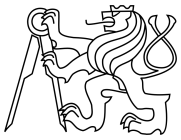
ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ



ZADÁNÍ DIPLOMOVÉ PRÁCE

Název:	Implementace škodné do projektu Clondike
Student:	Bc. Zdeněk Nový
Vedoucí:	Ing. Josef Gattermayer
Studijní program:	Informatika
Studijní obor:	Počítačová bezpečnost
Katedra:	Katedra počítačových systémů
Platnost zadání:	Do konce zimního semestru 2016/17

Pokyny pro vypracování

Hlavním cílem práce je zjednodušit práci se systémem Clondike v operačním systému. Dalším cílem je vymyslet a implementovat mechanizmy škodné a vymyslet algoritmy pro její odhalení. To znamená modifikovat migrační mechanizmus v Clondike takovým způsobem, aby daný uzel nebyl odhalen, i když prostředky clusteru neférově využívá a současně vymyslet algoritmy na detekci zlomyslného chování.Více o projektu Clondike na http://pcg.fit.cvut.cz/structure/clondike.

1) Proveďte refaktoring konfiguračních souborů Clondike.
2) Navrhněte a implementujte integraci Clondike do operačního systému.
3) Upravte záplatu Linuxového jádra pro verzi 3.18.21 a zdokumentujte provedené změny.
4) Navrhněte a implementujte několik variant škodné na úrovni user space.
5) Doplňte chybějící dokumentaci projektu Clondike.

Seznam odborné literatury

Dodá vedoucí práce.

L.S.

prof. Ing. Róbert Lórencz, CSc.
vedoucí katedry

prof. Ing. Pavel Tvrdík, CSc.
děkan

V Praze dne 24. února 2015

České vysoké učení technické v Praze
Fakulta informačních technologií
Katedra počítačových systémů

Diplomová práce

Implementace škodné do projektu Clondike

Bc. Zdeněk Nový

Vedoucí práce: Ing. Josef Gattermayer

11. ledna 2016

Poděkování

Děkuji svému vedoucímu diplomové práce Ing. Josefu Gattermayerovi za její metodické vedení a za vedení celého projektu Clondike. Děkuji všem, kteří mne ve studiu a psaní této práce podporovali.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval(a) samostatně a že jsem uvedl(a) veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 46 odst. 6 tohoto zákona tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou, a veškeré jejich dokumentace (dále souhrnně jen „Dílo“), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla, a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu), licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným způsobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 11. ledna 2016 .....

Odkaz na tuto práci

Nový, Zdeněk. *Implementace škodné do projektu Clondike.* Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2016.

Abstrakt

Diplomová práce se zabývá vytvořením nové záplaty pro linuxové jádro verze 3.18.21 v projektu Clondike. Práce popisuje sjednocení konfigurace do jednoho souboru a implementaci projektu Clondike jako služby operačního systému. Součástí diplomové práce je analýza škodných a možnosti jejich implementace v Uživatelském prostoru společně s návrhem obranného mechanismu. Dále se práce věnuje doplnění projektové dokumentace, konkrétně popisu nepreemptivní migrace procesu.

Klíčová slova Clondike, Nededikovaný cluster, Linux, Jádro, Škodná, Záznam transakcí, Cassandra

Abstract

The thesis deals with the creation of a new patch for the Linux kernel version 3.18.21 in project Clondike. It describes the unification of configuration into one file and implementation of the project Clondike as an operating system service. The thesis also includes an analysis of malicious nodes and the possibility of their implementation in Userspace, together with a draft of a defense mechanism. The thesis also deals with completion of project documentation, specifically description of non-preemptive process migration.

Keywords Clondike, Non-dedicated cluster, Linux, Kernel, Malicious nodes, Transaction log, Cassandra

Obsah

Seznam obrázků

Seznam tabulek

Úvod

Předložená práce se zabývá projektem Clondike. Název Clondike je odvozen z anglického ***CL****uster* ***O****f* ***N****on-****D****edicated* ***I****nteroperating* ***KE****rnels*, což volně přeloženo znamená shluk nededikovaných kooperujících jader [4].

Úvodní část se zabývá stručným popisem projektu a studiem jeho stavu po implementaci diplomových prací kolegů Ing. Pavla Tvrdíka [5] a Ing. Michala Saláta [6]. Součástí této práce je tvorba záplaty pro nové linuxové jádro verze 3.18.21, která byla v době zadání této práce považována za stabilní. Společně s tvorbou záplaty je popsána související úprava zdrojových kódů Clondike v Prostoru jádra.

Další kapitola je věnována nahrazení konfiguračních řetězců centrálním konfiguračním souborem. Účelem této implementace je substituce konfigurací umístěných na různých místech za transparentní konfigurační mechanismus celého Uživatelského prostoru. Kapitola zároveň obsahuje implementaci řídících skriptů, jejichž smyslem je zjednodušit práci se systémem Clondike.

Následující kapitola obsahuje analýzu a návrh škodné, která v kontextu této práce představuje uzly neférově využívající cluster. Aby nedošlo k jejich odhalení, snaží se modifikovat migrační mechanismus. V práci jsou popsány obě varianty škodné v Uživatelském prostoru i v Prostoru jádra. Dále je zmíněn možný způsob implementace škodné v Uživatelském prostoru společně s návrhem na její obranu.

Poslední kapitola doplňuje dokumentaci projektu. Detailně popisuje mechanismus migrace procesu včetně názvů volaných funkcí. Tento mechanismus zahrnuje rozhodnutí o migraci procesu, výběr Hostitelského uzlu, samotnou migraci procesu, vykonání procesu na Hostitelském uzlu, odeslání výsledku

a jeho přijetí. Dále je zdokumentován databázový systém Cassandra, na který se tato práce odkazuje.

Kapitola 1

Clondike

1.1 Popis projektu

Clondike je cluster nededikovaných počítačů s linuxovým operačním systémem distribuce Debian [7]. Název je odvozen z anglické zkratky „**CL**uster **O**f **N**on-**D**edicated **I**nteroperating **KE**rnels“, což volně přeloženo znamená shluk nededikovaných kooperujících jader [8].

Clondike využívá standardní počítače s operačním systémem Linux jako výpočetní jednotky. Unikátní vlastností tohoto clusteru je nededikovanost jednotlivých uzlů. To znamená, že počítač poskytuje svůj výpočetní výkon pouze v době, kdy není vytížen lokálními procesy [8]. Díky tomu je možné, aby na uzlu pracovali lokální uživatelé a nebyli zpomalováni migrovanými procesy.

Systém běží jako peer-to-peer (více o *P2P* architektuře v [9]) na úrovni operačního systému. To znamená, že mezi uzly v clusteru neexistuje hierarchie jako například klient a server, ale všechny uzly jsou si rovnocenné a fungují jako klient i server zároveň. Důsledkem této vlastnosti je možnost spustit migraci na kterémkoliv uzlu clusteru. Nejenže každý uzel clusteru poskytuje svůj výpočetní výkon, ale zároveň může výkon clusteru využívat. Tato vlastnost výrazně podporuje motivaci k zapojení do clusteru [8].

Uzel clusteru tedy nejen poskytuje svůj výkon, ale zároveň může výkon clusteru využívat, což výrazným způsobem podporuje motivaci k zapojení do clusteru [8].

1.1.1 Stručný popis komponent

Zdrojové kódy projektu jsou rozděleny na dvě části, podle toho, ve které části se v systému nacházejí. Jedná se o Prostor jádra (*Kernelspace*) a Uživatelský prostor (*Userspace*). Obě tyto části se podle funkcionality dále dělí na několik komponent.

Prostor jádra Modifikované linuxové jádro, do kterého jsou umístěny *háčky* (*hooks*). Tyto háčky jsou zodpovědné za volání určité funkcionality Clondike v Prostoru jádra [10].

- **KKC** – knihovna zajišťující síťovou komunikaci mezi jádry jednotlivých uzlů (kernel to kernel communication).
- **ProxyFS** – virtuální souborový systém umožňující používat speciální typy souborů jako roury, terminály nebo sokety.
- **CCFS** – speciální typ souborového systému sloužící jako vyrovnávací paměť (Clondike cache filesystem).
- **TCMI** – implementace migračního mechanismu procesů.
- **Director** – klient komunikující s Uživatelským prostorem pomocí knihovny Netlink.

Uživatelský prostor Uživatelská část implementace Clondike, která je napsána převážně v jazyce *Ruby*. Obsahuje aplikační rozhraní, které zprostředkovává komunikaci s Prostorem jádra [10].

- **Director API** – část aplikačního rozhraní napsaná v jazyce C zajištující komunikaci s jádrem pomocí knihovny Netlink.
- **Ruby Director API** – rozhraní mezi Director API (C) a Simple Ruby Directorem (Ruby) obsahující definici Callback funkcí s mapovacím mechanismem.
- **Simple Ruby Director** – uživatelská aplikace ve formě Ruby skriptů implementující správu uzlu, rozložení zátěže, certifikaci uzlu, komunikaci s jádrem (Netlink) a řízení procesu migrace.
- **Network Pipe File System (NPFS)** – server sdílející souborový systém přes 9P 1.2 protokol pomocí NFS 1.2.

1.2 Definice používaných pojmů

Operační systém Základní programové vybavení počítače, které umožňuje uživateli komunikovat s počítačem. Jedná se o komplexní sadu programů, které se starají o běh počítače a umožňují spouštět další programy [11].

Linux Linux je svobodně vyvíjený a volně šířený operační systém typu UNIX, jehož hlavní komponentou je jádro [11].

Shell Shell je základní interpretr příkazového řádku, který nabízí základní rozhraní pro uživatele a zpracovává jednotlivé příkazy. Sada příkazů v textovém souboru je nazývána skriptem [12].

Bash Zkratka z anglického ***B**ourne **a**gain **sh**ell*, je jeden z Unixových Shellů, který interpretuje příkazový řádek. Je kompatibilní s původním interpretrem příkazového řádku *Shell* a taktéž umožňuje psaní skriptů [13].

Ruby Interpretovaný objektový skriptovací jazyk s širokým použitím a jednoduchou syntaxí.

Proces Proces je instance běžícího programu [14].

PID Zkratka *process ID* je jednoznačné kladné celé číslo procesu, které je v rámci systému přiděleno každému procesu [15].

Migrace procesu Technika, která umožňuje vykonat proces na vzdáleném počítači.

UID Z anglického *User ID* je jednoznačný identifikátor uživatele v jádře systému, který je reprezentován kladným celým číslem [16].

GID Zkratka *group ID* je jednoznačný identifikátor skupiny v systému reprezentovaný kladným celým číslem [16].

Regulární výraz Speciální řetězec znaků, který představuje určitý vzor pro textové řetězce [17].

IP adresa Jedinečné číslo, které slouží pro jednoznačnou identifikaci počítače v dané síti. V kontextu této práce se jedná o IP adresu verze 4 [18]

Hash Je řetězec vytvořený Hashovací funkcí, která definuje jeho délku. Hashovací funkce je jednosměrná bezkolizní funkce, která binárnímu řetězci (*vzoru*) přiřadí bitovou posloupnost (*obraz*). Jednosměrná funkce je taková funkce, která ze vzoru jednoduše vytvoří obraz, ale je výpočetně neschůdné získat vzor z jeho obrazu. Bezkoliznost funkce znamená, že je výpočetně neschůdné nalézt dva různé vzory, které mají stejný obraz [19].

NFS Network File System je protokol souborového systému umožňující klientovi přistupovat k souborům v síti stejným způsobem, jako by se jednalo o lokální soubory [20].

9P Celým názvem *Plan 9 Filesystem Protocol* je protokol, který je využíván pro sdílení souborového systému pomocí NPFS [21].

Uzel V kontextu této práce uzel znamená počítač zapojený do clusteru [8].

Cluster Skupina síťově propojených počítačů, které mezi sebou spolupracují a vytvářejí dojem jediného výkonného systému (Single system image) [8].

Cluster nededikovaných počítačů Jedná se o typ clusteru, kde jednotlivé integrované počítače nejsou plně vlastněny clusterem. Tyto počítače mohou být používány jejich uživateli jako běžné pracovní stanice. Při vytížení počítače lokálními uživateli není clusteru nabízen výpočetní výkon [8].

Virtuální paměťový prostor Lineární paměťový prostor, který je mapován do fyzické operační paměti (RAM) pomocí mechanismů stránkování a segmentace [22].

Prostor jádra Virtuální paměťový prostor, který je oddělený od Uživatelského paměťového prostoru, ve kterém jsou spuštěny procesy a služby jádra. Paměťový Prostor jádra je přístupný přes systémová volání [23].

Uživatelský prostor Virtuální paměťový prostor, který je oddělený od Prostoru jádra, ve kterém jsou spouštěny uživatelské procesy [24].

Callback funkce Jedná se o funkci, která zajišťuje vykonání části kódu a kterou jí lze předat parametrem například jako ukazatel [10].

CTLFS Virtuální souborový systém sloužící ke komunikaci Uživatelského prostoru, respektive komponentou *Simple ruby director*, a Prostoru jádra. Je připojen do adresáře `/clondike`. Pomocí souborů v tomto adresáři lze nastavovat spojení a zjišťovat stav uzlů nebo ovládat migraci [25].

Bootstrap V kontextu této práce jde o proces zajišťující připojení k definovanému uzlu clusteru. Tento uzel následně poskytne informace pro připojení k ostatním uzlům v clusteru [5].

Bootstrap uzel Je uzel, ke kterému se uzel pokusí připojit. V případě úspěšného připojení tento uzel poskytne informace o ostatních uzlech v clusteru [5].

1.3 Identifikace stavu projektu

V prvotní fázi bylo nutné nastudovat práce kolegů Ing. Pavla Tvrdíka [5] a Ing. Michala Saláta [6]. Po nastudování prací a konzultacích s autory bylo zjištěno několik nedostatků, které musely být vyřešeny, přičemž některé jsou popsány dále v této práci. Účelem této části bylo úspěšně provést migraci procesu v clusteru tvořeném několika virtuálními stroji. Při pokusu provést migraci se objevilo několik nesrovnalostí, které bylo třeba upravit.

V Uživatelském prostoru bylo nutné odstranit volání komponenty `Clondike Cache FileSystem`, reprezentované Ruby třídou `CacheFSController`, která způsobovala selhání migrace. Jedná se o pomocnou komponentu a její odstranění nevedlo ke změně funkcionality.

Po nastudování mechanismu *DHT* byl zjištěn nedostatek při spojování více uzlů. Pro správnou funkčnost bylo nutné doplnit informování ostatních uzlů o naučených uzlech a vkládání těchto uzlů do třídy `NodeRepository`.

Po úpravě několika konfiguračních řetězců bylo možné odmigrovat testovací proces kompilace zdrojového souboru pomocí nástroje `gcc`.

Po poradě s vedoucím práce bylo rozhodnuto aktualizovat použitý operační systém Debian z verze *squeeze* (*Debian 6*) na novější *wheezy* (*Debian 7*). S přechodem na novější verzi operačního systému došlo zároveň k aktualizaci nainstalovaných balíčků dostupných z oficiálních repozitářů distribuce Debian. Určité problémy způsobila aktualizace nástroje `gcc`, který byl aktualizován z verze `4.4` na `4.7`. Do souboru `.migration.conf` v domovském adresáři uživatele `root` bylo proto nutné přidat cestu ke spustitelnému souboru `cc1` pro verzi 4.7. Při pokusu o migraci jádra verze *2.6.33.1* došlo k chybě při kompilaci. Po krátké studii bylo zjištěno, že k této chybě dojde i bez migrace procesů. Kompilátor `gcc` verze *4.7* a vyšší nepřeloží čisté jádro verze *2.6.33.1*. Pro kompilaci této starší verze linuxového jádra je nutné použít starší verzi, například *4.4*, nástroje `gcc`.

S přechodem na novější verzi operačního systému byl aktualizován balíček Ruby na verzi *1.9.1*. Díky této aktualizaci bylo provedeno několik drobných úprav v Uživatelském prostoru. Příkladem je třída `Identity`, ve které došlo ke změně třídy obsahující RSA klíč. Nově je RSA klíč uchováván přímo ve třídě `OpenSSL::PKey:RSA`.

Z důvodu přehlednosti byly upraveny úrovně několika logovacích zpráv. Některým méně důležitým zprávám byl snížen stupeň závažnosti pouze na varování nebo dokonce informaci a nepotřebné zprávy byly zakomentovány.

Po provedení těchto úprav bylo možné provést migraci kompilace pomocí nástroje `gcc`. Ovšem při pokusu o kompilaci jádra docházelo k chybě jádra, která je popsána v [10].

Kapitola 2

Tvorba nové záplaty

2.1 Příprava

Po dohodě s vedoucím práce bylo odsouhlaseno vytvoření nové záplaty do novějšího jádra. Účelem této aktualizace bylo zjištění podrobností nebo odstranění chyby jádra typu *race condition*[1], kterou popisuje sekce *Ověřovací měření opraveného systému* diplomové práce [10]. Jako vhodný kandidát bylo vybráno linuxové jádro verze *3.18.21*.

2.1.1 Základní pojmy

Linuxové jádro Je program, který představuje centrální prvek operačního systému. Jádro systému má plnou kontrolu nad systémem a ostatní aplikace jej využívají při svém běhu [27].

Čisté jádro Čisté (Vanilla) jsou jádra publikovaná na oficiálním webu `http://www.kernel.org/`. Jedná se o obecná jádra, ze kterých vycházejí specializovaná jádra jednotlivých distribucí [28].

Rozdíl Příkaz `diff` slouží k porovnání textových souborů po řádcích. Základním výstupem příkazu `diff` je výpis rozdílných řádků souborů, který lze ovlivnit množstvím přepínačů [29].

Clondike rozdíl Pro Clondike je použit příkaz `diff` s přepínači `-N` (s chybějícím souborem zacházet jako s prázdným), `-a` (vstupní soubory jsou textové), `-u` (výstup okolo změny nastaven na výchozí hodnotu tři řádky) a `-r` (procházet rekurzivně) [29].

[1] *Race condition* je chyba v systému, kdy ke stejným datovým strukturám přistupuje více procesů najednou. Chyba je způsobena chybnou kontrolou pořadí přístupů jednotlivých procesů [26].

Záplata Z anglického *patch* je textový soubor, který jednoznačně definuje rozdíly (změny) v souborech. Změny je možné aplikovat pomocí stejnojmenného příkazu *patch* s použitím různých parametrů.

Clondike záplata V případě Clondike se jedná o změny ve zdrojových kódech především v jádře, napsaném v jazyce *C*. Použití záplaty vygenerované příkazem *diff* se provádí příkazem *patch* s parametrem *-p* (hloubka cesty).

Konfigurace jádra Pomocí konfiguračního souboru `.config` je možné ovlivnit komponenty, které budou zkompilovány. Existují dvě možnosti kompilace dané části, přímo do jádra operačního systému (souboru `vmlinuz`) nebo jako modul. Pro konfiguraci je možné použít příkaz `make menuconfig`, který umožní editovat konfigurační soubor `.config` interaktivním způsobem.

Konfigurace jádra pro Clondike Clondike záplata přidává do konfiguračního souboru pět položek pro instalaci Clondike. Jedná se o instalaci háčků, podporu příchozích (CCN) a odchozích (PEN) migračních procesů. Pro správnou funkci clusteru je nutné, aby všechny tyto komponenty byly nainstalovány přímo do jádra operačního systému. Poslední volitelný parametr slouží pro implementaci rozsáhlých ladících výpisů do logovacích souborů systému. V repozitáři Clondike [30] se nacházejí předpřipravené konfigurační soubory.

Kompilace V kontextu této sekce znamená kompilaci a linkování souborů se zdrojovými kódy do jednoho spustitelného souboru jádra pomocí příkazu `make`. Výsledný soubor obrazu jádra se nazývá `vmlinux`, který je možné komprimovat. Takto zkomprimovaný soubor, který se nazývá `vmlinuz`, se zavádí při startu operačního systému [31].

2.2 Postup tvorby záplaty

V této části je popsán postup úpravy zdrojových souborů čistého jádra. Výsledkem je soubor záplaty `clondike_kernel_3.18.21.patch` v adresáři `patches/` repozitáře [30]. Relativní cesty uvnitř záplaty jsou vztaženy k adresáři se zdrojovými soubory jádra (`linux-3.18.21`).

Z oficiálního Linuxového repozitáře [28] byly staženy zdrojové soubory jádra 3.18.21. Na toto čisté jádro byly následně aplikovány změny, které vycházejí z existující záplaty pro jádro 3.6.11. Z důvodu odlišných zdrojových souborů bylo nutné některé změny upravit tak, aby v novém jádře plnily stejnou funkci jako v jádře původním. Objevily se problémy, kdy bylo třeba nalézt nové umístění původní změny v rámci stejné funkce, případně souboru. Mezi oběma verzemi jádra docházelo ke změně adresářové struktury, což vedlo k hledání souboru, který svoji funkcionalitou nahradil soubor původní. V určitých

Obrázek 2.1: Ukázka výpisu z příkazu patch.

```
1  patching file arch/x86/Kconfig
2  Hunk #1 succeeded at 2513 (offset 278 lines).
3  patching file arch/x86/kernel/entry_64.S
4  Hunk #1 succeeded at 1119 with fuzz 2 (offset -133 lines).
5  ...
6  patching file arch/x86/mm/fault.c
7  Hunk #1 succeeded at 10 with fuzz 1 (offset -1 lines).
8  Hunk #2 succeeded at 823 (offset 37 lines).
9  Hunk #3 succeeded at 1063 with fuzz 2 (offset 45 lines).
10 Hunk #4 succeeded at 1100 with fuzz 2 (offset 45 lines).
11 ...
12 patching file fs/exec.c
13 Hunk #2 succeeded at 1474 (offset -14 lines).
14 Hunk #3 FAILED at 1591.
15 ...
```

případech bylo dokonce třeba upravit původní kód jádra tak, aby bylo možné změnu implementovat.

Po aplikaci všech změn převzatých ze souboru záplaty pro verzi jádra 3.6.11 byla spuštěna kompilace nového jádra. Během této kompilace se objevilo mnoho syntaktických chyb, které souvisejí se změnami zdrojových kódů v jádře. Na základ vypsaných chybových informací, studiu změn ve zdrojových souborech [32] a informací o vývoji jádra [33] byly tyto chyby postupně opraveny. Bylo nutné provést rozsáhlé úpravy jak zdrojových kódů jádra, tak implementace Clondike.

Proces kompilace vždy vypíše jednu chybu (případně sadu chyb), kterou je potřeba opravit. Tento proces se opakuje až po odstranění všech chyb. Jednotlivé chyby a postupy oprav popisují sekce 2.2.1.2 a 2.2.2.

2.2.1 Úpravy čistého jádra

Tato část popisuje úpravy čistého (Vanilla) jádra. Relativní cesty k souborům jsou uváděny vzhledem k adresáři `linux-3.18.21`. Výsledkem této části je nová záplata.

2.2.1.1 Původní záplata

Některé změny bylo možné aplikovat pomocí původní záplaty jádra 3.6.11. Při aplikaci záplaty bylo vypsáno upozornění na změnu umístění (čísla řádku) v novém jádře, ale kontext[2]byl zachován.

Výpis 2.1 představuje ukázku vypsaného upozornění. Z výpisu je patrné, že v souboru `Kconfig` byl nalezen odpovídající kontext o 278 řádků níže než je definováno v souboru záplaty. Soubor `entry_64.S` obsahuje kontext o 133

[2]Kontextem záplaty se zde rozumí 3 řádky nad a pod úpravou.

Tabulka 2.1: Seznam automaticky aktualizovaných souborů pomocí původní záplaty.

Soubor	#	Soubor	#
arch/x86/Kconfig	1	fs/9p/fid.c	1
arch/x86/kernel/entry_64.S	1	fs/9p/v9fs.c	2
arch/x86/kernel/process_64.c	1	fs/9p/v9fs.h	1
arch/x86/kernel/x8664_ksyms_64.c	1	fs/9p/vfs_inode.c	5
arch/x86/mm/fault.c	8	fs/fcntl.c	1
include/linux/signal.h	1	fs/namei.c	2
include/clondike/tcmi/tcmi_dbg.h	1*	fs/namespace.c	2
include/clondike/tcmi/tcmi_hooks.h	1*	fs/open.c	1
include/clondike/tcmi/tcmi_struct.h	1*	kernel/capability.c	2
include/linux/init_task.h	1	kernel/exit.c	4
include/linux/sched.h	3	kernel/groups.c	2
clondike/tcmi/tcmi_hooks.c	1*	clondike/tcmi/Makefile	1*
clondike/tcmi/tcmi_dbg.c	1*	clondike/Kconfig	1*
clondike/Makefile	1*		
include/clondike/tcmi/tcmi_hooks_factory.h			1*

* Nově vytvářený soubor
\# Počet změn

výše než v původním souboru. Záplata souboru `fault.c` obsahuje několik změn, kdy každá změna je posunuta jinak. Seznam souborů, které se podařilo aktualizovat automaticky, se nachází v tabulce 2.1.

Dále je z výpisu 2.1 patrné, že soubor `exec.c` se nepodařilo automaticky upravit, protože kontext záplaty nebyl v novém souboru nalezen. Tento soubor a ostatní podobné je nutné upravit ručně.

2.2.1.2 Změněné soubory

Některé soubory nebylo možné automaticky upravit pomocí původní záplaty pro jádro 3.6.11. Tyto soubory bylo nutné upravit ručně, aby bylo možné vytvořit novou záplatu pro jádro verze 3.18.21. Ve většině případů se jedná o změnu kontextu přidáním nebo odebráním určité části kódu. V následující části jsou relativní cesty k souborům uváděny vzhledem adresáři `linux-3.18.21`.

fs/exec.c Přidání řádku `EXPORT_SYMBOL(do_execve);` je lokalizováno nad funkcí `do_execve()`. Této funkci byl v novém jádře odebrán parametr `regs`, což vedlo ke změně kontextu a selhání automatické aktualizace záplaty.

Z důvodu změny typu parametru `filename` z konstantního řetězce na specializovanou strukturu bylo ve funkci `do_execve_common` změněn parametr

volání Clondike háčku. Nově je předáván ukazatel na jméno souboru jako `filename->name`.

fs/fcntl.c Systémové volání `dup2` bylo přesunuto do souboru `file.c` ve stejném adresáři. V záplatě je definována direktiva exportu této funkce, kterou bylo potřeba také přesunout do nového souboru.

kernel/fork.c Ve funkci `do_fork()` byl změněn název volané funkce `ptrace_event` na `ptrace_event_pid`. Zároveň byl změněn jeden z předávaných parametrů z `pid` na `nr`.

Za koncem definice funkce `do_fork`, kde má být přidána direktiva `EXPORT_SYMBOL`, přibyly definice systémových volání.

kernel/sched/core.c Ze souboru byla odstraněna větší část kontextu uvedená v záplatě starého jádra. Bylo tedy nutné nalézt nové místo pro export symbolu `tasklist_lock`. Kvůli velkému množství preprocesorových podmínek bylo zvoleno umístění na konec souboru. Protože se jedná o export symbolu, není důležité, na jakém místě v souboru se tato direktiva nachází.

kernel/signal.c Ve funkci `get_signal()`, dříve pojmenované jako `get_signal_to_deliver`, má být provedeno volání TCMI háčku. V okolí této změny došlo k několika úpravám. Z volání funkce `ptrace_signal()` byly odstraněny parametry `regs` a `cookie` a struktura `info` typu `siginfo_t` se stala součástí struktury `ksignal`. Stejně tak další část záplaty selhala při změně struktury `info`.

Další selhání souvisí se změnou návratové funkce `get_signal`, která nyní vrací výsledek porovnání `ksignal->sig > 0`. Dříve byl vracen přímo parametr `sig` pojmenovaný `signr`.

kernel/sys.c Tento soubor slouží pro práci s UID a GID v celém systému. Úkolem záplaty je exportovat funkce do systému pro použití v Clondike.

Export systémového volání `setregid()` se nezdařil, protože následující implementované systémové volání se změnilo z `setuid()` na `setgid()`. Za voláním `setfsgid()` byla nově implementována další systémová volání, takže záplata nebyla aplikována kvůli změně kontextu. Export systémových volání `setgid()`, `setuid()` a `setreuid()` se nezdařil pouze díky změně v komentářích. Totéž platí pro dalších několik funkcí, jejichž seznam zde není nutné vypisovat. Úkolem záplaty je vyexportovat všechny zde implementované funkce.

kernel/timer.c Při záplatování tohoto souboru byla vypsána chyba 2.2, která poukazuje na neexistenci souboru. Při bližším studiu bylo zjištěno, že soubor byl přesunut do adresáře `kernel/time/`. Po úpravě záplaty na nové jméno souboru se objevilo několik dalších chyb.

Obrázek 2.2: Chybová hláška záplaty o souboru `kernel/timer.c`.

```
1 can't find file to patch at input line 1565
2 Perhaps you used the wrong -p or --strip option?
3 The text leading up to this was:
4 --------------------------
5 |diff -Naur linux-3.6.11-vanilla/kernel/timer.c \
6          linux-3.6.11/kernel/timer.c
7 |--- linux-3.6.11-vanilla/kernel/timer.c
  2012-12-17 18:27:45.000000000 +0100
8 |+++ linux-3.6.11/kernel/timer.c    2015-04-05 11:40:14.8 +0200
9 --------------------------
```

Systémová volání `getpid()`, `getppid()`, `getuid()`, `geteuid()`, `getgid()` a `getegit()` byla z tohoto souboru odstraněna a přesunuta do souboru `kernel/sys.c` mezi ostatní systémová volání. K těmto voláním implementuje záplata háčky ve formě příslušných systémového volání Clondike. Všechna tato volání jsou exportována pomocí direktivy `EXPORT_SYMBOL`.

Zároveň byla nalezena chyba v záplatě, kdy ze systémového volání `getuid()` byl volán Clondike háček `getgid()`. Tato chyba byla zdokumentována a opravena.

mm/memory.c Ve funkci `handle_pte_fault()` byl nahrazen přímý přístup k proměnné `*pte` za přístup pomocí makra `ACCESS_ONCE(*pte)`.

scripts/Makefile V definici proměnných je v novém jádře přidána položka `CONFIG_ASN1`. To změnilo kontext záplaty pro přidání položky `CONFIG_TCMI` s hodnotou cesty k souboru `dbgenv`.

arch/x86/vdso/vdso32-setup.c V tomto souboru byla záplatou exportována funkce `arch_setup_additional_pages`. Definice této funkce byla v novém jádře přesunuta z tohoto souboru do souboru `vma.c`, který se nachází ve stejném adresáři.

2.2.2 Úpravy implementace Clondike

V této části jsou popsány změny v implementaci Clondike pro jádro 3.18.21. Zdrojové soubory jsou uváděny vzhledem k adresáři `linux-3.18.21/clondike/src/`. Výsledkem této části jsou zdrojové kódy Clondike připravené pro jádro 3.18.21 v repozitáři [30] v adresáři `sources/kernel_3.18.21/src`

director/netlink/comm.c Změna názvu funkce `genl_register_ops()` na `genl_register_family_with_ops_groups()`. Přidána statická struktura typu `genl_multicast_group` nazvaná `gnl_clondike_mcgrps` jako parametr této

Obrázek 2.3: Upravená inicializační funkce Netlink komunikace.

```
1  int init_director_comm(void) {
2      ...
3      /* Register callback for daemin PID registration */
4      genl_register_family_with_ops_groups \
5          (&director_gnl_family, register_pid_ops, gnl_mcgrps);
6      genl_register_family_with_ops_groups \
7          (&director_gnl_family, send_user_message_ops, gnl_mcgrps);
8      ...
9  }
```

Obrázek 2.4: Statická struktura `genl_ops` a její naplnění hodnotami.

```
1  static struct genl_ops ops[] = {
2      {
3          .cmd = 0,
4          .flags = 0,
5          .policy = NULL,
6          .doit = NULL,
7          .dumpit = NULL,
8      }
9  };
10
11 struct genl_ops* genlmsg_register_tx_ops \
12  (struct genl_family *family, struct nla_policy* policy, u8 command) {
13      ops->cmd = command;
14      ops->policy = policy;
15      ops->doit = generic_message_handler;
16      ...
17 }
```

funkce. Struktura obsahuje parametr jméno netlink skupiny „clondike“. Ukázka upraveného souboru je znázorněna v kódu 2.3.

Podobným způsobem byla změněna funkce `genl_unregister_ops()`, která byla volána pro odregistraci každé zaregistrované operace. Nově je podle dokumentace volána pouze funkce `genl_unregister_family()`, která jedním voláním odregistruje všechny zaregistrované operace [32].

director/netlink/genl_ext.c Funkce `genlmsg_register_tx_ops()` vrací ukazatel na strukturu `genl_ops`. Tato struktura byla definována jako dynamická alokovaná pomocí funkce `kmalloc`. V nové implementaci knihovny *Netlink* je tato struktura přepsána na statickou. Bylo nutné definovat tuto strukturu jako statickou a ve funkci ji naplnit hodnotami. Ukázka inicializace funkce a její naplnění je na obrázku 2.4

Obrázek 2.5: Změněný parametr nr_ptes a přiřazení hodnoty.

```
1  // include/linux/mm_types.h
2  struct mm_struct {
3      ...
4      atomic_long_t nr_ptes; /* Page table pages */
5  }
6
7  // tcmi/ckpt/tcmi/ckpcomm.c
8  if ( current->mm ) {
9      atomic_long_set(&current->mm->nr_ptes, 0);
10 }
```

proxyfs/proxyfs_server.c Doplněna chybějící hlavička statického inline systémového volání `_syscall3()`:

```
static inline _syscall3(long, ioctl, unsigned int, \
    fd, unsigned int, cmd, unsigned long, arg);
```

Parametr `f_vfsmnt` struktury `file` byl zapouzdřen navíc do struktury `f_path` a přejmenován na `mnt`. Změna se projevila ve funkci `proxyfs_server_register_poll_callback()`.

tcmi/ckpt/tcmi_ckpt_vm_area.c Funkci `do_mmap_pgoff()` přibyl číselný parametr `populate` předaný pomocí reference. Byly upraveny všechny funkce, které obsahují volání `do_mmap_pgoff()`. Jedná se o funkce `tcmi_ckpt_vm_area_read_l()`, `tcmi_ckpt_vm_area_stack_fixup()` a `tcmi_ckpt_do_mmap()`.

tcmi/ckpt/tcmi_ckptcomm.c Ve funkci `tcmi_ckptcom_restart` se objevila tato chyba:

```
error: too many arguments to function 'do_execve'
```

Po bližším zkoumání problému bylo zjištěno, že parametr `regs`, obsahující hodnoty registrů procesoru, není v jádře od verze 3.8 podporován. Odstranění parametru `regs` vedlo k několika dalším úpravám díky zanoření tohoto parametru v dalších funkcích Clondike. Důvodem k odstranění byla podle [33] a [32] zastaralost parametru.

Ve stejné funkci byl změněn datový typ položky `current->mm->nr_ptes` z typu `int` na `atomic_long_t`. Se změnou datového typu souvisí změna přiřazení hodnoty. Datovému typu `atomic_long_t` není možné přiřadit hodnotu standardně pomocí příkazu `=`. Je nutné využít funkce `atomic_long_set` [32]. Nové přiřazení zobrazuje kód 2.5

/tcmi/ckpt/tcmi_ckpt.c Ve funkci `tcmi_ckpt_map_count()` kompilátor hlásil nedefinovanou proměnnou `VM_RESERVED`. Po přezkoumání bylo v [32]

Obrázek 2.6: Výpis chyb kompilace souboru tcmi_man.c (zkráceno).

```
1 error: macro "hlist_for_each_entry" passed 4 arguments, \
2         but takes just 3
3 In function 'tcmi_man_send_generic_user_message':
4  error: 'hlist_for_each_entry' undeclared (first use in this f...
5  error: (Each undeclared identifier is reported only once
6  error: for each function it appears in.)
7  error: expected ';' before 'break'
8 ...
```

Tabulka 2.2: Nahrazení parametrů int16_t na specializované xxid_t [1].

Popis	Původní dat. typ	Nový dat. typ
Efektivní identifikátor uživatele	int16_t	kuid_t
Efektivní identifikátor skupiny	int16_t	kgid_t
Identifikátor uživatele pro systém souborů	int16_t	kuid_t
Identifikátor skupiny pro systém souborů	int16_t	kgid_t

a [33] zjištěno, že tato konstanta je již od verze jádra 3.7 považována za zastaralou. Při zjišťování příznaků popisující adresní prostor (vma_area_struct->flags) byla odstraněna kontrola na příznak `VM_RESERVED`.

/tcmi/ctlfs/tcmi_ctlfs_entry.h Ze struktury `dentry` byla odstraněna zastaralá položka `d_count`. Byl upraven výpis ve funkci `tcmi_ctlfs_entry_put()`, aby tento člen nebyl vypisován.

tcmi/lib/util.h Položka `pid_ns` datové struktury `nsproxy` byla přejmenována na `pid_ns_for_children` [32].

tcmi/manager/tcmi_man.c Nalezené chyby zobrazuje výpis 2.6 Jedná se o změnu parametrů makra `hlist_for_each_entry`. Parametr `pos` (popsaný jako adresa struktury hlist_node pro použití iterátoru cyklu) byl označen jako zastaralý a byl z makra odstraněn [32].

tcmi/task/tcmi_shadowtask.c Ve funkci `tcmi_shadowtask_emigrate_p()` byl nalezen nesoulad typů několika parametrů, který ukazuje výpis 2.7 Z chybových zpráv je patrné, že byly změněny datové typy identifikátorů uživatelů a skupin z šestnácti-bitových znaménkových čísel na speciální datové typy. Tabulka 2.2 popisuje transformaci datových typů parametrů, která byla provedena v `tcmi/comm/tcmi_p_emigrate_msg.h` v definici struktury `tcmi_msg` [1].

Obrázek 2.7: Výpis chyb kompilace souboru tcmi_shadowtask.c (zkráceno).

```
task/tcmi_shadowtask.c: In function 'tcmi_shadowtask_emigrate_p':
task/tcmi_shadowtask.c:236: error: incompatible type \
                for argument 5 of 'tcmi_p_emigrate_msg_new_tx'
comm/tcmi_p_emigrate_msg.h:97: note: expected 'int16_t' \
                but argument is type 'kuid_t'
task/tcmi_shadowtask.c:236: error: incompatible type for \
                argument 6 of 'tcmi_p_emigrate_msg_new_tx'
comm/tcmi_p_emigrate_msg.h:97: note: expected 'int16_t' \
                but argument is type 'kgid_t'
task/tcmi_shadowtask.c:236: error: incompatible type for \
                argument 7 of 'tcmi_p_emigrate_msg_new_tx'
comm/tcmi_p_emigrate_msg.h:97: note: expected 'int16_t' \
                but argument is type 'kuid_t'
...
```

tcmi/comm/tcmi_p_emigrate_msg.c Parametry funkce `tcmi_p_emigrate_msg_new_tx()` jsou, shodně jako v případě 2.2.2, celočíselného typu `int16_t` místo specializovaných `kuid_t`, respektive `kgid_t`. Tyto parametry byly nahrazeny podle tabulky 2.2. Ve stejné funkci jsou tyto hodnoty přiřazovány do celočíselných proměnných. Namísto struktur je použito přiřazení její celočíselné položky `val`.

tcmi/migration/tcmi_migcom.c Ve výpisu jádra je pro nově zavedené typy podle tabulky 2.2 použit formátovací řetězec `%d`[3]. Nové datové typy jsou struktury, které mají celočíslený typ v položce `val`, proto bylo nutné upravit vypisované položky na `current_*id.val`.

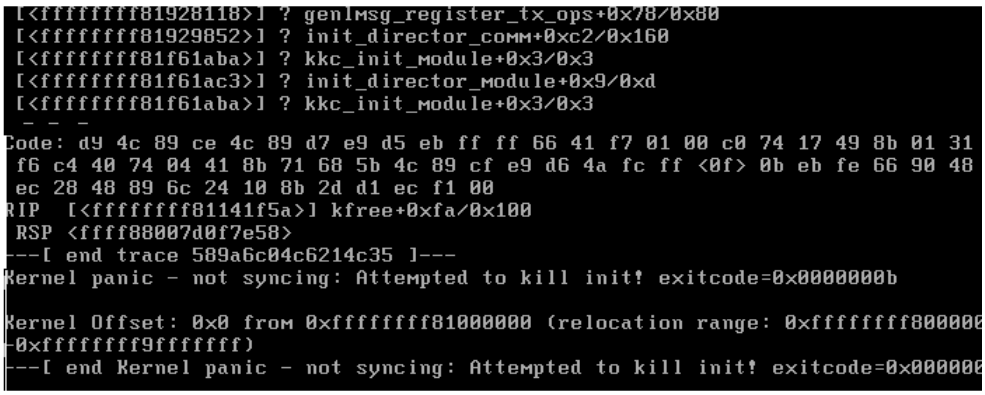
tcmi/migration/tcmi_mighooks.c Funkce `tcmi_try_npm_on_exec()` volá funkci direktoru `director_npm_check()` pro kontrolu migrace. Tato funkce ověří, zda se má proces nepreemptivně migrovat. Druhý předávaný parametr původně byl celočíselného typu, ale nově je typu `euid`. Stejně jako v případě 2.2.2 je nutné předat celočíselnou položku `val`.

tcmi/syscall/tcmi_shadow_fork_rpc.c V implementaci systémového volání `TCMI_SHADOW_RPC_GENERIC_CALL_DEF` je volána funkce `do_fork()`. V této funkci byl odstraněn zastaralý parametr `regs`. Z volání funkce byl tento parametr odstraněn.

arch/i386/regs.c Nebylo možné nalézt soubor `regs.h`, jak popisuje následující chyba:

[3]Formátovací řetězec `%d` se ve funkci `printf()` používá pro výpis celočíselné proměnné [34].

Obrázek 2.8: Výpis selhání jádra po startu systému s novým jádrem 3.18.21 (zkráceno).




```
In file included from clondike/src/arch/current/regs.c:8:
 clondike/src/arch/i386/regs.h:41:30: error: \
    arch/x86_64/regs.h: No such file or directory
```

Po přezkoumání problému se ukázalo, že v jádře je architektura pojmenována *x86* místo *x86_64*. Oprava této chyby znamenala pouze úpravu cesty vkládaného souboru.

2.2.3 Spuštění nového jádra

Po odstranění všech chyb kompilátoru a linkování byl vytvořen soubor nového jádra 3.18.21 (`vmlinuz-3.18.21`). Zároveň byly pro nové jádro vytvořeny soubory `Initrd` a `SystemMap`. Všechny tyto soubory byly, společně s dříve vytvořeným konfiguračním souborem jádra `config`, přejmenovány podle standardní konvence a přesunuty do adresáře `/boot/`. Po spuštění příkazu `update-grub` bylo nové jádro nalezeno a přidáno do startovací obrazovky.

Selhání systému Po spuštění systému s novým jádrem došlo k selhání systému[4]. Výpis procesoru a paměti je znázorněn na obrázku 2.8.

Analýza Byla provedena analýza chybového výpisu, která vedla do upravovaného souboru `clondike/src/director/netlink/genl_msg.c` Z výpisu zásobníku bylo zjištěno, že selhání nastalo ve funkci `genlmsg_registeer_tx_ops()`. Po bližší analýze byla odhalena příčina selhání. Jednalo se o nesprávné uvolňování paměti pomocí funkce `kfree()`, které zůstalo z předchozí implementace, jak je naznačeno v 2.9. Jak je popsáno v sekci 2.2.2, struktura `ops` byla

[4]Anglicky *kernel panic* je stav operačního systému, kdy je detekována chyba, kterou systém nedokáže překonat. Po této chybě systém nereaguje na uživatelský vstup, do konzole je vypsána část paměti a stavu procesoru a běh operačního systému je ukončen [35].

Obrázek 2.9: Ukázka kódu, který způsobil selhání po startu systému.

```
1  struct genl_ops* genlmsg_register_tx_ops( struct genl_family \
2          *family, struct nla_policy* policy, u8 command) {
3      struct genl_ops* ops;
4      int err;
5
6      ops->cmd = command;
7      ops->policy = policy;
8      ops->doit = generic_message_handler;
9      ops->flags = 0;
10     ops->dumpit = NULL;
11
12     if ( (err=genl_register_ops(family, ops)) )  {
13         kfree(ops); /* TOTO JE CHYBNE */
14         return NULL;
15     }
16     return ops;
17 }
```

změněna na statickou strukturu. Uvolnění statické struktury pomocí funkce `kfree()` v jádře způsobí selhání a pád celého systému.

Oprava Protože statické struktury jsou alokovány v datovém segmentu, nelze je uvolňovat. Paměť uvolňovaná pomocí `kfree()` musí být nejprve dynamicky alokována na haldě pomocí `kmalloc()` [36]. Oprava chyby spočívá pouze v odstranění řádku označeného v 2.9.

2.2.3.1 Ověření nového jádra

Po opravě chyby uvolňování statické paměti již bylo možné systém spustit a pokusit se o migraci procesu s novou verzí jádra. Při vynucené migraci pomocí exportu proměnné `EMIG` byla úspěšně odmigrována testovací úloha na druhý uzel, který také používá novou verzi jádra *3.18.21*. Při pokusu o kompilaci jádra se vyskytla chyba typu *race condition*, která se v Clondike vyskytuje už od jádra verze *3.6.11*. Detailně je tato chyba popsána v [4].

Kapitola 3

Sjednocení konfigurace

Cílem této sekce je analyzovat a sjednotit konfiguraci Clondike. Uživatel by měl pomocí jednoho souboru nastavit všechny potřebné parametry tak, aby nebylo třeba upravovat jednotlivé skripty a zdrojové kódy Clondike.

3.1 Analýza konfigurace

V současné verzi konfiguračních parametrů projektu Clondike se jedná o textové řetězce umístěné v jednotlivých skriptech a zdrojových souborech projektu. Některé parametry mají svůj speciální konfigurační soubor. Každý soubor má svoji syntaxi, která většinou není popsána ani zdokumentována.

Interface je použit ve skriptu `clondike-init.sh` jako parametr skriptu `parse-interface-ip.sh`, který vrací aktuální IP adresu, která se používá v dalších skriptech.

Protokol a port, na kterém Clondike naslouchá, lze předat jako parametry ve skriptu `clondike-init.sh` skriptu `listen.sh`. V současné podobě nejsou tyto parametry předávány a skript `listen.sh` používá výchozí hodnoty.

Cesta k souboru `listen` a `adresář` s konfigurací souborového systému jsou definovány ve skriptu `listen.sh` jako textové řetězce.

Bootstrap uzly jsou vypsány v samostatném souboru `BootstrapList.txt` v adresáři `simple-ruby-director`. Seznam Bootstrap uzlů má tvar `IP:port` a oddělovačem záznamů je nový řádek.

Programy pro migraci jsou vypsány v souboru `.migration.conf` v domovském adresáři uživatele *root*. Jedná se o cesty ke spustitelným souborům, které jsou plánovačem kontrolovány na migraci.

Obrázek 3.1: Konstruktor třídy ConfigurationParser.

```
1   #  config file to hash table
2   # @param string   name of config file to load
3   def initialize( configFile )
4      @@ya_directives = Hash.new
5
6      # Check file existence
7      config = Pathname.new(configFile)
8      if not config.exist?
9               error("config file #{configFile} does not exist", 1)
10     end
11
12     @@configFile=configFile
13     if ( ! loadConfigFile() )
14        error("Log file #{@@configFile} can not be loaded", 1)
15     end
16  end
```

3.2 Konfigurační soubor

Konfigurační soubor byl nazván `clondike.conf` a byl umístěn do adresáře `simple-ruby-director`. Do budoucna se počítá s přesunem tohoto konfiguračního souboru do standardního unixového adresáře s konfiguračními soubory `/etc`.

3.2.1 Formát

Pro konfiguraci byl použit jednoduchý textový soubor, který je možné měnit pomocí libovolného textového editoru. Jako struktura konfiguračního souboru byla zvolena textová databáze formátu *klíč, hodnota*, kde byl jako oddělovač použit znak `=`.

Konfigurační direktivy jsou jednořádkové a hodnoty by měly být odolné vůči řádkovým bílým znakům[5]. Na pořadí jednotlivých direktiv nezáleží. Hodnoty jednotlivých direktiv lze uzavřít do apostrofů nebo uvozovek. Konfigurační analyzátor dokáže zpracovat i hodnotu bez uzavíracích znaků, která by ale neměla obsahovat řádkové bílé znaky, protože pak není zaručena správná funkčnost analyzátoru[6].

V konfiguračním souboru je možné používat komentáře pomocí znaku `#`. Analyzátor je schopen rozpoznat jak celořádkový komentář, kde je `#` prvním alfanumerickým znakem na řádce, tak komentář, který začíná v libovolném místě na řádce.

3.2.2 Omezení konfiguračního souboru

Z důvodu přiřazení speciálního významu znakům apostrof ('), uvozovky ('') a křížek (#) nelze tyto znaky použít v hodnotě direktivy.

[5] Za řádkové bílé znaky jsou považovány mezery a tabulátory.
[6] Nezaručená funkčnost analyzátoru vychází z použitých funkcí jazyka Ruby.

3.2.3 Podporované direktivy

Rozhraní (interface) Je rozhraní využívané systémem Clondike. Z tohoto rozhraní bude vedena veškerá komunikace Clondike. IP adresa tohoto rozhraní bude sloužit jako jednoznačný identifikátor uzlu a bude se objevovat ve výzvě příkazového řádku[7].

Listenprot a listenport Představuje definici protokolu a portu, na kterém bude Clondike poslouchat pro komunikaci s ostatními uzly.

Bootstrap Definuje jeden nebo více Bootstrap uzlů. Bootstrap uzlem se v kontextu této práce rozumí uzel, který poskytne identifikátory ostatních uzlů v clusteru. V případě Clondike se jedná o kterýkoliv uzel připojený do clusteru [5]. V konfiguraci je možné definovat více Bootstrap uzlů, jejichž seznam je oddělen čárkou bez mezer.

Listenfile Představuje soubor v *CTLFS*, do kterého je při spuštění Clondike zapsána IP adresa a port ve tvaru IP:port. Tento soubor je používán pro zjištění IP adresy, která je používána v Clondike, například při výzvě příkazového řádku.

Mounterdir Určuje adresář s konfigurací souborového systému pro protokol *9P*. V souboru `fs-mount` se nachází definice typu síťového protokolu, v případě Clondike *9p-global*. Parametry připojení, jako jméno uživatele nebo port, jsou zapsány v souboru `fs-mount-options`. V souboru `fs-mount-device` se nachází IP adresa uzlu.

MaxImmigrationRequests Je maximální počet požadavků na migraci, které uzel přijímá od ostatních. V případě, že je počet požadavků větší než toto maximum, uzel bude další úlohy aktivně odmítat do té doby, než se počet přijatých úloh sníží pod tuto hodnotu.

3.2.4 Použití

V souvislosti s nasazením konfiguračního souboru bylo nutné vytvořit třídu v Uživatelském prostoru pro jeho zpracování. Byl vytvořen soubor `ConfigurationParser.rb` obsahující stejnojmennou třídu.

Tato třída je vytvořena při spuštění Clondike a v konstruktoru znázorněném na obrázku 3.1 zajistí načtení konfiguračního souboru předaného parametrem. Jednotlivé konfigurační direktivy si uloží do asociativního pole implementované v Ruby třídou Hash.

[7] Výzva příkazového řádku (anglicky *bash prompt*) je krátká textová zpráva, která se zobrazuje na začátku příkazového řádku před každým příkazem [37].

Obrázek 3.2: Implementace načítání direktivy *bootstrap* z globální konfigurace.

```
1  def getBootstrapNodes
2    bootstrapList = []
3    bootstraps = @configuration.getValue('bootstrap')
4
5    bootstraps = bootstraps.split(%r{\s*,\s*})
6    bootstraps.each do | bootstrap |
7      $log.debug("Parsing bootstrap #{bootstrap}")
8      ipAndPort = bootstrap.split(':')
9      ip = ipAndPort[0]
10     port = ipAndPort[1]
11     if( (ip !~ Resolv::IPv4::Regex) or ( !(Integer(port) rescue false) ) )
12       $log.warn "In reading configuration skip the bootstrap: '#{bootstrap}'"
13       next
14     end
15     bootstrapList.push( NetworkAddress.new(ip, port) )
16   end
17   return bootstrapList
18 end
```

3.2.4.1 Načtení konfiguračního souboru

Načtení konfiguračního souboru provádí metoda `loadConfigFile()` třídy ConfigurationParser. Metoda čte konfigurační soubor po řádcích. Jako první odstraní z řádky komentáře a znak nového řádku. Poté analyzátor rozpozná, v čem je uzavřena hodnota direktivy a podle oddělovače vyhodnotí direktivu a její hodnotu. Metoda používá pro analýzu řádku regulární výrazy, jejichž ukázka je na obrázku 3.3.

Příklad regulárního výrazu pro analýzu řádku, který má hodnotu uzavřenou v uvozovkách vypadá takto:

```
^#{REGEX_DIRECTIVE}=#{REGEX_SPACEANDTAB}*".+"#{REGEX_SPACEANDTAB}*$
```

3.2.4.2 Získání hodnoty

Získání hodnoty příslušné direktivy zajiťuje metoda `getValue()` s parametry `directive` a `defaultValue`. Tato metoda ověří, jestli se direktiva v konfiguraci nachází a pokud ano, vrátí hodnotu dané direktivy. V opačném případě vypíše do logu varování a vrátí `null`.

3.2.4.3 Změny implementace

V souvislosti se změnami v konfiguraci bylo nutné upravit implementace některých Ruby skriptů. V souboru `ClondikeInit.rb` jsou řetězce nahrazeny voláním metody `getValue()` s parametrem názvu direktivy a nepovinnou výchozí hodnotou.

Bootstrap uzly Změny byly provedeny při načítání bootstrap uzlů z nového konfiguračního souboru. V původní implementaci byly čteny jednotlivé řádky ze specializovaného konfiguračního souboru `BootstrapList.txt`

Obrázek 3.3: Příklad regulárních výrazů použitých ve třídě ConfigurationParser.

```
1   REGEX_CHAR               = '[A-Za-z0-9_-]'
2   REGEX_DIRECTIVE          = '\s*(' + REGEX_CHAR + '+)\s*'
3   REGEX_COMMENT            = '\s*#.*$'
4   REGEX_LINEOFWHITESPACES  = '^\s*$'
5   REGEX_SPACEANDTAB        = '[ \t]'
```

v adresáři `simple-ruby-director`, který obsahoval pouze konfiguraci Bootstrap. Nově je přečtena direktiva *bootstrap* z globálního konfiguračního souboru `clondike.conf`. Hodnota direktivy může obsahovat několik Bootstrap uzlů oddělených čárkou. Kód 3.2 zobrazuje způsob načítání Bootstrap uzlů z globálního konfiguračního souboru. Oproti původní implementaci byla na řádcích 11 – 14 přidána kontrola IP adresy a portu. V případě neplatné IP adresy nebo portu je tento záznam ignorován a do logovacího souboru je vypsáno varování.

Maximum přijatých migrací Ve třídě `LimitersImmigrationController` bylo místo konstantního počtu maxima přijatých úloh implementováno načtení příslušné hodnoty z globálního konfiguračního souboru. Do konstruktoru třídy bylo nutné přidat ukazatel na třídu s konfigurací, aby bylo možné volat její metodu `getValue()`.

3.3 Úprava řídících skriptů

Cílem této části je sjednotit řídící skripty projektu Clondike, aby bylo možné celý systém ovládat standardními Linuxovými nástroji. Sekce pojednává o studiu principů používaných v linuxovém prostředí a nahrazení řídících Shell skriptů za skripty napsané v jazyce Ruby.

3.3.1 Současný stav

V současné verzi systému Clondike se o běh starají Shell skripty, pomocí kterých lze Clondike zapnout, vypnout nebo restartovat. Tyto skripty jsou umístěny v adresáři `scripts/` společně s pomocnými a instalačními skripty.

parse-interface-ip.sh Jednoduchý Shell skript, který vyžaduje jako parametr název síťového rozhraní. Účelem skriptu je vypsat na standardní výstup IP adresu příslušného rozhraní s pomocí nástrojů `ifconfig`, `awk`, `sed` a `cut`. Pokud příslušná IP adresa neexistuje, skript tuto situaci nekontroluje a vypíše nepredikovatelný řetězec, který odpovídá posloupnosti příkazů.

clear-current-config.sh Shell skript, který smaže konfiguraci Clondike. Tato konfigurace je umístěna v adresáři `userspace/simple-ruby-director/conf/`.

Obsahuje privátní a veřejný klíč a certifikát lokálního uzlu. Dále pak obsahuje soubory `msg-seq` a `seq` s čísly, které identifikují další posílanou zprávu,

listen.sh Skript očekává až 3 vstupní parametry a jeho účelem je zapnout sdílení pomocí NPFS. První parametr je protokol (tcp nebo udp), pomocí kterého bude komunikace probíhat. Druhým parametrem je IP adresa lokálního uzlu a třetím port, na kterém má uzel naslouchat příchozím spojením. Skript zapíše parametry sdílení do souborů, které se nacházejí ve speciálním souborovém systému *CTLFS* v adresáří `/clondike/ccn/mounter/`. Zapsané jsou například parametry přípojného bodu nebo uživatele s jehož právy se klient připojí. Pro aktivaci naslouchání s danými parametry je nutné zapsat řetězec ve tvaru '*protokol:IP_adresa:port*' do souboru `/clondike/ccn/listen` a jádro systému zajistí spuštění sdílení. Správná funkčnost je ověřena výpisem jádra:

```
Starting listening on tcp:<IP_adresa>:<port>...
Done
```

Posledním krokem skriptu je namapování souborového systému typu *proxyfs* do adresáře `/mnt/proxy`.

clondike-init Inicializační skript, který slouží k zapnutí systému Clondike. Využívá skriptu `parse-interface-ip.sh` ke zjištění platné IP adresy na rozhraní, které se předává jako parametr. Tento parametr je nutné ručně upravit na síťové rozhraní, které bude využíváno pro komunikaci Clondike. Volání skriptu `clear-current-config.sh` způsobí smazání konfigurace Clondike. Pro správnou funkci je nutné, aby Clondike naslouchal příchozím spojením. To je zajištěno skriptem `listen.sh`. Posledním krokem pro start systému je spuštění *Simple ruby director* voláním Ruby skriptu `Director.rb` v adresáři `userspace/simple-ruby-director/`. Výstup tohoto Ruby skriptu je přesměrován do logovacího souboru `director.log` v adresáři `/tmp/`.

stop.sh Tento skript slouží k zastavení Clondike, rozpojení existujících spojení a k zastavení právě migrujících procesů. Zastavení Clondike probíhá pomocí příkazu `pkill` s argumentem `ruby`. Tento příkaz způsobí zastavení všech běžících Ruby skriptů, což může uživateli působit problémy. Odpojení uzlů zařídí jádro po zapsání jedničky do souborů `stop` všech uzlů umístěných v *CTLFS* v adresářích `pen/nodes/*/` a `ccn/nodes/*/`. Poté se opět volá příkaz `kill`, tentokrát s parametrem *-9 (SIGKILL)* pro případ, že by Clondike nereagoval na původní příkaz *terminate*. Pro případ neodpojení uzlů pomocí *stop* se stejným způsobem zapisuje do souboru `kill`. Nakonec se volá příkaz `pkill` na procesy `make` a `cc1` pro zastavení probíhajících migrací.

restart-director.sh Jednoduchý skript, který volá `stop.sh` 3.3.1 a `clondike-init` 3.3.1.

Obrázek 3.4: Ukázka nastavení výzvy příkazového řádku.

```
if [[ ${EUID} == 0 && -d /clondike/pen/nodes && -d /clondike/ccn/nodes ]]
then
  PS1="\[\033[01;31m\]$(cut -d: -f2 /clondike/ccn/listen) "
  PS1+="$(cat /clondike/pen/nodes/count)/$(cat /clondike/ccn/nodes/count)"
  PS1+="\[\033[01;34m\] \W $DLR\\$\[\033[00m\] "
else
  PS1='\[\033[01;32m\]\u@\h\[\033[01;34m\] \w $DLR\\$\[\033[00m\] '
fi
```

Obrázek 3.5: Ukázka inicializačního souboru npfsd.

```
#!/bin/sh
### BEGIN INIT INFO
# Provides:          npfsd
# Required-Start:    $remote_fs $syslog
# Required-Stop:
# Default-Start:     2 3 4 5
# Default-Stop:
# Short-Description: NPFS server
### END INIT INFO

echo "Starting NPFS server"
/root/npfs/npfs -w 1 -p 5577 &
```

bash_prompt.sh Skript sloužící pro nastavení Bash_prompt. Pokud v systému běží Clondike, nastaví Bash_prompt do tvaru `'<IP> <pen>/<ccn>'`, kde *IP* je IP adresa daného lokálního uzlu, *pen* je počet uzlů, ke kterým je lokální uzel připojen a *ccn* je počet uzlů připojených k lokálnímu uzlu. Nastavení výzvy příkazového řádku ilustruje obrázek 3.4. Volání tohoto skriptu po spuštění systému zajišťuje skript `.bashrc` uživatele *root*.

install.sh Instalační skript používaný při instalaci Clondike. Jeho účelem je vytvoření uživatele *clondike*, vytvoření příslušných adresářů a zapsání volání skriptu `bash_prompt.sh` 3.3.1 jako proměnná `PROMPT_COMMAND` do souboru `/root/.bashrc`.

configuration.sh Pomocný skript volaný téměř všemi výše uvedenými skripty, který dynamicky nastaví proměnné `CLONDIKE_HOME`, `CLONDIKE_SCRIPTS` a `CLONDIKE_SIMPLE_RUBY_DIRECTOR` na příslušné cesty.

devel/ Adresář s pomocnými skripty, které jsou používány při vývoji.

npfs Pro správný běh Clondike je nutné, aby byl spuštěný server pro protokol 9P *npfs*. Jeho spuštění zajišťuje skript `/etc/init.d/npfsd`, který je zobrazen ve výpisu 3.5. Démona *npfsd* není možné tímto skriptem ovládat a při volání příkazu `service npfsd *` je uživatel informován hláškou *Starting NPFS server*.

3.3.2 Implementace

Cílem této části je analyzovat nesrovnalosti ve výše uvedených skriptech a pokusit se o jejich opravu. Dosavadní skripty budou přepsány do jazyka Ruby, ve kterém je napsána uživatelská část Clondike. Zároveň si tato část klade za cíl zdokumentovat nově vytvořené Ruby skripty a maximálně zjednodušit jejich používání.

Pro řízení Clondike bylo navrženo použití standardních linuxových nástrojů. Clondike poběží v systému jako služba. Pro práci se službami se v distribuci Debian od verze 8 (Jessie) používá démon *systemd*[8]. Ovládání služeb s použitím *systemd* se provádí pomocí příkazu *systemctl*. V distribuci Debian je možné použít příkaz *service*.

3.3.2.1 Inicializační skript Clondike

Hlavní implementovanou komponentou je inicializační skript `clondike` napsaný v Shell.

Hlavička Na začátku skriptu je standardní definice inicializačního skriptu. Jsou zde definovány služby, na kterých závisí spuštění a které musí být zastaveny. Hlavička obsahuje seznam *runlevel*[9], ve kterých má být služba spuštěna a ve kterých má být naopak zastavena a stručný popis služby.

Podpůrné funkce Pro inicializační skripty jsou implementované základní funkce pro práci s démony, pro různé výpisy a podobně. Tyto funkce jsou implementovány v souboru `/lib/lsb/init-functions`. Jedná se například o funkci `log_begin_msg()` pro výpis zprávy při začátku spouštění a `log_end_msg()` při výpisu OK nebo FAIL v závislosti na návratové hodnotě spouštěcího příkazu.

Dalším podpůrným skriptem je `/lib/init/vars.sh`. Ten definuje parametry předané do jádra[10]a několik dalších parametrů definovaných při zavádění systému. Inicializační skript používá parametr `VERBOSE`, který ovlivňuje informativní výpis při zavádění systému.

Zámkový soubor Z důvodu možnosti dlouhého zpracování inicializačního skriptu pro Clondike je použit semafor pomocí zámkového souboru. Tento semafor umožňuje, že pouze jeden proces může vykonávat inicializační skript (bez ohledu na to, jakou část provádí)

[8] *Systemd* je manažer služeb pro Linux. Je kompatibilní s inicializačními skripty LSB a System V a je schopen je nahradit [38].

[9] *Runlevel* definuje procesy a služby, které mají být automaticky spuštěny po startu systému [39].

[10] Parametry jádra jsou definovány v programu `grub` a po startu systému se nacházejí v souboru `/proc/cmdline` [40].

Obrázek 3.6: Ukázka části kódu zajišťujícího vytvoření démona Clondike.

```
1 PIDFILE="/var/run/clondike.pid"
2 SRDIRECTOR_DIR="/root/clondike/userspace/simple-ruby-director"
3 CLONDIKE="ClondikeInit.rb"
4 CLONDIKE_START="${SRDIRECTOR_DIR}/${CLONDIKE}"
5 DIRECTOR_LOG="/tmp/director.log"
6
7 start-stop-daemon --start --make-pidfile --pidfile "${PIDFILE}" \
8 --exec "${CLONDIKE_START}" >"${DIRECTOR_LOG}" 2>&1 &
```

Obrázek 3.7: Ukázka kódu zajišťujícího zastavení démona.

```
1 # Pouzite promenne shodne s~3.6
2 start-stop-daemon --stop --quiet --pidfile "${PIDFILE}" \
3 --name "${CLONDIKE}" >/dev/null 2>&1
4
5 # Delete pidfile
6 rm "${PIDFILE}" >/dev/null
7
8 # Disconnect nodes
9 find /clondike/*/nodes/[0-9]*/stop -exec sh -c "echo 1 > {} " \; >/dev/null
```

Zamknutí Inicializační skript se na začátku pokusí pomocí příkazu `lockfile-create` vytvořit zámkový soubor. Následně soubor aktualizovat pomocí příkazu `lockfile-touch`, který soubor zamkne a zůstane běžet na pozadí se známým PID.

Druhý proces Kdyby se v tuto chvíli spustil inicializační skript podruhé jako jiný proces (s jiným PID), nemohl by vytvořit soubor a byl by uspán do té doby, než by první proces soubor uvolnil.

Odemknutí Pro odemknutí souboru je nutné nejprve odstranit proces vytvořený příkazem `lockfile-touch` běžící na pozadí pomocí příkazu `kill`. Tím je zámkový soubor uvolněn a je možné ho smazat pomocí příkazu `lockfile-remove`.

Start O start Clondike se stará funkce `do_start()`. Jejím úkolem je pomocí příkazu `start-stop-daemon` s parametrem `-start` vytvořit démona, který bude vykonávat uživatelskou část Clondike. Řádky 7,8 kódu 3.6 představují způsob vytvoření démona, který je reprezentován třídou `ClondikeInit.rb` představenou v sekci 3.3.3. Příkaz `start-stop-daemon` je spuštěn na pozadí s výstupem na standardní vstupy z důvodu, aby bylo možné logovat všechny komponenty do jednoho souboru. Dalším důvodem bylo zajištění funkčnosti mechanismu souborů PID tak, aby ve vytvořeném PID souboru bylo číslo procesu Clondike. Díky použití souboru `ClondikeInit.rb` jako spustitelného je ve výpisu procesů proces Clondike reprezentován jako `ClondikeInit.rb`.

Obrázek 3.8: Výpis logovacího souboru po spuštění systému.

```
1  15:42:21.700504 /etc/init.d/clondike s~parametrem  start
2  15:42:21.701666 /etc/init.d/clondike: before do_start
3  15:42:21.784021 /etc/init.d/clondike: after do_start
4  15:42:21.956102 /etc/network/if-up.d/clondike: start
5  15:42:21.974005 /etc/network/if-up.d/clondike: restart started
6  15:42:22.020278 /etc/init.d/clondike s~parametrem  restart
7  15:42:22.021302 /etc/init.d/clondike: before do_stop and do_start
8  15:42:22.044264 /etc/init.d/clondike: after do_stop and do_start
9  15:42:22.051296 /etc/network/if-up.d/clondike: restart ended
```

Stop Ukončení Clondike provádí funkce `do_stop()` pomocí stejného příkazu `start-stop-daemon` s parametrem `-stop`. Volání příkazu zobrazují první dva řádky kódu 3.7. Příkaz přečte PID z příslušného souboru a vyhledá proces podle jména. Tento proces následně bez výpisu ukončí. Po ukončení je smazán PID soubor a jsou odpojeny všechny připojené uzly.

Status Pro zjištění stavu clusteru je použit příkaz `status_of_proc`. Po předání parametrů PID soubor a jméno démona (`ClondikeInit.rb`) vypíše, jestli Clondike běží nebo je zastavený.

3.3.2.2 Podpůrný inicializační skript Clondike

V adresáři `/etc/network/if-up.d/` byl vytvořen skript `clondike`. Každá změna síťového rozhraní způsobí zavolání všech skriptů v tomto adresáři. Tabulka 3.1 definuje předané parametry a jejich možné hodnoty.

Účelem tohoto skriptu je po inicializaci rozhraní zajistit spuštění, respektive restartování Clondike. Skript reaguje na změny každého rozhraní kromě *loopback*[11]a v současné implementaci pouze pokud rozhraní má IPv4 adresu. V případě, že je Clondike spuštěn, po zapnutí rozhraní je zavolán inicializační skript 3.3.2.1 s parametrem `restart`. Pokud neběží, je použit parametr `start`.

Ověření funkčnosti Pro ověření správné funkčnosti byly do podpůrného a inicializačního skriptu přidány kontrolní výpisy. Oba skripty zapisovaly do speciálního logovacího souboru řádky ve tvaru *<datum> <skript> <akce>*. Z výpisu 3.8 je patrné, že nejprve byla systémem nastartována služba *clondike*. O chvíli později byl spuštěn podpůrný skript, který spustil inicializační skript s parametrem `restart`. Díky tomuto mechanismu je Clondike spuštěn po spuštění systému.

[11]Loopback je speciální rozhraní, které vrací data zpět do zdroje [41].

Tabulka 3.1: Parametry a jejich možné hodnoty [2].

Název	Popis	Možné hodnoty
IFACE	Fyzický název rozhraní	eth0, wlan0, lo...
MODE	Prováděná akce	start, stop
ADDRFAM	Adresní rodina rozhraní	inet, inet6, ipx...
METHOD	Název metody rozhraní	static, dynamic...
PHASE	Detail prováděné akce	pre-up, pre-down, post-up...
VERBOSITY	výpis na standardní výstup	pokud --verbose 1, jinak 0
PATH	Systémová proměnná $PATH	static, dynamic...
METHOD	Název metody rozhraní	static, dynamic...

Obrázek 3.9: Příkaz pro spuštění démona npfsd.

```
1    start-stop-daemon --start --quiet
2        --make-pidfile --pidfile "${PIDFILE}" --background
3        --exec "${NPFSD}" -- -w "${NPFS_THREADS}" -p "${NPFS_PORT}"
```

3.3.2.3 Inicializační skript npfs

V rámci úpravy řídících skriptů byl upraven řídící skript `/etc/init.d/npfsd`, který je svojí strukturou podobný inicializačnímu souboru Clondike 3.3.2.1. Díky úpravě skriptu je možné ovládat *npfs server* pomocí příkazu `service` s parametry `start`, `stop`, `restart` a zjišťovat jeho stav pomocí parametru `status`.

Skript obsahuje standardní hlavičku shodnou s inicializačním skriptem Clondike. Shodně načte soubory pro definici proměnných a pomocných funkcí, definuje funkce pro zamknutí, odemknutím, zapnutí a zastavení.

Zapnutí O zapnutí démona *npfsd* se stará funkce `do_start()`. Tato funkce spustí na pozadí npfs server a jako parametry předá počet obsluhujících vláken a port, na kterém má server naslouchat. PID spuštěného procesu se uloží do vytvořeného souboru pro pozdější zpracování. Příkaz pro spuštění démona npfsd je zobrazen ve výpisu 3.9.

Zjištění stavu Pomocí příkazu `service npfsd status` je možné zjistit stav, ve kterém se démon nachází. Interně je volán příkaz `status_of_proc` s parametrem souboru PID a názvem démona.

Zastavení Zastavení démona provádí funkce `do_stop()` a využívá k tomu vytvořený soubor, který obsahuje PID démona. Ukázka příkazu `start-stop-daemon`, který zajistí zastavení démona je na obrázku 3.10

Obrázek 3.10: Příkaz pro zastavení démona npfsd.

```
1    start-stop-daemon --stop --quiet
2        --pidfile "${PIDFILE}" --name "${NPFS}"
```

Tabulka 3.2: Přehled Shell skriptů a jejich mapování na metody třídy Clondike.

Shell skript	Ruby implementace[12]	Stručný popis
parse-interface-ip.sh	getLocalIP()	vrátí IP adresu rozhraní
clear-current-config.sh	clearCurrentConfig()	smaže soubory konfigurace
listen.sh	listen() mount()	zapíše naslouchací řetězec soubory pro 9P a proxyfs
clondike-init	`service clondike start`	start clusteru 3.3.2.1
stop.sh	`service clondike stop`	zastavení clusteru 3.3.2.1
restart-director.sh	`service clondike restart`	stop a start clusteru
configuration.sh	třída Clondike a konfigurace	definuje potřebné proměnné
bash_prompt.sh	neimplementováno	definuje formát výzvy 3.3.1
install.sh	neimplementováno	prvotní instalace

3.3.3 Úpravy Uživatelského prostoru

V souvislosti se změnou konfigurace bylo nutné změnit vstupní bod do uživatelské části Clondike. Původní vstupní bod `Director.rb` byl nahrazen novým skriptem `ClondikeInit.rb`. Účelem vytvoření skriptu ClondikeInit je nahrazení Shell skriptů, o kterých pojednává kapitola 3.3.1 a zakomponování konfiguračního souboru popsaném v kapitole 3.2.

3.3.3.1 Přesunutí vstupního bodu

V souvislosti s řízením uživatelské části Clondike bylo nutné změnit vstupní Ruby skript z `Director.rb` na `ClondikeInit.rb`. Vytvoření objektu třídy `Logger` zůstalo stejné, pouze úroveň logovaných zpráv je nyní čtena z konfiguračního souboru `clondike.conf`. Prvním vytvářeným objektem je nově vytvořený objekt třídy Clondike, který popisuje sekce 3.3.3.2. Dále je vytvořen objekt třídy `Director`, zavolána jeho metoda `start()`. Metoda start zajistí vytvoření několika vláken, která se starají o chod uživatelské části Clondike. Metoda `waitForFinished()` zajistí čekání na ukončení stejně jako v původní variantě.

3.3.3.2 Nahrazení Shell skriptů

Soubor ClondikeInit obsahuje třídu `Clondike`, která zastupuje funkci několika původních Shell skriptů. Mapování Shell skriptů na metody třídy Clondike zobrazuje tabulka 3.2.

3.3.3.3 Úprava logování

Z důvodu přílišného množství informací logovaných do hlavního logovacího souboru `/tmp/director.log` byl vytvořen dílčí log. Každý spuštěný proces je testován na migraci. Třída v *simple-ruby-director* `ExecutionTimeTracer` obsahuje metody `newTask()` a `endTask()`. Tyto metody jsou volány při vytvoření, respektive ukončení procesu. V metodě endTask jsou vypisovány informace o procesu do logovacího souboru.

Ke každému procesu je vypsán jeho identifikátor PID, cesta ke spuštěnému souboru, návratový kód, IP adresa uzlu, na kterém byla úloha vykonána a doba trvání spuštění úlohy. Pro lepší orientaci v hlavním logovacím souboru byly tyto zprávy přesměrovány do speciálního logovacího souboru `clondike-taskLog` v adresáři `/var/log/`.

[12] Ve většině případů jako metoda třídy Clondike.

Kapitola 4

Návrh a implementace škodné

Cílem této kapitoly je analyzovat a implementovat různé varianty škodné v Uživatelském prostoru systému Clondike. Součástí kapitoly jsou zároveň způsoby a algoritmy pro její odhalení. Prerekvizitou pro algoritmy na odhalování je použití distribuovaného logu migračních transakcí popsaného v sekci 4.4.1.

Pro analýzu a návrh škodných bylo nutné analyzovat průběh migračního procesu. Migrační proces je analyzován v kapitole 5 v sekci 5.1.

4.1 Pojmy

Škodná Škodnou se v kontextu této práce rozumí modifikace zdrojových souborů na jednom uzlu takovým způsobem, že uzel může ovlivnit migrační mechanizmus nebo chování celého clusteru.

Hostitelský uzel Jinak nazýván také *příjemce* úlohy, je uzel, kterému je odeslán požadavek na migraci. Jeho úkolem je v ideálním případě požadavek přijmout, vykonat zadaný kód a výsledek vrátit odesílateli. V Clondike je tento uzel nazýván *PEN*.

Domovský uzel Nebo také *odesílatel* úlohy, je uzel, který spustil úlohu, která má být migrována na jiný (hostitelský) uzel. Migrace úlohy je podmíněna rozhodovacím procesem, který na základě dostupných informací vyhodnotí, zda bude úloha migrována či nikoliv. V Clondike je tento uzel nazýván *CCN*.

Návratový kód Je číselná hodnota, kterou vrací poslední ukončený proces.

Návratový kód úspěchu V prostředí Linux je návratový kód úspěchu roven nule. Tento návratový kód označuje, že proces byl ukončen korektně bez chyb.

Jiffies Je 64-bitová celočíselná neznaménková (unsigned long) hodnota definovaná v souboru `linux/jiffies.h`. Tato hodnota je při startu operačního systému vynulována a vyjadřuje počet tiků hodin od začátku nabíhání operačního systému [42].

Checkpoint soubor Je soubor, který představuje obraz procesu. Soubor obsahuje kontext procesu a samotný vykonavatelný kód.

záznam transakcí Je databázový seznam kroků všech provedených migrací. Obsahuje informace o jednotlivých krocích migrací provedených v clusteru společně s Návratovými kódy, identifikací a časem. Technologický popis je uveden v sekci 4.4.1.

Standardní proces migrace Je označení úspěšného provedení všech kroků migrace. Ve standardním procesu migrace je příjemci migrované úlohy odeslán Checkpoint soubor, který je následně na Hostitelském uzlu rozbalen a spuštěn. Hostitelský uzel vykoná definovaný kód, případně zapíše do souborového systému Domovského uzlu, a odešle Návratový kód úlohy zpět na Domovský uzel. Záznam transakcí standardního procesu migrace je znázorněn v tabulce 4.1.

4.2 Škodná v Prostoru jádra

První možnou variantou injekce škodné je úprava zdrojových kódů linuxového jádra. Po poradě s vedoucím práce bylo rozhodnuto, že úpravy jádra není nutné v kontextu této práce implementovat. Pokud je uživatel schopen měnit zdrojové soubory a následně kompilovat vlastní jádro, má nad uzlem plnou kontrolu a tedy možnost implementovat téměř libovolnou funkcionalitu. Stejný případ platí i pro standardní linuxové jádro bez systému Clondike. Každý uživatel má možnost upravit zdrojové kódy a implementovat libovolnou vlastní funkcionalitu.

4.2.1 Příklady škodných v Prostoru jádra

V této části je naznačeno několik možných variant škodné, které by bylo možné implementovat v Prostoru jádra. Pro implementaci těchto škodných je nutné zkompilovat vlastní jádro s funkcionalitou Clondike. To znamená získat zdrojové kódy čistého jádra (například z [28]), aplikovat Clondike záplatu, získat zdrojové soubory Clondike (z [30]) a zkompilovat vlastní jádro. Tento postup kompilace vlastního jádra s Clondike je popsán v návodu v [30].

4.2.1.1 Nevykonání migrované úlohy

Jedná se o pasivní odmítnutí vykonání migrované úlohy Hostitelským uzlem. Hostitelský uzel přijme úlohu a Checkpoint soubor od Domovského uzlu. V tomto okamžiku nastává změna standardní funkcionality Clondike. Hostitelský uzel úlohu ihned po příjmu odstraní, neprovede rozbalení Checkpoint souboru a nezačne vykonávat kód migrované úlohy. Domovský uzel čeká na provedení úlohy a na zprávu z jádra Hostitelského uzlu pomocí *KKC*. Po uplynutí časového limitu Domovský uzel ukončí čekání na výsledek a považuje migraci úlohy za nezdařenou.

Záznam transakcí Domovský uzel v každém případě zapíše do Záznamu transakcí neúspěšný příjem zprávy s výsledkem úlohy (čtvrtý řádek tabulky 4.1 s Návratovým kódem rozdílným od nuly). Hostitelský uzel může díky modifikaci jádra do Záznamu zapsat libovolný výsledek přijetí úlohy a také odeslání odpovědi. V případě, že je úloha na Hostitelském uzlu hned po přijetí odstraněna, není do Záznamu transakcí zapsán ani jeden z těchto kroků. Výsledný Záznam pro tuto transakci by byl první a poslední datový řádek v tabulce 4.1 s Návratovým kódem na čtvrtém řádku rozdílným od nuly.

Vylepšená varianta škodné by simulovala úspěšné přijetí úlohy. Záznam transakcí by poté vypadal jako kompletní tabulka 4.1 s nenulovým Návratovým kódem na čtvrtém řádku. Stejně by vypadal Záznam, pokud by se při standardním migračním procesu zpráva s legitimní odpovědí reálně ztratila v síti. Případ, kdy hostitelský uzel zároveň odešle falešnou odpověď a Návratový kód na Domovský uzel, je popsán v 4.2.1.3.

4.2.1.2 Zamlčení výsledku migrace

Jedná se o útok na Záznam transakcí, který je možné implementovat na Domovském uzlu, kdy je úspěšně migrovaná úloha vydávána za neúspěšnou. Podle standardního průběhu migrace Domovský uzel odešle požadavek na migraci, Hostitelský uzel jej přijme, vykoná kód a zašle zpět Návratový kód. Ačkoli Domovský uzel úspěšně Návratový kód přijme, do Záznamu transakcí odešle záznam, jako by Návratový kód migrace nebyl odeslán.

Záznam transakcí Záznam transakcí se od standardního 4.1 liší ve čtvrtém kroku, kde je Návratový kód různý od 0. V tomto případě je Záznam transakcí shodný s vylepšenou variantou škodné 4.2.1.1, kdy Hostitelský uzel úlohu neprovede a odešle falešný Návratový kód. Podle Záznamu lze usuzovat, že Hostitelský uzel úlohu nevykonal korektně, ale ve skutečnosti se jedná o snahu Domovského uzlu tento uzel zdiskreditovat. Pokud si Domovský uzel navíc upraví plánovač, může postup opakovat vždy na jeden libovolný Hostitelský uzel a tím jej kompromitovat.

Tabulka 4.1: Ukázka Záznamu transakcí pro případ standardní migrace.

Odesílatel (veřejné klíče)	Příjemce (veřejné klíče)	Časové razítko (Unix formát[13])	Operace (4.2)	ID úlohy (proces-jiffies)	Výsledek (0-255[14])
VK1	VK2	14504436	1	9-gcc-6453154	0
VK2	VK1	14504691	2	9-gcc-6453154	0
VK2	VK1	14504973	3	9-gcc-6453154	0
VK1	VK2	14505108	4	9-gcc-6453154	0

4.2.1.3 Odeslání chybného výsledku

Pomocí modifikace zdrojových souborů jádra je možné upravit proces takovým způsobem, že Hostitelský uzel neprovede kód zaslaný Domovským uzlem, ale pouze odešle libovolný Návratový kód, nejspíše nulu jako úspěch, zpět Domovskému uzlu. Tato modifikace způsobí, že Domovský uzel dostane přes *KKC* zprávu s Návratovým kódem 0. Díky této zprávě nabude Domovský uzel dojmu, že migrovaný proces proběhl úspěšně a jeho výsledek (například zkompilovaný objektový soubor) je zapsán v souborovém systému Domovského uzlu.

Hostitelský uzel ovšem díky úpravě jádra nevykoná přijatý kód, ale pouze odešle na Domovský uzel Návratový kód úspěchu. Domovský uzel není schopen tuto informaci ověřit a do Záznamu transakcí zaznamenává úspěšné přijetí úlohy. Problém je objeven až tehdy, kdy je detekováno selhání následných úloh závislých na neexistujícím výstupu.

Záznam transakcí Hostitelský uzel zapíše oba kroky (operace 2 a 3) jako úspěšné díky modifikaci příslušných zdrojových kódů. Domovský uzel zapíše úspěšné odeslání požadavku na migraci a úspěšné přijetí výsledku migrované úlohy. V Záznamu transakcí je tedy tato transakce zapsána jako úspěšná ve všech krocích jak ukazuje tabulka 4.1.

4.2.1.4 Změna časového razítka

Oba uzly, které se účastní migrace, mají možnost pozměnit časová razítka vkládaná do Záznamu transakcí. Tato modifikace časových razítek může způsobit zhoršení orientace v Záznamu transakcí.

Hostitelský uzel Například Hostitelský uzel má možnost změnit časové razítko příjmu úlohy. Pokud je časové razítko přijetí úlohy posunuto o několik sekund dopředu, případně čas odeslání úlohy posunut dozadu, je čas mezi přijetím úlohy a odesláním výsledku zkrácen. Zkrácení rozdílu mezi odesláním

[13]Čas v sekundách od počátku epochy 1. ledna 1970 [43]

[14]Návratová hodnota, kde nula znamená, že operace proběhla úspěšně a cokoliv jiného chybu.

Obrázek 4.1: Funkce `connectorImmigrationRequestCallbackFunction()` třídy `NetlinkConnector`.

```
def connectorImmigrationRequestCallbackFunction
                  (uid, pid, slotIndex, name, jiffies)
  $log.info("Immigration request for process #{pid} #{name} #{jiffies}")
  result = true
  @immigrationHandlers.each do |handler|
    node = @membershipManager.detachedManagers[slotIndex].coreNode
    result = result && handler.onImmigrationRequest(node, name)
    break if !result
  end
  if result
  else
    $log.info("Immigration request for process #{name} REJECTED!")
  end
  return result
end
```

výsledku a přijetím úlohy se v Záznamu transakcí projeví tak, že Hostitelský uzel vykonal úlohu rychleji, než ve skutečnosti.

Domovský uzel Domovský uzel může změnou vkládaných záznamů ovlivnit délky trvání zpráv. Posunutím času o odeslání úlohy může v Záznamu transakcí zvyšovat nebo snižovat dobu trvání do přijetí úlohy. Změnou času přijetí výsledku má Domovský uzel možnost zvýšit nebo snížit časovou prodlevu doručení výsledku.

4.3 Škodná v Uživatelském prostoru

Druhou možností tvorby škodné je úprava zdrojových kódů v Uživatelském prostoru. Jedná se převážně o kód napsaný v jazyce Ruby. Protože Ruby je interpretovaný jazyk [44], po změně zdrojových souborů není nutná jeho kompilace. Změny se projeví okamžitě po restartu Clondike.

4.3.1 Příklady škodných v Uživatelském prostoru

Tato část se zabývá uvedením několika škodných, které je schopen uživatel jednoho uzlu implementovat v Uživatelském prostoru a pokusit se tím ovlivnit chování clusteru. Změna zdrojových kódů v Uživatelském prostoru je poměrně snadná a nevyžaduje rekompilaci. Pro každou škodnou je naznačen způsob, jakým by bylo možné ji implementovat. Ochraně clusteru před zde uvedenými škodnými se věnuje sekce 4.4.1.4.

4.3.1.1 Zatěžování clusteru

Z podstaty a návrhu systému Clondike, popsané v úvodní kapitole 1 vyplývá, že každý uzel může využívat výpočetní výkon clusteru. Počet migrovaných úloh ani požadavků na migraci není nijak omezen. Je tedy možné generovat

Obrázek 4.2: Příklad skriptu pro zatěžování clusteru.

```
export EMIG=1
while : ; do
    gcc code.c &
done
```

libovolně výpočetně náročné úlohy a požadovat jejich migraci. Tím je možné zahltit celý cluster takovým způsobem, že ostatní uzly nebudou moci migrovat své procesy.

Tato implementace škodné je, jak popisuje článek [8], náročná pro zdrojový uzel. Důvodem je výsledek měření, který ukazuje, že zátěž Domovského uzlu je vyšší, než zátěž Hostitelských uzlů.

Implementace Pokud je cílem uživatele pouze zatěžování clusteru, může napsat jednoduchý Bash skript, který bude opakovat proces vhodný k migraci. Příklad 4.2 ukazuje způsob, jak spustit nekonečný cyklus, který bude spouštět procesy kompilace libovolného zdrojového souboru `code.c`. Export proměnné `EMIG` s hodnotou `1` slouží k vynucení migrace. Bez této proměnné by plánovač vybral pro provedení lokální uzel.

4.3.1.2 Změna identity

V konfiguraci uzlu se nachází jeho identifikátor ve formě veřejného klíče. Tento veřejný klíč je uložen v adresáři `conf/` v Uživatelském prostoru v souboru `public.pem` společně s privátním klíčem `private.pem`. Pokud při startu Clondike klíče v adresáři chybí, je náhodně vygenerován nový pár klíčů, které jsou nadále používány k identifikaci uzlu v clusteru. Vlastník uzlu má možnost tyto klíče kdykoliv smazat, nechat si vygenerovat nové a vystupovat v clusteru jako jeho nový člen.

Implementace Pro odstranění identity stačí zastavit Clondike příkazem `service clondike stop`, smazat obsah adresáře `clondike/userspace/simple-ruby-director/conf/` a znovu nastartovat službu Clondike.

4.3.1.3 Aktivní odmítnutí

Hostitelský uzel má možnost příchozí požadavek na migraci aktivně odmítnout několika způsoby. Jakmile Hostitelský uzel přijme požadavek na migraci procesu, posílá zpět Domovskému uzlu zprávu o odmítnutí požadavku. Ve výchozím stavu je toto odmítnutí zapsáno do Záznamu transakcí. Tento zápis je prováděn v Uživatelském prostoru ve třídě `NetlinkConnector` a je možné jej jednoduše odstranit.

Implementace pomocí konfigurace První možností, jak požadavek aktivně odmítnout, je pomocí parametru `MaxImmigrationRequests` v globálním konfiguračním souboru 3.2.3. V případě, že je parametr nastaven na nulu nebo dokonce záporné číslo, budou aktivně odmítány všechny příchozí požadavky. Je také možné nastavit parametr na nízké číslo (například 1). Pokud v tomto případě přijde požadavek na migraci a na uzlu již je aktivní jiný migrovaný proces, bude nový požadavek aktivně odmítnut.

Implementace úpravou kódu Pro aktivní odmítání požadavků je možné upravovat zdrojové kódy v Uživatelském prostoru. Tyto úpravy nevyžadují změnu jádra a díky implementaci v jazyce Ruby není nutná ani kompilace zdrojových souborů. Aktivní odmítnutí všech úloh je možné docílit úpravou třídy `NetlinkConnector` takovým způsobem, aby funkce `connectorImmigrationRequestCallbackFunction` vrátila `false`. Ukázka funkce je zobrazena na 4.1. Například je možné na čtvrtém řádku inicializovat proměnnou `result` na hodnotu `false`, která v použité implementaci nemůže být nikdy změněna na `true` a je použita jako návratová hodnota z funkce.

4.4 Odhalení škodné

Tato sekce pojednává o způsobech, jakými je možné odhalit škodné v Uživatelském prostoru, které jsou uvedené v sekci 4.3.1. Účelem je vymyslet a nastínit mechanismy, které odhalí uzly neférově využívající prostředky clusteru nebo jinak poškozují ostatní uživatele clusteru.

Obecným řešením tohoto problému je nastavení důvěry mezi jednotlivými uzly. V současném stavu je úloha migrována na libovolný uzel, který byl vybrán plánovačem, ačkoliv se může jednat o uzel, který do této doby ještě nikdy nepřijal žádnou migrovanou úlohu. Tento uzel může pouze využívat výpočetní výkon ostatních uživatelů clusteru. V případě existence nastavení důvěryhodnosti uzlů by se tento uzel jevil jako nedůvěryhodný, úlohy by na něj nebyly migrovány a zároveň by ostatní uzly jeho úlohy nepřijímaly.

Každý proces migrace s sebou nese velký počet informací o zúčastněných uzlech, jako identifikátory obou uzlů, identifikátor transakce, čas a výsledek provedení. Navrženým řešením evidence důvěryhodnosti jednotlivých uzlů je sdílený Záznam migračních transakcí. Hlavní myšlenkou je využít některého z řešení sdílené databáze, která by uchovávala informace o velkém počtu migračních transakcí. Z těchto informací by bylo možné pomocí dotazů vyhodnotit důvěryhodnost daného uzlu.

4.4.1 Sdílený záznam migračních transakcí

Jedná se o databázi jednotlivých kroků velkého množství migrovaných transakcí, která je sdílena v celém clusteru. Účelem této databáze je uchovávat data o co největším počtu transakcí, která budou dostupná pro všechny uzly.

Z důvodu velkého množství dat bylo po dohodě s vedoucím práce rozhodnuto o použití *NoSQL* databáze[15]. Jednotlivé NoSQL databáze se svojí implementací často liší. Pro projekt Clondike byla vybrána databáze *Cassandra*, která je krátce popsána v sekci 5.2.

4.4.1.1 Představa použití

Každý uzel by měl být schopen pomocí dotazu do sdíleného Záznamu transakcí zjistit důvěryhodnost libovolného uzlu. Důvěryhodnost je hodnota reprezentovaná znaménkovým číslem, u kterého platí, čím větší je jeho hodnota, tím je uzel důvěryhodnější. Střední hodnota důvěryhodnosti je vyjádřena hodnotou 0, ale nemusí se jednat o inicializační hodnotu.

Uzel má možnost před migrací procesu na vybraný uzel zjistit veškeré informace o migracích, ve kterých tento uzel figuroval. Na základě získaných dat by mělo být možné spočítat hodnotu důvěryhodnosti. Na základě této hodnoty se uzel může rozhodnout změnit vybraný Hostitelský uzel nebo úlohu odmigrovat. Stejný výpočet důvěryhodnosti uzlu může vypočítat Hostitelký uzel a na základě získaných výsledků rozhodnout, jestli danou úlohu přijme.

4.4.1.2 Použité údaje

Tato část analyzuje údaje, které je potřeba ukládat do Záznamu transakcí, aby bylo možné zjistit hodnotu důvěryhodnosti libovolného uzlu. Pro analýzu na úrovni uzlů je nutné, aby databáze obsahovala identifikátory obou aktérů migrace procesu, identifikátor úlohy a další údaje.

Krok migrace V rámci jedné migrace je nutné zaznamenat několik kroků. Prvním krokem migrace, který je zaznamenán, je rozhodnutí o samotné migraci. Tento záznam vkládá Domovský uzel a je zde definován název úlohy a vybraný Hostitelský uzel. Druhý záznam je do záznamu vkládán Hostitelským uzlem. Z tohoto záznamu musí být zřetelné, jestli Hostitelský uzel úlohu přijal nebo odmítl vykonat. Po vykonání migrované úlohy Hostitelským uzlem musí být zaznamenáno odeslání výsledku úlohy zpět na Domovský uzel. Posledním krokem je zaznamenání přijetí výsledku migrace Domovským uzlem.

Pro uložení kódu migrace do Záznamu transakcí byl použit číselný údaj. Použité mapování číselných údajů na operace je znázorněno v tabulce 4.2.

Uzly Každý záznam musí obsahovat jednoznačné identifikátory obou aktérů migrace, tedy Domovského i Hostitelského uzlu. Pro jednoznačnou identifikaci jsou použity veřejné klíče, které již v implementaci Clondike existují. Tyto

[15] *Not only SQL* přeloženo jako *Nejen SQL* databáze je databázový koncept, který nepoužívá tabulková schémata jako relační databáze. Jedná se o škálovatelné databáze s možností distribuovaného umístění. Data jsou uspořádána s různou možností strukturalizace, většinou typu *klíč-hodnota* [45].

Tabulka 4.2: Seznam operací (kroků) prováděných během migračního procesu.

Kód	Autor	Popis
1	Domovský uzel	Odeslání úlohy k migraci
2	Hostitelský uzel	Přijetí migrované úlohy
3	Hostitelský uzel	Odeslání výsledku migrované úlohy
4	Domovský uzel	Přijetí výsledku migrované úlohy

údaje jsou nazvány jako *Odesílatel* a *Příjemce* a jejich přiřazení závisí na konkrétním kroku migrace.

Identifikátor úlohy Jako identifikátor úlohy slouží číslo a název migrovaného procesu. Použití těchto údajů není dostatečné, protože není zaručena jejich jednoznačnost. Je tedy nutné přidat do položky identifikátoru jednoznačný údaj. Jako jednoznačný údaj byl vybrán parametr `jiffies`, který označuje počet tiků hodin od začátku nabíhání operačního systému. Jednoznačný identifikátor úlohy má tedy tvar `'<pid>-<název_procesu>-<jiffies>'`. Podobný tvar identifikátoru (bez PID) je používán jako název checkpoint souboru. Parametr `jiffies` je popsán v kapitole *Návrh a implementace ukládání dat do DB Cassandra* diplomové práce [10].

Časové razítko Každý vkládaný záznam obsahuje aktuální datum a čas. Jedná se o datum a čas vložení příslušného záznamu do databáze. Pomocí těchto časových údajů lze k jednotlivým transakcím dopočítat například dobu trvání migrované úlohy na Hostitelském uzlu.

Výsledek Každý záznam s sebou nese hodnotu výsledku provedeného kroku, která se vztahuje k uzlu, který daný záznam vkládá. Pro uložení výsledku je použit Návratový kód, kde nulový kód znamená, že operace byla provedena bez chyb a nenulový kód znamená chybu. Pomocí posloupnosti výsledků jednotlivých kroků migrace by mělo být možné zjistit, zda migrace proběhla úspěšně nebo neúspěšně. V případě neúspěšné migrace by mělo být zjistitelné, kde a z jakého důvodu migrace selhala.

4.4.1.3 Implementace

Implementace Záznamu transakcí vyžaduje mnoho dat, které se vyskytují na různých místech. Třída, která bude zapisovat údaje do databáze, bude umístěna v komponentě *Simple ruby director*. Z tohoto důvodu je nutné všechny potřebné údaje odesílat příslušnými zprávami do této komponenty.

V této části je stručně popsán mechanismus odesílání zpráv z Prostoru jádra a následný příjem těchto zpráv v Uživatelském prostoru. Obsáhlejší popis mechanismu odesílání a příjmu zpráv je popsán v diplomové práci [10].

Jiffies Tento parametr se vyskytuje pouze ve zprávě, která informuje Hostitelský uzel o názvu vytvořeného *checkpoint* souboru. Pro použití Jiffies napříč systémem Clondike byl tento údaj přidán do struktury jádra `task_struct`. Ve funkci `tcmi_mighooks_execve()`, která je zavolána hned po spuštění procesu, je provedeno přiřazení hodnoty pomocí stejnojmenné funkce jádra. Byla upravena tvorba checkpoint souboru ve funkci `tcmi_taskhelper_checkpoint()`, aby byl použit uložený parametr `jiffies` struktury `task_struct`.

Odeslání Bylo nutné přidat parametr Jiffies do struktury `tcmi_ppm_p_migr_back_guestreq_procmsg`. Tato struktura je posílána jako informační zpráva od Hostitelského k Domovskému uzlu. V této struktuře se nachází mimo jiné PID procesu a jeho název. Struktura tedy obsahuje všechny tři části jednoznačného identifikátoru úlohy. S přenosem parametru pomocí zprávy souvisí jeho extrahování pomocí zaregistrované funkce. Pro extrakci parametru Jiffies byla implementována funkce `tcmi_p_emigrate_msg_ckpt`, která vrátí hodnotu parametru Jiffies.

Tento parametr je dále přidán do zpráv, které jsou posílány z jádra do Uživatelského prostoru. Přidání do zpráv zajišťuje funkce `nla_put_u64()`.

Příjem Při příjmu zpráv je nutné implementovat extrakci parametru Jiffies a přiřazení do interních struktur v Uživatelském prostoru. Příjem zpráv zajišťují funkce `handle_*()`, kde názvy korespondují s funkcemi pro odeslání `director_*()`. Pomocí funkce `nlmsg_find_attr()` je pomocí identifikátoru ve zprávě nalezen ukazatel na parametr `jiffies`. Pomocí funkce `nla_get_u64()` je z tohoto ukazatele získána hodnota parametru. Získaná hodnota je předána do funkcí typu Callback a použita v komponentě *Simple ruby director*.

PID Je celočíselný parametr identifikující konkrétní proces.

Odeslání Parametr PID bylo nutné přidat do zprávy typu `immigration_request`. Ve funkci `director_immigration_request()` byl údaj PID přidán do volání funkce `immigration_request()`. Protože parametr PID je typu 32-bitového celého čísla, je pro přidání do zprávy použita funkce `nla_put_32()`.

Příjem Příjem a extrakce atributu PID je shodná s příjmem parametru Jiffies Rozdíl je pouze v použité funkci pro extrakci hodnoty z důvodu rozdílných datových typů. Pro parametr PID je použita funkce `nla_get_32()`.

Jméno (name) Jedná se o řetězcový parametr uchovávající jméno migrovaného procesu.

Tabulka 4.3: Použité parametry, jejich datové typy v jazycích C a Ruby a příslušné převodní funkce.

Parametr	C	Ruby	Převodní funkce
Jiffies	unsigned long	Fixnum	ulong2num()
PID	int	Fixnum	int2num()
Name	char*	String	rb_str_new2()

Odeslání Parametr byl přidán do zprávy, která potvrzuje migraci procesu. Ve funkci `director_immigration_confirmed()` byla upravena volání přísušných funkcí. Dále byl parametr přidán do zprávy, která informuje o chybě při migraci. Tato úprava byla implementována do funkce `director_emigration_failed()`. Umístění řetězce do zprávy zprostředkovává funkce `nla_put_string()`.

Příjem Stejně jako u předchozích parametrů je použita funkce pro nalezení ukazatele podle identifikátoru. Extrakci hodnoty zajišťuje funkce `nla_data()`.

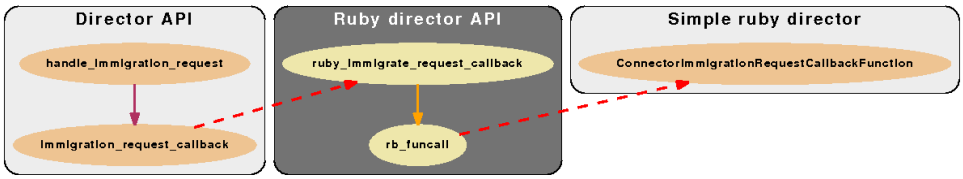
Ruby director API Bylo potřeba upravit komponentu *Ruby diretor API*, která zajišťuje komunikaci v Uživatelského prostoru mezi jazyky C a Ruby pomocí Callback funkcí. Těmto funkcím byly předány nově extrahované parametry, které bylo nutné převést na odpovídající datové typy, které používá jazyk Ruby. Datové typy obou jazyků a odpovídající převodní funkce ukazuje tabulka 4.3. Se změnou parametrů souvisí úprava počtu předávaných parametrů ve funkci `rb_funcall()`, kdy počet předávaných parametrů byl vždy navýšen o odpovídající počet přidaných parametrů. Tyto změny je nutné zkompilovat do knihovny `directorAPI.so`.

Implementací tohoto mechanismu jsou hodnoty předány do komponenty *Simple ruby director*, konkrétně do třídy `NetlinkConnector`. Tato třída registruje své metody jako Callback funkce pomocí následujícího volání.

```
DirectorNetlinkApi.instance.registerNpmCallback(instance, <callback_function>)
```

Tyto metody třídy `NetlinkConnector` mají svůj název ve tvaru `Connector<Callback_funkce>Function`. Parametry těchto funkcí musí odpovídat extrahovaným parametrům v `DirectorApi.so`. Celý mechanismus volání funkcí v Uživatelském prostoru po příchodu zprávy z Prostoru jádra je znázorněn na obrázku 4.4.1.3. Konkrétně se jedná o příchod zprávy typu `DIRECTOR_IMMIGRATION_REQUEST`.

Simple ruby director V komponentě *Simple ruby director* jsou nyní dostupné všechny potřebné údaje pro ukládání dat do sdíleného Záznamu transakcí. Konkrétně se údaje nachází ve třídě `NetlinkConnector`, která zpracovává



Obrázek 4.3: Volání funkcí v Uživatelském prostoru pro zpracování příchozí zprávy.

příchozí zprávy. Tato komponenta bude při příjmu zpráv volat metodu, která předané parametry uloží do databáze.

Cassandra Jako databázové úložiště pro Záznam transakcí byl zvolen projekt *Cassandra* zastřešovaný společností *Apache*. Pro práci s touto databází v jazyce Ruby existuje nástroj *Datastax Ruby Driver for Apache Cassandra* označovaný *CQL3Driver*, který je popsán v [46]. Instalace tohoto nástroje se provádí příkazem `gem install cassandra-driver`. Stažený soubor `CQL3Driver.rb` obsahuje stejnojmennou třídu a třídu `TaskRecord`. V konstruktoru třídy `CQL3Driver` jsou definovány parametry připojení k databázi, kontrola a případné vytvoření `Keyspace` a příslušné tabulky. Třída obsahuje metodu `createRecord()`, která z předaných parametrů vytvoří příslušný záznam a vloží jej do databáze. Databáze Cassandra je blíže popsána v sekci 5.2.

4.4.1.4 Odhalení škodné

Tato část krátce popisuje konkrétní způsoby odhalení škodných v Uživatelském prostoru, uvedených v sekci 4.3.1, s použitím Záznamu transakcí. Implementace těchto algoritmů závisí na implementaci Záznamu transakcí a je nad rámec této práce.

Odhalení škodné vychází z analýzy Záznamu transakcí. Výsledkem analýzy záznamu transakcí je ohodnocení zkoumaného uzlu vyjádřeného číselnou hodnotou důvěryhodnosti. Obecně platí, že čím vyšší hodnota, tím je uzel důvěryhodnější. Hodnota důvěryhodnosti 0 znamená, že je uzel považován za středně důvěryhdný. Záporné hodnoty znamenají nízkou důvěryhodnost uzlu.

Zatěžování clusteru Každá migrovaná transakce, ve které uzel vystupuje jako Domovský, sníží jeho hodnotu důvěryhodnosti. Tím je docíleno, že při velkém počtu vytvořených transakcí je uzel považován za méně důvěryhodný a pro získání kladné důvěryhodnosti musí úlohy také přijímat.

Aktivní odmítnutí Aktivní odmítnutí je penalizováno negativní důvěryhodností, která je úměrná hodnotě důvěryhodnosti druhého uzlu účastnícího

Tabulka 4.4: Příklad hodnocení uzlů při různých průbězích migrace.

Migrace	Domovský	Hostitel
Úspěšná	-1	+4
Přijatá, neodeslaný výsledek	0	-3
Aktivně odmítnutá	0	0 – (-4)
• $(-\infty, -15>$	0	0
• $(-15, 0>$	0	-1
• $(1, 15>$	0	-2
• $(15, 30>$	0	-3
• $(30, \infty)$	0	-4

se migrace. Čím vyšší hodnota důvěryhodnosti druhého uzlu, tím je uděleno vyšší záporné ohodnocení uzlu, který migraci aktivně odmítl. Pro nedůvěryhodné uzly je odmítnutí penalizováno nízkou nebo dokonce žádnou hodnotou.

Změna identity Úroveň důvěryhodnosti nově připojeného uzlu by měla být mírně záporná. Účelem tohoto opatření je zamezení uzlu využívat cluster a poté smazat své identifikátory a opět vystupovat jako nově připojený uzel. Nově připojený uzel si nejprve musí získat svoji důvěryhodnost výpočtem migrovaných úloh. Pro tyto případy musí být tyto nově připojené uzly preferovány jako Hostitelské uzly, aby měly možnost svoji důvěryhodnost zvýšit. V případě problémů je možné nové uživatele clusteru schvalovat, ověřovat a nastavovat jejich důvěryhodnost na neutrální hodnotu 0.

Příklad ohodnocení V tomto odstavci je uveden příklad bodového ohodnocení důvěryhosti používající pro ohodnocení celá čísla. Jedná se pouze o návrh, jehož účelem je pouze ukázat princip Záznamu migračních transakcí na příkladu. Uvedené hodnoty mají pouze informativní charakter. Pro zjištění reálných hodnot by bylo nutné provést důkladnou analýzu a studii, která je nad rámec této práce.

Ohodnocení Základním prostředkem k získání důvěryhodnosti je přijetí a úspěšné vykonání migrované úlohy, které je hodnoceno 4 kladnými body. Nový uzel připojený do clusteru je ohodnocen důvěryhodností -15. Pro dosažení kladného ohodnocení stačí vystupovat v roli Hostitelského uzlu ve 4 úspěšných migracích. Každá úspěšná odchozí migrace je hodnocena 1 záporným bodem, neúspěšná migrace hodnocena není. Neúspěšná migrace, kdy Hostitelský uzel úlohu přijme, ale neodešle výsledek, je penalizována 3 důvěryhodnostními body. Aktivní odmítnutí migrace Hostitelským uzlem znamená ztrátu 0 až 4 bodů, podle úrovně důvěryhodnosti Domovského uzlu. Celý příklad hodnocení shrnuje tabulka 4.4.

Kapitola 5

Dokumentace

Tato kapitola doplňuje dokumentaci projektu Clondike. V první části je podrobně zdokumentován migrační mechanismus. Podle tohoto popisu by mělo být možné správně implementovat Záznam transakcí popsaný v 4.4.1. Další část je pro porozumění myšlenky Záznamu transakcí věnována seznámení s databázovým systémem Cassandra.

5.1 Migrační mechanismus

V této části je popsán mechanismus nepreemptivní migrace procesu. Preemptivní migrace není v systému Clondike zatím podporována.

Migrační mechanismus obsahuje několik hlavních kroků. Na Domovském uzlu se jedná o kontrolu migrace, samotnou migraci úlohy, postup přijetí migrované úlohy na Hostitelském uzlu, její vykonání a odeslání výsledku. Posledním krokem je přijetí výsledku migrace na Domovském uzlu.

5.1.1 Kontrola migrace

Tato část popisuje postup ověření migrace na Domovském uzlu, který začíná spuštěním procesu a jeho kontrolou na migraci. Následuje výběr vhodného kandidáta pro migraci.

Migrační mechanismus začíná v Prostoru jádra spuštěním procesu, který znamená systémové volání. Toto volání je obsluhováno funkcí `sys_execve()`. Jsou volány další funkce `do_execve()` a následně `do_execve_common()`. Do funkce `do_execve_common()` je Clondike záplatou vložen háček ve formě `TCMI_HOOKS_CALL(execve,...)`.

Pomocí tohoto háčku je volána funkce `tcmi_mighooks_execve()`. Tato funkce je vstupním bodem do rozsáhlé implementace Prostoru jádra Clondike. Funkce `tcmi_mighooks_execve()` je rozdělena na tři hlavní části. Protože se

jedná o domovský uzel, je vybrána část, kdy se uzel pokusí proces migrovat zavoláním funkce `tcmi_try_npm_on_exec()`.

Další volanou funkcí je `director_npm_check()`, která se nachází v souboru `clondike/src/director/director.c`. Jejím úkolem je volání funkce `npm_check()` a případně i `npm_check_full()`. Tato funkce posílá pomocí funkce
`msg_transaction_do()` zprávu typu `DIRECTOR_NPM_CHECK` do Uživatelského prostoru pomocí komponenty *Netlink*.

Zpráva `DIRECTOR_NPM_CHECK` je v jádře zpracována funkcí `handle_npm_check()`, která se nachází v Uživatelském prostoru v *director-api* v souboru `npm-check.c`. Tato funkce následně volá zaregistrovanou funkci `npm_callback()`, která je zaregistrována v *Simple ruby director* ve třídě `NetlinkConnector` jako metoda `ConnectorNpmCallbackConnector()`. Princip mapování funkcí mezi jazyky *C* a *Ruby* popisuje sekce *Propojení Ruby a C* v kapitole *Dokumentace* diplomové práce [10].

V komponentě *Simple ruby director* je volána metoda `onExec()` pro všechny zaregistrované třídy. Jedná se o metodu `onExec()` třídy `ProcTrace`, která zajišťuje shromažďování statistických údajů. Tato funkcionalita je v současné implementaci Clondike potlačena.

Ve třídě `TaskRepository` je kontrolováno jméno migrovaného procesu proti regulárnímu výrazu. V současné implementaci má regulární výraz tvar `.*`, což znamená, že je vždy vrácena hodnota `true`, která povoluje pokračování v procesu migrace.

Metoda `onExec()` třídy `LoadBalancer` volá pro Domovský uzel metodu `OnExecCore()` a ta kromě jiného hledá vhodného kandidáta pro migraci pomocí `getEmigrationTarget()`. Nejdříve je v metodě
`isEmigrationPreffered()` testována hodnota proměnné prostředí `EMIG`. Pokud je tato proměnná rovna jedné, je proces automaticky migrován. Pokud není proměnná nastavena, rozhoduje o migraci samotná třída vyrovnání zátěže (`Load balancer`). Ta podle příslušných parametrů zajistí buď lokální provedení procesu nebo jeho migraci na nejvhodnější uzel.

V Clondike jsou implementovány čtyři třídy různých strategií vyrovnání zátěže pojmenované `*BalancingStrategy`. Současná implementace využívá třídu `QuantityLoadBalancingStrategy`, která pro rozložení zátěže využívá počet právě vykonávaných migrovaných úloh. O nalezení vhodného kandidáta pro migraci se stará metoda `findMigrationTarget()`. Interně tato metoda používá pro nalezení nejlepšího kandidáta metodu `findBestTarget()`, která rozhodne o provedení úlohy lokální nebo její migraci. Zde je opět využit mechanismus proměnné prostředí `EMIG`. Pokud je proměnná nastavena na 1, tato kontrola je přeskočena a úloha je migrována. Pro nalezení nejvhodnějšího vzdáleného uzlu je volána metoda `findBestTargetRemoteOnly()`. Tato metoda ještě prostupuje do další třídy `TargetMatcher` a následně `UserConfiguration`, kde

Obrázek 5.1: Strom volání funkcí při migraci procesu.

```
tcmi_try_npm_on_exec()
└─ tcmi_man_emig_npm()
   └─ self.ops.emigrate_npm() → tcmi_migcom_emigrate_ccn_npm()
      └─ tcmi_shadowtask_new()
         ├─ tcmi_task_submit_method()
         └─ tcmi_taskhelper_attach()
└─ → tcmi_task_emigrate_npm()
   └─ tcmi_shadowtask_emigrate_p()
      ├─ tcmi_taskhelper_checkpoint()
      └─ tcmi_p_emigrate_msg_new_tx()
         └─ tcmi_msg_init_tx()
```

je prováděna kontrola na název migrovaného procesu proti konfiguračnímu souboru `.migration.conf`.

Výsledkem celého popsaného procesu ověřování a hledání vhodného kandidáta pro migraci je návratová hodnota metody `findMigrationTarget()`, který prostoupí zpět až do metody `onExecCore()` ve třídě `LoadBalancer`. Pokud je vrácena prázdná hodnota `null`[16], znamená to, že proces nebude migrován a bude vykonán na Domovském uzlu. V případě vrácení kladného celého čísla se jedná o index do struktury `detachedNodes`, na který bude proces migrován. Seznam `detachedNodes` se nachází ve třídě `CoreNodeManager` implementované v souboru `Manager.rb`. Ostatní třídy pro vyrovnání zátěže se liší pouze ve způsobech hledání nejlepšího kandidáta pro migraci a jejich analýza je nad rámec této práce.

Kontext je pomocí návratových hodnot vrácen do komponenty *director-api* do funkce `handle_npm_check()`, kde je odeslána zpráva `DIRECTOR_NPM_RESPONSE` zpět do jádra pomocí knihovny *Netlink*. Součástí zprávy je i hodnota `DIRECTOR_A_DECISION_VALUE`, která může nabývat hodnoty 0 pro pokračování v migraci nebo jiné celočíselné hodnoty označující konkrétní chybu. Případná chyba způsobí přerušení celého migračního mechanismu.

5.1.2 Provedení migrace

Po kontrole migrace se Domovský uzel pokusí proces odmigrovat na vybraný Hostitelský uzel. Na obrázku 5.1 je znázorněno volání jednotlivých funkcí.

[16] V jazyce Ruby se tato konstanta nazývá `nil`.

Po návratu z funkce `director_npm_check()` z předchozí části je kontext přepnut zpět do funkce `tcmi_try_npm_on_exec()`. Zde následuje větvení, zda se jedná o standardní migraci nebo migraci zpět[17]. Proces standardní migrace pokračuje voláním funkce `tcmi_man_emig_npm()`, která je implementována v souboru `tcmi_man.c`. V této funkci je vyhledán a zaregistrován slot pro manažera řídící migraci (*migman*). Slot obsahuje *slot vektor* (`slotvec`), ve kterém se nacházejí informace o migraci a ukazatele na operace (*handlers*). Tento slot je zaregistrován do virtuálního seznamu. Konkrétní manažer poté volá jednu ze zaregistrovaných operací `emigrate_npm()`.

V souboru `tcmi_ccnman.c` je jako obsluha této operace definována funkce `tcmi_migcom_emigrate_ccn_npm()`. Obslužná funkce vytvoří pomocí funkce `tcmi_shadowtask_new()` stínový (*shadow*) proces, ze kterého bude vytvořen *checkpoint soubor*. Tomuto procesu je funkcí `tcmi_task_submit_method()` nastavena migrační metoda `tcmi_task_emigrate_npm()`, která je spuštěna přidáním prováděcího vlákna do fronty operací pomocí funkce `tcmi_taskhelper_attach()`. Tím dojde k přepnutí kontextu a případ, kdy dojde k návratu zpět do funkce `tcmi_migcom_emigrate_ccn_npm()`, znamená chybu migrace.

Ve vykonávání migrovaného procesu pokračuje funkce `tcmi_task_emigrate_npm()`, která pouze volá další funkci `tcmi_shadowtask_emigrate_p()` ve stejném souboru `tcmi_shadowtask.c`. Tuto funkci lze považovat za řídící funkci celé migrace. Jejím úkolem je nejprve vytvořit *checkpoint soubor* voláním `tcmi_taskhelper_checkpoint()`. Tento soubor, obsahující všechny potřebné informace o procesu, je uložen do adresáře `/home/clondike/` pod názvem `<název_procesu>.<PID>.<jiffies>`. Pomocí funkce `tcmi_p_emigrate_msg_new_tx()` je vytvořena nová zpráva Hostitelskému uzlu identifikována jako `TCMI_P_EMIGRATE_MSG_ID`. Obsahem zprávy je název *checkpoint souboru*, jméno procesu, lokální PID a jiné identifikační údaje. Odeslání zprávy zajistí komponenta *KKC* společně s *tcmi/comm*, konkrétně je volána funkce `tcmi_msg_init_tx()`, která rozezná, že se jedná o transakci a vytvoří zároveň i identifikátor transakce.

Vytvoření *checkpoint souboru* a odeslání zprávy znamená korektní vytvoření požadavku na migraci Domovským uzlem. Tím je prozatímní aktivita Domovského uzlu ukončena.

5.1.3 Přijetí procesu

Na Hostitelský uzel je doručena zpráva typu `TCMI_P_EMIGRATE_MSG_ID`, která informuje o příchozím požadavku na migraci. V této části je popsán proces příjmu migračního požadavku probíhající na Hostitelském uzlu.

[17] Migrace zpět není v současné verzi implementována.

Zpráva je přijata a zpracována komponentami *tcmi/comm* a *KKC*. Po přijetí zprávy je zpracování předáno do souboru `tcmi_penmigman.c` funkci `tcmi_penmigman_process_msg()`, kde je rozpoznán typ zprávy a zavolána obslužná funkce. Obslužnou funkcí pro tento typ zprávy je funkce `tcmi_migcom_immigrate()`, která se nachází v souboru `tcmi_migcom.c`. V případě selhání této funkce je vytvořena a odeslána chybová zpráva zpět na Domovský uzel.

Dotaz do Uživatelského prostoru Prvním úkolem obslužné funkce je dotaz do Uživatelského prostoru, jestli se má migrace přijmout. Volaná funkce `director_immigration_request()` volá obecnou funkci `immigration_request()`. Ta připraví a odešle zprávu typu `DIRECTOR_IMMIGRATION_REQUEST` do Uživatelského prostoru pomocí komponenty *Netlink*. Pro tento typ zpráv je v komponentě *director-api* registrována obslužná funkce `handle_immigration_request()`. Tato funkce extrahuje ze zprávy potřebné atributy a provede volání Callback funkce `immigration_request_callback()` V souboru `directorApiExt.c` je jako obslužná funkce zaregistrována `ruby_immigrate_request_callback()`, jejíž implementace se nachází ve stejném souboru. Jejím úkolem je registrovat funkci jako metodu třídy `NetlinkConnector`. Pomocí mapování identifikátoru `registerImmigrateRequestCallback` je volána metoda `connectorImmigrationRequestCallback()` třídy `NetlinkConnector`.

Tato metoda volá metodu `onImmigrationRequest()` všech zaregistrovaných tříd. Registrace tříd je implementována v řídící třídě `Director` a v současném stavu je zaregistrována pouze třída `LimmitersImmigrationController`, registrace třídy `CacheFSController` je zakomentována. Uvedená metoda třídy `LimmitersImmigrationController` opět volá metodu `maximumAcceptCount()` několika v `Director` zaregistrovaných omezovačů (limmiters). Jedná se o třídy `TaskNameBasedAcceptLimiter` a `MeasurementAcceptLimiter`, která ovšem slouží pouze pro měření a v současném stavu neimplementuje žádné omezení. Třída `TaskNameBasedAcceptLimiter` omezuje pomocí regulárních výrazů přijímané migrační procesy podle jejich názvů. Výčet povolených názvů je ve formě pole uveden ve třídě `Director`. Samotná třída `LimmitersImmigrationController` implementuje pouze maximum počtu právě přijatých procesů podle hodnoty v konfiguračním souboru menší než 100. Metoda `maximumAcceptCount()` této třídy vrací `true` pouze pokud požadavek vyhověl všem omezovačům.

Stejným způsobem logického *and* je vytvořena návratová hodnota metody `connectorImmigrationRequestCallback()`. Tato návratová hodnota je odeslána do Prostoru jádra jako zpráva typu `DIRECTOR_IMMIGRATION_REQUEST_RESPONSE`. Pokud je hodnota různá od nuly, je proces migrace ukončen a je poslána zpráva Domovskému uzlu.

Připojení souborového systému V případě povolení příchozí migrace od Uživatelského prostoru je z funkce `tcmi_migcom_immigrate()` pomocí volání `get_new_mounter` získána struktura typu `Mounter`. Tato struktura obsahuje ukazatel na funkci `global_mount()` z komponenty *9P*. Funkce `global_mount()` zařídí připojení souborového systému Domovského uzlu do přípojného bodu `/mnt/clondike/<IP>_<UID>_<GID>` s parametry definovanými v souborovém systému *CTLFS* v `/clondike/ccn/mounter/`.

Spuštění procesu Dalším krokem je vytvoření nového vlákna pomocí nativní funkce jádra `kernel_thread()`. Vláknu je jako kořen souborového systému pomocí předaných parametrů nastaven adresář `/mnt/clondike/<IP>_<UID>_<GID>` připojený na Domovský uzel. Běh vlákna je okamžitě po vytvoření pozastaven voláním funkce `wait_for_completion_interruptible()`.

Pomocí funkce `tcmi_guesttask_new()` je vytvořen nový proces typu *guest*. Tento *guest* proces je nahrán do čekajícího vlákna pomocí funkce `tcmi_taskhelper_attach()`. Vlákno je spuštěno voláním funkce jádra `complete()` a aktuální vlákno čeká na vyzvednutí úlohy jádrem. V inicializační fázi běhu vlákna je formou háčku volána funkce `tcmi_ckptcom_restart()`, která zajistí import dat z příslušného *checkpoint souboru*.

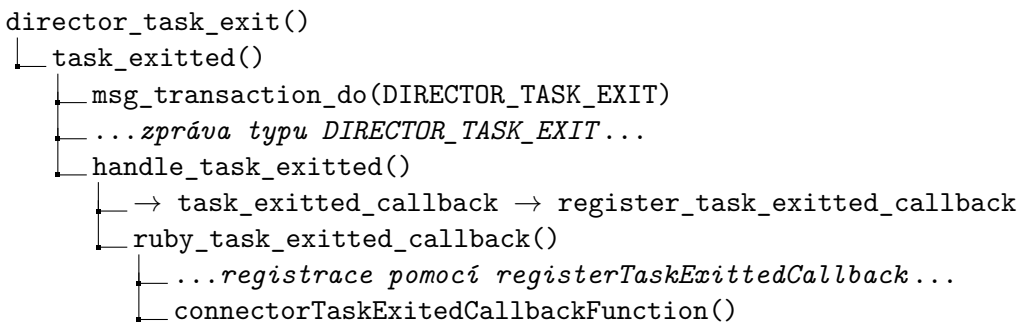
Zpráva do Uživatelského prostoru Posledním krokem funkce `tcmi_migcom_immigrate` je informování Domovského uzlu o úspěšném převzetí migrované úlohy pomocí funkce `director_immigration_confirmed()`, která volá `immigration_confirmed()`. Tato funkce odešle pomocí volání funkce `msg_transaction_do()` zprávu typu `DIRECTOR_IMMIGRATION_CONFIRMED` do Uživatelského prostoru, kde se o její obsluhu stará funkce `handle_immigration_confirmed` implementovaná v souboru `immigration-confirmed.c`. Volaná Callback funkce se nazývá `immigration_confirmed_callback()` a přes identifikátor `register_immigration_confirmed_callback` je volána funkce `ruby_immigration_confirmed_callback()`. Tato funkce je zodpovědná za registraci a volání metody `connectorImmigrationConfirmedCallbackFunction()` třídy `NetlinkConnector` v komponentě *Simple ruby director*.

Metoda `connectorImmigrationConfirmedCallbackFunction()` volá metodu `onImmigrationConfirmed()` všech zaregistrovaných tříd. V současné implementaci je jedinou zaregistrovanou třídou `ImmigratedTasksController`, která uloží migrovanou úlohu do pole `immigratedTasks`.

5.1.4 Ukončení procesu

Po vykonání úlohy na Hostitelském uzlu je pro ukončení procesu volána nativní funkce jádra `do_exit()`. Do této funkce je vložen háček, který způsobí volání

Obrázek 5.2: Strom volání funkcí z `director_task_exit()`.



funkce `tcmi_mighooks_do_exit()`. Tato funkce řídí proces ukončení procesu v Clondike.

Zpráva do Uživatelského prostoru První volanou funkcí je `director_task_exit()`, která zajistí informování Uživatelského prostoru. Postup volání navazujících funkcí je znázorněn na obrázku 5.2. Podrobný popis mechanismu volání a registrace funkcí při volání metody třídy `NetlinkConnector` již byl popsán výše. Metoda `connectorTaskExitedCallbackFunction()` volá metodu `onExit()` zaregistrovaných tříd. Obě zaregistrované třídy `TaskRepository` a `ImmigratedTasksController` odstraní proces ze svého seznamu migrovaných procesů.

Zpráva na Domovský uzel Voláním `tcmi_taskhelper_notify_current()` je vytvořen obal pro funkci `tcmi_task_exit()`, který je následně zařazen do fronty operací. Z této funkce je volána specifická ukončovací funkce `exit()` zaregistrovaná v seznamu `ops`. Ta zajistí odeslání zprávy Domovskému uzlu, která obsahuje Návratový kód migrované úlohy pomocí komponent *tcmi/-comm* a *KKC*.

Dalším krokem je volání funkce `director_disconnect()` a následně `disconnect_director()`. Funkce `disconnect_director()` zajistí uvolnění PID procesu ve strukturách Clondike. Posledním krokem je vyjmutí procesu z obousměrného spojového seznamu procesů. Po ukončení funkce `tcmi_mighooks_do_exit()` je řízení vráceno nativní funkci `do_exit()`, která zajistí korektní ukončení procesu.

Domovský uzel přijme zprávu, ukončí svůj *shadow* proces a přijatou hodnotu považuje za návratovou hodnotu vykonaného procesu.

5.2 Cassandra

Cassandra je distribuovaná *NoSQL* databáze se širokými sloupci (Wide column) typu *klíč – hodnota*. Jedná se o distribuovanou databázi replikovanou napříč množinou uzlů, která je vhodná k ukládání a zpracování velkého množství dat. Pro uložení dat používá tabulky, řádky a sloupce, ale na rozdíl od relačních databází mohou být řádky jedné tabulky značně odlišné.

5.2.1 Vznik a historie

Cassandra byla vyvinuta v roce 2008 za účelem efektivního úložiště uživatelských zpráv v sociální siti *Facebook*. Její vývoj byl inspirován dvěmi NoSQL databázemi *Google BigTable* a *Amazon DynamoDB*. O rok později byly uvolněny zdrojové kódy a projekt byl převzat organizací *Apache Software Foundation*. V současné době je Cassandra projektem s otevřenými zdrojovými kódy, který je vyvíjen společnostmi *Acun* a *Datastax* [3].

5.2.2 Nejdůležitější vlastnosti

Tato část popisuje nejdůležitější vlastnosti databázového systému Cassandra a její odlišnosti od relačních databází [47].

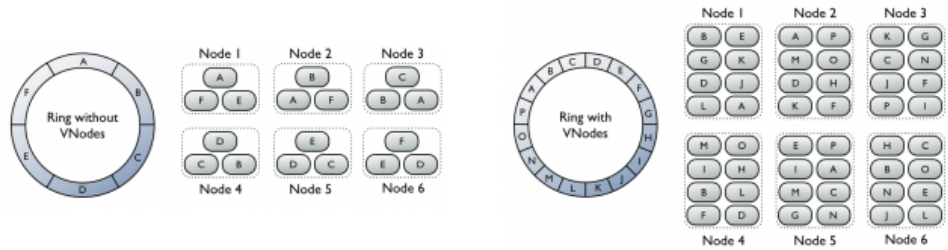
Denormalizovaná a redundantní data Obecně v *NoSQL* databázích neexistuje příkaz `join`, který spojuje více tabulek dohromady, ani žádná jeho alternativa. Z tohoto důvodu je v těchto databázích běžná redundance dat. Nároky na získání chybějících dat jsou mnohem větší než nároky na udržování redundantních dat. Ze stejného důvodu je zároveň nevhodná normalizace dat.

Neexistence jednoho bodu selhání Díky architektuře není struktura databáze Cassandra ani její data ohrožena při výpadku jakéhokoliv uzlu. Data jsou v Cassandra replikována na množinu uzlů. Počet udržovaných kopií je dán *replikačním faktorem*, který je možné měnit i za běhu databáze. Replikační faktor je blíže popsán v sekci 5.2.3.

Škálovatelnost Vytvořený kód funguje stejně na clusteru o jednom lokálním uzlu jako na rozsáhlém clusteru o tisících uzlech.

Cassandra Query Language Jazyk pro tvorbu databázových dotazů pro databázi Cassandra, zkráceně *CQL*. Jazyk *CQL* je velmi podobný jazyku *Structured Query Language* (*SQL*) známého z relačních databází.

Obrázek 5.3: Ukázka rozdílu rozložení dat na fyzických a virtuálních uzlech.



5.2.3 Architektura

Data jsou symetricky rozložena napříč všemi uzly v clusteru, které jsou vzájemně rovnocenné. Důvodem tohoto návrhu je předpoklad nízké spolehlivosti použitého software a hardware [48].

Uspořádání uzlů Databáze Cassandra používá uspořádání uzlů do kruhu, kde každému uzlu přísluší určitá část dat. Přiřazení dat k uzlům probíhá pomocí Hashovací funkce, kdy každému uzlu připadá interval Hashovacích řetězců. Vstupem Hashovací funkce je primární klíč záznamu a výsledný Hash je podle své hodnoty umístěn na příslušný uzel. Tomuto uzlu jsou doručena data a stává se zodpovědným za tvorbu replik podle Replikačního faktoru. Tento uzel obsahuje jednu repliku, ale nejedná se o primární repliku, protože všechny repliky dat jsou rovnocenné [48].

Virtuální uzly V novějších verzích Cassandry je každý uzel rozdělen na několik virtuálních uzlů a teprve těmto virtuálním uzlům je přiřazena určitá část prostoru Hashovacích řetězců. Tím lze dosáhnout lepších výkonnostních výsledků například při obnově znovu připojeného uzlu. Tento princip zároveň umožňuje škálování pomocí mapování různého počtu virtuálních uzlů na jeden fyzický uzel podle jeho výkonu. Rozložení dat na fyzických a virtuálních uzlech je znázorněno na obrázku 5.3 [48].

Replikační faktor Je kladné celé číslo, které určuje počet replikací dat. Hodnota replikačního faktoru se může lišit pro každou databázi. Při použití virtuálních uzlů jsou repliky umisťovány nejen na různé virtuální, ale i fyzické uzly. Všechny repliky dat jsou navzájem rovnocenné a neexistují žádná primární data ani primární repliky.

Úroveň konzistence Jedná se o parametr databáze, který určuje pravděpodobnost správnosti vráceného výsledku. Správným výsledkem se rozumí

Tabulka 5.1: Různé úrovně konzistence pro čtení [3].

Úroveň	**Počet odpovědí**
ONE, TWO, THREE	Definovaný počet (1, 2, 3)
QUORUM	Větší polovina uzlů majících tato data
LOCAL_QUORUM	Větší polovina ze stejného umístění (datacentra)
EACH_QUORUM	Větší polovina z různých umístění (datacenter)
ALL	Všechny odpovědi

Tabulka 5.2: Různé úrovně konzistence pro zápis [3].

Úroveň	**Počet odpovědí**
ANY	Alespoň jeden uzel[18]
ONE, TWO, THREE	Definovaný počet (1, 2, 3)
QUORUM	Větší polovina uzlů
LOCAL_QUORUM	Větší polovina ze stejného umístění
EACH_QUORUM	Větší polovina z různých umístění
ALL	Všechny odpovědi

platná a aktuální data. Úrovně konzistence pro čtení definuje tabulka 5.1 a pro zápis tabulka 5.2.

5.2.4 Klientské požadavky

Požadavek na čtení nebo zápis dat může přijít od kteréhokoliv a na jakýkoliv uzel v clusteru. Příjemce požadavku se stává jeho *koordinátorem* a přebírá odpovědnost za jeho korektní vyřízení [48].

Čtení Koordinátor operace odešle požadavek na zpřístupnění dat všem uzlům, které vlastní požadovaná data. Podle parametru Úrovně konzistence čeká Koordinátor na definovaný počet aktuálních odpovědí. Získaná data poté Koordinátor posílá klientovi.

Zápis Prvním úkolem Koordinátora operace je určit uzly, na které mají být data podle Replikačního faktoru a rozdělovací strategie zapsána. Následně Koordinátor pošle požadavek na zápis dat na všechny tyto uzly. Podle nastavené Úrovně konzistence čeká na odpovědi o úspěšném zapsání od definovaného počtu uzlů. Data jsou považována za zapsaná, pokud se uzlu podaří zapsat data do paměti a seznamu transakcí (*commit logu*). Teprve potom je odeslána zpráva Koordinátorovi o úspěšném zápisu dat. Koordinátor po přijetí

[18] Pokud není ani jeden uzel dostupný, data jsou zapsána do odkladového úložiště a nejsou přístupná až do opravení alespoň jednoho uzlu.

dostatečného množství odpovědí prohlásí požadavek za úspěšný a informuje klienta.

5.2.5 Využití v Clondike

Hlavní myšlenkou využití databáze Cassandra v systému Clondike je vytvoření společného decentralizovaného systému důvěry pomocí Záznamu transakcí, který je popsán v sekci 4.4.1.3. Pomocí tohoto sdíleného Záznamu transakcí by měl být každý uzel schopen vytvořit databázový dotaz, díky kterému by zjistil důvěryhodnost libovolného uzlu. Podle zjištěné důvěryhodnosti by mohl učinit rozhodnutí, jestli má úlohu od daného uzlu přijmout, případně na tento uzel migrovat svoji úlohu.

Závěr

Před samotným zpracováním zadaného tématu muselo být provedeno několik úprav, jejichž výsledkem bylo úspěšné provedení migrace procesu. Po rozsáhlém studiu vnitřních principů a mechanismů byla implementována záplata do novějšího jádra verze 3.18.21. S vývojem záplaty byly úspěšně modifikovány zdrojové kódy Prostoru jádra Clondike, které byly společně se záplatou umístěny do repozitáře [30]. Dále byla otestována funkčnost jádra 3.18.21 a na systému s tímto jádrem byl úspěšně migrován testovací proces kompilace. Během tvorby záplaty se objevilo velké množství chyb, pro jejichž opravu bylo nutné nastudovat a pochopit dílčí funkcionality jádra.

Do Uživatelského prostoru Clondike byl přidán konfigurační soubor `clondike.conf`, který obsahuje základní nastavení Clondike na jednom místě. Díky použité architektuře je možné jednoduše rozšiřovat podporované konfigurační direktivy a implementovat nové možnosti nastavení. Nově je možné ovládat systém Clondike pomocí řídících skriptů jako službu operačního systému příkazem `service`. Stejným způsobem byl jako služba implementován síťový server *npfs*.

Práce obsahuje různé způsoby implementace škodné, které mohou negativně ovlivnit chod clusteru. Práce popisuje škodné v Prostoru jádra i v Uživatelském prostoru. U škodné v Uživatelském prostoru je uvedena jejich možná implementace a je navržen jeden z možných způsobů obrany pomocí sdíleného Záznamu transakcí. Součástí práce je i dokumentace databázového systému Cassandra, na základě kterého je možné Záznam transakcí realizovat.

Práce doplňuje dokumentaci projektu Clondike o detailní popis nepreemptivní migrace procesu. Je zdokumentován celý průběh zahrnující rozhodnutí o migraci procesu, výběr Hostitelského uzlu, samotnou migraci procesu, vykonání procesu na Hostitelském uzlu, odeslání výsledku procesu a jeho přijetí.

Budoucí práce

Chyba v jádře operačního systému Nejzávažnější chybou, kterou je potřeba z projektu odstranit, je chyba jádra typu *race condition*. Jedná se o chybu, která vzniká použitím nulového ukazatele (*null pointer dereference*) a způsobuje selhání systému. Tato chyba se vyskytuje za běhu systému a v náhodných časových intervalech. Díky vícevláknovému zpracování nelze tuto chybu jednoduše vyřešit otestováním na hodnotu *null*. Detailně je chyba popsána v [4].

Sdílený Záznam transakcí Clondike je v současné době připraven na implementaci sdíleného Záznamu migračních transakcí. Pro jeho nasazení je nutné nainstalovat do systému databázový systém Cassandra a povolit ukládání informací o jednotlivých migracích.

Algoritmy pro odhalení škodné Po implementaci Záznamu transakcí je možné vytvořit algoritmy, které budou na základě Záznamu transakcí zjišťovat důvěryhodnost jednotlivých uzlů. Pomocí těchto algoritmů by mělo být možné odhalit případné škodné v clusteru, které popisuje kapitola 4 této práce.

Rozšiřování konfigurace Pomocí implementovaného mechanismu jednotné konfigurace je možné pokračovat ve sjednocování konfigurace Uživatelského prostoru Clondike. Jednotlivé řetězcové konstanty, umístěné v různých zdrojových souborech, lze snadno přesunout do jednotného konfiguračního souboru.

Literatura

[1] Veselský, J.: *LINUX: dokumentační projekt*. Computer Press, druhé vydání, 2003, ISBN 80-7226-761-2, [cit. 2015-12-13].

[2] Aoki, O.: Debian Reference. 2013, [cit. 2015-12-06]. Dostupné z: `https://www.debian.org/doc/manuals/debian-reference/ch05.en.html#_scripting_with_the_ifupdown_system`

[3] Veselý, D.: *Distribuované ukládání a zpracování velkého množství dat - případové studie*. Diplomová práce, České vysoké učení technické v Praze, Fakulta informačních technologií, 2014, [cit. 2015-04-26]. Dostupné z: `https://dip.felk.cvut.cz/browse/pdfcache/veseldom_2014dipl.pdf`

[4] Parallel Computing Group: Clondike. 2011, [cit. 2015-10-12]. Dostupné z: `http://pcg.fit.cvut.cz/structure/clondike`

[5] Tvrdík, P.: *Implementace BitTorrent discovery protokolu do Clondike*. Diplomová práce, České vysoké učení technické v Praze, Fakulta informačních technologií, 2014, [cit. 2015-11-16]. Dostupné z: `https://dip.felk.cvut.cz/browse/pdfcache/tvrdipa1_2014dipl.pdf`

[6] Salát, I. M.: *Rešerše kešovacích mechanizmů v systému Clondike*. Diplomová práce, České vysoké učení technické v Praze, Fakulta informačních technologií, 2014, [cit. 2015-11-16]. Dostupné z: `https://dip.felk.cvut.cz/browse/pdfcache/salatmic_2014dipl.pdf`

[7] Software in the Public Interest, Inc.: Debian – The Universal Operating System. 2015, [cit. 2015-11-28]. Dostupné z: `https://www.debian.org/index.en.html`

[8] Gattermayer, J.; Tvrdík, P.: Different approaches to distributed compilation. In *Parallel and Distributed Processing Symposium Workshops and PhD Forum (IPDPSW)), 2012 IEEE 26th International*, IEEE, 2012, s. 1128–1134, [cit. 2015-10-27].

[9] Rouse, M.: TechTarget - peer-to-peer. 2005, [cit. 2015-10-27]. Dostupné z: http://searchnetworking.techtarget.com/definition/peer-to-peer

[10] Rákosník, J.: *Úpravy jádra operačního systému pro Clondike*. Diplomová práce, České vysoké učení technické v Praze, Fakulta informačních technologií, 2015, [cit. 2015-10-25].

[11] Canonical-Ltd.: Official Ubuntu Documentation - What is GNU/Linux? 1999, [cit. 2015-10-27]. Dostupné z: https://help.ubuntu.com/lts/installation-guide/powerpc/what-is-linux.html

[12] Bourne, S. R.: The Unix Shell. *BYTE, The small systems journal*, ročník 8, č. 10, 1983: s. 187–202, [cit. 2015-11-27]. Dostupné z https://archive.org/stream/byte-magazine-1983-10/1983_10_BYTE_08-10_UNIX#page/n189/mode/2up.

[13] Free Software Foundation: GNU Bash. 2014, [cit. 2015-11-27]. Dostupné z: https://www.gnu.org/software/bash/

[14] Rouse, M.: TechTarget - process. 2005, [cit. 2015-10-27]. Dostupné z: http://whatis.techtarget.com/definition/process

[15] Trdlička, J.: Procesy a vlákna (vznik, stavy, atributy), signály, 2011, [cit. 2015-12-06]. Dostupné z: https://edux.fit.cvut.cz/oppa/BI-UOS/prednasky/BI-UOS-pr07-processes.pdf

[16] The Linux Information Project: User ID Definition. 2005, [cit. 2015-12-27]. Dostupné z: http://www.linfo.org/uid.html

[17] The IEEE and The Open Group: Regular Expressions. 2013, [cit. 2015-11-25]. Dostupné z: http://pubs.opengroup.org/onlinepubs/9699919799/

[18] Postel, J.: RFC 791 - Internet Protocol. 1981, [cit. 2015-10-27]. Dostupné z: http://tools.ietf.org/html/rfc791

[19] Lórencz, R.: Bezpečnost - Hašovací funkce, MD5, SHA, HMAC, 2011, [cit. 2015-04-26]. Dostupné z: https://edux.fit.cvut.cz/oppa/BI-BEZ/prednasky/bez5.pdf

[20] B. Pawlowski and Ch. Juszczak and P. Staubach and C. Smith and D. Lebel and David Hitz : NFS version 3, Design and Implementation. 1994, [cit. 2015-11-27]. Dostupné z: https://www.usenix.org/legacy/publications/library/proceedings/bos94/full_papers/pawlowski.ps

[21] Alcatel-Lucent: Plan 9 from Bell Labs. 2009, [cit. 2015-11-29]. Dostupné z: http://plan9.bell-labs.com/plan9/about.html

[22] Schluting, C.: Understand Linux Virtual Memory Management. 2008, [cit. 2015-11-11]. Dostupné z: http://www.enterprisenetworkingplanet.com/netsysm/article.php/3741281/Understand-Linux-Virtual-Memory-Management.htm

[23] Bellevue, L.: Kernel Space Definition. 2005, [cit. 2015-11-11]. Dostupné z: http://www.linfo.org/kernel_space.html

[24] Bellevue, L.: User Space Definition. 2005, [cit. 2015-11-11]. Dostupné z: http://www.linfo.org/user_space.html

[25] Rákosník, J.: *Úpravy Clondike pro představení open source komunitě.* Bakalářská práce, České vysoké učení technické v Praze, Fakulta informačních technologií, 2013, [cit. 2015-11-24]. Dostupné z: https://dip.felk.cvut.cz/browse/pdfcache/rakosjir_2013bach.pdf

[26] Kolář, P.: Operační systémy. 2005, [cit. 2016-01-03]. Dostupné z: http://www.nti.tul.cz/~kolar/os/os-s.pdf

[27] The Linux Information Project: Kernel Definition. 2004, [cit. 2015-12-12]. Dostupné z: http://www.linfo.org/kernel.html

[28] Linux Kernel Organization, Inc.: The Linux Kernel Archives. [cit. 2015-12-12]. Dostupné z: https://www.kernel.org

[29] The IEEE and The Open Group: Diff. 2001, [cit. 2015-12-12]. Dostupné z: http://pubs.opengroup.org/onlinepubs/9699919799/utilities/diff.html

[30] (PCG), P. C. G.: FIT-CVUT/clondike. 2012, [cit. 2015-10-12]. Dostupné z: https://github.com/FIT-CVUT/clondike

[31] Bellevue Linux: Vmlinuz Definition. 2005, [cit. 2015-12-12]. Dostupné z: http://www.linfo.org/vmlinuz.html

[32] Free Electrons: Linux Cross Reference. 2012, [cit. 2015-12-12]. Dostupné z: http://lxr.free-electrons.com/

[33] Kernel.org git repositories. 2013, [cit. 2015-12-12]. Dostupné z: https://git.kernel.org

[34] MacKenzie, D.: Printf manual. 2015, [cit. 2015-12-13]. Dostupné z: http://www.gnu.org/software/coreutils/printf

[35] Rouse, M.: Kernel panic definition. 2007, [cit. 2015-12-16]. Dostupné z: http://searchenterpriselinux.techtarget.com/definition/kernel-panic

[36] Ferres, L.: Memory management in C: The heap and the stack. 2010, [cit. 2015-12-16]. Dostupné z: http://www.inf.udec.cl/~leo/teoX.pdf

[37] Project, T. L. I.: Command Prompt Definition. 2005, [cit. 2015-11-24]. Dostupné z: http://www.linfo.org/prompt.html

[38] Escoffier, D.: Systemd - system and service manager. 2015, [cit. 2015-11-30]. Dostupné z: https://wiki.debian.org/systemd

[39] Dorn, D.: RunLevel. 2013, [cit. 2015-11-30]. Dostupné z: https://wiki.debian.org/RunLevel

[40] Escoffier, D.: Systemd. 2015, [cit. 2015-12-02]. Dostupné z: https://wiki.debian.org/systemd

[41] IANA: *RFC 3330 - Special-Use IPv4 Addresses*. Internet Engineering Task Force, 2002, [cit. 2015-12-06]. Dostupné z: https://tools.ietf.org/html/rfc3330

[42] Team LiB: Jiffies. 2014, [cit. 2015-12-23]. Dostupné z: http://www.makelinux.net/books/lkd2/ch10lev1sec3

[43] The IEEE and The Open Group: General Concepts. 2013, [cit. 2015-12-18]. Dostupné z: http://pubs.opengroup.org/onlinepubs/9699919799/basedefs/V1_chap04.html#tag_04_15

[44] Ruby Programming Language. 2001, [cit. 2015-10-27]. Dostupné z: https://www.ruby-lang.org/

[45] MongoDB, Inc.: NoSQL Databases Explained. 2015, [cit. 2015-12-20]. Dostupné z: https://www.mongodb.com/nosql-explained

[46] DataStax, I.: Datastax Ruby Driver for Apache Cassandra. 2013, [cit. 2015-10-23]. Dostupné z: http://datastax.github.io/ruby-driver/

[47] Ellis, J.: The Apache Cassandra Project. 2012, [cit. 2015-04-26]. Dostupné z: http://cassandra.apache.org/

[48] DataStax: Datastax. 2015, [cit. 2015-04-26]. Dostupné z: http://www.datastax.com/

[49] Linn, J.: *RFC 1421 - Privacy Enhancement for Internet Electronic Mail: Part I: Message Encryption and Authentication Procedures*. Internet Engineering Task Force, 1993, [cit. 2015-12-20]. Dostupné z: https://tools.ietf.org/html/rfc1421

Příloha A

Seznam použitých zkratek

CCN Cluster Core Node – Domovský uzel

PEN Process Execution Node – Hostitelský uzel

CTLFS Controller Filesystem – Konfigurační souborový systém [25]

PEM Privacy Enhanced – internetový standard používaný pro zabezpečení elektronické pošty pomocí certifikátů [49]

UID User ID – identifikátor uživatele [16]

GID Group ID – identifikátor skupiny [16]

NFS Network file system – Síťový souborový systém[20]

Příloha **B**

Obsah přiloženého CD

- `readme.txt` — stručný popis obsahu CD
- `clondike_repository` — oficiální repozitář Clondike
  - `backup` — archiv souborů z původního kořene
  - `devel` — adresář se soubory pro vývoj (měření a testy)
  - `doc` — dokumentace generovaná pomocí nástroje doxygen
  - `etc` — konfigurační a inicializační skripty
  - `patches` — záplaty pro různé verze linuxových jader
  - `root` — adresář uživatele *root* obsahující konfiguraci a server *npfs*
  - `scripts` — pomocné ovládací skripty
  - `sources` — zdrojové soubory jádra pro různé verze a knihovny Netlink
  - `userspace` — zdrojové soubory Uživatelského prostoru
    - `director-api` — soubory pro komunikaci přes Netlink
    - `ruby-director-api` — rozhraní mezi jazyky C a Ruby
    - `simple-ruby-director` — skripty Uživatelského prostoru v Ruby
  - `INSTALL` — návod na instalaci
  - `README.md` — popis repozitáře
- `diplomova_prace`
  - `DP_Novy_Zdenek_2015` — zdrojová forma práce ve formátu LaTeX
  - `DP_Novy_Zdenek_2015.pdf` — text práce ve formátu PDF
  - `DP_Novy_Zdenek_2015.ps` — text práce ve formátu PS
  - `Zadani_DP_Novy_Zdenek_2015.pdf` — zadání práce ve formátu PDF