgy a bejegzés nem kerül naplózásra. ["debug"]

- printLog : true/false - Amennyiben igaz, és a program előtérben fut, úgy minden naplóbejegyzés amit a program írna, a képernyőre is kiírásara kerül. [false]

- noCleanUp : true/false - Amennyiben true, úgy az induláskor, nem törli a 'logChildLogDir' allatti naplófájlokat. [true]

- noDaemon : true/false - Amennyiben az értéke true, úgy a szerver előtérben fog futni (nem daemonként) [false]

- directory : könyvtár elérési útvonala : Az a könyvtár amelybe megpróbál belelépni a program demonizáláskor (chdir függvénnyel). Ha null van megadva (vagy nincs megadva), akkor a program nem vált könyvtárat. [null]

- pidFile : egy fájl elérési útvonala - Amegadott fájlba, a demonizált folyamat beleírja a saját pid-ját. ["/tmp/PHPVisor/phpvdaemon.pid"]

- user : felhasználónév vagy uid - A folyamat megadott használóként való futtatása. (root-ként való futtatás szükséges lehet ez esetben) [jelenlegi username][1]

- umask : maszk érték (szám) - A demonizálás előírása szerinti végső maszkoláskor, a megadott maszk használata. [0]

2.4.1.2. Socket konfiguráció

Az egyes socketek beállításait is amelyeken a kommunikáció zajlik, a konfigurációs fájlban kell megadni. Ez kötelező mező, a program egy kivétellel jelzi amennyiben, kevesebb mint egy socketet építene fel induláskor. A fájlban a `servers` kulcson jelölt listában definiálhatóak a socketek, egy-egy objektumként. A socketek egyes beállítási lehetőségei a következők:

- protocol : "tcp"/"udp"/"unix" - A socket típusa. (Unix socket esetén, a port beállítás értelmét veszti fájl mivolta miatt)

- host : socket fájl elérési útvonala vagy tcp/udp ip cím vagy hostnév. (Pl.: "127.0.0.1" vagy "/tmp/PHPVisor/comm.sock")

- port : meghatározott port szöveges értékként - A meghatározott host, meghatározott portja amelyre a socket hallgat. (Pl.:"8080"),

[1] Az aktuális felhasználó aki a programot futtatja

- canReUseSocketAddress : true/false - Amennyiben az alkmazásunkat, sokszor indítjuk illetve leállítjuk (interupt) egymás után, a létrejött kapcsolatokat a kernel még nem zárta megfelelően, így ahhoz, hogy újra bindolható legyen a socket a már bindolt címre, a SOCK_REUSADDR flag átadására van szükség. Ezzel a beállítással ez megtörténik.

- username : tetszőleges felhasználónév - Az a felhasználónév amellyel be lehet kérdezni a portra. Ha nincs megadva, akkor bármilyen felhasználóval be lehet kérdezni, nyitott portról van szó. (Legfeljebb 20 karakter hosszú lehet)

- password : tetszőleges jelszó - Az előbbi felhasználóhoz tartozó jelszó. Itt is ugyanaz érvényes mint a felhasználónévnél, ha megvan adva, akkor csak a megfelelő jelszóval, lehet bekérdezni a szerverre. (Legfeljebb 20 karakter hosszú lehet)

Megjegyzés: Jelenleg a program, a stream illetve a datagram alapú socket-kommunikációt támogatja. Ezek közül is jelenlegi működésében a TCP, az UDP, illetve a UNIX socket típusok választhatóak.

2.4.1.3. Folyamat konfiguráció

Az egyes folyamatokat a konfigurációs fájl `processes` kulcsa alatti listában adhatjuk meg, objektumként. Külön ellenőrzés, nincs arra vonatkozólag, hogy megadásra került-e egyáltalán bármennyi folyamat, ha nincs megadva egy se, úgy a program, folyamatok indítása nélkül fog futni és csak újbóli beállítás és indítás után van lehetőség folyamatok hozzáadására.

A folyamatok opcióinak listája: [alapértelmezett érték, ha van]

- command : futtatandó parancs - Az a parancs, amelyet szeretnénk folyamatként futtatni (az érték a escapeshellcmd függvényen kersztül kerül meghívásra)

- autoStart : true/false - A beállítás megmondja, hogy a folyamatot a szerver komponens indításakor el szeretnénk-e indítani vagy nem. [false]

- delay : nem negatív egész szám : Ennyi másodpercet vár a folyamatkezelő, mielőtt ténylegesen elindítja a folyamatot. [2]

- autoRestartOn : "BOTH"/"NONE"/"EXPECTED"/"UNEXPECTED" - Újraindítási stratégiák. Milyen körülmény esetén indítsa újra a szever a folyamatot. Both: Váratlan és várt leállás esetén is. None: egyik esetén sem. Expected:

Az elvárt programleállások esetén. Unexpected: A nem várt leállások esetén. ["none"]

- exitCodes: egész számok listája : Azon program-kilépési kódok listája, amelyeket az elvárt kategóriába sorolunk. [[0]]
- stopWaitSecs : nem negatív egész szám : Ennyi másodpercet vár a kezelő, mielőtt normális módon leállítja a futó folyamatot. (Bevárja a lefutás végét) [10]
- customSignal : tetszőleges szignálnév vagy utótag - Az a szignál, amelyet a kliens alakalmazás Custom Signal néven küldhet a folyamatnak. ["QUIT"]
- termSignal : tetszőleges szignálnév vagy utótag - Az a szignál, amelyet a program terminálása (azonnali leállítása) céljából küldhet a kliens a folyamatnak. ["TERM"]
- directory : mappa elérési útvonala - Az a mappa, ahol a folyamatot futtatni szereténk. Ha nincs megadva akkor a getcwd függvény eredménye lesz az értéke. [null]
- termWaitSecs : nem negatív egész szám : Ennyi időt vár a kezelő, míg terminálja a kívánt folyamatot. (Azonnal leállítja) [1]
- priority : [-20..20]-béli szám : Relatív prioritás, amelyet az adott folyamatra beállít a kezelő. [0]
- groups : szabadszavas lista : Azon csoportnevek, címkék, amelyekkel összefogható egy csoportba több folyamat. Ha egy csoportnév, több folyamatnál, is megvan adva, az azt jelenti, hogy az adott csoportban, annyi folyamat van. (Legfeljebb 15 karakter hosszú lehet egy címke) [[]]
- name : szöveges érték : A folyamat tetszőleges neve. (Ez alapján (is) lehet indítani folyamatot a kliensről. Legfeljebb 15 karakter hosszú lehet)
- stdErrorFile : null vagy fájl elérés útvonala - Az a fájl ahová a folyamat a standard error leíróját irányítja a program. Ha nincs megadva, akkor a folyamathoz tartozó naplófájlba kerül átirányításra. Ha az is le van tiltva, a hibák el lesznek "nyelve". [null]
- stdInFile : null vagy fájl elérési útvonala - Az a fájl, amelynek tartalmát stream-ként beleirányítjuk a folyamat standard inputjába, ha az értéke nem null. [null]

- stdOutLogFilePath : "AUTO"/fájl elérési útvonala - A folyamat standard outputjára kapcsolt fájl. Amennyiben "AUTO", úgy a folyamatkezelőben meghatározott childLog könyvtár alá a program létrehoz egy <folyamat neve>.log állományt, és az lesz a napló. Ha az értéke null, a folyamathoz naplófájl nem keletkezik. ["AUTO"]

- stdOutLogMaxBytes : fájlméret : Az előbb említett napló maximális mérete. (Byte-okban vagy megadott mértékegységgel) ["50KB"]

- stdOutLogMaxBackups : nem negatív egész szám : A backup naplók maximális száma a folyamat naplóira vonatkozólag. [2]

- logMode : "normal"/"pure" - Naplózási mód. Amennyiben normal, úgy a folyamat életciklusát lekövető bejegyzések is naplózásra kerülnek, amennyiben pure, úgy a napló fájl tisztán csak a folyamat outputját tartalmazza. [normal]

- logLevel : debug/notice/warning/error - Naplózási szint. A folyamat mely saját eseményeit naplózza. (Amennyiben a logMode normal.) [debug]

- envVariables : kulcs-érték párok listája (JSON objektum) - Egy tömb reprezentációja, azoknak a környezeti változóknak, amelyeket az adott folyamat indulásakor át akarunk adni neki. (Ennek formátuma speciálisan a folyamat igényeitől függhet.) [[]]

2.4.1.4. Parancssori beállítások

A program lehetőséget biztosít, parancssori kapcsolók használatára, amellyekkel a kívánt módon befolyásolhatjuk, illetve konfigurálhatjuk a főprogramunkat. A szerver-komponens az alábbi parancssori funkciókat támogatja:

Rövid név	Hosszú név	Paraméter	Funkció
c	configuration	FILENAME	Program indítása megadott FILENAME konfigurációs állomány alapján.
n	nodaemon		Amennyiben adott, úgy a program előtérben fut.
h	help		Kiírja a program a megadható parancssori kapcsolók listáját leírásukkak, majd leáll.
v	version		Kiírja a program verzióját, majd leáll.
u	user	USER	Program futtatása USER felhasználói nevű vagy USER uid-val rendelkező felhasználóval.
m	umask	MASK	A megadott MASK maszkolás használata, a demonizált alfolyamaton.
d	directory	DIRECTORY	Könyvtár, amibe átnavigálunk, demonizáció esetén.
p	print_log		Minden bejegyzés, ami naplózásra kerülne kiíródik a képernyőre is (csak ha előtérben fut).
l	logfile	FILENAME	A megadott FILENAME fájl használata, mint napló.
y	logfile_max_bytes	BYTES	A napló maximális méretének meghatározása.
z	logfile_backups	NUM	Mennyi archív, telített naplót engedünk tárolni.
e	loglevel	LEVEL	Milyen naplózási szintet használjon a program. Azon bejegyzéseket amik nem érik el a megadott szintet, nem kerülnek naplózásra.
q	childlogdir	DIRECTORY	Könyvtár amelybe a gyermek folyamatok, logjai kerülnek.
k	nocleanup		Ennek megadásával megakadályozzuk, hogy a program indulásakor kitörölje a folyamatok naplóit.

Paraméterek

- FILENAME - Egy fájlt reprezentál, annak teljes abszolút elérési útvonalával (pl.: /tmp/PHPVisor/log/server.log)
- USER - Egy felhasználót reprezentál, vagy a felhasználó nevével vagy annak uid-val (pl.: root vagy 0)
- MASK - Egy maszkot reprezentáló számsorozat (pl.: 022)
- DIRECTORY - Egy könyvtárat reprezentál (abszolút útvonallal) (pl.: /tmp/PHPVisor)
- BYTES - Fájlméretet reprezentáló érték, amennyiben egy szám, úgy bájtban értendő, de lehetőség van szöveges érték megadására is, amelnyek prefixe egy szám, a szuffixe pedig a megfelelő bájt-mértékegység (pl.: 5MB)
  Elfogadható mértékegységek: B, KB, MB, GB, TB, PB, EB, ZB, YB.
- NUM - Egy egyszerű darabszámot reprezentáló érték (pl.: 5)
- LEVEL - Naplózási szintet reprezentál, amelyet szövegesen lehet megadni.
  Elfogadható naplózási szintek: debug, notice, warning, error.

2.4.1.5. Futtatás

Az alkalmazás indításához szükséges állomány a főkönyvtár */app* alkönyvtárában található. Az indításhoz a Server.php szkriptet kell futtatni, amire két lehetőség adott: Az */app* könyvtárban állva,

- a parancssorba kiadni a `php Server.php` parancsot vagy
- a parancssorba kiadni a `./Server.php` parancsot, (amennyiben a */usr/bin/php* útvonalról elérhető a php)

Az alkalmazás mivolta miatt, kilépés vagy bezárás funkció nincs implementálva a programban. Az első üzembehelyezéskor elég lefutattni a fent említett parancsot, s így a következő konfiguráció váltásig (vagy valamely rendszerhibáig) a komponens futni fog. Jelenleg nem ad lehetőséget a program arra, hogy futás közben újrakonfigurálható legyen a program. Új konfiguráció, csak új futtatással eszközölhető. A daemonizált folyamat a beállított pidfile zárolása, miatt garantáltan egy példányban fog futni. A szerver komponens futásásnak, valamely folyamatkezelő parancson keresztül történő leállítással (pl.: "kilövésével") vethetünk véget.

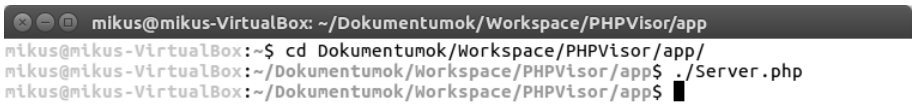
2.4.1.6. Működés

Amint a futatási parancs kiadásra került, a program detektálja, majd értelmezni kezdi a megadott konfigurációk összességét. A program általános célú működése mellett, két további funkcióval is rendelkezik:

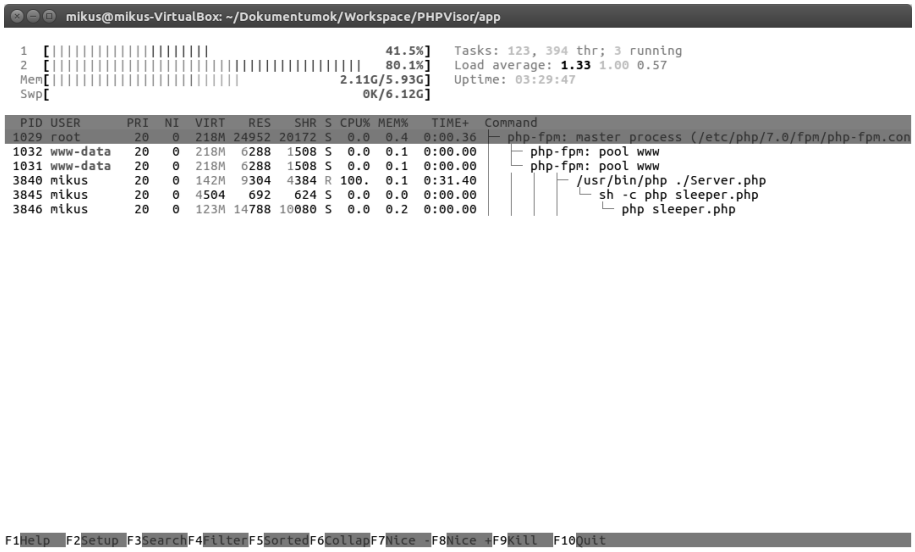
- Lekérhető a programhoz tartozó menü/súgó, amely a konzolról állítható kapcsolókat írja a képernyőre a hozzájuktartozó mininális segédleírással. Ekkor a program az üzenet kiírása után leáll, folyamatok nem indulnak el.
- Továbbá lekérhető a program verziója, amelyet a program az alkalmazás gyökérkönyvtárában lévő *.version* fájlból nyer ki. A program kiírja a verziót, a képernyőre majd leáll, folyamatok ekkor sem indulnak. (Amennyiben nem érhető el a megfelelő helyen, a verziófájl, úgy a program a "Version UNKNOWN" feliratot írja ki)

A szerver komponensre jellemző alkalmazási környezet, hogy a célszámítógépen, a program *daemon*-ként fut. Ekkor a folyamat leválasztásra kerül a terminálról és háttérfolyamatként fut tovább.

Ilyenkor a program működésének állapotairól, csak a naplóbejegyzések segítségével tájékozódhat a felhasználó.



2.1. ábra. Példa az alkalmazás daemon-ként való futtatására. Miután a parancsot kiadtuk, viszakapjuk a promptot, s a futó program a háttérbe került.

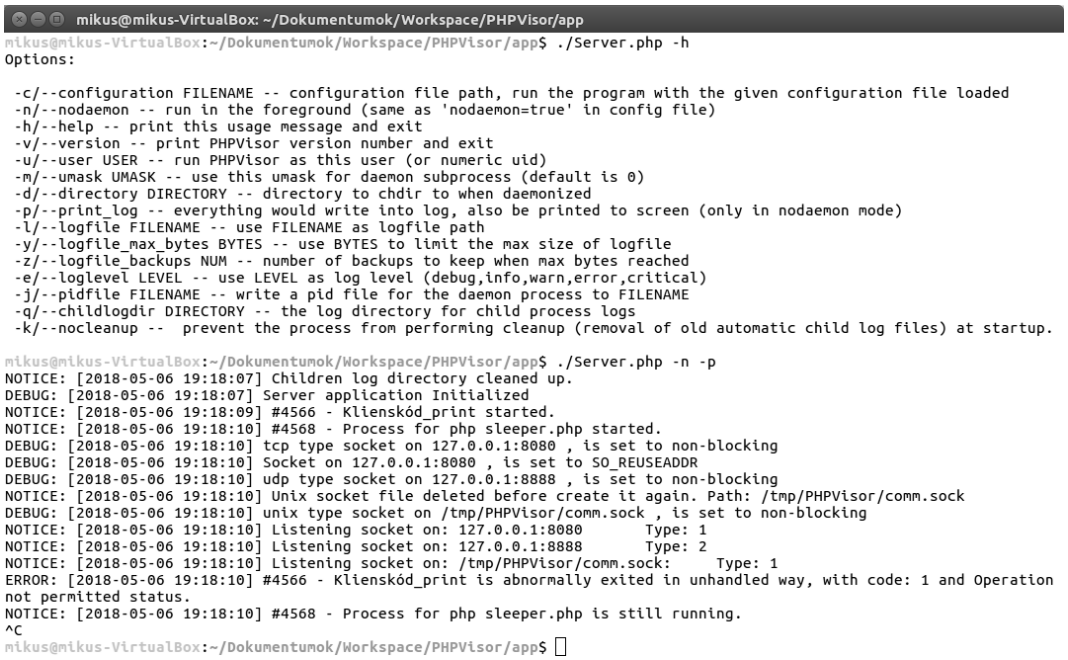


2.2. ábra. Folyamatkezelőben láthatjuk (htop), hogy a program a háttérben tovább fut.

Ahhoz, hogy a program működése jobban követhető legyen, a *nodaemon* illetve a *print_log* opciók megadására/engedélyezésére van szükség. Ekkor jobban megfigyelhető a progam működése közben, ugyanis a nodaemon beállítással, az előtérben, azaz az aktuális terminál ablakon fut az alkalmazás, továbbá a printLog-gal elérhetjük, hogy az aktuális naplóbejegyzéseket a terminálra is kiírja a program. Így valós időben megfigyelhető az alkalmazás.

Megyjegyzés: A printLog opció nem használható a háttérben való futtatás esetén, mivel az *echo* függvény arra a terminálra írná az üzeneteket, amelyről leválasztottuk a programot.

A 2.3. ábán látható, hogy miután elindult a program sorban jelennek meg a fontosabb műveletekhez kapcsolódó rendszerüzenetek. A program a beállított naplózási szintnek megfelelő időbélyeggel ellátott bejegyzéseken keresztül leírja mikor inicializálja magát a szerver komponens, milyen folyamatokat indított el a rendszer, milyen portokat nyitott ki, illetve melyeken hallgat. A naplókon nyomon követhetőek *debug* naplózási szinttel, a klienssel való kommunikáció nyomai is. A naplók rögzítik mikor egy kapcsolat létrejön, illetve a kommunikáció tárgyát képező üzeneteket is. (2.4. ábra)



2.3. ábra. Az alkalmazáson belüli súgó, illetve a program futásának követésére szolgáló funkciók szemléltetése.

2.4.1.7. Hibaüzenetek

Mindkét program (szerver-kliens), kivételek, azaz exception-ök segítségével jelzi hibaállapotait. A kivételek jellemzően adatbeviteli vagy kommunikációs mechanizmusokból eredhetnek. A kivétel keletkezése, azonnal leállítja a program futását. Amennyiben nem konfigurációs hiba keletkezik, a rendszer az okozó kivételt, a napló végére írja, majd leáll. A szerver komponens lehetséges hibaüzenetei:

- *InvalidArgumentException:* Olyan kivételek, amelyek azt hivatottak jelezni a felhasználónak, hogy konfiguráció során invalid adatot adott meg a megjelölt helyen.

- *RuntimeException:* Ezeket a kivételeket, akkor dobja a program, ha valamely PHP-s függvény nem az elvárt működést produkálja. Ez lehet akár az, hogy az operációs rendszer már nem tud *fork*-olni, de akkor is ilyen típusú hibát kapunk, amennyiben kommunikációs port megadása nélkül indítanánk a programot.

2.4. ábra. A naplóbejegyzéseken nyomon követhetőek a klienssel kommunikált üzenetek is.

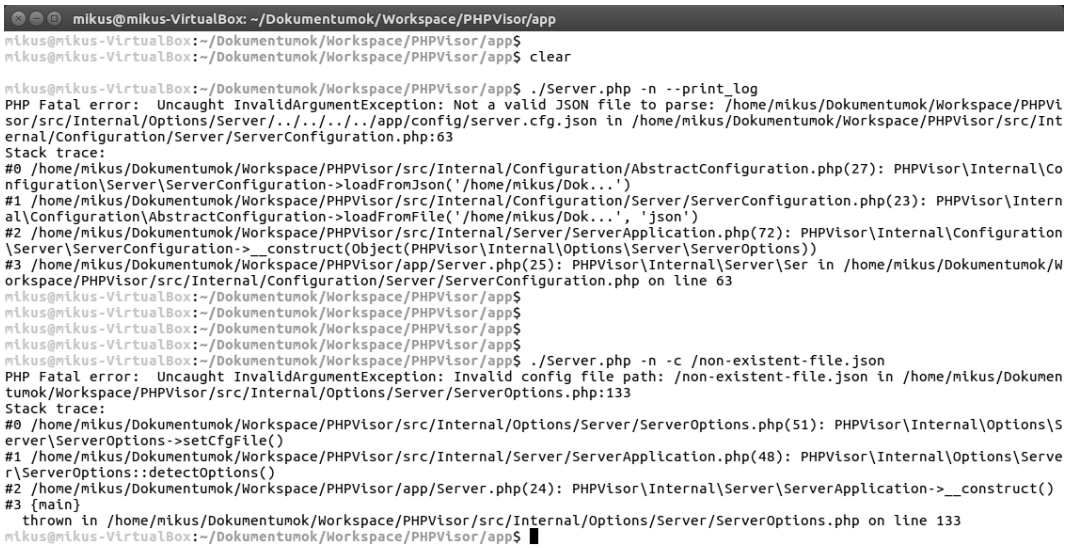
2.4.2. Client

2.4.2.1. Konfiguráció

A kliens komponensnek lényegesen kevesebb konfigurálási lehetősége van, mint a szervernek. Itt a konfigurációban kvázi az előre egyeztetett adatokat találhatjuk amelyet a kliens a szervertől kap. A formátuma szintén JSON, illetve a megfelelő szabályok itt is érvényesek. A példa (alapértelmezett) konfigurációs állományt, a már említett *config* könyvtárban találjuk. Definiálásra kerülnek a következő opciók:

- serverUrl : Ezen beállítás határozza meg azt a kommunikációs címet, amelyre szerenténk csatlakozni. Amennyiben TCP kapcsolatot határozott meg a szerver, úgy elég megadni az IP/port párost a szokásos alakban. (Például: `"127.0.0.1:8080"`). Ha UDP vagy Unix socketen keresztül csatlakozna a kliens, akkor a protokollt is a címbe kell írni. UDP esetén `udp://127.0.0.1:8888`, Unix socketnél `"unix:///tmp/PHPVisor/comm.sock"`. Ez utóbbinál a port szám jellmezően elhagyható, hiszen egy fájlra tett hivatkozásról van szó.

- username : A felhasználónév alapértelmezett értéke null érték. Viszont csak akkor lesz a kívánt működésű a programok kommunikációja, amennyiben az itt megadott érték pontosan megegyezik a szerver konfigurációjában megadottal. A felhasználónév, illetve jelszó párossal minden egyes beérkező kérést authentikál a szerver, így csak a megfelelő információk birtokában nyílik lehetőség a



2.5. ábra. Példa helytelen konfigurációs adat megadásakor keletkező kivételekre.

folyamatok tényleges manipulációjára. (Legfeljebb 20 karakter hosszú lehet.)

- password : A felhasználónévhez hasonlóan, ennek a is a szerveren megadott adattal kell egyeznie. (Legfeljebb 20 karakter hosszú lehet.)

2.4.2.2. Parancssori konfiguráció

Természetesen a kliens komponens is kihasználja a már említett parancssori opciók nyelvi támogatottságát és a fentebb említett három opció, (plusz a konfigurációs állományt meghatározó opció) szintén megadható a kapcsolók szintjén. Ekkor a szokásos hierarchia lép életbe a konfigurációs állomány, illetve a konzolos kapcsolók között. (A kapcsolók előnyt élveznek.) A kliens az alábbi kapcsolókat ismeri:

Rövid név	Hosszú név	Paraméter	Funkció
c	configuration	FILENAME	A megadott FILENAME konfigurációs állomány használata
h	help		Kiírja a programhoz tartozó súgó üzenetet, a parancssori kapcsolókról, leírásukkal.
l	url	URL	A megadott URL használata mint serverUrl. A cím, amelyre a kliens csatlakozik.
u	user	USERNAME	A jogosultság ellenőrzéshez használt felhasználónév.
p	pwd	PASSWORD	A jogosultság ellenőrzéséhez használt jelszó.

Paraméterek

- FILENAME : A szerver konfigurációjának leírásával analóg módon, itt is a kívánt fájl elérési útvonalát határozhatjuk meg. Az alapértelmezett fájl a *config* mappa alatt található *client.cfg.json*.
- URL : A konfigurációs részben tagalatak szerinti szabályok alapján formázott, kommunikációs cím, amelyen keresztül a szerverhez kapcsolódik a kliens.
- USERNAME : A megadott felhasználónév, amellyel a kommunikációt szeretnénk végezni. (Legfeljebb 20 karakter hosszú lehet)
- PASSWORD : Az előbbi felhasználónévhez tartozó jelszó. (Legfeljebb 20 karakter hosszú lehet)

2.4.2.3. Futtatás

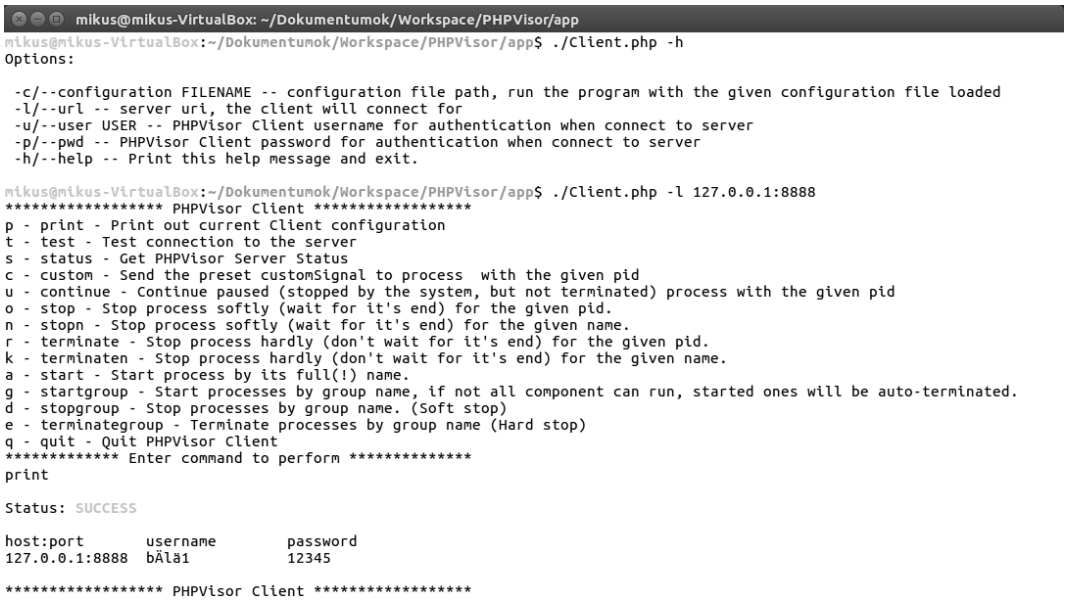
Az alkalmazás indításához szükséges állomány a *főkönyvtár/app* alkönyvtárában található. Az indításhoz a Client.php szkriptet kell futtatni, amire két lehetőség adott: A */app* könyvtárban állva,

- a parancssorba kiadni a `php Client.php` parancsot vagy
- a parancssorba kiadni a `./Client.php` parancsot, (amennyiben a */usr/bin/php* útvonalról elérhető a php)

A program használati célja, hogy a szerver-komponenssen keresztüli kommunikáció során, információt szolgáltasson a felhasználójának, a szerveren beállított folyamatokról, illetve adott parancsokat közvetítsen feléjük. Az implementált akciólistából válogathat a menün keresztül a felhasználó, amelyet a billentyűzet segítségével irányíthat. Egy lehetőség kiválasztásához a kiválasztott menüpont betű vagy szókódját kell begépelni, majd Enter-t nyomni. Amennyiben ismeretlen opciót választunk a program erről tájékoztatást ad majd újra lehet próbálkozni. Vannak olyan akciók is amelyek további paraméterek megadását követelik (például folyamat indítása név alapján). Ekkor a megfelelő menüpont kiválasztása után a program tájékoztatást ad, arról, hogy milyen értéket vár. A kívánt értéket a billentyűzeten beírva, majd Enter-t ütve lehet az akciónak átadni.

2.4.2.4. Működés

A program elindulását követően, rögtön kiírja a képernyőre a menü-t, amelynek segítségével tájékozódhat a felhasználó az lehetséges opciókat illetően. Amennyiben



2.6. ábra. A kliens program indításának szemléltetése kapcsolók használatával. A menün keresztül látható milyen utasításokat lehet használni.

a szerver alkalmazás nem fut, úgy a kapcsolódási kísérleteket tartalmazó akciók, egy kivétel dobásával közlik, hogy a kapcsolódás meghiúsult. A program lehetőséget nyújt a kapcsolat tesztelésére anélkül, hogy valamely folyamatot érintő akcióval kellene ezt letesztelni. A program menüpontjai a következők:

- print : Ezen menüponton keresztül, az aktuális konfigurációt lehet megtekinteni. A program kiírja a csatlakozási pont hivatkozását, a fehasználónevet és a jelszót is a képernyőre. (2.6. ábra)
- test : Lehetőség van tesztelni a szerverhez való csatlakozás sikerességét. Ilyenkor siker esetén egy üres üzenetet kapunk vissza a szervertől, hiba esetén egy hibajelzést.
- status : Ezzel az opcióval lekérdezhetőek a szeveroldalon futtatott folyamatok státuszai. A program oszlopokba szedve kiírja a képernyőre a futtatott folyamatok azonosítóit, illetve állapotukat is. (Az egyes oszlopok részletes jelentésének leírása a 2.4.2.5. pont alatt olvasható.)
- custom : A funkció egy előre (szerver oldalon, a folyamat konfigurációjában) meghatározott szignált küld az adott folyamatnak.
- continue : Amennyiben az operációs rendszer egy folyamatot megállít (szünetelteti), ezzel a funkcióval egy folytatásra késztető szignált küld az adott folyamatnak.

- stop : A kliensben a stop fogalma, eltér az operációs rendszer stop fogalmához képest. Itt nem áll meg a folyamat tényleges futása, hanem a `proc_open` függvény hívásával, a program egy futásának a végét megvárva leállítódik. Ezen opció lehetőséget nyújt, egy ilyen típusú leállításra a folyamat pid-a alapján.

- stopn : Az előbb leírt működést produkálja, csak az előzővel ellentéteben itt a program neve alapján van lehetőség leállítani a folyamatot, nem a pid-a alapján.

- terminate : Ezen funkció a stop működésnek a kényszerített verziója. A megadott pid-ú folyamat terminálására van lehetőségünk a `proc_terminate` függvény segítségével, amely a folyamat konfigurációjában megadott terminálló szignált küldi a folyamatnak, ezzel az azonnali megállítását okozva.

- terminaten : Az előbb említett terminálási funkció azzal a kivétellel, hogy itt a folyamat neve alapján van lehetőségünk terminálni.

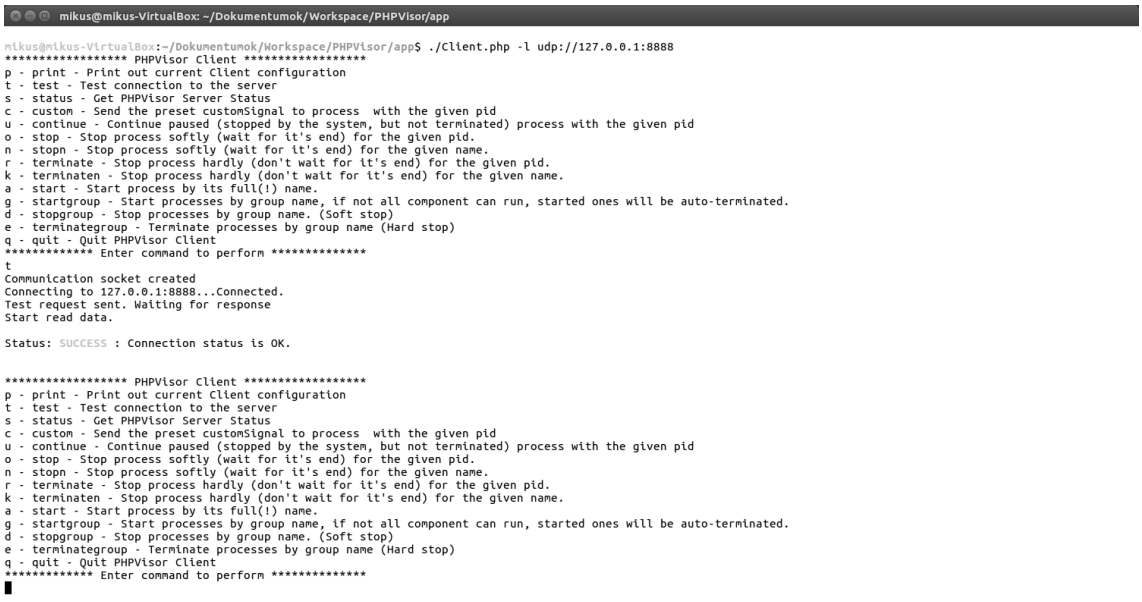
- start : Ez az opció lehetőséget ad számunkra, hogy a szerveroldalon előre megadott folyamatok közül elindítsunk egyet. Az akciónak megadott folyamatnév alapján kerül a folyamat elindítása. (A teljes név megadása szükséges, ezért célszerű lehet nem a rendszerre bízni, a process nevek automatikus kiosztását.)

- startgroup : A program a szerver leírásban említett címkék/csoportnevek alapján is el tud indítani esetleg megállítani programokat. A program a megadott csoportnév, alapján a konfiguráció sorrendjében elindítja a folyamatokat. Amennyiben egy folyamat már futott a csoportból, ahhoz nem nyúl. Amennyiben az adott kérés során indított folyamatok közül, valamelyiket nem sikerült elindítani, a szerver leállítja ezeket az elindított folyamatokat is. (Amelyek a kérés előtt is futottak, azokat nem.)

- stopgroup : Ezen lehetőség segítségével a fentebb említett csoportnév alapján lehet kérni azok (soft) normál leállítását. (Hasonlóan a stop lehetőséghez, csak itt csoportnévvel szűrünk a leállítandó folyamatokra.)

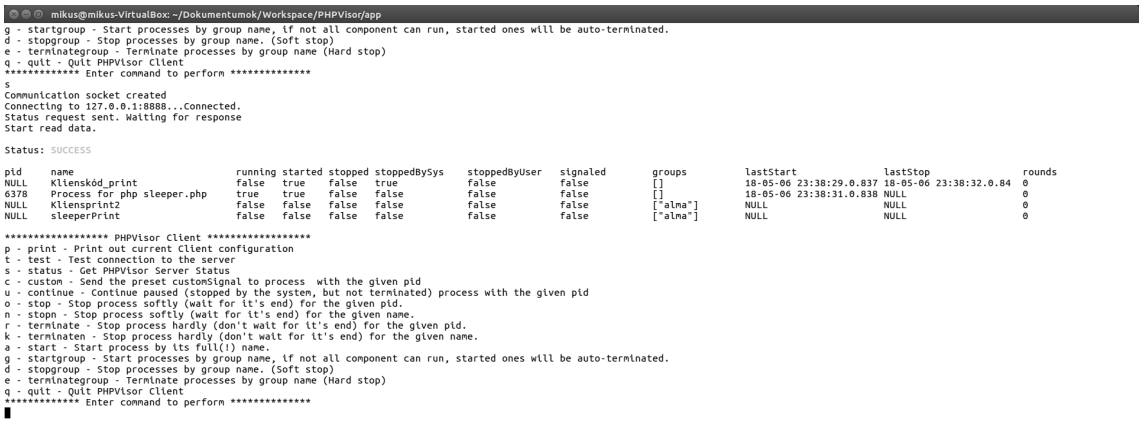
- terminategroup : Ennek a parancsnak a segítségével, lehetőség nyílik megadott csoportnév alapján, kikényszerített leállítást kezdeményezi a csoport folyamataira.

- quit : Ezen menüpont választásával a program kilép.



2.7. ábra. A kliensnek a szerverrel való kapcsolatának tesztelése a kliens program test opciója alapján.

2.4.2.5. Felület értelmezése



2.8. ábra. A kliensen keresztül lekérhető, az aktuálisan konfigurált folyamatok állapota.

A 2.8. ábrán látható, amikor a kliens lekérdezi, az aktuális folyamatok állapotát. Ekkor összesen 12 oszlopban tájékoztatást kapunk, a legfontosabb információkról, állapotokról. Az első oszlopban látható, a folyamatok *pid*-a, ha van. Amennyiben egy folyamat aktuális nem rendelkezik pid-val, úgy NULL-ként jelenik meg, ez jelzi, hogy a folyamat nem fut. Mivel nagyon gyorsan lefutó folyamatok is lehetnek a konfiguráció részei, ezért ez az adat nem tekinthető aktuálisnak. Ez az oka amiért nem csak az azonosító szerinti manipulációt, de a név szerintit is támogatja a rendszer. A második oszlopban a program megnevezése található (*name*). A formázott kiíratásnak karakterszám-béli korlátja is van, ezért ajánlott rövid és tömör elneve-

zéseket használni a folyamatokhoz. (Érdemes azért is ügyelni a rövid, egyértelmű és lehetőleg egyedi nevek használatára, mivel a programban nincs implementálva egyediséget ellenőrző logika, ezért a név alapján talált első azonosított folyamattal fog a rendszer dolgozni, ha az kérdéses helyzetbe kerülne.) A harmadiktól a nyolcadik sorig a következő állapotjelző információkat találhatjuk:

- running : Arról ad tájékoztatást, hogy a megadott folyamat fut-e vagy sem.
- started : Ha igaz, akkor az alkalmazás el lett indítva, de nem feltétlenül fut. Ekkor érdemes tüzetesebb vizsgálatnak alá tenni, a folyamat saját naplóját. (A természetes megállás, újraindításkor megfelelően kapcsolja ennek az értékét.)
- stopped : Amennyiben az operációs rendszer megállítja a folyamat futását, T jelzésű (stopped) állapotba helyezi a folyamatot, ekkor a folyamat tulajdonképpen tovább fut. Ilyenkor lehetőség van megkísérleni a folyamat folytatását a megfelelő folytatást jelentő szignál küldésének segítségével, amire a program lehetőséget is ad.
- stoppedBySys : Amennyiben igaz, az a következőket jelentheti. Amennyiben a program normálisan (a tőle elvárt módon) megállt, ez az érték igaz lesz, miközben a running, és a started hamisra vált, továbbá a rounds oszlopba 0-nál nagyobb szám kerül. Ha program abnormálissan állt meg, (például egy szignál kényszerítette megállásra) az értéke igaz lesz, ahogyan a signaled oszlop értéke is.
- stoppedByUser : Ez az állapotjelző azt fejezi ki, hogy a felhasználó a kliensen keresztül állította le a programot. (Vagy terminálta azt.)
- signaled : Amennyiben igaz, a programot valamely szignál megállásra késztette. (Ez az érték újra hamisra vált, ha egy stopped folyamatnak folytatási kérelmet küldünk.)

A képernyőre írt információk között, a *groups* kulcs alatt megjelenik az egyes folyamatok címkelistája is, amely segítséget nyújthat, amennyiben csoportnév alapú műveleteket szeretnénk végrehajtani. További két oszlopban időbélyegekkel ellátott jelzőket találhatunk, a *lastStop* értéke bárminemű fentebb említett leállás legutóbbi idejét, míg a *lastStart* a legutóbbi indításának időpontját szimbolizálja. Az utolsó oszlopban a már említett *rounds*, az egyes futásokat számlálja. Egy teljesértékű, normális lefutás e számláló eggyel való növekedését okozza.

A státuszinformációk kinyerésén kívül folyamatok futását befolyásoló parancsok közlésére is van lehetőség. Ezek, paraméterhez kötött akciók amelyek esetében a paraméterek az akció hatáskörébe eső folyamatokat határozzák meg.

A szerverrel való kommunikáció eredményéről a kliens, minden esetben tájékoztat. *Success* állapottal jelzi, amennyiben a kérés sikeresen célt ért, végrehajtódott és a megfelelő sikeres válaszüzenetet kapta vissza.(2.12. ábra) Amennyiben valamely olyan eset lépett fel amelyet a szerver hibásnak ítélt meg, (például elrontott felhasználó-jelszó páros), a válaszban érkező hibaüzenetet a kliens az *Error* állapottal jeleníti meg. (2.11. ábra)

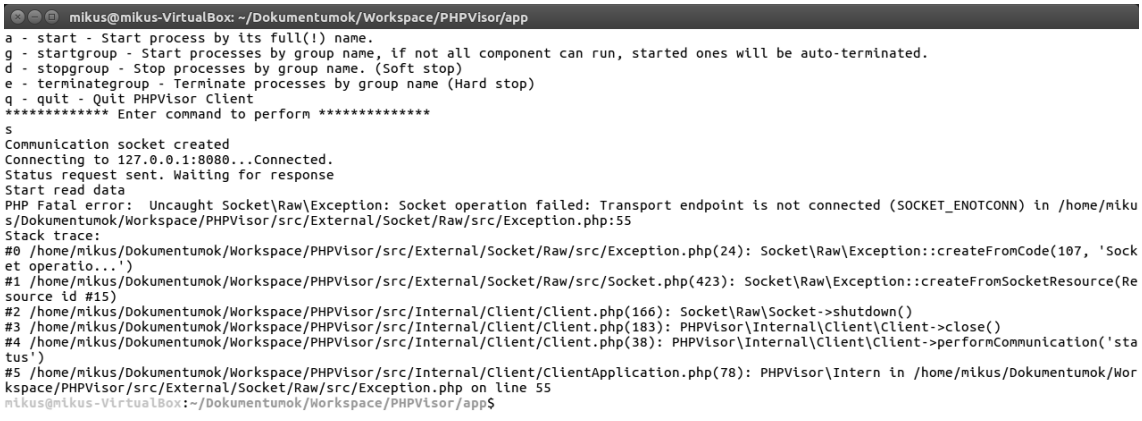
2.4.2.6. Hibaüzenetek

Abban esetben, ha üzenetváltás során, elszállna a szerverkomponens, miközben a kliens a válaszra vár, az nem kapja meg a megfelelő méretű üzenetet, s *Error* státuszú hibaüzenet ír ki a képernyőre, miszerint nem tudta megfelelően átvenni az üzenetet a szervertől. A kliens 60 másodpercig vár, egy-egy kérés utána a válaszra, és amennyiben az lejár, s nem érkezik megfelelő válasz, ugyanúgy az előbb említett hibaüzenet jelenik meg a képernyőn.

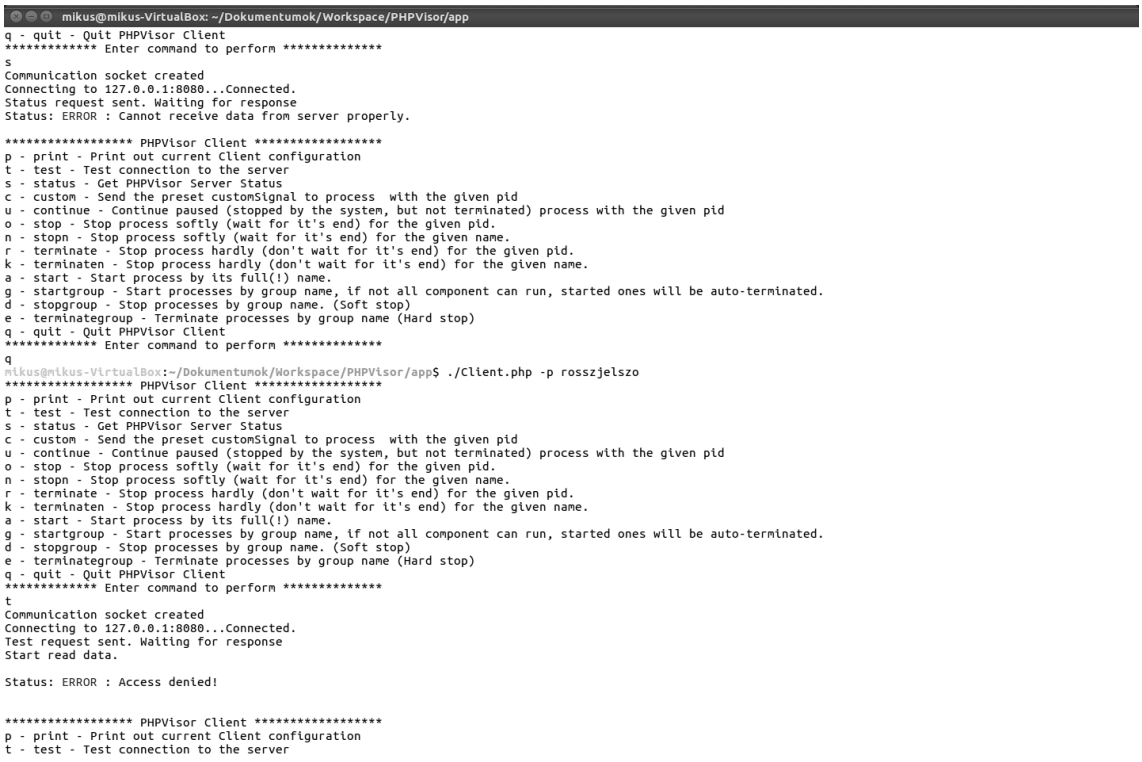
A program működése során nem csak az imént említett hibák jelenhetnek meg. Az alkalmazás a szerver komponenshez hasonlóan kivételekkel jelzi, amennyiben a kiépített kommunikációs csatáronán hiba lépett fel. A *Connection refused* kódú hibaüzenet (2.9. ábra), azt jelenti, hogy a kliens nem tud kapcsolatot létesíteni a kívánt host-tal (például amennyiben a szerver nem fut és csatlakozni próbálunk rá), míg a *Not connected* kódú hibaüzenet (2.10. ábra) azt hivatott jelezni, hogy a kliens és a szerver között megszakadt a kommunikációs kapcsolat.

```
mikus@mikus-VirtualBox: ~/Dokumentumok/Workspace/PHPVisor/app
mikus@mikus-VirtualBox:~/Dokumentumok/Workspace/PHPVisor/app$ ./Client.php
****************** PHPVisor Client ******************
p - print - Print out current Client configuration
t - test - Test connection to the server
s - status - Get PHPVisor Server Status
c - custom - Send the preset customSignal to process  with the given pid
u - continue - Continue paused (stopped by the system, but not terminated) process with the given pid
o - stop - Stop process softly (wait for it's end) for the given pid.
n - stopn - Stop process softly (wait for it's end) for the given name.
r - terminate - Stop process hardly (don't wait for it's end) for the given pid.
k - terminaten - Stop process hardly (don't wait for it's end) for the given name.
a - start - Start process by its full(!) name.
g - startgroup - Start processes by group name, if not all component can run, started ones will be auto-terminated.
d - stopgroup - Stop processes by group name. (Soft stop)
e - terminategroup - Terminate processes by group name (Hard stop)
q - quit - Quit PHPVisor Client
************* Enter command to perform **************
t
Communication socket created
Connecting to 127.0.0.1:8080...PHP Fatal error:  Uncaught Socket\Raw\Exception: Socket operation failed: Connection refused (SOCKET_ECONNREFUSED)
 in /home/mikus/Dokumentumok/Workspace/PHPVisor/src/External/Socket/Raw/src/Exception.php:55
Stack trace:
#0 /home/mikus/Dokumentumok/Workspace/PHPVisor/src/External/Socket/Raw/src/Exception.php(24): Socket\Raw\Exception::createFromCode(111, 'Socket o
peratio...')
#1 /home/mikus/Dokumentumok/Workspace/PHPVisor/src/External/Socket/Raw/src/Socket.php(109): Socket\Raw\Exception::createFromSocketResource(Resour
ce id #15)
#2 /home/mikus/Dokumentumok/Workspace/PHPVisor/src/Internal/Client/Client.php(103): Socket\Raw\Socket->connect('127.0.0.1:8080')
#3 /home/mikus/Dokumentumok/Workspace/PHPVisor/src/Internal/Client/Client.php(176): PHPVisor\Internal\Client\Client->connect()
#4 /home/mikus/Dokumentumok/Workspace/PHPVisor/src/Internal/Client/Client.php(33): PHPVisor\Internal\Client\Client->performCommunication('test')
#5 /home/mikus/Dokumentumok/Workspace/PHPVisor/src/Internal/Client/ClientApplication.php(71): PHPVisor\Inte in /home/mikus/Dokumentumok/Workspace
/PHPVisor/src/External/Socket/Raw/src/Exception.php on line 55
mikus@mikus-VirtualBox:~/Dokumentumok/Workspace/PHPVisor/app$
```

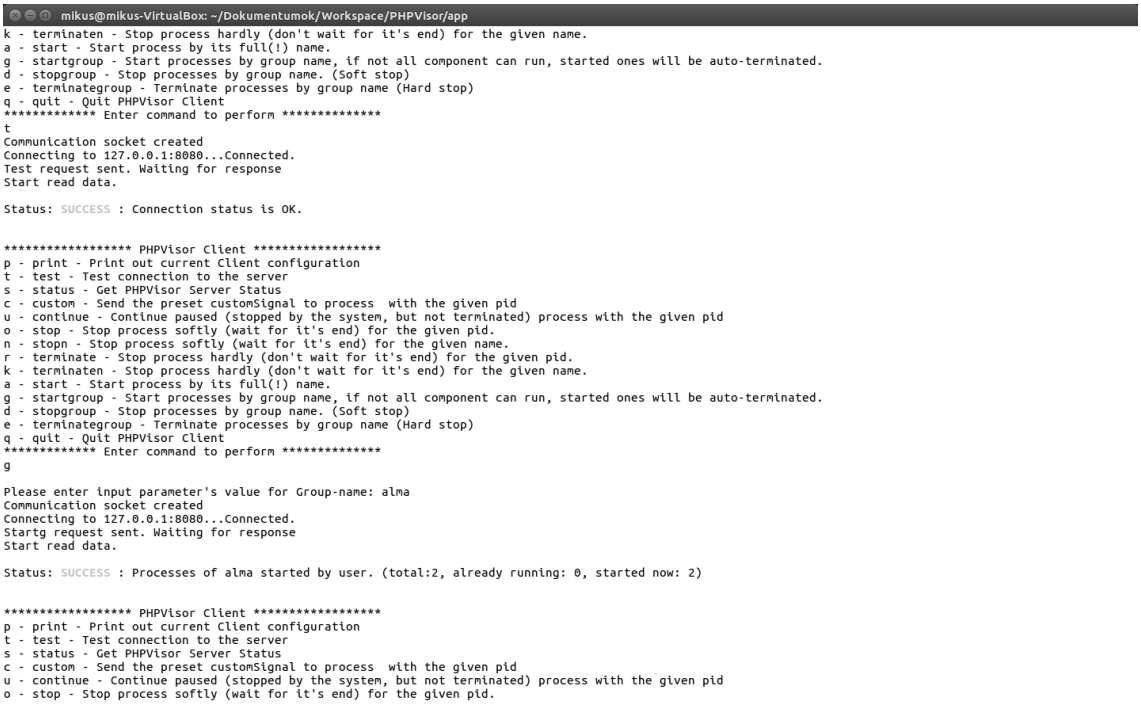
2.9. ábra. Példa a *Connection refused* kivétel megjelenésére



2.10. ábra. Példa a *Not connceted* kivétel előfordulására



2.11. ábra. Példa hibás üzenet fogadásakor vagy timoutkor keletkező hibára, illetve téves jogosultsági adatok esetén keletkezp hibára.



2.12. ábra. Példa sikeres üzenetváltásokra.

3. fejezet

Fejlesztői dokumentáció

Ezen fejezetben bemutatásra kerül a programok szerkezete, felépítése, illetve a megvalósításnak bizonyos elemei is. Továbbá megemlítésre kerülnek komolyabb problémák, amelyek a rendszer körül adódhatnak, érdekesebb algoritmusok, nyelvi korlátok és az is, hogy az alkalmazások milyen objektumokból épülnek fel. Megjegyzés szintjén jövőbe mutató javaslatokat, ötleteket is közlök, amelyek útmutatásul szolgálhatnak a későbbi fejlesztési időszakokban.

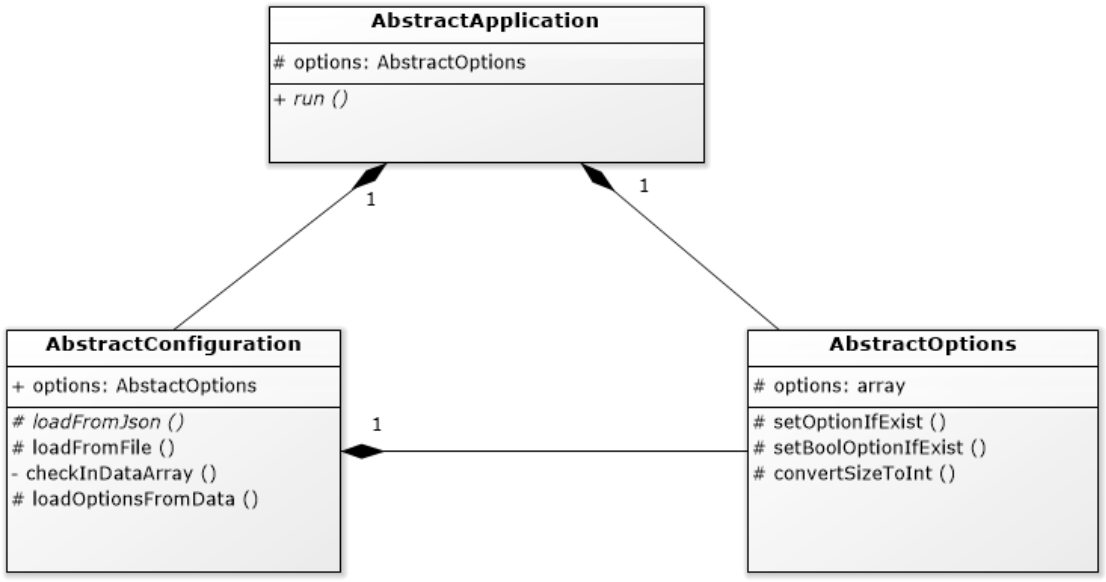
Az alkalmazások tervezése és megvalósítása objektum-orientált szemléletben történt, aminek eredményeképp sokféle osztály került megvalósításra. Ezek hierarchiába illetve öröklődési láncokba szervezve működnek együtt.

A fejlesztés során, a kommunikációs alapokhoz egy alacsonyszintű külső könyvtárat használtam, amely egy minimális OOP szemléletű réteget helyez a PHP szolgáltatta socket függvények fölé. A könyvtár nyílt forráskódú, MIT licesszel ellátott project, amely megtalálható a GitHub-on.[6] Segítségével, könnyedén megvalósíthatóak a szerver illetve a kliens objektumok, amelyek a kommunikáció alapját képzezik. Használatát tekintve egyszerű. Egy ún. Factory objektumon keresztül példányosítja a megfelelő komponenst, ami a Socket osztály egy példánya. A külső forrásanyag, teljes project-ként megtalálható az alkalmazás *src* mappájának *External* almappája alatt.

3.1. Tervezés

A feladat részletes leírása, a 1.2. pontban található. A program architektúrája miatt, két alapvető egységre bomlik: kliensre és szerverre. A szerver egy olyan program, amely futása közben felhasználói interakciót nem biztosít, így tulajdonképpen csak egy konfigurálási réteggel lép kapcsolatba a felhasználó. A kliens ezzel szemben, egy főképp felhasználói interakcióra épülő alkalmazás, amelynek lehetőséget kell biztosítania az elvárt akciók végrehajtására és közlésére a szerver felé.

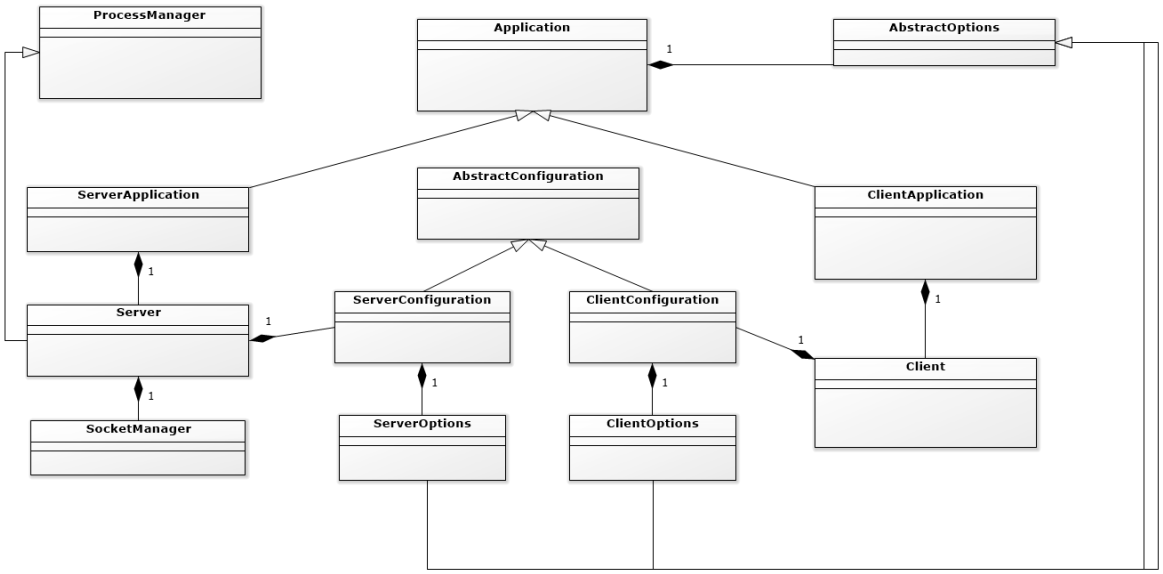
Mindkét program egy-egy objektumént való reprezentálásával, egy könnyen alkalmazható ún. alkalmazási réteg jön létre. Ez azt jelenti, hogy az egyes futást kiváltó állományokban, csupán a megfelelő alkalmazás-objektum példányosítására majd elindítására van szükség. Ezen alkalmazásrétegbe épül be a konfigurációs-réteg és a program lényegét adó metódusok is. Az alapkoncepciója a programoknak, hogy az *Application* programváz, a megfelelő *AbstractOptions* tulajdonságokkal ruházza fel magát, a beállítások meghatározását követően. Az *AbstractOptions* reprezentál-



3.1. ábra. A programok alapstruktúrájának vázlata.

ja, az alkamazások által használt beállításokat, amelyket az *Application* detektál induláskor. Ezután a megfelelő alkalmazásobjektum, létrehozza a lényeget képező objektumot (Kliens objektum - Szerver objektum). Ezen objektumok, a hozzájuk tartozó *AbstractConfiguration* segítségével jönnek létre, ugyanis ezen konfigurációs réteg valósítja majd meg a fájlból történő információ kinyerését, amely adatokkal aktualizálja a megfelelő *AbstractOptions* objektumot. A konfigurációs, illevte opciós szintek elkülönítésére, azért van szükség, mert a program indításakor megadott esetleges opciók között lehetnek olyanok is amik függetlenek az elvárt szerver- vagy

kliensműködéstől (Például a *help* menü megjelentése, majd kilépés funkció). Továbbá az egymást átfedő beállítások kezelésére is megoldást nyújt a struktúra, mert ezáltal elkülönítve töltjük fel információval az opcióhalmazt például a konzolról, illetve fájlból. A 3.1. ábrán megjelenő gráf, alsó útja - az options-től a configuration-ig, majd tovább az application-be - a fájlból való konfiguráció, a jobb oldali útja - az options-ből egyenesen az application-be - a parancssori konfiguráció egy interpretációja.



3.2. ábra. A rendszerben előforduló fontosabb objektumok hierachiáját és kapcsolatát szemléltető ábra

3.1.1. Server

A szerver három legnagyobb feladata a socketek kezelése, a folyamatok felügyelete és a beérkező kapcsolatok lebonyolítása. A socketet létrehozása a SocketManager objektum feladata, amelyet a Server objektum konstruálásakor hozunk létre. A Server önmagában a ProcessManager osztályból származik, aminek feladata a folyamathoz kötődő funkciók kezelése, mint például folyamatok elindítása, megállítása. A klienssel való kommunikációt önmagában valósítja meg a Server.

Megjegyzés: Javaslat refaktorizációs fejlesztéshez, hogy az imént említett kommunikációs műveletek leválasztásra kerüljenek a Server objektum törzséről.

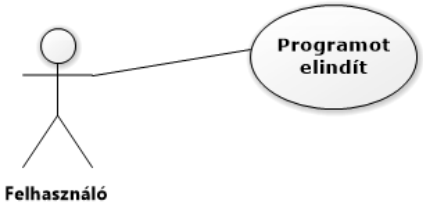
Ezen felül rendelkeznie kell a feladatban definiált naplózási funkciókkal, amelyeket a *Logger* objektum hivatott kezelni. A Logger feladata a megfelelő naplózási opci-

ók (*LogOptions*) szerinti naplóbejegyzések készítése. (Ezen bejegyzéseket szintén a szerver vezérli).

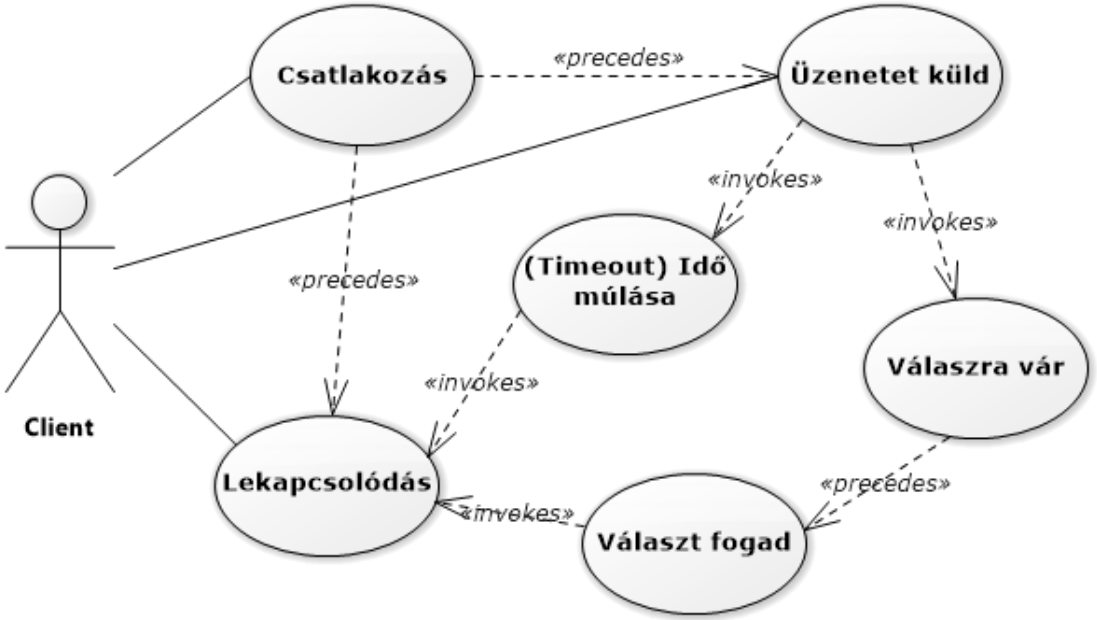
A szerver komponens önmagában használható funkciókat nem biztosít, csak a kliens általi vezérlés lehetséges. A kliensoldalról érkezett üzeneteket JSON-ben kódolva kerülnek a szerveroldalra, ahol a megfelelő feldolgozó műveletek és a válasz előállítása után szintén JSON-ben a szerver válaszol a kliens-nek.

3.1.1.1. Felhasználói esetek

A felhasználói eset diagrammok segítségével postosabb képet kaphatunk a komponens működésével kapcsolatban.



3.3. ábra. Ömagában a szever nem rendelkezik különösebb intrakciót igénylő használati esetekkel, a program elindításán kívül



3.4. ábra. Amennyiben a kliens-t tekintjük actornak, úgy több használati eset is szemléltethető

3.1.1.2. Osztályok

A szerver az lábbi osztályokból épül fel:

ServerApplication Ezen osztály képzi a szerver komponens alkalmazás rétegét. Az objektum feladata, érzékelni a parancssori beállításokat. Ha verziót, kérik le akkor az ApplicationVersion osztályból kinyerhető az aktuális verziószám, ha a súgót kérik le a paraméterekhez írt segédleírás íródik aki a képernyőre. Mindkét esetben a program leáll bármilyen folyamat indítása nélkül. Egyéb esetben a program létrehoz a kinyert opciókból egy ServerConfiguration-t, amelyet aztán a Server létrehozásához használ fel.

ServerOptions Ezen osztály tartalmazza a lehetséges Server beállításait. Az osztály felel a parancssori kapcsolók kinyeréséért is. A legtöbb argumentum rendelkezik alapértelmezett értékkel ami felülírásra kerül, amennyiben egy adott opciót megadunk.

ServerConfiguration Az *AbstractConfiguration* leszármazottjaként, egy *AbstractOptions* paraméterből konstruált osztály, ami felhasználva az aktuális opciókat (konfigurációs fájl) betölti, a fájlban található beállításokat. A ServerOptions, LogOptions, ProcessOptions osztálokat látja el adattal.

Megjegyzés: Az alkalmazás jelenlegi funkcionalitásához képest, a jövőben akár célszerű lehet más formátumú konfigurációs fájlok feldolgozását is támogatni. Ekkor az osztályban egy újabb metódus felvitlére van csak szükség, a `loadFromJson` metódus mintájára.

Server A komponens magját képező objektum. Ezen objektum kezeli a fontosabb rendszernaplókat, és írja azokat a Logger segítségével. Az osztály legofontosabb metódusai a futást jelző `run` és a `runForever`. A run elindítja a szerver által vár működéssorozatot: elindulnak a megfelelő folyamatok és kinyílnak akommunikáció ablakok. A runForever metódus feladata pedig, hogy egy végtelen ciklus segítségével, minden iterációban megvizsgálja a socketeket és a programokat. Továbbá az osztályban definiáltak a beérkező üzeneteket feldolgozó metódusok, ahogyan a kommunikációs interfészt kiszolgáló funkciók is. Ezen kívül a szerver daemonizálható is, így az ezt megvalósító függvény helye is itt van.

ProcessManager Az előbb taglalt Server objektum őse. Az objektum rendelkezik két tárolóval. A *pool* nevű gyűjtőbe, minden megkreált Process objektum bekerül.

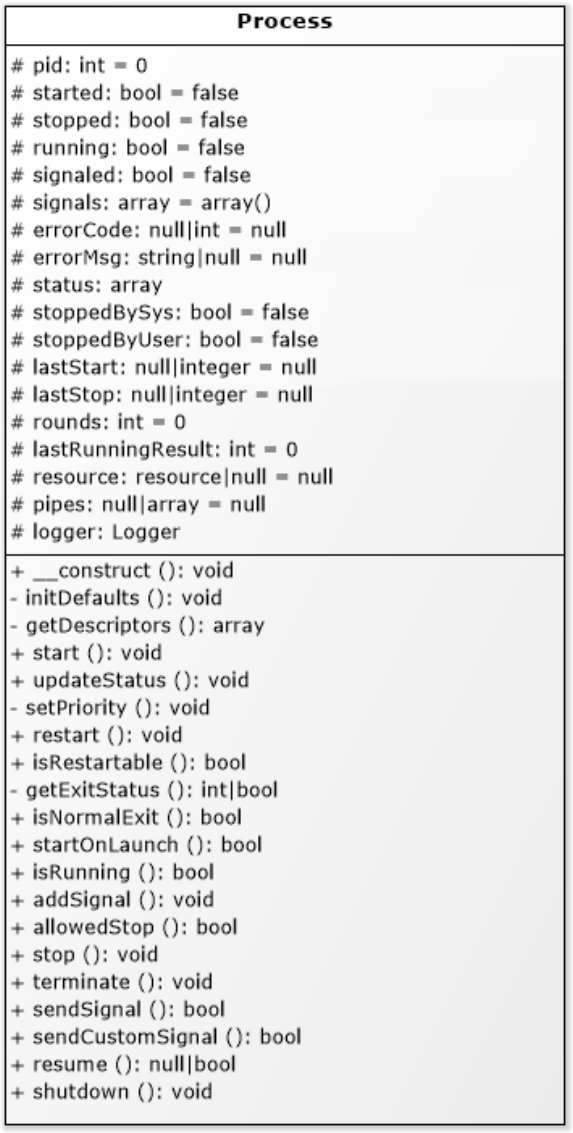
3.5. ábra. A Server osztályt ábrázoló diagramm.

A *runningPool*, pedig az aktuálisan futó folyamatok tárhelye. Ez a réteg biztosítja a Server számára, a megfelelő akciók működését. (elindítás, leállítás, terminálás)

ProcessOptions Ez az osztály reprezentálja, a megadható folyamat-beállításokat. Ezen opció-réteg-beli objektumok feladata, az adott lehetőségek beállítása, illetve validálása.

Process Az egyes folyamatokat képző folyamatokat reprezentálja ez az osztály. Feladata alapvető kezelési funkciók biztosítása, (elindíátsa, leállítása), továbbá a folyamatok állapiotainak változását aktualizáló működés megvalósítása.

ApplicationVersion A szerver alkalmazás verziószámát szolgáltató objektum. A feladata, a verziószám meghatározása. Jelenleg a működése abból áll, hogy a megfelelő verzófájlból kiolvassa az verziót képző string értéket.



3.6. ábra. A Process osztály fontosabb metódusait ábrázoló diagramm.

Megjegyzés: Fejlesztési lehetőségként feltüntetném, hogy az osztály könnyen tovább bővíthető lehetne, valamely olyan szogáltatással, amivel a megfelelő verzióérték automatikusan lekövetését garanatálja. Ezzel tulajdonképpen a verzió, verziófájlba történő írását automatizálnánk.

SockerManager A socketek létrehozásáért felelős manager ostály, ami a socket objektumokkal kapcsolatos kollektív műveleteket implementálja, mint például a konfiguráció alapján való létrehozás, beállítás vagy hallgatás (STREAM_SOCKET esetén). Az objektum naplózási funkcióit a Server objektumttól átvett Logger osztály segítségével valósítja meg. Az objektum a socketek felépítésekor, a megadott beállításokon kívül a nem blokkoló tulajdonság beállítását is eszközöli. Amelyre azért van szükség, hogy a szerver alapműködését garantáló főciklus, futása ne blokkoljon,

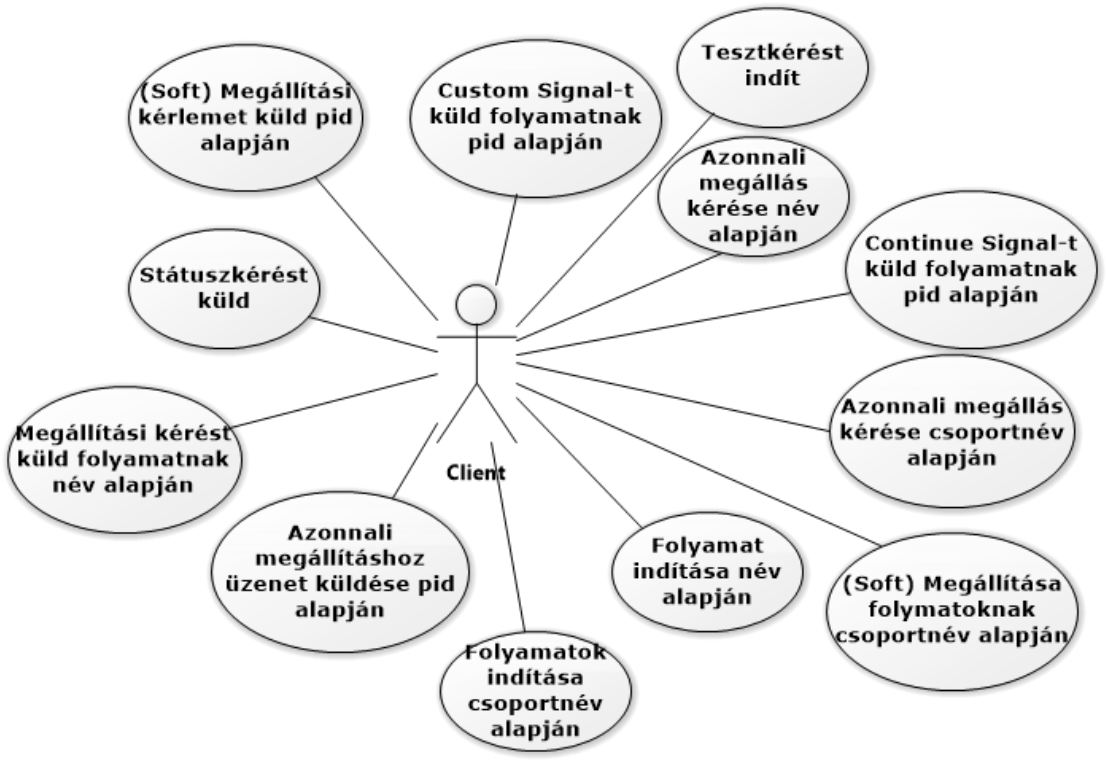
amikor a socketen kömmunikáció zajlik. Az osztály a külső (*External*) könyvtár *Factory*-ából származik le. Ennek oka, hogy a megfelelő socket beállítások összegyűjtése után sorban létrehozza a Socketeket, amelyeket aztán az osztályban megvalósított tárolóban helyez el.

Megjegyzés: Az alkalmazás egyéb fontosabb objektumai mind valamely Options példányban implementált validálási és feldolgozó logika után kapnak értéket, majd kerülnek alkalmazásra. A kevés számú opcióra való tekintettel ezen objektum eseténben ez a *parse logic* ebben az osztályban valósul meg. A cél egy későbbi *SocketOptions* objektummal való együttműködés lenne.

LogOptions Ezen objektum hivatott, az egyes naplózási lehetőségeket összefogni. Az általa definiált lehetőséget több helyen is megjelennek az alkalmazásban. A Server komponens alapvető naplóin kívül (ServerOptions-nek is része egy ilyen LogOptions példány), a folyamatoknak is saját naplói kell legyenek. Az objektum segítségével, megvalósítható az alkalamzás bármely területén, a megfelelően konfigurált Logger objektum létrehozása. Jelenleg az említett két kontextusra felkészítő logikát valósít meg, miszerint elkülönül a Server objektum naplóihoz, illetve a gyermek folyamatok naplóihoz szükséges adatok feldolgozása.

Logger A naplózást végző objektum. Az előre definiált naplózási szintekhez tartozó bejegyzések kerülnek a naplóba program futása során. A napló lehetőséget nyújt biztonsági mentések ("backup") készítésére, ezen felül a `cleanUp` funkcióval, a gyermekeknek létrehozott további naplók törölhetők, a program új indulása esetén. A már említett `printLog` funkció, a nem demonizált futtatás esetén használható arra, hogy a naplóba rögzíteni kívánt bejegyzést a képernyőre is kiírja a program (Ellenkező esetben a megfelelő argumentum, automatikusan letiltásra kerül).

3.1.1.3. Felhasználói esetek



3.7. ábra. A teljes kommunikációs interfész funkcionalitását bemutató ábra

3.1.2. Client

A kliens komponens legfőbb feladata, hogy a feladat által megfogalmazott kommunikációs interfészt biztosítsa a felhasználójának. Ebbe beletartozik, a kapcsolat megvalósítása socketen keresztül, továbbá az is, hogy felhasználói felületet biztosítson a program kezeléséhez.

3.1.2.1. Osztályok

A kliens az lábbi osztályokból épül fel:

ClientApplication A ServerApplication osztály mintjárára, a kliens komponensben ez az osztály képezi az alkalmazásréteget. A feladata, hogy az érékelt parancssori paraméterek megadása után, a konfigurációs állományt feldolgozva olyan állapotba juttassa az alkalmazást, hogy az a működés magját alkotó ciklus számára alkalmas legyen. A főciklus feladata, hogy minden egyes kívánt iteráció esetén egy parancsot hajtson végre. Ezen parancsok az iteráció elején képernyőre írt menü alapján vezérelhetők. A paraméter táblázatból (2.4.2.2. bekezdés) látható, hogy a *help* kapcsoló

megadása esetén, a program a fentebb említett ciklust mellőzve, a program paraméterezésének kiírása után leáll. Minden egyéb funkció esetén, a program a ciklusba belép, és a megfelelő navígációs utasításra vár. Az itt megadható lehetőségeket vezérlését, az ezekre kapott válaszok feldoglozását és a megfelelő I/O műveletek végrehajtását is ezen réteg adja. Lényegében a felhasználó ezen a rétegen keresztül vezérli a rendszert.

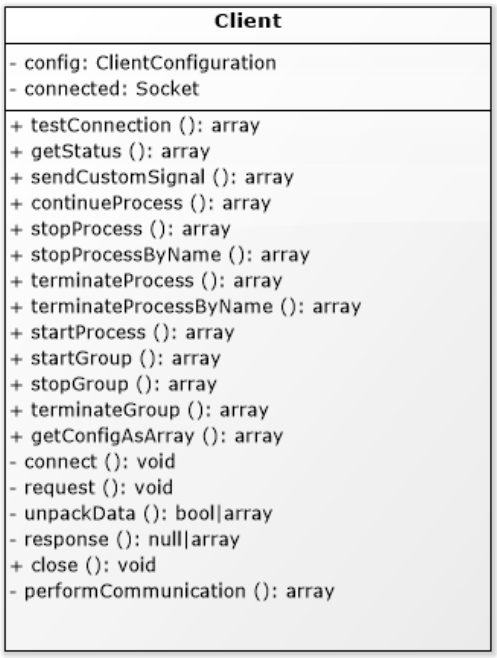
ClientOptions A kliens komponens esetében, is azon alapelvek érvényesek amelyek a szerver komponens esetében meg lettek határozva. A működés magját képező Client objektum is bizonyos opciók alapján inicializálja jellemző működését. Ezen opciókból itt lényegesen kevesebb található, mint a szerver béli megfelelőjében. Ugyanis a megfelelő csatlakozási címen kívül, az alkalmazás konfigurácós állományának illetve a hitelesítéshez szükséges felhasználónév és jelszó páros meghatározására van lehetőség.

ClientConfiguration Ezen osztály a ClientOptions segítségével jön létre, célja, hogy a megadott konfigurációs fájlban található beállításokat betöltse a megfelelő Options argumentumba.

Megjegzés: Itt is megjegezném, hogy a konfigurációs réteg kialakításakor figyelembe lett véve, hogy esetleg JSON-től különböző állomány betöltésére is legyen a későbbiekben lehetőség.

Client A komponens központi működését szolgáltató objektum, aminek a feladata a kommunikációs réteg magvalósítása. Fontosabb funkciója a `performCommunication`, ami egy kommunikációs kapcsolat életciklusát implementálja. Azaz kapcsolódik a kívánt címre, majd a kiválasztott menü alapján, a megfelelő üzenetet küldi el, illetve fogadja a választ, és dekódolja ClientApplication számára értelmezhető formátumra. Ezen kommunikációs ciklusok körökre osztottak, minden egyes kérés küldésre, pontosan egyetlen választ vár, majd miután azt megkapta bontja a kapcsolatot.

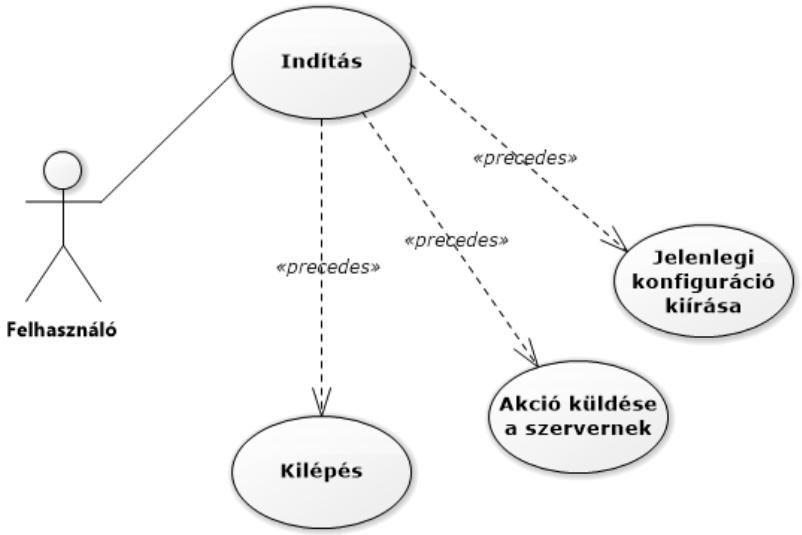
Megjegyzés: Minden szerver felé közölt kérés rögzített kódú paranccsal történik, és ezáltal az átküldött üzenet mérete becsülhető felső korláttal. Ezt használja ki a Server komponens a kommunikációhoz, mert az az objektum 1024 bájtban meghatározott fogadási bufferen várja a kódolt üzenetet. Ugyanakkor, mivel a válaszüzenet mérete jóval nagyobb lehet ennél, ezért a kliens oldalon, ciklusban fogadjuk a socketen érkező adatokat, ugyanúgy 1024 bájt méretű buffer segítségével.



3.8. ábra. A Client osztályt ábrázoló diagramm.

Megjegyzés: A kommunikáció köztes pontján, amikor a szerver dolgozik, a kliens legfeljebb 60 másodpercig vár arra, hogy a megfelelő socket olvasásra felszabaduljon.

3.1.2.2. Felhasználói esetek



3.9. ábra. A kliensprogram használati esetei.

3.2. Megvalósítás

A rendszer implementációja során a magasszintű konfigurálhatóság, illetve az objektum elvű szemléletek voltak előtérbe helyezve. A megvalósított konzolos alkalmazások indító szkriptjei, a megfelelő már bemutatott objektumot példányosítva majd azt elindítva működnek. A névterek kialakítása a PSR-4-es szabvány szerint történt, és az ezeket feloldó *AutoLoader* objektum, a statikus metódusai segítségével regisztrálja a megfelelően hivatkozott névtereket. Az osztály implementációja, a *src* mappa gyökerében található.

Megjegyzés: A manapság szokásos standardok szerint, dependency-managerre bízott telepítés integrálásával, a rendszerből kiemelhetővé (dependency-vé) válik az External könyvtár alatt található forrás-projekt. Ez egy későbbi fejlesztési időszak tárgyát képezheti. (Pl.: Composer alá)

Kommunikáció A kommunikációt a kliens komponens kezdeményezi. Jelenleg a datagram és a stream alapú socketek kommunikációja lehetséges a rendszben. Ezen socketek a kapcsolat létrejötte után, kérdés-válasz jellegű adatcserét végeznek. A szerveroldali socketek, legfeljebb 1024 bájt méretű adatban fogadják az üzeneteket. A kliens oldali socket időkorlátos megvalósítással rendelkezik, ami azt jelenti, hogy amennyiben a megadott időn belül (60 másodperc) nem olvasható a válasz a socketen, bontja a kapcsolatot. A kommunikáció során, az egyes felek leellenőrzik, hogy a teljes üzenet átérkezését, s amennyiben ez teljesül, sikeresnek tekinthető az üzenetfogadás.

Üzenetek Az üzenetek alapvetően string formába csomagolt adatok. Ezen string egy mindkét komponensben definiált tokennel kezdődik illetve végződik. A token (tag) közrezárja az egy üzenetben lévő `json_encode` által szöveggé alakított tömböt. A tömb reprezentálja az üzenetet, aminek tartalma a végrehatjani kívánt akció neve, annak paramétere (ha van), továbbá a konfigurációban meghatározott felhasználónév és jelszó. (A kiolvasási buffer mérete, ami miatt a megfelelő szöveges paraméterek (name, groupname), a felhasználónév és a jelszó is karakterlimittel van validálva.) Az üzenetek fogadó felén, az említett token erősíti meg az üzenet megérkezését, majd a token-t eltávolítva az üzenet elejéről és végéről, a tényleges adat is kinyerhető. (Az adat `json_decode` segítségével kerül elemezhető formába.)

Egy lehetséges üzenetre példa:

```
<PHPVisor>{"action":"test", "params":[], "username": null, "password": null}</PHPVisor>
```

3.2.1. Server

A szerver komponens megvalósítása közben felmerül egy-két olyan fontosabb probléma, amelyekről érdemes említést tenni.

Az öröklődési probléma Az alábbi jelenség, amiről szó van a folyamatok új számtási szálra való lehelyezése miatt fordulhat elő. A probléma az, hogy a `proc_open` függvény, a *fork* alkalmazásával helyezi le az új szálra a megadott futtatandó parancsot. A fork működése miatt [7], a gyermekfolyamatok megöröklik a szülőfolyamat fájlleíróit. Ez azt jelenti, hogyha valamilyen oknál fogva a program futásakor, a szűlőfolyamatot lelövi az operációs rendszer, de egy gyermekfolyamat még várakozik (és fut), akkor az a folyamat megörökli a nyitott socketekhez tartozó leírókat is. Ezáltal a socketet még nyitottnak látszódnak, pedig tulajdonképpen nincs mögöttük a működést biztosító szál.

Mivel a PHP jelenleg nem nyújt lehetőséget a megfelelő közbenjárásra a problémával kapcsolatban, ezért csak *workaround* eszközölhető. Ez eset miatt, a szerver indulásakor elinduló folyamatokat előbb indítjuk, mint a socketeket, de ez nem valódi megoldás a problémára. A megoldáshoz az FD_CLOEXEC descriptor flag alkalmazására lenne szükség, amellyel elérhető, hogy a megfelelő leíró bezárásra kerüljön, és ne történjen meg az öröklődés.

Megjegyzés: Amennyiben a PHP lehetőséget tud nyújtani, a fent említett problémára a programot érdemes továbbfejleszteni, a megjelölt hibalehetőség kizárásával.

Folyamatok kezelése A folyamatok alapvető kezelését PHP-s függvényekkel oldottam meg. A folyamatok létrehozását a `proc_open` függvény végzi. A `proc_close` és a `proc_terminate` függvények segítségével a normális (soft), illetve az azonnali (hard) leállítás funkciók is implementálásra kerültek. A kiválasztott futtatandó parancsot, megfelelő `escapeshellcmd` függvényen átvezetve passzoljuk a `proc_open` számára. A folyamatok indításuk után folyamatosan vizsgálódnak, a `proc_get_status` függvény segítségével. Bizonyos értékekre, a függvény nem aktuális adatokat szolgáltat, így ezt szem előtt tartva lett megvalósítva a Process objektum updateStatus, státusz frissítést végző metódusa. Ezen státusz változása esetén, a megfelelő állapotot kifejező argumentum is változtatásra kerül az adott Process objektumban, hogy

az, megfeleltethető legyen a folyamat aktuális állapotával.

A demonizáció A demonizáció algoritmusa a szakirodalmakban megtalálható lépéseket követi. A demonizáció lényege, hogy a háttérbe szorítani kívánt futási szálat leválasztjuk az aktuális terminálról, ezzel háttérbe helyezve.
A lépések a következők[8]:

1. Fájlleírók bezárása
2. Vezérlő terminálról való leválasztása
3. Munkamenet és folyamatcsoportok elvesztése
4. Aktuális munkakönyvtár megváltoztatása
5. Újraállítani a fájlok létrehozási maszkját.
6. Gyermekkel kapcsolatos szignál kezelése. (SIGCHLD)

Ezen általános leírás nem minden lépésére van szükség a tényleges implementációban. A fájlleírók és a terminálról való leválasztás után (`posix_setsid` függvény), a Server alkalmazás a megadott könyvtárba navigál, majd elkészíti a program pid fájlját, amibe a demonizált folyamat pid-át írja bele. Végül alkalmazza a megadott umaszk értéket, amennyiben erre van jogosultsága a futtató felhasználónak.

Megjegyzés: W. Stevens az Advanced Unix könyvében, SVR4 okokra, hivatkozva a SIGHUP figyelmen kívül hagyását és újra-forkolást javasol a setsid hívást követően (13.3 fejezetben), de ezen működés nem része az implementációnak.[9]

3.2.2. Client

A kliens lényegében egy komolyabb mechanizmust valósít meg, ami a kommunikáció. Önmagában a program célja, hogy egy kezelhető felületet valósítson meg felhasználója számára, amivel vezérelni tudja a szerver komponenst.

Kommunikáció A kommunikációs réteg megvalósításáról már több szó is esett. A program rögzített karakterszámú üzeneteket továbbít, illevte időkorláttal vár az érkező válaszra, aminek eredményéről és sikerességéről a felhasználó számára kiírt üzenetekben tájékoztat. Az egyetlen érdekesnek mondható metódus, az adatok fogadása, mivel nem tudhatjuk megkkora választ kell fogadnunk. Ezt egy ciklusos olvasással oldjuk meg, a PHP-s `socket_recv` függvény felhasználásával, amely megkülönbözteti, azt aktuálisan nem kapott értékek esetét az üzenet végét jelentő esetektől. (Egy beolvasás kör bufferének mérete itt is 1024 bájt.)

4. fejezet

Tesztelés

A programok tesztelését kézzel végeztem, de megtalálható több speciális konfigurációs állomány a *test* könyvtár alatt, amelyek segítségvél ellenőrizhető a program egy-egy opció alkamazása esetén. A rendszer megfelelő működéséről funkciói segítségével tájékozódhatunk. A szerver esetén a képernyőre vagy naplóba írt bejegyzések segítik a futás nyomon követését, míg a kliensnél a képernyőre írt információk teszik egyértelművé a futás állapotait.

Megjegyzés: Egy PHP-s tesztelést segítő keretrendszer integrálásával, későbbi fejlesztések alkalmával sor kerülhet egységtesztek bevezetésére. A PHPUnit[10] egy szabadon használható szoftver, amely támogatja objektumok egységteszteinek írását *assert*-ek segítégével.[1]

4.1. Server

4.1.1. Feketedoboz tesztesetek

- Nem JSON konfigurációs fájl (nonjson.test)
- Nem megfelelő környezetben való futtatás
- Üres fájl (empty.test)
- Üres JSON fájl (empty.test.json)
- Invalid érték a logikai értékek helyett (invalidValues.test.json)
- Invalid érték a szám értékek helyett (invalidValues.test.json)

[1] Az egyes tesztesetek a *test* könyvtárban találhatóak

4.1.2. Fehérdoboz tesztesetek

- Minimális kötelező konfigurációval való működés (requireds.test.json)
- Hiányzó socket konfiguráció (misssock.test.json)
- Hiányzó process konfiguráció (missproc.test.json)
- Hiba esetén a naplóba íródik a megfelelő hibaüzenet. (misssock.test.json)
- Túl hosszú limitált érték (pl. name) (tooLongName.test.json)
- Megfelelő újraindítása szabálok lépnek életbe, a program egy lefutásakor (run.test.json)

4.2. Client

4.2.1. Feketedoboz tesztesetek

- Hibás parancssori konfiguráció beütése
- Nem JSON konfigurációs fájl (nonjson.test)
- Üres fájl (empty.test)
- Üres JSON fájl (empty.test.json)
- Túl hosszú felhasználónév megadása fájlban (tooLongUsername.test.json)
- Túl hosszú akcióparaméterek tesztje
- Hibás menüopció választása

4.2.2. Fehérdoboz tesztesetek

Mivel a kliensnek meglehetősen kevés konfiguációs beállítása van, ezért az itt ugyanúgy érvényes teszteseteket nem sorolom fel. A kliens funkcionális tesztelése keretében a letesztelt funkciók:

- Test menü működése
- Custom menü működése
- Folyamatok indítása pid/név/csoportnév alapján
- Folyamatok leállítása pid/név/csoportnév alapján

- Folyamatok terminálása pid/név/csoportnév alapján
- Folyamat folytatása Stopped állapot esetén
- Állapot lekérdezése
- Kapcsolat megszakadása esetén érkező hibák

4.3. Hatékonyság

A program hatékonyságát több szempont alapján elemezhetjük. A kommunikációs részprogram erőforráskímélő, mivel a randevú a lehető legrövidebb tart. Ugyanakkor a hálózat paraméteritől is függ, hogy milyen sebességgel képes továbbítani az adatokat az adott komponens. A folyamatkezelő mechanizmus a futó folyamatokat, pid-re kulcsolt tömbben tárolja, így azok elérése egy-egy interakció esetén a lehető leggyorsabb. Ugyanakkor az összes folyamatot tároló tömb nem kulcsolt, így név vagy csoportnév alapján folyamatok meghatározása keresést igényel. A rendszer megvalósítása során, a már nem használt objektumok/referenciák felszabadításra kerülnek, ezáltal a PHP nyújtotta Garbage Collector szolgáltatásra bízva a takarítást.

5. fejezet

Zárszó

A PHPVisor megszületését, az ösztönözte, hogy komplexebb összetételű alkalmazások működése során külső, illetve független programkomponensek lefutására lehet szükség. Az egyes futtatandó programok, ideális esetben ritkán változnak, ezzel ellentétben a lefutásukra jelentkező igény különféle üzleti helyzetekben, különféle ütemezést jelenthet. Ilyen esetekben egy megfelelő folyamatkezelést, vezérlést biztosító program megléte rendkívül megkönnyítheti az üzemeltetést. A program implementációjából fakadóan, minimális függőséget jelenthet alkalmazási környezettől függően.

A tervezett céljaim a program megvalósítása során teljesültek. Az eleinte kitalált funkcionalitások implementálásra kerültek, továbbá olyan funkciók is megvalósültak, amelyek a kényelmesebb használatot teszik lehetővé. Az alapvető irányvonal a fejlesztés során az volt, hogy egy könnyen kezelhető (alacsony szintű) programot hozzak létre, ami magasszintű konfigurálhatóságot biztosít a felhasználójának. A fejlesztés jelenlegi iránya, hogy még több olyan funkció kerüljön a rendszerbe, amelyek elősegítik a gördülékeny használatot.

A jövőre mutató tervek között szerepel egy automatikus verziózó szolgáltatás implementálása a szerver komponens számára, de egy alternatív kliens is, ami egy grafikus felületű weboldalt szolgáltat a program kezeléséhez. További fejlesztési munkálatok során ki lehetne terjeszteni a szerver kommunikációs lehetőségeit más specifikusabb protkollokra, HTTP-n keresztüli üzenetváltási standardeket implementálni, esetleg egy összetettebb folyamatcsoportokat kezelő logikát integrálni.

6. fejezet

Irodalomjegyzék

[1] Supervisor honlapja és GitHub oldala:
http://supervisord.org/ (2018. április 29.)
https://github.com/Supervisor/supervisor (2018. április 29.)

[2] PHP CLI honlapja:
http://php.net/manual/en/features.commandline.php (2018. május 03.)

[3] PHP PCNTL:
http://php.net/manual/en/book.pcntl.php (2018. május 03.)

[4] PHP POSIX:
http://php.net/manual/en/book.posix.php (2018. május 03.)

[5] PHP Socket:
http://php.net/manual/en/book.sockets.php (2018. május 03.)

[6] Clue GitHub oldalán a php-socker-raw library:
https://github.com/clue/php-socket-raw (2018. május 05.)

[7] Fork manual oldala:
https://linux.die.net/man/2/fork (2018. május 05.)

[8] Daemonizálási lépések leírása az UWE oldalán:
http://www.cems.uwe.ac.uk/~irjohnso/coursenotes/lrc/system/daemons/d3.htm (2018. május 06.)

[9] W. R. Stevens: Advanced programming in the UNIX environment - Third Edition -, Addison Wesley, 2013, [994], ISBN-13: 978-0-321-63773-4

[10] PHPUnit tesztelési keretrendszer oldala:
https://phpunit.de/ (2018. május 09.)