UNIVERZA V LJUBLJANI
FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

Tadej Kanižar

VZPOREDNO IZVAJANJE UKAZOV V LUPINAH RAČUNALNIKOV SISTEMA PLANETLAB

DIPLOMSKO DELO NA
VISOKOŠOLSKEM STROKOVNEM ŠTUDIJU

MENTOR: doc. dr. Boštjan Slivnik

Ljubljana, 2010

Univerza
v Ljubljani

Fakulteta za računalništvo
in informatiko

Tržaška 25
1000 Ljubljana, Slovenija
telefon: 01 476 84 11
faks: 01 426 46 47
www.fri.uni-lj.si
e-mail: dekanat@fri.uni-lj.si

Št. naloge: 00520/2010
Datum: 05.04.2010

Univerza v Ljubljani, Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Kandidat: **TADEJ KANIŽAR**

Naslov: **VZPOREDNO IZVAJANJE UKAZOV V LUPINAH RAČUNALNIKOV SISTEMA PLANETLAB**

PARALLEL EXECUTION OF SHELL COMMANDS ON PLANETLAB NODES

Vrsta naloge: Diplomsko delo visokošolskega strokovnega študija

Tematika naloge:

Programe na porazdeljenem sistemu PlanetLab tipično poganjamo z uporabo računalnika, ki ni vključen v PlanetLab, zato dostopamo do teh računalnikov preko storitve SecureShell (SSH). Sestavite vmesnik, ki omogoča vzporedno izvajanje istega ukaza v ukazni lupini na večih računalnikih sistema PlanetLab. Vmesnik naj omogoča tudi vzporedni prenos datotek med uporabnikovim računalnikom in računalniki v sistemu PlanetLab ter zaznavanje napak in upravljanje z oddaljenimi procesi.

Mentor:
B. Slivnik
doc. dr. Boštjan Slivnik

Fakulteta za računalništvo in informatiko • UNIVERZA V LJUBLJANI • 2

Dekan:
prof. dr. Franc Solina

I Z J A V A O A V T O R S T V U

diplomskega dela

Spodaj podpisani/-a Tadej Kanižar,

z vpisno številko 63030111,

sem avtor/-ica diplomskega dela z naslovom:

Vzporedno izvajanje ukazov v lupinah računalnikov sistema PlanetLab

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal/-a samostojno pod mentorstvom (naziv, ime in priimek)

  doc. dr. Boštjan Slivnik

  in somentorstvom (naziv, ime in priimek)

- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela

- soglašam z javno objavo elektronske oblike diplomskega dela v zbirki »Dela FRI«.

V Ljubljani, dne _____ Podpis avtorja/-ice: _____

ZAHVALA

Zahvalil bi se vsem, ki so me med vsemi leti študija podpirali in nad mano niso obupali.

KAZALO

SEZNAM UPORABLJENIH KRATIC IN SIMBOLOV

BASH	*Bourne-again shell.* Lupina v sistemih Unix (in podobnih).
FTP	*File transfer protocol.* Protokol za prenos datotek.
HUP	Okrajšava za *SIGHUP*. Signal, poslan procesu, ko se terminal, ki ga nadzoruje, zapre.
IFS	*Internal field separator.* Znak oz. znaki, ki določajo presledek med vrednostmi. Privzeto sta to ponavadi presledek in nova vrstica.
PEER-TO-PEER	Distribuirana aplikacijska arhitektura, ki razdeli naloge med vrstniki. Vsak opravi en del skupne naloge, tako je delo opravljeno hitreje.
SCP	*Secure copy*. Varen prenos datotek med lokalnim in oddaljenim računalnikom oz. med dvema oddaljenima računalnikoma.
SSH	*Secure shell*. Omrežni protokol, ki omogoča varno izmenjavo podatkov med dvema povezanima napravama.
STDERR	*Standard error*. Izhodni podatkovni tok, tipično uporabljen za izpis sporočil napak ali diagnostiko.
STDIN	*Standard input*. Vhodni podatkovni tok, pogosto tekst.
STDOUT	*Standard output*. Izhodni podatkovni tok, kamor programi izpisujejo svoje izhodne podatke.
XML	*Extensible markup language*. Preprost računalniški jezik, podoben *HTML*-ju, ki nam omogoča opisovanje strukturiranih podatkov.
XML-RPC	Protokol za klic oddaljene metode. Za kodiranje klicev uporablja *XML*, kot transportni mehanizem pa *HTTP*.

POVZETEK

Cilj te diplomske naloge je rešitev problema vzporednega izvajanja programov na računalnikih sistema *PlanetLab*. Tipično se ti programi izvajajo iz računalnika, ki ni povezan v ta sistem.

Uporabniki sistema *PlanetLab* trenutno prenašajo in izvajajo programe ročno. Ker je število računalnikov, ki so nam na voljo v naši rezini, ponavadi več 100, je upravljanje z njimi dolgotrajno opravilo. V tem leži naš problem.

Za namene rešitve tega problema je potrebno vzpostaviti skripte za prenos programov in podatkov na vse računalnike ter izvajanje teh programov. Po končanem izvajanju prenesemo rezultate na računalnik uporabnika. Če se kakšen program ni končal, ga moramo prisilno končati in le-to izpisati na ekran kot napako. Prav tako moramo izpisati napako, če se na računalnik nismo mogli povezati oz. je prišlo do kakršnega koli problema.

Povezovanje in izvajanje programov poteka preko protokola *SSH*, prenašanje datotek pa preko protokola *SCP*.

Naš problem smo rešili s tremi skriptami, napisanimi v ukazni sintaksi lupine *BASH*. S tem je omogočeno enostavno in hitro komuniciranje, z mnogo računalniki sistema *PlanetLab* hkrati.

KLJUČNE BESEDE

PlanetLab

Bash

Ssh

Scp

Vzporedno izvajanje

ABSTRACT

The aim of this thesis is to solve the problem of parallel execution of programs on the computers of *PlanetLab*.

Users of *PlanetLab* must currently transfer and run the programs by hand. As the number of computers available to us, in our slices, is usually over 100, managing them can be time-consuming. In this lies our problem.

For the purposes of the solution to this problem, it is necessary to set up scripts to transfer programs and data to all the computers as well as executing those programs. After the completion of execution, we transfer the results to the user's computer.
If a particular program failed to stop, we have to force it to end and output that onto the screen as an error. We also need to display an error in case we failed to connect to a computer or any other problems were encountered.

Connecting and executing of programs is done via *SSH* protocol, while transferring the files is done via *SCP* protocol.

We solved our problem with three scripts written in the command syntax *BASH*. This allows for a quick and easy communication with many *PlanetLab* computers simultaneously.

KEYWORDS

PlanetLab

Bash

Ssh

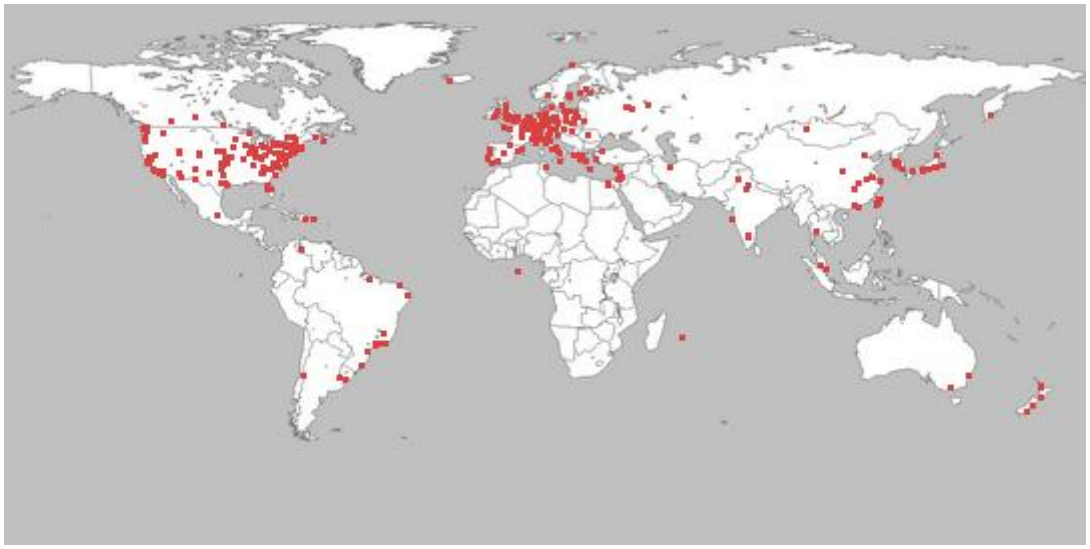
Scp

Parallel execution

1. UVOD

Razvoj novega, boljšega interneta.
Drzen stavek, a ravno s tem se vsak dan ukvarja mnogo raziskovalcev po celotnem svetu. Večina od nas jemlje internet že kot nekaj vsakdanjega, kot nekaj, kar preprosto obstaja, in obstaja samo zato, da služi nam.
A resnica je drugačna. Izkaže se namreč, da je arhitektura, in storitve, ki tečejo na njej, zelo stara, saj ima korenine že v 60. letih. Nekatere storitve (npr. *ftp*) so se v podobni obliki, kakršno imajo še danes, pojavile že v 70. letih. [1]
Problemi se pojavijo, ko upoštevamo neverjetno rast interneta. Ocena o populaciji interneta, narejena 30. junija 2010, ocenjuje **število uporabnikov na 1,97 milijarde**. [2]. Spletna stran www.youtube.com je že presegla **2 milijardi obiskov dnevno** [3].
Takšne številke hitro pokažejo, da se morajo trenutne omrežne storitve že zelo truditi, da bi obdržale vse skupaj v delujočem stanju. In ravno to je razlog, zakaj potrebujemo nove spletne storitve, ki bodo bolj prilagojene takšnim uporabam.
Teoretični razvoj takšnih storitev poteka neprestano, a zatakne se pri praktičnem razvoju. Problem je namreč v tem, da je zelo težko razvijati storitve, ki bodo podpirale 2 milijardi uporabnikov, in te storitve testirati, saj ima večina raziskovalcev na voljo le nekaj računalnikov, pa še to v istem omrežju, itd. Nesmiselno je seveda pričakovati, da bi takšno, zelo omejeno, testiranje delovalo tudi na ravni celotnega interneta.
Na srečo so ta problem nekateri raziskovalci ugotovili že pred več leti.

Na tej točki se leta 2003 pojavi projekt *PlanetLab*, ki omogoča raziskovalcem dostop do naraščajočega številka računalnikov, porazdeljenih po celotnem svetu. V času pisanja te diplomske naloge, obsega ***PlanetLab* 1125 računalnikov, na 511 različnih lokacijah** [4].



Slika 1: Porazdelitev lokacij računalnikov, vključenih v *PlanetLab*

PlanetLab je torej porazdeljen sistem računalnikov, s katerim rešujemo probleme, povezane z omrežji.

Podobna omrežja so še SETI@Home, Folding@Home, FightAIDS@Home in podobno (glej http://en.wikipedia.org/wiki/List_of_distributed_computing_projects).

A v nasprotju s SETI@Home, in podobnimi sistemi, pa *PlanetLab* računalniki ne prevzemajo svojih nalog avtomatično, pač pa je potrebno programe in podatke na vsakega izmed njih prenesti ročno, ponavadi s pomočjo orodij *ssh* in *scp*. Nekaj orodij za pomoč pri tem opravilu, ki bodo bolj podrobno predstavljena v nadaljevanju, že obstaja, a nisem našel nobenega, ki bi zadovoljil podane potrebe po enostavnosti in nadzoru.

Moja diplomska naloga torej poskuša rešiti problem upravljanja z računalniki sistema *PlanetLab*.

2. SISTEM *PLANETLAB*

2.1. Kaj je *PlanetLab*?

Sistem *PlanetLab* raziskovalcem po vsem svetu omogoča lažje testiranje in razvoj novih povezanih storitev. Od začetka projekta, leta 2003, si je z njim pomagalo več kot 1.000 raziskovalcev, ki so ga uporabili za razvoj novih tehnologij za distribuirane shranjevanje podatkov, kartiranje omrežij, *peer-to-peer* sisteme, in podobno. [4]

PlanetLab je v osnovi samo skupek računalnikov, na katerih teče operacijski sitem *Linux*, virtualizacijo pa ponuja preko projekta *Linux-VServer*. To zagotavlja popolno (kolikor je mogoča) neodvisnost več uporabnikov na danem računalniku. Vsak lahko namešča programe, spreminja datoteke in podobno, kot želi, pri tem pa ne moti ostalih uporabnikov.
Za administrativno upravljanje s temi računalniki, se uporablja sistem *MyPLC* [5]. Le-ta omogoča osnovno postavitev strežnika, kompatibilnega s sistemom *PlanetLab*.

Slika 2: *PlanetLab* logotip

2.2. Osnovni problem

Osnovni problem, na katerega naletijo vsi uporabniki sistema *PlanetLab* je, kako na vse te računalnike najlažje in najhitreje naložiti potrebne programe in podatke, kako jih nato zagnati ter seveda, kako pridobiti rezultate izvajanj teh programov.
Ponavadi morajo uporabniki namreč uporabljati 100 in več računalnikov tega sistema, kar ni najlažje opravilo, če sta njihovi orodji samo *ssh* in *scp*.

2.3. Obstoječe rešitve problema

2.3.1. Splošen opis

Seznam obstoječih orodij za upravljanje z računalniki sistema *PlanetLab* se nahaja na spletnem naslovu http://www.planet-lab.org/tools. V sledečih točkah bom na kratko opisal nekaj tistih, ki so po značilnostih najbolj podobna naši ciljni rešitvi – enostavnost, napisana v programskem jeziku *bash* (če je mogoče), hitrost (dosežena z vzporednostjo povezav), izvajanje arbitrarnih programov.

2.3.2. pShell

pShell je vmesnik, podoben ukazni vrstici za operacijski sistem *Linux*, ki ponuja nekaj osnovnih ukazov za delo z računalniki sistema *PlanetLab*, med drugim *slist*, s katerim dobimo seznam vseh vozlišč (drugo ime za računalnike) znotraj dodeljene rezine (angl. s*lice*), *put* in *get*, ki nam omogočata prenašanje poljubnih datotek iz našega računalnika na oddaljeno vozlišče in nazaj, *cmd* in *chkproc*, s katerima lahko poganjamo programe in pregledujemo njihove statuse. [6]

pShell je razvit v programskem jeziku *Python*, hoteli pa smo imeti, če je le mogoče, skripte s čim manj odvisnosti od drugih paketov.

2.3.3. Nixes

Nixes je skupek *bash* skript za namestitev, vzdrževanje, upravljanje z in spremljanje stanja aplikacij. Za to si pomaga z orodji *yum*, *gzip* in *java*.
Privzeto deluje s 30 paralelnimi procesi, kar poveča performančno zmogljivost orodij.

Tukaj je problematična predvsem uporaba orodij *yum* in *java*. Dokumentacija orodja pravi, da je zaradi licence *jave* potrebno njen namestitveni paket prenesti na nek lasten strežnik, ta skripta pa nato *javo* namesti iz tega paketa [7].
Poleg tega nekateri poročajo, da je *yum* spominsko zelo zahteven, in je zato nepriporočljiv na računalnikih sistema *PlanetLab*.

2.3.4. pssh

Orodja *ssh*, *scp*, *rsync*, *nuke* in *slurp* so pri povezovanju na računalnike sistema *PlanetLab* precej uporabna. A ne toliko, kot bi lahko bila, če bi imela možnost delovati v vzporednem načinu.
Ravno to rešuje paket *pssh*, saj vsebuje vsa omenjena orodja, v paralelni verziji. Napisana so v programskem jeziku *Python*. [8]

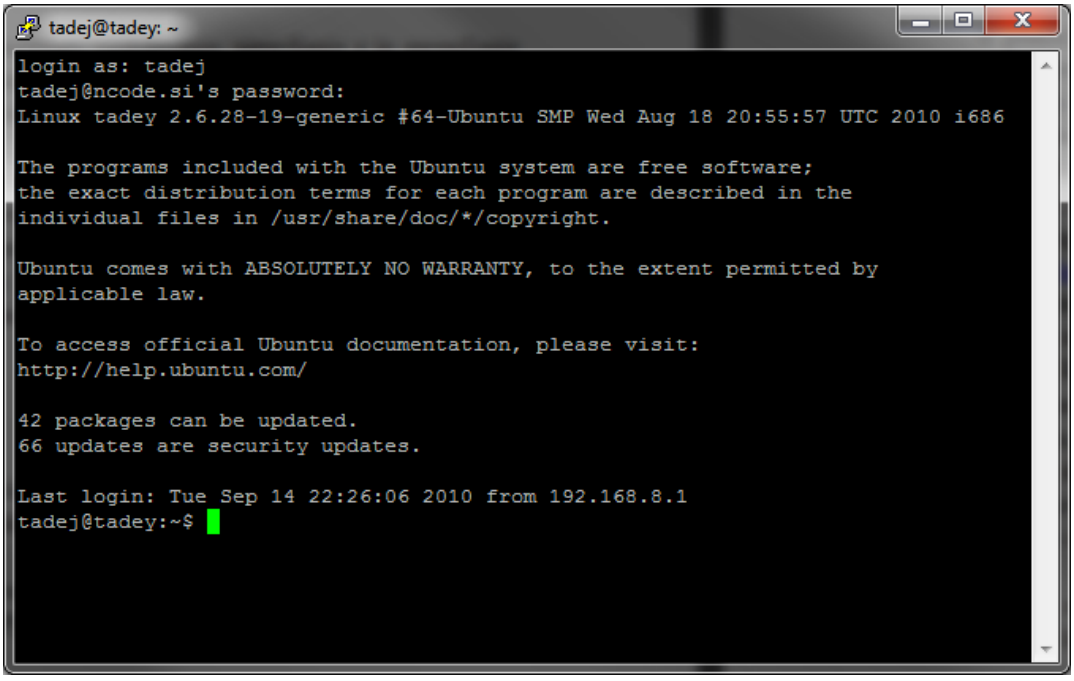
Ta orodja so še vedno zelo osnovna, saj je potrebno na roke poganjati *scp*, nekako pridobiti seznam vseh aktivnih vozlišč, in podobno.
Zato za naš namen niso uporabna, saj je eden od naših ciljev preprostost uporabe.

3. OPIS IZDELAVE SKRIPT

3.1. Opis povezovanja v sistem *PlanetLab*

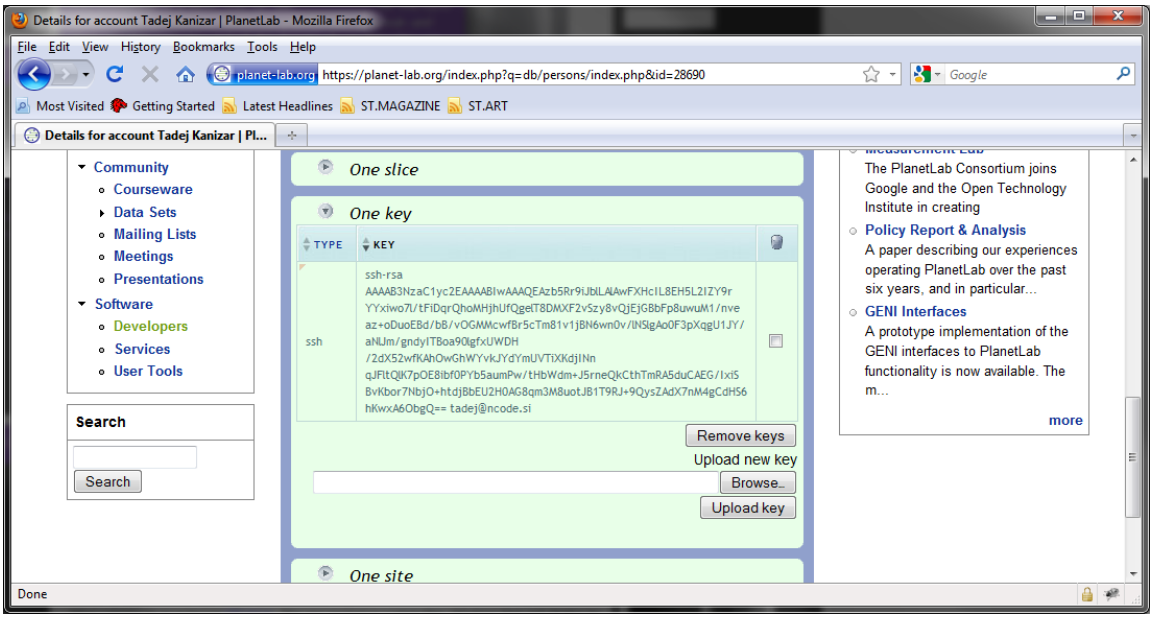
3.1.1. Orodji *ssh* in *scp*

Osnovni orodji za dostop do računalnikov sistema *PlanetLab* sta *ssh*, ki se uporablja za splošno povezovanje v oddaljene računalnike, ter *scp*, s katerim (varno) prenašamo datoteke med našim in oddaljenim računalnikom. Obe orodji sta standardni v okoljih **unix* (tj. *Unix*, *Linux*, *MacOSX* in podobno), v okolju *Windows* pa malo manj poznani, ponavadi pa preko orodij *WinSCP* (za protokol *scp*) ter *PuTTY* (za protokol *ssh*).
Orodji sta sorodni, zato bom bolj podrobno opisal le povezovanje preko orodja *ssh*.



Slika 3: Primer povezave preko *ssh*

S pomočjo tega orodja se lahko uporabniki povezujejo na oddaljene strežnike (če imajo seveda ustrezen dostop) in na njih delajo, kot bi bili fizično za tipkovnico. Tako je lahko na fizični računalnik hkratno prijavljenih veliko različnih uporabnikov, ki (načeloma) ne motijo eden drugega.
Ssh podpira več možnosti avtentikacije, v sistemu *PlanetLab* pa se uporablja samo metoda javnega ključa (angl. *publickey*), kjer ustvariš ključ, sestavljen iz dveh delov – privatnega in javnega. Javni del nato vpišeš v svojo uporabniško stran na spletni strani www.planet-lab.org, po (največ) nekaj dneh se ta del uporabniškega ključa prenese na vse strežnike, do katerih ima uporabnik dostop.



Slika 4: Objavljanje javnega dela ključa na www.planet-lab.org

Ko ima računalnik javni del ključa, se lahko vanj prijavimo s pomočjo privatnega dela ključa. Ker lahko ob izdelavi ključa pustimo geslo prazno, nas ob prijavi ne bo popolnoma nič vprašal (razen tega, če smo prepričani, da sprejmemo aventičnost strežnika, na katerega se povezujemo). In ravno na tem temelji moja skripta (in tudi nekatere druge), saj bi v nasprotnem primeru moral, če bi se povezoval na 100 strežnikov, 100-krat vpisati svoje geslo, kar do uporabnika ne bi bilo zelo prijazno in bi kršilo enega od naših osnovnih ciljev – enostavnost.

Ti dve orodji predstavljata osnovo moje skripte, njuno vključenost vanjo pa bom opisal v kasnejših točkah.

Slika 5: *OpenSSH* logo

3.1.2. *PlanetLab* vmesnik *API*

PlanetLab središčni *API* (angl. *PlanetLab Central API*) oz. *PLCAPI* je vmesnik, preko katerega lahko dostopamo do glavne baze podatkov sistema *PlanetLab*. S pomočjo tega vmesnika lahko pridobivamo različne informacije (kot npr. seznam vseh vozlišč, seznam oseb in podobno) ter spreminjamo nekatere podatke, seveda glede na naše

dodeljene pravice (npr. v rezine (angl. *slice*) dodajamo vozlišča, dodajamo javne ključe in podobno). [9]
Osebno me je zanimala predvsem zmožnost pridobivanja seznama vseh vozlišč, saj brez teh moja skripta ne zna delovati, ročno pa jih seveda v datoteko ne bo nihče prepisoval.

Dostop do omenjenega vmesnika *API* poteka preko standardnega protokola *XML-RPC*, ki določa, da se komunikacija vrši v obliki *XML*, ki je v zadnjih letih postal zelo pogosti način takšnih komunikacij (tj. med različnimi sistemi, različnimi programskimi jeziki, ipd.).

Slika 6: *XML-RPC* logotip

Primeri v dokumentaciji vmesnika *API* sistema *PlanetLab* so napisani v jeziku *Python*, saj le-ta zelo dobro podpira takšno komunikacijo. Stanje v programskem jeziku *bash* je precej drugačno, saj ga le-ta sploh ne podpira, kot tudi ne razčlenjevanje dokumentov *XML*.
Zaradi tega sem moral najti alternativo.
Za pošiljanje zahtevkov je bilo to orodje *netcat* [10], pri katerem preprosto na roke napišemo zahtevano vsebino *XML*, nato pa jo pošljemo na ciljni strežnik.
Pri obdelavi rezultatov (spet dokument v formatu *XML*) pa sem uporabil zelo zanimiv način – v *bashu* lahko s preprosto zanko *for* pregledamo vse besede v nizu. To lahko storimo zato, ker *bash* privzeto obravnava presledke, tabulatorje in nove vrstice kot ločnico med besedami. Če uporabimo trik, pri katerem spremenimo posebno spremenljivko *IFS* na »>«, nam le-ta omogoča, da napišemo kodo, kjer za vsak par ključa-vrednosti v »<ključ>vrednost</ključ>«, dobimo ključ v eno spremenljivko, vrednost pa v drugo.
S tem dobimo splošno orodje za razčlenjevanje datotek *XML*.

Naslednja koda prikazuje razčlenjevanje vsebine *XML*, ki jo dobimo kot rezultat klica metode *XML-RPC GetNodes*, in je vzeta iz skripte *get_nodes*, s katero lahko uporabnik dobi seznam vseh (njemu) zanimivih vozlišč. Spremenljivka *tmp2* v tem primeru vsebuje pot do (začasne) datoteke z vsebino *XML*.

```
rdom () { local IFS=\> ; read -d \< E C ;}

searching_for_slice_id="0"
include_next_site="0"
prev_name=""
while rdom; do
    if [[ $E = "string" ]]; then
        if [[ $prev_name = "hostname" ]]; then
           if [[ $include_next_site = "1" ]]; then
              echo $C
           fi
           include_next_site="0"
        fi
```

```
    elif [[ $E = "name" ]]; then
        if [[ $C = "slice_ids" ]]; then
           searching_for_slice_id="1"
        fi
        prev_name=$C
    elif [[ $E = "node_id" ]]; then
        searching_for_slice_id="0"
    elif [[ $E = "int" ]]; then
        if [[ $searching_for_slice_id = "1" ]]; then
           #echo "slice_id: $C"
           if [[ $C = $slice_id ]]; then
               include_next_site="1"
           fi
        fi
    fi
done < $tmp2
rm $tmp2
```

3.1.3. Težave

Pri povezovanju na vozlišča smo se želeli povezati predvsem na vozlišča naše rezine, a po dolgem iskanju nisem uspel odkriti neposrednega načina pridobivanja takšnega seznama. Zato sem uporabil splošno metodo *GetNodes* v vmesniku *API*, pri kateri sem s parametri oblikoval obliko seznama vozlišč v rezultatu klica te metode, in sicer tako, da je, poleg drugih podatkov, vsebovala tudi podatek o *id-jih* rezin, katerim posamezno vozlišče pripada. Te podatke sem moral potem še pregledati in iz njih izločiti le tista vozlišča, ki so pripadala naši želeni rezini.

Druga težava se je pojavila, ko sem poskušal dobljeni seznam vozlišč uporabiti, tj. povezati se na vozlišča iz seznama. Kot se je izkazalo, so lahko (vsaj nekateri) računalniki v tem sistemu zelo nezanesljivi – v nekaterih primerih računalniki preprosto niso bili dosegljivi (najverjetneje so bili ugasnjeni), v drugih primerih pa je njihov odziv trajal tako dolgo, da je skripta nad njim obupala - prišlo je do ti. časovnega izteka (angl. *timeout*).

```
-254 ulj_lalg@planetlab4-dsl.cs.cornell.edu Error connecting
-254 ulj_lalg@planetlab5.cs.cornell.edu Error connecting
-254 ulj_lalg@ricepl-3.cs.rice.edu Error connecting
-254 ulj_lalg@plgmu3.ite.gmu.edu Error connecting
-254 ulj_lalg@planetlab2.cs.ubc.ca Error connecting
-254 ulj_lalg@planlab1.cs.caltech.edu Error connecting
-254 ulj_lalg@pl1.unm.edu Error connecting
-254 ulj_lalg@planetlab1.pop-rs.rnp.br Error connecting
-254 ulj_lalg@planlab2.cs.caltech.edu Error connecting
-254 ulj_lalg@planetlab3.cs.uiuc.edu Error connecting
-254 ulj_lalg@planetlab1.pop-ce.rnp.br Error connecting
-254 ulj_lalg@dplab03.kookmin.ac.kr Error connecting
-254 ulj_lalg@csplanetlab3.kaist.ac.kr Error connecting
-254 ulj_lalg@planetlab-1.man.poznan.pl Error connecting
-254 ulj_lalg@planetlab1.elet.polimi.it Error connecting
-254 ulj_lalg@planetlab-2.vuse.vanderbilt.edu Error connecting
-254 ulj_lalg@charon.cs.binghamton.edu Error connecting
-254 ulj_lalg@planetdev05.fm.intel.com Error connecting
0 ulj_lalg@planet3.cs.ucsb.edu
-254 ulj_lalg@hptest-1.cs.princeton.edu Error connecting
-254 ulj_lalg@planetlab2.cs.cornell.edu Error connecting
-254 ulj_lalg@planetlab1.cs.cornell.edu Error connecting
-254 ulj_lalg@ds-pl2.technion.ac.il Error connecting
-254 ulj_lalg@planet1.cs.rochester.edu Error connecting
```

Slika 7: Primer izpisa neodzivnih računalnikov

3.2. Dogodivščine v *bashu* oz. kako so iz ideje nastale skripte

3.2.1. Ideja

Naša osnovna ideja je bila napisati enostavno skripto, z naslednjimi značilnostmi:

- uporabljal bi jo lahko vsak navaden uporabnik (tj. uporabnik brez znanja o uporabi orodij *ssh, scp* in podobnih),
- na oddaljene računalnike bi lahko prenašal poljubne datoteke (predvidoma programe in potrebne podatkovne datoteke),
- zaganjal naložene programe,
- preverjal stanje (program teče, se je končal),
- vse programe, ki bi tekli predolgo (in so se predvidoma prenehali odzivati), na silo zaprl,
- prenašal datoteke z rezultati iz oddaljenih računalnikov na svojega,
- skripta bi bila prenosljiva (brez, da bi moral uporabnik prevesti izvorno kodo),
- zaradi povečanja hitrosti izvajanja, bi uporabili metodo vzporednega izvajanja ukazov na različnih računalnikih (s tem počasni računalniki ne zaustavijo celotnega postopka, pač pa samo eno vzporedno izvajanje),
- skripta bi se povezala na računalnike iz seznama, ki bi ga dobila kot parameter.

Po premisleku sem sprejel odločitev, da je, zaradi organizacije kode, najboljše da je skripta razdeljena v dve skripti, kjer ena kliče drugo – skripti *p_ctl* ter *remote_ctl*.
Poleg tega sem ugotovil, da bomo potrebovali pomožno skripto za pridobitev seznama vozlišč (tj. računalnikov), saj ne bi bilo zelo učinkovito, če bi moral uporabnik ta seznam napisati na roke – skripta *get_nodes*.

3.2.2. Razvoj skript *p_ctl* in *remote_ctl*

Kot sem omenil že pri prejšnji točki, sem se arhitekturno odločil za dve skripti, imenovani *p_ctl* (okrajšava za *PlanetLab control* oz. nadzor sistema *PlanetLab* in *Parallel control* oz. vzporedni nadzor) ter *remote_ctl* (okrajšava za *Remote control* oz. oddaljen nadzor).

Razloga za razdelitev kode na dva dela sta bila dva:
- lažje vzdrževanje kode, še posebej na dolgi rok,
- logična razdelitev funkcionalnosti:
    - v skripti *remote_ctl* se nahaja splošna funkcionalnost, nepovezana s sistemom *PlanetLab*, ter je popolnoma neodvisna,
    - v skripti *p_ctl* se nahajajo funkcionalnosti, tesno povezane s sistemom *PlanetLab*, skripta je odvisna od skript *remote_ctl* ter *get_nodes*.

Za skripto *remote_ctl* sem se torej odločil, da jo napišem zelo splošno, tj. neodvisno od sistema *PlanetLab*. Vanjo sem zato dodal vse značilnosti, povezane z orodji *ssh* in *scp*.
Skripta *remote_ctl* ima posledično naslednje funkcije:
- *find_remote_pid*: preko *ssh* izvede ukaz *ps* ter z njim ugotovi *PID* (okrajšava za *Process ID*) podanega procesa – to nam pomaga pri dveh stvareh:
    - izvemo, če proces teče (če smo dobili njegov *PID*, očitno teče),
    - ko izvemo njegov *PID*, lahko ta proces s pomočjo tega podatka na silo zaustavimo (»ubijemo«),
- *kill_remote_process*: za podani proces ugotovi njegov *PID*, nato pa ga s pomočjo *kill* ukaza, preko *ssh*, ustavi,
- *exec_remote_process*: na oddaljenem računalniku zažene podani ukaz, ter ga pusti, da se izvaja v ozadju,
- *upload_file*: iz lokalnega računalnika, s pomočjo *scp*, prenese podane datoteke v oddaljeno mapo,
- *download_file*: deluje podobno kot *upload_file*, a v obratni smeri – iz oddaljenega računalnika, s pomočjo *scp*, prenese podane datoteke v ciljno (lokalno) mapo,
- *help*: pomožna funkcija, s pomočjo katere lahko, za vsako izmed zgornjih funkcij, izpišemo pomoč.

Skripta *remote_ctl* ni bila zastavljena z mislijo na uporabnika (čeprav ima funkcijo *help* za izpis pomoči), pač pa bolj kot skripta, katero uporablja neka druga skripta (ali pa bolj izkušen uporabnik).

Zaradi tega razloga obstaja še *p_ctl*, ki je dosti bolj prijazna do uporabnika in ima, v osnovi, uporabniški vmesnik, kjer lahko uporabnik zelo preprosto, korak za korakom,

izpolni potrebne informacije, potem pa se, na podlagi teh informacij, izvedejo druge skripte.

Skripta *p_ctl* ima tri osnovne načine delovanja, glede na to, s kakšnimi parametri je bila poklicana:

1) uporabnik skripto pokliče brez parametrov (oz. s premalo parametri): dobi tekstovni vmesnik, kjer ga skripta vpraša o potrebnih informacijah (npr. uporabniško ime), na to pa mu na izbiro ponudi šest možnosti, med katerimi se mora odločiti (npr. »Update nodes file«),
2) uporabnik kot prvi parameter ne poda načina izvajanja: skripta pokliče sama sebe, kjer za prvi parameter (način izvajanja) poda vrednost »multi«, vse skupaj pa pošlje programu *xargs*, ki poskrbi za vzporedno izvajanje ukazov – ta način klicanja primarno ni namenjen uporabnikom, pač pa ga pokliče skripta iz točke 1,
3) uporabnik skripto pokliče s parametri, pri čemer je prvi parameter način izvajanja (*single* oz. *multi*): v primeru načina izvajanja *single*, skripta sama izvede potrebne ukaze; v primeru načina izvajanja *multi* pa skripta le izpiše potrebne ukaze (predvidoma bodo podani ukazu *xargs*, ki jih bo vzporedno izvedel).

Primarni način delovanja skripte *p_ctl*:

```
echo "Welcome to PARALLEL PLANETLAB PROCESS EXECUTION"
echo "We need some basic data to get started. To exit the application
at any time press CTRL+C."
read -e -p "Identity file (example: id_rsa): " identity_file
read -p "Username (name of slice; example: ulj_lalg): " username
read -e -p "Nodes list file (example: nodes.txt): " ip_list_file

while [ 1 ]
do
    echo ""

    echo "Please select one of the following actions (enter 0 to
exit):"
    echo "1) Update nodes file";
    echo "2) Upload files";
    echo "3) Execute remote process";
    echo "4) Kill remote process";
    echo "5) Download files";
    echo "6) Find remote process id (pid)";
```

Največji izziv pri razvoju skripte *p_ctl* je bilo poganjanje ukazov na vzporedni način, pri čemer bi lahko omejil, koliko se jih hkrati izvaja.

Drugi programski jeziki, kot npr. *Java*, *C#*, *Python* in podobni, imajo ponavadi vgrajeno zelo dobro podporo za nitenje (angl. *threading*), kar bi bila popolna rešitev za moj problem s hkratnim izvajanjem ukazov.

A ker je *bash* zelo osnovno programsko okolje, takšne funkcionalnosti nima. Zelo veliko se namreč opira na obstoječa orodja, kar ni slabo, a včasih zelo omejeno.

V mojem primeru bi neko osnovno vzporedno izvajanje lahko dosegel s pomočjo ukaza *wait*, a bi le-ta povzročil, da bi v nekaterih primerih ukazi še vedno čakali eden

na drugega. Težava je namreč v tem, da *wait* čaka, dokler se ne končajo vsi procesi v ozadju (angl. *background processes*). V primeru, če bi hkrati izvedli 5 ukazov, bi šesti čakal, dokler se **vseh** 5 ne bi končalo. In če je med njimi takšen, ki bi za svoje delovanje porabil veliko časa, bi ga naslednjih 5 čakalo, kar brez dvoma ne bi bila pravilna rešitev.
Druga možnost bi bila izvajanje vseh ukazov hkrati, v ozadju. Delovala bi sicer (vsaj okvirno) tako, kot bi želeli, a bi se pri večjemu številu ukazov (računalnikov, na katere se povezujemo, je lahko tudi več kot tisoč) pojavil problem, saj bi lahko preobremenili lokalni računalnik, kot tudi, v primeru prenosa večjih datotek, celotno lokalno omrežje.

Kot že omenjeno, se je rešitev vzporednosti izvajanja ukazov pokazala v uporabi programa *xargs*.
Xargs je program na operacijskih sistemih *Linux, Unix* in podobnih, ki primarno omogoča, da nekemu poljubnemu programu podamo vse, kar dobimo od drugega programa kot izhod, pri čemer lahko s parametrom *–n* določimo, koliko parametrov naj mu naenkrat poda.
Če uporabimo še parameter *–P*, s katerim mu povemo, koliko procesov naj hkrati zažene, dobimo natančno to, kar iščemo – vzporedno izvajanje ukazov v lupini.
Da pa sem to funkcionalnost tudi dosegel, sem moral najprej izpisati vse vrstice, ki jih je potrebno izvesti, jih ujeti in podati programu *xargs*, z že omenjenimi parametri.

Koda za izpis teh vrstic izgleda takole:

```
for ip in $ip_arr
do
...
    elif [ "$mode" = "multi" ]
    then
        echo "$action $identity_file $username@$ip $args"
    fi
done
```

Potem pa jih podam programu *xargs* takole:

```
nr_of_args=$(( $# - 1 )) #this is actually 3 (action, identity_file,
username@ip) + number of arguments - 1

args=""
for arg
do
    args="$args \"$arg\""
done

cmd="$SHELL p_ctl multi $args | xargs -P $CONCURRENT_PROCS -n
$nr_of_args $SHELL remote_ctl"
eval $cmd | while read line
do
    echo $line
```

```
done
```

Na še en zanimiv izziv sem naletel pri razvoju skripte *remote_ctl*. Problem je namreč ta, da se *ssh* ponavadi konča šele, ko se konča program, ki ga prek *ssh* zaženemo.
Radi pa bi seveda, da *ssh* program na »drugi strani« požene, ga pusti, da teče v ozadju, *ssh* pa se konča (in nadaljuje z naslednjim vozliščem).
Seveda sem najprej pomislil na to, da bi oddaljeni program zagnal preprosto tako, da ukazu dodam znak &, kar bi ta program postavilo v ozadje in ga »odklopilo« od terminala. Ta metoda ni delovala tako, kot je bilo pričakovano, zato sem preiskal še druge rešitve, in našel programa *nohup* in *screen*.
Prvi naj bi povzročil, da zagnan program prezre *HUP* signal in s tem nadaljuje z delovanjem tudi po tem, ko se je uporabnik odjavil. A tudi ta ni deloval tako, kot sem upal.
Zato sem pregledal še ukaz *screen*, a se je hitro izkazalo, da tudi ta ne bo ustrezal kot rešitev mojega problema.
Po nadaljnjem dolgotrajnem iskanju sem končno našel rešitev. Izkazalo se je namreč, da *ssh* čaka na program samo zato, ker so z njim povezani trije podatkovni tokovi (angl. *data streams*) – *stdin* (vhodni tok), *stdout* (izhodni tok) in *stderr* (izhodni tok za napake). V skripti sem tako moral poskrbeti za to, da so bili vsi trije tokovi programa preusmerjeni nekam drugam, tj. na posebno »napravo«, imenovano tudi »črna luknja« - */dev/null*, kar je posebna datoteka, ki preprosto sprejme vse, kar dobi, a hkrati s tem nič ne naredi. Poleg tega sem program postavil v ozadje.
A če bi vse tri tokove vedno preusmeril na */dev/null*, bi to pomenilo, da uporabnik, kot del ukaza, teh tokov ne bi mogel preusmerjati (kar se zelo pogosto uporablja). Zato sem moral ukaz, ki naj bi se izvedel oddaljeno, pregledati, če je katerega od tokov že preusmeril. V tem primeru sem pustil ta tok preusmerjen tako, kot je, in ga nisem preusmeril na */dev/null*.

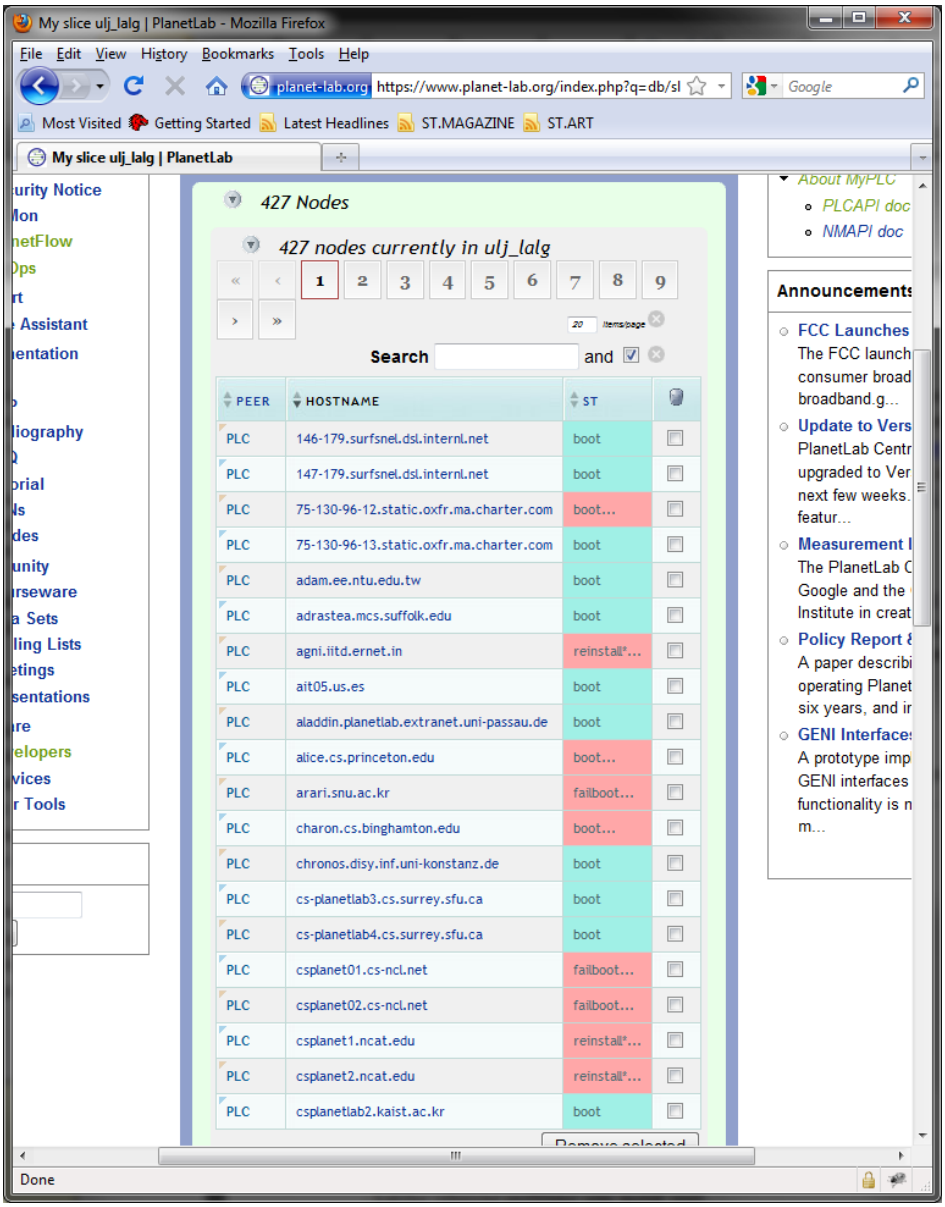
Koda, ki rešuje zgornji problem, je sledeča:

```
command="$3 "
if [[ "$command" == "${command/ >/}" ]]; then
    command="$command >/dev/null "
fi
if [[ "$command" == "${command/2>/}" ]]; then
    command="$command 2>/dev/null "
fi
if [[ "$command" == "${command/</}" ]]; then
    if [[ "$command" == "${command/|/}" ]]; then
        command="$command </dev/null "
    fi
fi

output=`ssh $OPTIONS -i $1 $2 "$command &"`
```

3.2.3. Razvoj skripte *get_nodes*

Na začetku razvoja se nisem veliko ukvarjal s tem, kje bomo dobili seznam vseh računalnikov (tj. vozlišč), na katere se bomo povezovali, saj sem verjel, da takšen seznam brez dvoma že obstaja, nekje na spletni strani www.planet-lab.org.
A ko sem prešel v poznejše faze razvoja, kjer sem opravljal bolj obsežna testiranja, se je hitro pokazalo, da takšen seznam ne obstaja, vsaj ne v obliki, ki bi bil, za namene tega diplomskega dela, uporaben.
Potrebovali smo namreč seznam vseh vozlišč v naši dodeljeni rezini, in edini seznam teh vozlišč, ki sem ga uspel najti, je prikazan na naslednji sliki:



Slika 8: Seznam vozlišč rezine

Kot je razvidno že iz slike, je ta seznam namenjen uporabnikom, ne računalniški obdelavi (npr. prek skript). Zadovoljen bi bil že s tem, če bi našel seznam, katerega bi

lahko uporabnik ročno kopiral in prilepil v ustrezno datoteko, a žal seznam iz zgornje slike niti za to ni primeren.
Zato sem moral rešitev najti drugje. Prebrskal sem dokumentacijo in odkril, da lahko do nekaterih informacij sistema *PlanetLab* pridem preko njihovega vmesnika *API*, imenovanega *PLCAPI*. Mojo pozornost je vzbudila predvsem metoda *GetNodes*, ki vrne vsa vozlišča. Žal ne podpira filtriranja po rezinah.
Zato sem moral takšno filtriranje izvesti sam – nastala je skripta *get_nodes*. Ta skripta s pomočjo zahtevka *XML-RPC* (več o tem v 3.1.2.) pridobi seznam vseh vozlišč in iz njih, na podlagi šifre rezine (*slice_id*), ustvari seznam le želenih vozlišč.

Primer, kako ročno napisati vsebino *XML*, za klic metode oblike *XML-RPC*:

```
tmp=`mktemp`
echo "<?xml version=\"1.0\"?>
<methodCall>
  <methodName>GetNodes</methodName>
  <params>
    <param>
      <value>
        <struct>
          <member>
            <name>AuthMethod</name>
            <value><string>password</string></value>
          </member>
          <member>
            <name>Username</name>
            <value><string>$username</string></value>
          </member>
          <member>
            <name>AuthString</name>
            <value><string>$password</string></value>
          </member>
        </struct>
      </value>
    </param>
    <param>
      <value>
        <struct>
          <member>
            <name>run_level</name>
            <value><string>boot</string></value>
          </member>
        </struct>
      </value>
    </param>
    <param>
      <value>
        <array>
          <data>
            <value><string>node_id</string></value>
            <value><string>hostname</string></value>
            <value><string>slice_ids</string></value>
          </data>
        </array>
      </value>
    </param>
```

```
  </params>
</methodCall>" > "$tmp"
```

Skripto *get_nodes* lahko uporabnik uporabi samostojno, ali pa v okviru skripte *p_ctl*.

3.2.4. Težave

Brez težav ne gre, se včasih tolažimo, kadar naletimo na kopico težav. In res je, težave so zmeraj prisotne, ne glede na situacijo. Včasih so manjše in jih je malo, drugič so večje in jih je veliko.
Tudi v programiranju ni nič kaj drugače, a resnici na ljubo, so vedno nove težave ravno tisto, kar večino programerjev privlači k njihovem delu, saj te težave predstavljajo vedno nove izzive.
Tako je bilo tudi pri pisanju tega diplomskega dela. In ker sem programiral v jeziku, v katerem ne programiram pogosto (*bash*), ter se povezoval na sistem, ki ga prej nisem poznal (*PlanetLab*), je bilo teh težav še mnogo več.

Bash je ukazna sintaksa (včasih poimenovana kar programski jezik), s katero lahko v operacijskih sistemih *Linux*, *Unix* in podobnih, napišemo skripte za različna opravila, od zelo enostavnih do precej kompleksnih.
Ne glede na to, da je *bash* osnova skript v zgoraj omenjenih operacijskih sistemih, pa ima nekaj slabosti oz. napak, na katere sem naletel tudi sam:

- kakršen koli grafični vmesnik je zelo težko napisati (razen z uporabo dodatnih knjižnic),
- sintaksa je težja, kot v večini modernih programskih jezikov (je bolj striktna, manj intuitivna, itd.),
- izvajanje kode je počasnejše kot pri jezikih z (vsaj delnim) prevajanjem,
- nekatere manjkajoče funkcionalnosti (sam sem imel problem predvsem z manjkajočo podporo pri klicanju metod *XML-RPC* ter s parsanjem datotek *XML).*

Težave sem imel tudi s sintakso, saj mi je v nekaterih primerih vzelo veliko časa, da sem ugotovil, kako neko preprosto stvar napisati v *bashu*.
Še en zanimiv problem mi je predstavljalo prenašanje dvojnega narekovaja (ki se ponavadi uporablja za pisanje parametrov s presledki) iz skripte v skripto. Dvojni narekovaj se je vmes namreč izgubil, zato sem moral parametre razstaviti in ponovno sestaviti, kot je razvidno iz sledeče kode:

```
args=""
shift 5
for arg
do
    args="$args \"$arg\""
done
```

Kot sem omenil že v nekaterih prejšnjih poglavljih, sem imel nekaj težav tudi s samim sistemom *PlanetLab*. Težave so izvirale predvsem iz moje nevedenosti v zvezi s tem

sistemom, in je tako npr. (pre)dolgo trajalo, da sem ugotovil, da moram za uporabniško ime, pri povezovanju prek *ssh*, podati ime rezine, ne moje lastno uporabniško ime.
Presenečen sem bil tudi nad tem, koliko truda sem moral vložiti, da sem dobil preprost seznam vseh vozlišč, pri čemer mi sistem ni znal filtrirati niti po trenutnem stanju vozlišč (da bi lahko dobili samo vozlišča v delujočem stanju).

3.3. Obstoječe pomanjkljivosti in možne dodelave

Kot vsi programi, imajo tudi moje skripte (znane) pomanjkljivosti.

Ena od večjih pomanjkljivosti je tako pomanjkanje grafičnega vmesnika. Kot omenjeno v točki 3.2.4., predstavlja razvoj grafičnega vmesnika v *bashu* precejšen izziv, in bi bilo v tem primeru morda lažje celoten sistem skript implementirati v programskem jeziku *Python*.
Pomanjkanje grafičnega vmesnika zlahka odvrne nekatere uporabnike, zato se mi zdi to ena od bolj pomembnih možnih dodelav.

Prav tako so skripte trenutno težje prenosljive na nekatere operacijske sisteme, kot npr. *Microsoft Windows*. V primeru *Microsoft Windows* to še vseeno ni zelo težko, saj obstaja dovolj programov, ki poznajo *bash* sintakso. Da omenim samo nekatere: *Cygwin, MinGW, Win-bash* projekt, itd.

Pametno bi bilo implementirati avtomatičen nadzor nad časom povezovanja, kar bi uporabili za avtomatično odstranjevanje vozlišč, saj se trenutno zlahka zgodi, da imamo vozlišča, ki so sicer aktivna, a tako počasna, da mora uporabnik predolgo čakati na odgovor. Še posebej, kadar je takšnih vozlišč veliko.
Prav tako bi se morala vozlišča, ki so bila prej neaktivna/počasna, sedaj pa so »zdrava«, avtomatično dodati na seznam vozlišč, saj nas zanima, da bi se naš poskus izvajal na čim večjem številu računalnikov.

3.4. Testiranje

3.4.1. Metodologija

Za testiranje skript *p_ctl* in *remote_ctl* sem najprej potreboval seznam (izbranih) vozlišč. Zato je bilo smiselno le, da sem najprej testiral skripto *get_nodes*, s katero sem, poleg tega, da sem jo testiral, dobil tudi najnovejši seznam.

Testiranje te skripte je potekalo zelo preprosto – večkrat sem jo zagnal in opazoval, če mi bo vsakič naredila datoteko z izbranimi vozlišči.

Zaradi lažjega testiranja, sem iz celotnega seznama naključno izbral le nekaj vozlišč. Na tej množici sem potem izvajal teste, ki so bili sestavljeni iz poganjanja različnih funkcij skripte *p_ctl*. Za začetek sem vsem poskusil naložiti neko naključno datoteko in jo prenesti nazaj k meni.
Potem sem na vsa vozlišča naložil preprost ukaz in ga izvedel.
Nadaljeval sem z nalaganjem programa, ki se izvaja dalj časa:

```
echo "start @ `date`"
sleep 60
echo "end @ `date`"
```

Izpis programa sem preusmeril v datoteko output.txt in jo po končanem izvajanju prenesel na lokalni računalnik. V vsakem sta bila 2 zapisa: »start @ trenutni datum« ter »end @ trenutni datum«
Poleg tega sem zopet zagnal ta program, a sem, preden se je zaključil, poklical ukaz *kill*, nato pa spet prenesel output.txt na lokalni računalnik. Tokrat je bil v datotekah samo en zapis: »start @ trenutni datum«.
S tem sem potrdil, da je ukaz *kill* opravil svoje.

Zanimalo me je tudi, koliko pridobimo z vzporednostjo. Zato sem število vzporednih procesov spreminjal in meril čas, ki ga potrebuje skripta, da izve *PID* procesa *bash*. Namesto tega bi lahko uporabil tudi funkcijo nalaganja datotek na oddaljeni računalnik, a bi v tem primeru preveliko vlogo igrala moja povezava v splet.
Vsakič, ko sem spremenil število vzporednih procesov, sem zagnal ukaz:

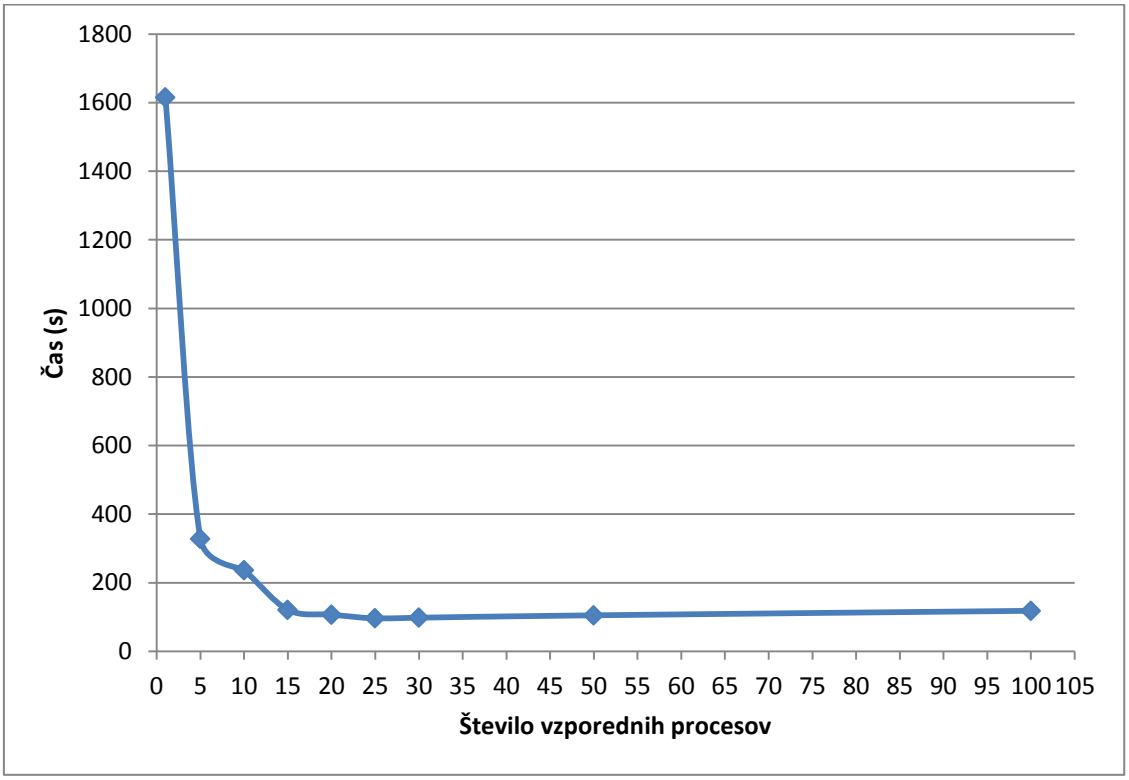
```
START=$(date +%s) && /bin/bash p_ctl 'nodes.txt' find_remote_pid 'id_rsa'
'ulj_lalg' 'ps' && END=$(date +%s) && echo $(( $END - $START ))
```

Ta ukaz mi je na koncu povedal, koliko sekund se je skripta izvajala.
Testiral sem na 345 vozliščih.
Rezultati testiranj so predstavljeni v naslednji točki, tj. 3.4.2.

3.4.2. Rezultati

Tabela 1: Tabela odvisnosti časa izvajanja od števila vzporednih procesov

Št. vzp. procesov	Čas izvajanja (s)
1	1615
5	328
10	236
15	121
20	107
25	96
30	98
50	105
100	118



Slika 9: Graf odvisnosti časa izvajanja od števila vzporednih procesov

Opazimo lahko, da čas izvajanja pada do 25 vzporednih procesov, potem pa začne spet počasi naraščati. Sklepam, da je razlog za to hitrost testnega računalnika ter omejitve omrežnih naprav.
Vidimo pa, da že 5 vzporednih procesov zelo pohitri celotno izvajanje skripte, s čimer smo brez dvoma rešili enega izmed naših glavnih problemov.

3.5. Uporabniški priročnik

3.5.1. Skripta *p_ctl*

Pred začetkom uporabe skripte, je potrebno objaviti javni del ključa na www.planet-lab.org (glej 3.1.1.). Poleg tega se je, v ukazni vrstici, potrebno postaviti v mapo, kjer se nahajajo vse tri skripte. Alternativno jih lahko skopirate v */usr/bin*, zaradi česar bodo potem dostopne iz katere koli mape.

Za osnovno delo s skripto, jo preprosto poženemo: `./p_ctl`.
Prikaže se osnovni uporabniški vmesnik, ki nas vpraša o vseh potrebnih informacijah.
Le-te so sledeče:

- *identity file*: datoteka z vašim privatnim delom ključa, katerega javni del ste že predhodno naložili na www.planet-lab.org,

- uporabniško ime: uporabniško ime, s katerim se lahko preko *ssh* prijavite na oddaljene računalnike. To je kar ime vaše rezine,
- datoteka s seznamom vozlišč: datoteka, ki vsebuje vsa vozlišča, s katerimi se želite povezovati.

Skripta nam nato ponudi meni, kjer lahko izbiramo med šestimi možnostmi:

```
[tadej@tadej-virtualc Diploma]$ ./p_ctl
Welcome to PARALLEL PLANETLAB PROCESS EXECUTION
We need some basic data to get started. To exit the application at any time press CTRL+C.
Identity file (example: id_rsa): id_rsa
Username (name of slice; example: ulj_lalg): ulj_lalg
Nodes list file (example: nodes.txt): nodes.txt

Please select one of the following actions (enter 0 to exit):
1) Update nodes file
2) Upload files
3) Execute remote process
4) Kill remote process
5) Download files
6) Find remote process id (pid)
```

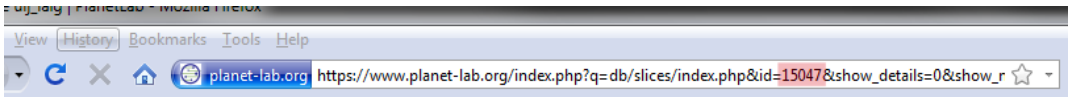
Slika 10: Glavni meni skripte *p_ctl*

Šest možnosti, ki jih imamo na voljo, in njihova razlaga:

1) *Update nodes file*: osvežimo izbrano datoteko s seznamom vozlišč, ki so nam trenutno na voljo. Ob uporabi te možnosti, morate izpolniti naslednja polja:
   - številka rezine (*slice id*), za katero naj se osvežijo vozlišča,
   - vaše uporabniško ime za dostop do sistema *PlanetLab*,
   - vaše geslo za dostop do sistema *PlanetLab*,
   - izhodna datoteka, kamor se bodo vozlišča zapisala.

   Ko izpolnete zadnje polje, in pritisnete *enter*, program začne delovati, vmes pa vidite napredek prenosa. Po končanem prenosu in obdelavi pa vam zopet izpiše glavni meni.
   Če se zgodi kakšna napaka, jo skripta takrat tudi izpiše.

   Kje najdem številko rezine?
   Ko se prijavite na www.planet-lab.org, pojdite na »My Slices« in kliknite na ime želene rezine. Potem poglejte v naslov *URL*, kjer piše »...index.php?id=15047&show_details=...«. Številka rezine je številka, ki se nahaja za »id=«. V našem primeru je to »15047«.



Slika 11: Kako najdemo številko rezine?

2) *Upload files*: izbrane datoteke prenesemo iz našega računalnika na oddaljenega. Ob uporabi te možnosti, morate izpolniti naslednja polja:

- oddaljeni direktorij: ciljni direktorij na oddaljenem računalniku, kamor se bodo izbrane datoteke prenesle. Pot je lahko relativna (npr. »./«) ali pa absolutna (npr. »/home/«).
- seznam datotek: vse datoteke, ki bi jih želeli prenesti na oddaljeni računalnik, v mapo, ki ste jo določili v prejšnjem koraku.

Ko izpolnete zadnje polje, in pritisnete *enter*, program začne delovati, izpisal bo le napake. Ob koncu vam zopet izpiše glavni meni.

```
2
Remote directory (example: ./): ./
List of files to be uploaded (example: /tmp/file1 ./file2): test.txt
-254 ulj_lalg@host4-plb.loria.fr Error connecting
-254 ulj_lalg@planetlab4.lublin.rd.tp.pl Error connecting
-254 ulj_lalg@planetlab3.gdansk.rd.tp.pl Error connecting
-254 ulj_lalg@planetlab13.millennium.berkeley.edu Error connecting
-254 ulj_lalg@iason.inf.uth.gr Error connecting
-254 ulj_lalg@planetlab2.ece.ucdavis.edu Error connecting
```

Slika 12: Primer vnosa opcije *upload files*

3) *Execute remote process*: na oddaljenem računalniku zaženemo program. To je lahko vgrajen program (npr. »ls«) ali pa naš program, ki smo ga s pomočjo *Upload files* možnosti, prenesli na oddaljeni računalnik. Ob uporabi te možnosti, morate izpolniti naslednje polje:
   - ukaz: ukaz, ki naj se na vseh računalnikih izvede. Vsebuje lahko tudi parametre in preusmeritve podatkovnih tokov. V primeru, da izhodnega toku (angl. *stdout*) ne preusmerite, izhoda ukaza ne boste videli nikjer, saj ga skripta v takšnem primeru avtomatično preusmeri na */dev/null*. Enako stori tudi z vhodom (angl. *stdin*) in izhodom za napake (angl. *stderr*).

   Ko izpolnete zadnje polje, in pritisnete *enter*, program začne delovati, izpisal bo le napake. Ob koncu vam zopet izpiše glavni meni.

```
3
Command to be executed (example: ./script.sh): ls > ls_output.txt
-254 ulj_lalg@planetlab3.gdansk.rd.tp.pl Error connecting
-254 ulj_lalg@planetlab4.lublin.rd.tp.pl Error connecting
-254 ulj_lalg@planetlab13.millennium.berkeley.edu Error connecting
-254 ulj_lalg@iason.inf.uth.gr Error connecting
-254 ulj_lalg@planetlab2.ece.ucdavis.edu Error connecting
```

Slika 13: Primer vnosa opcije *execute remote process*

4) *Kill remote process*: če ugotovimo, da se proces na oddaljenih računalnikih ne bo končal, ga lahko končamo na silo, z uporabo ukaza *kill*. Ob uporabi te možnosti, morate izpolniti naslednje polje:
   - ime procesa: ime procesa, katerega naj se konča. To ime mora biti enako tistemu, ki ga vrne ukaz *ps*.

   Ko izpolnete zadnje polje, in pritisnete *enter*, program začne delovati, izpisal bo le napake. Ob koncu vam zopet izpiše glavni meni.

```
4
Process to be killed (example: script.sh): test
-254 ulj_lalg@planetlab3.gdansk.rd.tp.pl Error connecting
-254 ulj_lalg@planetlab4.lublin.rd.tp.pl Error connecting
-254 ulj_lalg@planetlab13.millennium.berkeley.edu Error connecting
-254 ulj_lalg@iason.inf.uth.gr Error connecting
-254 ulj_lalg@planetlab2.ece.ucdavis.edu Error connecting
```

Slika 14: Primer vnosa opcije *kill remote process*

5) *Download files*: izbrane datoteke prenesemo iz vseh oddaljenih računalnikov na naš računalnik. Ob uporabi te možnosti, morate izpolniti naslednja polja:
   - ciljna oblika imena datotek: ker bomo enako datoteko prenešali iz več različnih računalnikov v eno mapo na našem računalniku, moramo te datoteke nekako ločiti med seboj, drugače lahko obstaja samo ena izmed njih (tj. zadnja). Zaradi tega je potrebno vpisati to polje, kjer lahko ime datoteke oblikujemo poljubno, pri tem pa lahko uporabimo dve posebni označbi, ki bosta pozneje zamenjani s trenutnima vrednostima. To sta »%HOST%«, ki bo zamenjan z naslovom oddaljenega računalnika, in »%FILENAME%«, ki bo zamenjan z nazivom datoteke.
     Primer: če vpišemo »%HOST%__%FILENAME%«, bodo datoteke imele imena, podobna kot »planetlab1.wroclaw.rd.tp.pl__nodes.txt«,
   - lokalni direktorij: direktorij na našem računalniku, kamor bodo shranjene vse datoteke,
   - seznam datotek: seznam vseh datotek, ki bi jih želeli prenesti iz oddaljenih računalnikov na lokalnega. Pot je lahko relativna ali absolutna.

   Ko izpolnete zadnje polje, in pritisnete *enter*, program začne delovati, izpisal bo le napake. Ob koncu vam zopet izpiše glavni meni.

```
5
Destination filename format (%HOST% will be replaced with the host, %FILENAME% will be re
placed with the filename; example: %HOST%_%FILENAME%): %HOST%__%FILENAME%
Local directory (example: ./): ./tmp/
List of files to be downloaded (example: /tmp/file1 file2): ls_output.txt
-254 ulj_lalg@host4-plb.loria.fr Error connecting
-254 ulj_lalg@planetlab3.gdansk.rd.tp.pl Error connecting
-254 ulj_lalg@planetlab4.lublin.rd.tp.pl Error connecting
-254 ulj_lalg@planetlab13.millennium.berkeley.edu Error connecting
-254 ulj_lalg@iason.inf.uth.gr Error connecting
-254 ulj_lalg@planetlab2.ece.ucdavis.edu Error connecting
```

Slika 15: Primer vnosa opcije *download files*

6) *Find remote process id (pid)*: pomožna možnost, s katero lahko izvemo številke oddaljenih procesov (angl. *process id* oz. *PID*). Vkolikor proces ne teče, vrne skripta številko 0. Če je prišlo do napake, vrne negativno vrednost. Ob uporabi te možnosti, morate izpolniti naslednje polje:
   - ime procesa: ime procesa, ki se mora ujemati z izpisom ukaza *ps*, katerega *PID* naj skripta vrne.

Ko izpolnete zadnje polje, in pritisnete *enter*, program začne delovati. Za vsak oddaljeni računalnik bo izpisal številko procesa in naslov tega računalnika. Če pride do napake, vrne poleg omenjenih dveh podatkov, še tekst napake.
Primer: »-254 ulj_lalg@pli2-pa-3.hpl.hp.com Error connecting«.

```
6
Process whose PID to find (example: script.sh): bash
29079 ulj_lalg@planetlab2.it.uc3m.es
2961 ulj_lalg@planetlab1.it.uc3m.es
13685 ulj_lalg@onelab3.warsaw.rd.tp.pl
22634 ulj_lalg@dplanet1.uoc.edu
18790 ulj_lalg@host4-plb.loria.fr
12267 ulj_lalg@host3-plb.loria.fr
-254 ulj_lalg@planetlab3.gdansk.rd.tp.pl Error connecting
-254 ulj_lalg@planetlab4.lublin.rd.tp.pl Error connecting
```

Slika 16: Primer vnosa opcije *find remote process id (pid)*

3.5.2. Skripta *remote_ctl*

Ker je ta skripta namenjena za vključevanje v druge skripte oz. za napredne uporabnike, je bolj podrobno opisana v tehničnem priročniku (glej 3.6.).

3.5.3. Skripta *get_nodes*

Ker je ta skripta namenjena za vključevanje v druge skripte oz. za napredne uporabnike, je bolj podrobno opisana v tehničnem priročniku (glej 3.6.).

3.6. Tehnični priročnik

3.6.1. Skripta *p_ctl*

To skripto lahko zaženemo na tri različne načine, glede na podane parametre:

1) brez kakršnih koli parametrov: uporaba tega načina je bolj podrobno opisana v uporabniškem priročniku (glej 3.5.). Ta način interno pokliče 2. način uporabe, opisan spodaj,
2) z zahtevanimi parametri, a prvi parameter ni »single« ali »multi«: zahtevani parametri so:
   - datoteka z naslovi oddaljenih računalnikov,
   - »funkcija« (ena izmed funkcij, definiranih v *remote_ctl*, npr. »download_files«),
   - privatni del ključa za povezovanje,
   - uporabniško ime (tj. ime rezine),

- ostali parametri so odvisni od izbrane »funkcije«, zato se je potrebno pri tem sklicevati na poglavje tehničnega priročnika o skripti *remote_ctl* (glej 3.6.2.).

Ta način interno pokliče 3. način uporabe, opisan spodaj. Rezultat klica 3. načina se pošlje ukazu *xargs* s parametrom »-P«, s katerim določimo, koliko procesov naj *xargs* vzporedno izvaja. To število se prilagaja s spreminjanjem spremenljivke CONCURRENT_PROCS.

3) z enakimi parametri kot pri 2. načinu uporabe, le da kot prvi parameter podamo metodo izvajanja (»single« ali »multi«). Pri »single« oz. posameznem načinu izvajanja, skripta vse ukaze izvrši neposredno, enega za drugim. V primeru »multi« oz. več-ukaznega načina izvajanja, pa skripta vse ukaze le izpiše, po enega v vsako vrstico, saj je namen tega, da se izhod tega načina uporabe, uporabi za vhodne podatke drugih programov (v primeru 2. načina uporabe, je to ukaz *xargs*).

3.6.2. Skripta *remote_ctl*

Prvotno je bila namenjena kot podporna skripta za druge, kot npr. za *p_ctl*. A ker je napisana zelo splošno (tj. neodvisno od sistema *PlanetLab*), prav tako pa ima pomoč (dosegljivo preko parametra »help«), je uporabna tudi za napredne uporabnike, ki bi raje uporabljali to enostavno sintakso, kot pa sintakso ukazov *ssh* in *scp*.
Skripta se privzeto povezuje preko vrat 22, kar so privzeta vrata za protokol *ssh*. Če želimo ta vrata spremeniti, lahko to storimo na dva načina:

1) to lahko storimo globalno, za vse naslove oddaljenih računalnikov: spremenimo spremenljivko DEFAULT_PORT,
2) to lahko storimo tudi samo za specifične naslove: pri teh naslovih dodamo dvopičje, za njim pa želena vrata. Primer: planetlab4.gdansk.rd.tp.pl:2121.

Pri povezovanju dodamo, kot parameter, koliko časa lahko traja zahteva, preden se odločimo, da je to predolgo. Privzeto je ta parameter nastavljen na 5 sekund, a se ga zlahka spremeni, s spremembo spremenljivke TIMEOUT.
Za dodajanje dodatnih parametrov ukazoma *ssh* in *scp*, je najbolj primerna prilagoditev spremenljivke OPTIONS.

Skripta *remote_ctl* ima naslednje načine klicanja:

1) klic brez parametrov: izpiše, da je to nepravilna uporaba, ter da lahko več informacij dobimo, če klicu dodamo parameter »help«, opisan v naslednji točki,
2) klic s parametrom »help«, brez dodatnih parametrov: izpiše, kako uporabljati pomoč, kar je opisano v naslednji točki,
3) klic s parametroma »help« in funkcija (oz. akcija): če podana funkcija obstaja, potem o njej izpiše pomoč,
4) klici z imenom funkcije, kot prvim parametrom, ostali parametri, kot jih zahteva izbrana funkcija (glej pomoč, 3. način klicanja te skripte): če ta funkcija obstaja, jo pokliče in ji poda ustrezne parametre.

Skripta ima naslednje funkcije:

1) *help*: izpiše pomoč. Za več informacij glej načine klicanja te skripte,
2) *find_remote_pid*: s pomočjo *ssh* se poveže na oddaljeni računalnik, izvede ukaz *ps* (z nekaj parametri; glej kodo), in z njegovo pomočjo ugotovi *PID* želenega procesa,
3) *kill_remote_process*: najprej pokliče funkcijo *find_remote_pid*, s pomočjo katere ugotovi *PID* podanega procesa, nato se s *ssh* poveže na oddaljeni računalnik, in izvede ukaz *kill* (z ustreznimi parametri),
4) *exec_remote_process*: podani ukaz, ki ga želimo izvesti na oddaljenem računalniku, skripta najprej pregleda, in mu doda manjkajoče preusmeritve podatkovnih tokov. Privzeto preusmeri vse tokove na */dev/null*, saj lahko pusti ukaz v ozadju le, če so vsi tokovi preusmerjeni (ni pomembno kam). Zatem se preko *ssh* poveže na oddaljeni računalnik in izvede podani ukaz, ki ga pusti v ozadju,
5) *upload_file*: s pomočjo *scp* prenese podane datoteke iz lokalnega na oddaljeni računalnik, v podano mapo,
6) *download_file*: s pomočjo *scp* prenese podane datoteke iz oddaljenega na lokalni računalnik, v podano mapo, pri tem pa vsako prenešeno datoteko preimenuje, glede na podano masko.

Vse funkcije vrnejo ustrezno kodo, tj. 0 v primeru, če je bil klic uspešen, ter vrednost, večjo od 0, če ni bil.

3.6.3. Skripta *get_nodes*

Zastavljena je kot podporna skripta, a jo napredni uporabniki lahko uporabljajo tudi neposredno.

Skripta sprejme naslednje tri parametre:

- uporabniško ime: uporabniško ime uporabnika v sistemu *PlanetLab*,
- geslo: geslo uporabnika, podanega v prvem parametru,
- številka rezine: številka rezine, katere vozlišča želimo dobiti.

Skripta deluje tako, da najprej sestavi datoteko *XML*, kjer poda uporabniško ime, geslo in katere lastnosti želimo, da dobimo kot rezultat klica. V našem primeru nas zanimata predvsem *hostname* in *slice_ids*, saj je *hostname* tisti, ki ga iščemo, v *slice_ids* pa se nahajajo vse številke rezin, v katerih je to vozlišče prisotno, kar uporabimo za filtriranje – s tem dobimo iz seznama vseh le tiste, ki jih iščemo (glede na podano številko rezine).
Datoteko *XML* pošljemo na *URL* »https://www.planet-lab.org/PLCAPI/«, s čimer smo izvedli *XML-RPC* klic metode (v našem primeru smo poklicali metodo *GetNodes*).

Ker so podatki, ki jih dobimo, kot rezultat zgornjega klica, v obliki *XML*, nam to v samem *bashu* zelo malo pomaga. Zato moramo iz oblike *XML*, podatke pretvoriti v drugo obliko, berljivo s pomočjo ukazov *basha*.
Za to bi lahko uporabili enega izmed mnogih ukazov, ki so na voljo, a so žal vsi dodatni (tj. ne vgrajeni v operacijski sistem) in jih je zato potrebno namestiti dodatno. Ker bi to povečalo odvisnosti od drugih programov/knjižnic, sem se odločil to narediti v samem *bashu*.

Čeprav sam *bash* oblike *XML* ne zna prebrati, pa je zato zelo fleksibilen, in omogoča, da prilagodimo *IFS* (interno ločilo med polji oz. angl. *internal field separator*). Na takšen način sem lahko uspešno prebral vrnjene podatke in iz njih izbral le želene.

4. SKLEPNE UGOTOVITVE

Na začetku sem se spraševal, zakaj sistem *PlanetLab* nima nekih privzetih orodij, s katerimi bi se dalo upravljati z vozlišči v tem sistemu. Ko sem tekom pisanja te diplomske naloge pregledoval že obstoječa orodja, bral dokumentacije, in podobno, sem ugotovil, da za to obstaja razlog.
PlanetLab je namreč zelo fleksibilen sistem, zaradi česar je omogočil vrsto različnih uporab računalnikov v njem. Kot se izkaže, nobeno izmed orodij ni dovolj fleksibilno, da bi omogočilo uporabo vseh teh različnih načinov uporabe.
Zato obstoja veliko orodij za vsa ta opravila. A to ni slabo, saj raznolikost pomeni razvoj novih idej, vedno novih načinov uporabe – in ravno to je osnovni cilj sistema *PlanetLab*.

Ugotavljam tudi, da morda *bash* ni bil najbolj ustrezen jezik za ta program, čeprav je na začetku izgledalo, da je za takšno opravilo idealen - brez potrebe po prevajanju kode, brez zunanjih orodij, brez namestitve, namenjen za hiter razvoj prav takšnih skript.
Skripte sicer delujejo, a so v nekaterih smereh že na meji zmožnosti – npr. *GUI*, datoteke *XML*, vzporedno izvajanje, in podobno.

Tako bi za nadaljnji razvoj, vkolikor bi le-ta vključeval večje dodelave, priporočal raje programski jezik *Python*.

Nadaljnje dodelave bi lahko vključevale grafični vmesnik, za še lažjo uporabo ter boljšo integriranost v današnje grafične operacijske sisteme.
Za boljši izkoristek internetne povezave uporabnika, bi bilo verjetno smiselno dodati možnost povezovanja *peer-to-peer*, kjer bi si oddaljeni računalniki sami pošiljali datoteke med seboj.

5. PRILOGE

Priloga A: izvorna koda skripte *p_ctl*

```
#!/bin/bash

if [ $# -lt 3 ]
then
    echo "Welcome to PARALLEL PLANETLAB PROCESS EXECUTION"
    echo "We need some basic data to get started. To exit the application
at any time press CTRL+C."
    read -e -p "Identity file (example: id_rsa): " identity_file
    read -p "Username (name of slice; example: ulj_lalg): " username
    read -e -p "Nodes list file (example: nodes.txt): " ip_list_file

    while [ 1 ]
    do
        echo ""

        echo "Please select one of the following actions (enter 0 to
exit):"
        echo "1) Update nodes file";
        echo "2) Upload files";
        echo "3) Execute remote process";
        echo "4) Kill remote process";
        echo "5) Download files";
        echo "6) Find remote process id (pid)";

        read answer
        case "$answer" in
            "1")
                read -p "Slice id (example: 15047): " slice_id
                read -p "Your PlanetLab username (example:
user@domain.com): " pl_username
                read -p "Your PlanetLab password (example: mypassword): "
pl_password
                read -e -p "Destination nodes filename (example:
nodes.txt): " nodes_filename

                cmd="$SHELL get_nodes '$pl_username' '$pl_password'
'$slice_id' > '$nodes_filename'"
                eval $cmd
                ;;
            "2")
                read -p "Remote directory (example: ./): " remote_dir
                read -e -p "List of files to be uploaded (example:
/tmp/file1 ./file2): " files

                cmd="$SHELL p_ctl '$ip_list_file' upload_files
'$identity_file' '$username' '$remote_dir' '$files'"
                eval $cmd
                ;;
            "3")
                read -p "Command to be executed (example: ./script.sh): "
cmd_name

                cmd="$SHELL p_ctl '$ip_list_file' exec_remote_process
'$identity_file' '$username' '$cmd_name'"
                eval $cmd
```

```
                ;;
            "4")
                read -p "Process to be killed (example: script.sh): "
cmd_name

                cmd="$SHELL p_ctl '$ip_list_file' kill_remote_process
'$identity_file' '$username' '$cmd_name'"
                eval $cmd
                ;;
            "5")
                read -p "Destination filename format (%HOST% will be
replaced with the host, %FILENAME% will be replaced with the filename;
example: %HOST%_%FILENAME%): " format
                read -e -p "Local directory (example: ./): " local_dir
                read -e -p "List of files to be downloaded (example:
/tmp/file1 file2): " files

                cmd="$SHELL p_ctl '$ip_list_file' download_files
'$identity_file' '$username' '$format' '$local_dir' '$files'"
                eval $cmd
                ;;
            "6")
                read -p "Process whose PID to find (example: script.sh): "
cmd_name

                cmd="$SHELL p_ctl '$ip_list_file' find_remote_pid
'$identity_file' '$username' '$cmd_name'"
                eval $cmd
                ;;
            *)
                exit
                ;;
        esac
    done

    exit
fi

method=$1
CONCURRENT_PROCS=20

mode=""
case "$method" in
    "single")
        mode="single"
        ;;
    "multi")
        mode="multi"
        ;;
    *)
        nr_of_args=$(( $# - 1 )) #this is actually 3 (action,
identity file, username@ip) + number of arguments - 1

        args=""
        for arg
        do
            args="$args \"$arg\""
        done

        cmd="$SHELL p_ctl multi $args | xargs -P $CONCURRENT_PROCS -n
$nr_of_args $SHELL remote_ctl"
```

```
        eval $cmd | while read line
        do
            echo $line
        done

        ret_status=$?
        exit $ret_status
        ;;
esac

ip_file=$2
action=$3
identity_file=$4
username=$5

args=""
shift 5
for arg
do
    args="$args \"$arg\""
done

exec 6<&0
exec < $ip_file

ip_arr=""
while read ip
do
    ip_arr="$ip_arr$ip
"
done

exec 0<&6 6<&-

for ip in $ip_arr
do
    if [ "$mode" = "single" ]
    then
        cmd="$SHELL remote_ctl $action $identity_file $username@$ip $args"
        eval $cmd | while read line
        do
            echo $line
        done
    elif [ "$mode" = "multi" ]
    then
        echo "$action $identity_file $username@$ip $args"
    fi
done
```

Priloga B: izvorna koda skripte *remote_ctl*

```
#!/bin/bash

DEFAULT_PORT=22
TIMEOUT=5
OPTIONS="-o StrictHostKeyChecking=no -o PreferredAuthentications=publickey
-o ConnectTimeout=$TIMEOUT -q"

## MAIN FUNCTIONS ##
help()
{
    case "$1" in
        "find_remote_pid")
            find_remote_pid
            ;;
        "kill_remote_process")
            kill_remote_process
            ;;
        "exec_remote_process")
            exec_remote_process
            ;;
        "upload_files")
            upload_file
            ;;
        "download_files")
            download_file
            ;;
    esac
}
find_remote_pid ()
{
    if [ $# -lt 3 ]
    then
        echo "Usage: remote_ctl find_remote_pid identity_file username@host
process_name"
        exit 2
    fi

    pid=0
    pid_arr=`ssh $OPTIONS -i $1 $2 "ps -A -o pid,comm= | grep '$3'"`
    retval=$?

    if [ "$retval" -eq 0 ]
    then
        for var in $pid_arr
        do
            pid=$var
            break
        done
        echo "$pid $2"
    elif [ "$retval" -eq 255 ]
    then
        msg="-254 $2 Error connecting"
        echo "$msg" >&2
        logger -- "$msg"
        exit 254 #this can't be 255, because xargs would stop processing
any more lines
    else
        echo "0 $2"
        exit 0
```

```
    fi
}
kill_remote_process ()
{
    if [ $# -lt 3 ]
    then
        echo "Usage: remote_ctl kill_remote_process identity_file
username@host process_name"
        exit 2
    fi

    output=`bash remote ctl find remote pid "$1" "$2:$port" "$3"`
    pid=0
    host=""
    c=0
    for var in $output
    do
        c=$(( $c + 1 ))
        if [ "$c" -eq 1 ]
        then
            pid=$var
        elif [ "$c" -eq 2 ]
        then
            host=$var
        fi
    done

    if [ "$pid" -gt 0 ]
    then
        `ssh $OPTIONS -i $1 $2 "kill $pid" >/dev/null 2>&1`
        retval=$?
        if [ "$retval" -eq 255 ]
        then
            msg="-254 $2 Error connecting"
            echo "$msg" >&2
            logger -- "$msg"
            exit 254
        fi
        exit $retval
    fi
    exit 0
}
exec_remote_process() {
    if [ $# -lt 3 ]
    then
        echo "Usage: remote_ctl exec_remote_process identity_file
username@host process_name"
        exit 2
    fi

    command="$3 "
    if [[ "$command" == "${command/ >/}" ]]; then
        command="$command >/dev/null "
    fi
    if [[ "$command" == "${command/2>/}" ]]; then
        command="$command 2>/dev/null "
    fi
    if [[ "$command" == "${command/</}" ]]; then
        if [[ "$command" == "${command/|/}" ]]; then
            command="$command </dev/null "
        fi
```

```
    fi

    output=`ssh $OPTIONS -i $1 $2 "$command &"`
    retval=$?
    if [ "$retval" -eq 255 ]
    then
        msg="-254 $2 Error connecting"
        echo "$msg" >&2
        logger -- "$msg"
        exit 254
    fi
    exit $retval
}
upload_file ()
{
    if [ $# -lt 4 ]
    then
        echo "Usage: remote_ctl upload_files identity_file username@host
remote_directory file1 [file2 [file3 ...]]"
        exit 2
    fi

    `scp $SCP_OPTIONS -i $1 "$4" $2:"'$3'" >/dev/null 2>&1`
    retval=$?
    if [ "$retval" -gt 0 ]
    then
        msg="-254 $2 Error connecting"
        echo "$msg" >&2
        logger -- "$msg"
        exit 254
    fi
    exit $retval
}
download_file ()
{
    if [ $# -lt 5 ]
    then
        echo "Usage: remote_ctl download_files identity_file username@host
dest_filename_format local_directory file1 [file2 [file3 ...]]"
        exit 2
    fi

    host_key="%HOST%"
    filename_key="%FILENAME%"

    user_and_host=$2
    format=$3
    hostname=${user_and_host#*@}

    local=${format//$host_key/$hostname}
    local=${local//$filename_key/$5}

    `scp $SCP_OPTIONS -i $1 $2:"'$5'" "$4$local" >/dev/null 2>&1`
    retval=$?
    if [ "$retval" -gt 0 ]
    then
        msg="-254 $2 Error connecting"
        echo "$msg" >&2
        logger -- "$msg"
        exit 254
    fi
```

```
    exit $retval
}

## HELPER FUNCTIONS ##
extract_host()
{
    echo $(extract_val 1 $1)
}
extract_port()
{
    if [[ $1 == *:* ]]
    then
        port=$(extract_val 2 $1)
    else
        port=$DEFAULT_PORT
    fi
    echo $port
}
extract_val()
{
    if [ "$1" -eq 1 ]
    then
        val=${2%:*}
    elif [ "$1" -eq 2 ]
    then
        val=${2#*:}
    fi
    echo $val
}

## ENTRY STUFF ##
action=$1
host_and_user=$(extract_host $3)
port=$(extract_port $3)
OPTIONS="$OPTIONS -p $port"
SCP_OPTIONS="-P $port -q"

case "$action" in
    "help")
        if [ "$#" -lt 2 ]
        then
            echo "Usage: remote_ctl help action"
            echo "Available actions: find_remote_pid, kill_remote_process,
exec_remote_process, upload_files, download_files"
            exit 1
        else
            help $2
        fi
        ;;
    "find_remote_pid")
        find_remote_pid "$2" "$host_and_user" "$4"
        ;;
    "kill_remote_process")
        kill_remote_process "$2" "$host_and_user" "$4"
        ;;
    "exec_remote_process")
        exec_remote_process "$2" "$host_and_user" "$4"
        ;;
    "upload_files")
        a=$2
        b=$host_and_user
```

```
        c=$4
        shift 4

        for arg
        do
            upload_file "$a" "$b" "$c" "$arg"
        done
        ;;
    "download_files")
        a=$2
        b=$host_and_user
        c=$4
        d=$5
        shift 5

        for arg
        do
            download_file "$a" "$b" "$c" "$d" "$arg"
        done
        ;;
    *)
        echo "Unknown action (action was $action)"
        echo "Usage: remote_ctl action [action_argument1 [action_argument2
[...]]]"
        echo "Execute remote_ctl help for further instructions"
        exit 255
        ;;
esac
```

Priloga C: izvorna koda skripte *get_nodes*

```
#!/bin/bash
if [ $# -lt 3 ]
then
    echo "Usage: get_nodes username password slice_id"
    exit 1
fi

username=$1
password=$2
slice_id=$3

tmp=`mktemp`
echo "<?xml version=\"1.0\"?>
<methodCall>
  <methodName>GetNodes</methodName>
  <params>
    <param>
      <value>
        <struct>
          <member>
            <name>AuthMethod</name>
            <value><string>password</string></value>
          </member>
          <member>
            <name>Username</name>
            <value><string>$username</string></value>
          </member>
          <member>
            <name>AuthString</name>
            <value><string>$password</string></value>
          </member>
        </struct>
      </value>
    </param>
    <param>
      <value>
        <struct>
          <member>
            <name>run_level</name>
            <value><string>boot</string></value>
          </member>
        </struct>
      </value>
    </param>
    <param>
      <value>
        <array>
          <data>
            <value><string>node_id</string></value>
            <value><string>hostname</string></value>
            <value><string>slice_ids</string></value>
          </data>
        </array>
      </value>
    </param>
  </params>
</methodCall>" > "$tmp"
```

```
tmp2=`mktemp`
curl -d "@$tmp" -H "Content-Type: text/xml" https://www.planet-
lab.org/PLCAPI/ > $tmp2
rm $tmp

rdom () { local IFS=\> ; read -d \< E C ;}

searching_for_slice_id="0"
include_next_site="0"
prev_name=""
while rdom; do
    if [[ $E = "string" ]]; then
        if [[ $prev_name = "hostname" ]]; then
           if [[ $include_next_site = "1" ]]; then
              echo $C
           fi
           include_next_site="0"
        fi
    elif [[ $E = "name" ]]; then
        if [[ $C = "slice_ids" ]]; then
           searching_for_slice_id="1"
        fi
        prev_name=$C
    elif [[ $E = "node_id" ]]; then
        searching_for_slice_id="0"
    elif [[ $E = "int" ]]; then
        if [[ $searching_for_slice_id = "1" ]]; then
           if [[ $C = $slice_id ]]; then
               include_next_site="1"
           fi
        fi
    fi
done < $tmp2
rm $tmp2
```

6. VIRI

[1] File Transfer Protocol - Wikipedia, the free encyclopedia. Zadnji dostop: 10.8.2010. Dostopno na: http://en.wikipedia.org/wiki/Ftp

[2] World Internet Usage Statistics News and World Population Stats. Zadnji dostop: 10.8.2010. Dostopno na: http://www.internetworldstats.com/stats.htm

[3] Neowin.net - YouTube has surpassed two billion daily hits. Zadnji dostop: 15.8.2010. Dostopno na: http://www.neowin.net/news/youtube-has-surpassed-2-billion-daily-hits

[4] PlanetLab | An open platform for developing, deploying, and accessing planetary-scale services. Zadnji dostop: 30.8.2010. Dostopno na: http://www.planet-lab.org/

[5] About PlanetLab | PlanetLab Zadnji dostop: 1.9.2010. Dostopno na: http://www.planet-lab.org/about

[6] pShell:: Home Page. Zadnji dostop: 5.9.2010. Dostopno na: http://cgi.cs.mcgill.ca/~anrl/projects/pShell/index.php

[7] AquaLab's Nixes. Zadnji dostop: 6.9.2010. Dostopno na: http://www.aqualab.cs.northwestern.edu/nixes.html

[8] parallel-ssh - Project Hosting on Google Code. Zadnji dostop: 6.9.2010. Dostopno na: http://code.google.com/p/parallel-ssh/

[9] PlanetLab Central API Documentation | PlanetLab. Zadnji dostop: 16.9.2010. Dostopno na: http://www.planet-lab.org/doc/plc_api

[10] XML RPC with bash/netcat. Zadnji dostop: 6.9.2010. Dostopno na: http://www.acooke.org/cute/XMLRPCwith0.html