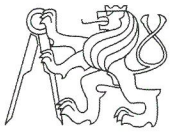
ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ



ZADÁNÍ DIPLOMOVÉ PRÁCE

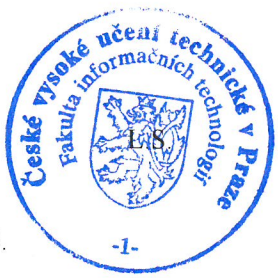
Název:	Úpravy jádra operačního systému pro Clondike
Student:	Bc. Jiří Rákosník
Vedoucí:	Ing. Josef Gattermayer
Studijní program:	Informatika
Studijní obor:	Počítačové systémy a sítě
Katedra:	Katedra počítačových systémů
Platnost zadání:	Do konce letního semestru 2015/16

Pokyny pro vypracování

Práce se zabývá několika dílčími úkony v prostoru jádra operačního systému Linux.
Pokyny:
1) Analyzujte chybu vzniklou voláním funkcí tcmi_transaction_timer a tcmi_transaction_put. Navrhněte řešení, implementujte jej a výsledek otestujte měřením.
2) Analyzujte chybu způsobující rozpojení uzlů. Navrhněte řešení, implementujte jej a výsledek otestujte měřením.
3) Analyzujte chybu task tcmi_commd_xx:xxxx blocked for more than 120 seconds. Navrhněte řešení, implementujte jej a výsledek otestujte měřením.
4) Navrhněte a implementujte mechanizmus ukládání klíčových parametrů migrace z jádra do DB Cassandra.
5) Doplňte dokumentaci projektu Clondike.

Seznam odborné literatury

Dodá vedoucí práce.



Ing. Tomáš Zahradnický, Ph.D.
vedoucí katedry

prof. Ing. Pavel Tvrdík, CSc.
děkan

V Praze dne 28. ledna 2015

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

Diplomová práce

Úpravy jádra operačního systému pro Clondike

Bc. Jiří Rákosník

Vedoucí práce: Ing. Josef Gattermayer

29. dubna 2015

Poděkování

Veliké poděkování patří vedoucímu práce Ing. Josefu Gattermayerovi za vedení této práce i celého projektu, další veliké poděkování patří mým rodičům, kolegům a přátelům za jejich podporu, bez které by tato práce nemohla vniknout.

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 29. dubna 2015 .....

Odkaz na tuto práci

Rákosník, Jiří. *Úpravy jádra operačního systému pro Clondike.* Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2015.

Abstrakt

Cílem této práce je analýza, návrh řešení a oprava dosud známých chyb v uživatelském prostoru a v prostoru jádra operačního systému projektu `Clondike`. V této práci je uveden celý postup provedení opravy jednotlivých chyb a aktualizací. Stručně je zde popsán projekt `Apache Cassandra` a jeho možné propojení a využití. Součástí je dokumentace použité knihovny `Netlink` a obecný princip propojení jazyka C a Ruby.

Klíčová slova Apache Cassandra, Clondike, Jádro operačního systému Linux, Nededikovaný cluster, Netlink

Abstract

The goal of this thesis is an analysis, design solutions and repair of known bugs in userspace and kernelspace of the operating system, especially for the project `Clondike`. In this work is presented a whole repair process each of bugs and updates. Briefly is described here Apache Cassandra project and its possible connections and usage. Included is documentation of used library Netlink and general principle of linking the C language and Ruby.

Keywords Apache Cassandra, Clondike, Kernel, Netlink, Non-dedicated cluster

Obsah

Seznam obrázků

Úvod

Úvodní část práce se zabývá stručným představením projektu `Clondike`[3] a objasňuje pojmy, které jsou důležité k pochopení celé problematiky. Samotný název projektu vychází z anglické zkratky „**CL**uster Of **N**on-**D**edicated **I**nteroperating **KE**rnels“, což volně přeloženo znamená cluster nededikovaných kooperujících jader.

Celý projekt `Clondike` je především unikátní tím, že poskytuje běžným uživatelským stanicím možnost stát se členem skupiny zařízení, které spolu navzájem sdílí hardware pro realizaci výpočetně náročných úloh tvořených větším množstvím procesů v daný okamžik a to bez uživatelské interakce.

Vlivem několikaletého vývoje tohoto projektu se vyskytla množina chyb a nedostatků, které omezují v některých případech zcela znemožňují původní funkcionalitu samotného projektu. Řešení těchto chyb a nedostatků je věnována velká část této práce. Další podstatnou částí je dokončení aktualizace použité knihovny `Netlink` na nejnovější podporovanou verzi a vytvoření evidence počtu připojených uzlů.

Předposlední část je věnována oblasti kolem využití a ukládání dat do databáze distribuovaného systému, který poskytuje projekt `Apache Cassandra`[4]. V poslední části této práce je uvedena dokumentace dosud nepopsaných principů a mechanismů využitých při komunikaci mezi jádrem operačního systému a uživatelským prostorem.

Kapitola 1

Analýza projektu

1.1 Příprava

Tato práce započala přípravnou fází, která mimo jiné obsahovala nastudování magisterských prací kolegů Ing. Pavla Tvrdíka[5] a Ing. Michala Saláta[6]. Především byla snaha ověřit funkčnost celého systému migrací a zjistit chyby a nedostatky již na počátku. Při kontrole bylo zjištěno mnoho malých a velkých chyb. Malé chyby byly záhy vyřešeny konzultací se zmíněnými kolegy. Příkladem je chyba špatného spojení vývojových větví ve vývojovém repositáři GITHub, kde jsou zdrojové kódy projektu `Clondike` uloženy. Velké chyby byly obecně známy, a proto předmětem této práce je konkrétní chyby analyzovat a opravit.

Další podstatnou částí byla aktualizace linuxové distribuce `Debian` z verze 6 na verzi 7 `(squeeze -> wheezy)` z důvodů instalace software `Apache Cassandra`, jehož základní popis je popsán v kapitole 3.3. Po aktualizaci linuxové distribuce byl celý projekt `Clondike` a příslušně upravené jádro verze 3.6.11 otestováno a nebyl na první pohled shledán žádný nový problém s nekompatibilitou.

Bylo zjištěno od kolegů, že jejich diplomové práce byly vypracovány s původní starou verzí jádra 2.6.33.1, což nebylo úplně správné, ale vzhledem k vývoji pouze v uživatelském prostoru není tento fakt zásadní. Na základě tohoto faktu bylo provedeno testování na novém jádře 3.6.11, které se prokázalo až na drobnosti a velké chyby bezproblémové.

Dále během testování různých komponent včetně kompilace jader bylo zjištěno, že kompilátor `gcc-4.7` není schopen přeložit staré jádro 2.6.33.1, pro jeho správnou kompilaci byl použit kompilátor `gcc-4.4`.

V poslední řadě byla dočasně vyřazena implementace `Cache Clondike File System (CCFS)`, která způsobovala pády migračního procesu. Tato komponenta není základním pilířem celého projektu Clondike, složí pouze k optimalizaci a kešování, což vzhledem k závažnějším problémům bude řešeno v budoucnu.

Byla vytvořena minimalizovaná funkční konfigurace jádra v3.6.11, pro rych-

lejší vývoj a kompilaci do budoucna vytvářených změn zdrojových kódů projektu `Clondike`.

1.2 Základní pojmy

V této části práce jsou uvedeny základní pojmy, které se vyskytují na řadě míst a je vhodné tedy vysvětlit jejich význam v kontextu této práce. U některých pojmů se nejedná o jedinou definici, mnoho pojmů v různých oblastech IT mohou být definovány odlišně. U většiny pojmů jsou uvedeny citace, ze kterých bylo čerpáno nejen pro samotnou definici, ale i pro studium problematiky s pojmem spojené.

Uzel V tomto kontextu uzel pojmenovává počítač připojený do počítačové sítě [7].

Cluster Skupina síťově propojených počítačů kooperujících na stejném cíli [7].

Nededikovaný cluster Typ clusteru, ve kterém nejsou počítače plně vlastněny clusterem, a proto mohou být v daný okamžik používány jako běžné pracovní stanice. Procesy nejsou clusterem migrovány na tyto stanice (uzly) v okamžiku, kdy je uživatelé aktivně používají. [7]

Proces Instance počítačového programu, který lze vykonat [8].

Operační systém Základní programové vybavení počítače, které uživateli umožňuje práci s počítačem. Jedná se o komplexní software zajišťující vytvoření procesů, správu systémových prostředků a hardwaru [9].

Linux Operační systém vycházející z `UNIX` [10] svobodně vyvíjen a šířen. Hlavní jeho komponentou je linuxové jádro (`kernel` [11]).

Ruby Interpretovaný objektový skriptovací jazyk, jehož největší výhodou je jednoduchá syntaxe a široké použití [12] [13].

Jazyk C Procedurální, nízkoúrovňový, kompilovaný, relativně minimalistický programovací jazyk, vznikl pro potřeby operačního systému `UNIX`. [14] a [15]

Migrace procesu Technika umožňující vykonání procesů na vzdálených počítačích. Existují dva základní typy migrace procesů **preemptivní** a **nepreemptivní**.

Preemptivní migrace procesu Tato technika umožňuje přesunout proces mezi dvěma uzly v jakémkoli čase jejich vykonání. Jedná se o mocnou techniku, která je složitější na implementaci, ale je více flexibilnější a umožňuje systém přizpůsobit distribuci procesu na základě aktuálních podmínek. Toto je obzvláště důležité v situaci, kdy na nepoužívané pracovní stanici je vykonáván migrovaný proces a uživatel této stanice se rozhodne ji opět používat [7].

Nepreemptivní migrace procesu Základní typ migrace umožnující vykonání procesů na vzdálených strojích, ale stroj musí být vybrán na začátku vykonání procesů. Tato technika může být provedena i bez změny kódu v režimu jádra, ale klade větší důraz na plánovače, pakliže dojde ke špatnému rozhodnutí, tak jej nelze jednoduše opravit [7].

Broadcast paket Speciální typ síťového paketu, který obdrží všechna zařízení v síti [16].

IP adresa Tento projekt zatím využívá pouze adresy `IPv4` [17], zkratka `IPv4` znamená Internet Protokol verze 4.

Patch V překladu záplata. Jedná se o soubor nebo více souborů obsahující rozdíly oproti původním zdrojovým souborům.

Checkpoint Obraz procesu ve formě souboru obsahujícího kontext migrovaného procesu a samotný vykonavatelný kód.

DHT Distribuovaná hašovací tabulka[5], jejíž záznamy jsou ve formě:

```
{klíč->hodnota}
```

Soket Koncový bod mezi procesové komunikace, je zajišťován operačním systémem.

Netlink Protokol poskytující komunikaci mezi uživatelským prostorem a prostorem jádra operačního systému.[18]

Virtuální paměťový prostor Lineární paměťový prostor, který je mapován do reálné operační paměti mechanismy segmentace a stránkování[19]

Uživatelský prostor Virtuální paměťový prostor, který je oddělený od jádra operačního systému, používají jej procesy neběžící s neprivilegovanými právy[19]

Prostor jádra Virtuální paměťový prostor, který je oddělený od uživatelského paměťového procesu, používají jej procesy s privilegovanými právy - služby operačního systému, ovladače hardwaru, atd.[19]

Uváznutí Je stav procesu, do kterého se dostane za podmínky, kdy úspěšné dokončení první akce je podmíněno předchozím dokončením druhé akce, přičemž druhá akce může být dokončena až po dokončení první akce.

Kontext procesu Je sada struktur, které obsahují stav daného procesu, je možné pomocí těchto struktur pozastavit a obnovit běh procesu.

API Rozhraní pro programování aplikací - sada procedur, struktur, protokolů, které programátor může využívat při implementaci aplikace

Callback funkce Funkce, která zajišťuje vykonání části kódu (funkce), kterou jí lze předat parametrem například jako ukazatel.

Hook - „háček" Jedná se přidání volání konkrétní funkce ze zdrojových kódů linuxového jádra, které jsou vyvíjeny open-source komunitou[11]

Log Záznam o běhu aplikace a operačního systému

Linux Console Terminal Dále jen `terminál` je jedna ze systémových konzolí poskytovaná jádrem operačního systému Linux. Jedná se o médium pro vstupní a výstupní operace v linuxovém systému.[20]

ACK - Acknowledgement Potvrzení o korektním přijetí paketu/zprávy při síťové nebo soketové komunikaci.

Hash Datová struktura v jazyce Ruby, která je ekvivalentem asociativního pole se záznamy typu `(klíč -> hodnota)`, nemá sekvenční uspořádání a její klíč může být libovolného typu.[13]

Spojový seznam Dynamická datová struktura, jejíž základní stavební jednotkou je struktura jednotného datového typu obsahující kromě uchovávaných dat odkaz (odkazy), pomocí kterého je provázána s ostatními stavebními jednotkami.

Makro Pravidlo, jehož obsah (zdrojový kód) je vložen v místě, kde je makro použito (expanze makra).

Base64 Kódování, které převádí binární data na posloupnosti tisknutelných znaků. [21]

PEM Privacy-Enhanced Mail je formát, který obsahuje zakódovaný certifikát pomocí `Base64`, uzavřený mezi řádky uvozující a ukončující samotný řetězec `Base64`. [22]

Makefile Konfigurační soubor nástroje `make` [23], který poskytuje efektivní kompilaci velkých projektů.

Bash **B**ourne **A**gain **SH**ell - jeden z unixových shellů, který interpretuje příkazový řádek a umožňuje skriptování.[24]

I-node Fundamentální datová struktura uchovávající metadata (data o datech) souborů a adresářů souborového systému. Například uchovává typ souboru, práva, časy změn, velikost a mnoho dalších metadat kromě názvu souboru.

1.3 Stručný popis projektu Clondike

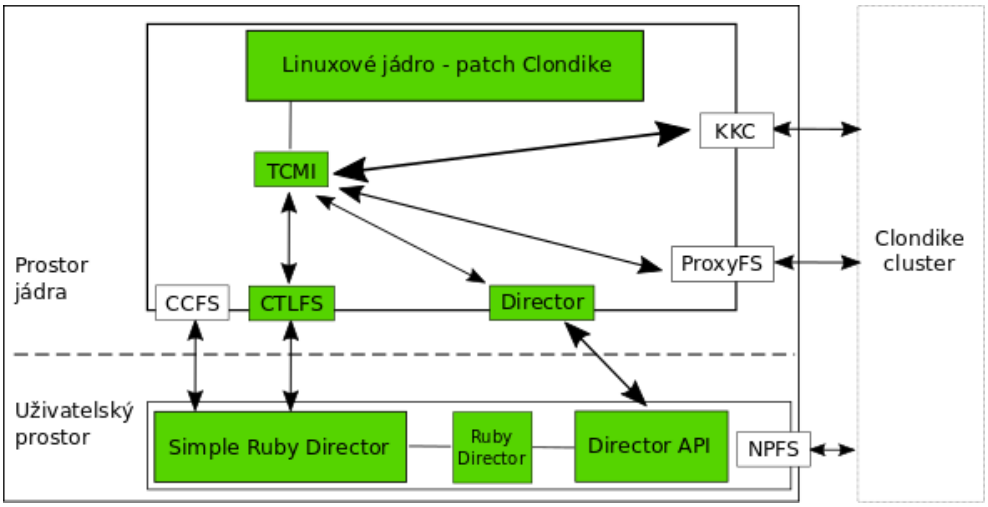
V této části práce je popsán stručně projekt `Clondike`, protože detailnímu popisu projektu byla věnována bakalářská práce [25]. Jsou zde uvedeny pouze komponenty, které se bezprostředně týkají problematiky spojené s touto prací.

Celý projekt můžeme rozdělit na dvě části dle virtuálního paměťového prostoru:

- **Prostor jádra**
  - **KKC** - Knihovna zajišťující síťovou komunikaci mezi jádry uzlů clusteru
  - **ProxyFS** - Virtuální souborový systém umožňující používat speciální typy souborů (roury, terminály, sokety ...)
  - **CCFS** - Speciální typ souborového systému sloužící jako vyrovnávací paměť
  - **TCMI** - Samotná implementace zajišťující migraci procesů
  - **Director** - Klient komunikující s uživatelským prostorem prostřednictvím protokolu `Netlink`
- **Uživatelský prostor**
  - **Director API** - Část aplikačního rozhraní napsaná v jazyce `C` zajišťující 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 uzlů, rozložení zátěže, certifikaci uzlů, komunikaci s jádrem (`Netlink`) a řízení procesu migrace.
- **NPFS** - Server sdílející souborový systém 9Plan protokol

Následující obrázek 1.1 ukazuje vzájemné zapojení zmíněných částí projektu `Clondike`. V této práci jsou implementačně upravovány všechny zeleně vyznačené části uživatelského prostoru a pouze dvě části v prostoru jádra, čili `TCMI` a `Director`.



Obrázek 1.1: Schéma komponent projektu `Clondike`

1.4 Analýza chyb

Většinu analýz na tomto projektu je možno provádět pouze pozorováním vývojářských výpisů z logů spolu s metodou reverzního inženýrství, které lze využít na studium zdrojových kódů jednotlivých komponent. Mezi výhodu logů patří důmyslně implementovaný kategorizovaný a odstupňovaný výpis. Každý záznam v logu obsahuje typ komponenty, které se týká, stupeň závažnosti od oznamovací přes upozorňující až po chybovou, či fatální a v poslední řadě stupeň závažnosti 1-4. Nevýhodou logů je vzhledem k aplikaci v prostoru jádra s paralelními procesy nemožnost synchronizace výpisů, čili jejich správná posloupnost. Tento typ analýzy je velmi časově náročný a u aplikací, které jsou s více paralelními procesy a ještě v prostoru jádra to platí dvojnásob.

1.4.1 Chyba vznikající voláním funkcí `tcmi_transaction_timer` a `tcmi_transaction_put`

Analýza této chyby započala nastudováním mechanismu komponenty `TCMI`, která obsahuje více částí, které navzájem spolu ne zrovna jednoduše kooperují na výsledném migračním mechanismu. Základem `TCMI` je sedm částí jak bylo detailně popsáno v mé bakalářské práci[25]. Pro připomenutí je uveden pouze stručný výčet jednotlivých částí:

- **CKPT** - Checkpoint - Uložení kontextu procesu do souboru
- **COMM** - Komunikační transakce a zprávy
- **CTLFS** - Konfigurační souborový systém
- **LIB** - Pomocné knihovny
- **MANAGER** - Manažeři migračního procesu
- **MIGRATION** - Implementace funkcí, které jsou typu `hooks` (háčky)
- **SYSCALL** - Implementace systémových volání
- **TASK** - Vytváření a ukončování migrovaných procesů

Z těchto částí se daná chyba vyskytuje u implementace v oblasti komunikačních transakcí, respektive v `COMM`. Postupně bylo zjištěno, že při každé komunikaci mezi jádry dvou uzlů je vytvořena komunikační transakce, která má určitou platnost a stav. Platnost transakce určuje již implementovaná komponenta v jádře `timer` [19], která v jednoduchosti umožňuje nastavit čas a ukazatel na funkci, která se má zavolat v případě, že čas uběhne nebo dojde k přerušení pomocí signálu. Tento čas je v současné chvíli nastaven na 10 sekund u transakcí nepreemptivního typu migrace.

Každá `TCMI` transakce se skládá ze dvou typu zpráv žádost a odpověď. Během svého života má čtyři možné stavy:

- `TCMI_TRANSACTION_INIT` - vytvoření transakce, než se naplní interní struktury a dojde k odeslání zprávy
- `TCMI_TRANSACTION_RUNNING` - odeslání žádosti a čekání na odpověď
- `TCMI_TRANSACTION_COMPLETE` - příjem odpovědi s úspěchem
- `TCMI_TRANSACTION_ABORTED` - příjem odpovědi s chybou nebo žádná odpověď nepřišla

Při zkoumání mechanismu transakcí bylo zjištěno, že došlo k vypršení času platnosti transakce, čili její stav se změnil z `TCMI_TRANSACTION_RUNNING` na `TCMI_TRANSACTION_ABORTED`. Tento fakt způsobil v migračním mechanismu řadu následků, které nebyly pro úspěšné dokončení migrace ideální. Bylo vše na straně odesílacího uzlu korektně ukončeno, veškeré alokované paměťové struktury a prostředky byly uvolněny příkladem je volání funkce `tcmi_transaction_put`, jenomže přijímající uzel byl z neznámých důvodů ve stavu, kdy pokračoval v mechanismu migrace, jako kdyby k žádnému vypršení doby platnosti této transakce nedošlo.

Přijímající uzel tedy samozřejmě zjistil, že dané sdílené prostředky například soubor `checkpointu` a filesystém `ProxyFS` nemohl připojit z výše popsaných důvodů, a proto veškerý migrační mechanismus ukončil podobným způsobem jako odesílající uzel, čili skončil s chybovým záznamem v logu a k migraci procesu tedy ve výsledku vůbec nedošlo. Přestože to na první pohled uživatele vypadalo, že proces byl migrován na jiný uzel jen s delší časovou prodlevou.

V této chvíli bylo jasné, proč je tato chyba vypisována do logu, ale nebylo jasné, co způsobuje nedoručení zprávy typu odpověď na odesílající uzel, aby komunikační transakce mohla být korektně dokončena. Spolu s touto chybou se objevovalo v logu mnoho dalších chyb, ať už to byly chyby vyvolané následky této, nebo i jiné. A tento fakt byl předmětem dalšího zkoumání, které vedlo k prapůvodní chybě této chyby a následné lavinové reakci.

Zkoumání vyústilo v myšlenku, že všechno v tomto projektu se vším více či méně souvisí, což platí i o chybách a tato zásadní myšlenka vedla k úvaze, že by tato chyba mohla být zaviněna prozatím neznámou chybou, která způsobuje při mechanismu spojování uzlů komplikace v podobě poměrně dlouhých odezev a reakcí samotného operačního systému na běžné požadavky uživatele. Například stisknutím klávesy *Enter* v terminálu dochází cca k 5 sekundovému blokování jakékoli práce s terminálem.

Hledání se posunulo na základě předchozích faktů na úroveň uživatelského prostoru. Z počátku bylo třeba zjistit, kdy přesně dojde k 5 sekundovým blokacím systému, což nebylo jednoduše identifikovatelné. Vzhledem k předchozím zkušenostem ladění na projektu `Clondike` byl vybrán směr postupného zjednodušení uživatelské komponenty `Simple Ruby Director`, který obsahuje řadu pro tento problém nadbytečných mechanismů. Tyto mechanismy mezi než patří například certifikace uzlů, kontrola procesů před migrací, měřící subsystém, apod. byly postupně vyřazeny z činnosti zakomentováním příslušných inicializací a volání jejich metod. Tímto vznikla pouze jednoduchá aplikace s funkcionalitou vyhledávání `DHT` a spojování uzlů `CTLFS`.

Komponenta `DHT` byla považována za otestovanou a plně funkční vzhledem k magisterské práci kolegy Ing. Pavla Tvrdíka [5]. Další krok vedl k prozkoumání komponenty `FileSystemConnector` v Ruby skriptech, která má na starosti zápis a čtení ze speciálního konfiguračního souborového systému `CTLFS`. Tato komponenta využívá k zápisu do tohoto souborové systému volání ex-

terního terminálového příkazu `echo`[1], který jak se ukázalo později způsoboval výše zmíněný problém s blokací operačního systému.

Pomalu se ukazovalo, že celý problém je právě v zápisu do souborového systému `CTLFS`, který je poskytován stejnojmennou komponentou na úrovni jádra operačního systému a tudíž to na první pohled vedlo k řešení problému v jádře. Ukázalo se po detailnějším zkoumání, že pokud `Simple Ruby Director` není zapnutý, lze simulovaně zapsat do souboru `/clondike/pen/connect` v `CTLFS` obyčejným příkazem `echo` z terminálu a k žádné blokaci operačního systému nedochází. Uzly se tímto simulovaným zápisem korektně spojí a zůstanou spojeny, dokud nedojde k opětovnému zápisu do příslušného souboru `/clondike/pen/nodes/*/stop`, kde hvězdička vyjadřuje index připojeného uzlu.

Simulovaným zápisem do souborů v `CTLFS` bylo potvrzeno, že problém není ve stejnojmenné komponentě prostoru jádra, ale někde mezi Ruby a vyvoláním externího příkazu `echo`. Další směr vedl k vyzkoušení volání jakéhokoli známého terminálového příkazu z `Simple Ruby Directoru`. Ukázalo se, že to způsobují všechny externí příkazy, respektive obecně zavolání externího příkazu z Ruby, ale pouze při běhu Directoru v uživatelském prostoru. Log `Simple Ruby Directoru` (`/tmp/director.log`) zaznamenával chyby `Error code read messsage netlink -22` v okamžiku zavolání externího příkazu.

Další analýza vedla dle předchozího chybového výpisu do komponenty `Director API`, která implementuje obsluhu Netlinkové komunikace. Tato komunikace probíhá v podobě posílání zpráv mezi Directorem jádra a komponentou `Director API`, což je Netlink Director v uživatelském prostoru. Dle zdrojových kódů bylo zjištěno, že k výpisu této chyby dochází při příjmu zprávy, která je typu `ERROR`. Dle dokumentace Netlink[2] je její chybový kód různý od 0 (0 = zpráva typu `ACK`), čili chyba 22 má sémantický text **invalid argument**. Bylo tedy na základě tohoto zjištění otestovat komunikaci přes Netlink a analyzovat, proč přicházejí chybové zprávy z jádra operačního systému.

Testování Netlink komunikace bylo provedeno testovacím Ruby skriptem `Ruby Director API/test.rb`, který implementuje triviálního Netlink klienta vypisující každou příchozí a odchozí zprávu na standardní výstup. Dle výpisu běžné komunikace, která probíhá na základě jakéhokoli vytvoření procesu, což způsobí odeslání zprávy typu `NPM_CHECK nebo NPM_CHECK_FULL` z jádra do uživatelského prostoru, nebyla shledána žádná chybová zpráva s kódem chyby 22. Na základě hlubšího studia implementace `Director API`, je evidentní, že mechanismus příjmu a odeslání zprávy je globální pro všechny typy zpráv a tudíž kdyby byl problém v samotné implementaci příjmu či odeslání, tak by se chyby tohoto typu objevovaly u každé komunikační transakce.

Pro umělé vyvolání chybové zprávy bylo do výše zmíněného Ruby skriptu `test.rb` implementováno vlákno, stejně jak je tomu v `Simple Ruby Directoru`, ve kterém se zavolá externí terminálový příkaz. Chybová zpráva byla vyvo-

[1] **echo** - interní příkaz `Bash` vypisující požadovaný řetězec na standardní výstup

lána a na základě programátorské dokumentace Netlink knihovny[18] bylo do komponenty `Director API` při příjmu a před odesláním zprávy přidáno volání funkce `nl_msg_dump`, která vypíše obsah zprávy v přehledném formátu. Po dalším pozorování typů posílaných zpráv a analýze kódu bylo zjištěno, že zprávy typu `NPM_CHECK nebo NPM_CHECK_FULL` tuto chybu nevyvolávají, ale s nimi spojené zprávy typu `DIRECTOR_NPM_RESPONSE`, které potvrzují a zároveň odesílají informaci do jádra, zda-li se bude nebo nebude migrovat aktuální proces, tuto chybu způsobí. Tento fakt vede k úvaze zda-li zpráva směřující do jádra odejde korektním způsobem, zda jsou všechny argumenty zprávy korektní a v pořádku.

Argumenty, respektive parametry, které zpráva obsahuje jsou v pořádku, protože v 99% případů, kdy jsou transakce tohoto typu bezproblémově dokončeny dokazují správnost implementace a naplnění zpráv parametry. Jediná výjimka je u 1% zpráv, které ovšem vznikají v důsledku vytvoření procesu, který je vytvořen operačním systémem na základě volání externího příkazu z `Ruby` v současném běhu testovacího Directoru i netestovacího `Simple Ruby Directoru` a `Netlink` komunikace. Tento zjištěný fakt vyvolává otázku, proč je vlastně proces, který vznikne z interního procesu projektu `Clondike` testován na migraci? Bohužel je to dáno nedostatečným stávající řešením.

Koncepční řešení tohoto problému na základě této analýzy je vytvořit mechanismus, který zajistí, aby každý externí proces, vyvolaný z `Simple Ruby Directoru`, nebyl testován na migraci. Bohužel v současné implementaci neexistuje toto opatření. Výsledkem této analýzy je požadavek na vytvoření tohoto mechanismu, který je navržen a popsán v následující kapitole 2.1.

1.4.2 Chyba způsobující rozpojení uzlů

Analýza problému rozpojování uzlů vychází z předešlé analýzy, protože se předchozího problému okrajově týká. Uzly jsou v projektu `Clondike` hledány a spojovány prostřednictvím komponenty `Simple Ruby Director`, a proto byla analýza zaměřena právě na tyto `Ruby` skripty. Pro přehled je zde uveden stručný výpis a popis jednotlivých Ruby skriptů, které byly analyzovány.

- `dht/Bootstrap.rb` - Hledání uzlů algoritmus Kademlia [5]
- `dht/DHTMessageDispatcher.rb` - Odesílání zpráv [5]
- `dht/DHTMessages.rb` - Jednotlivých typů zpráv pro DHT [5]
- `trust/AuthenticationDispatcher.rb` - Mechanismus autentizace uzlů
- `trust/CertificatesDistributionStrategy.rb` - Mechanismus rozesílání veřejných klíčů uzlů
- `trust/Identity.rb` - Identita uzlu, uložení veřejného a privátního klíče

- `trust/Session.rb` - Autentizační session
- `trust/trustManagement.rb` - Hlavní implementace trust managementu
- `trust/trustMessages.rb` - Jednotlivé typy zpráv pro mechanismus autentizace uzlů

Po detailním studiu skriptů `MembershipManager.rb` a `ManagerMonitor.rb` bylo zjištěno několik zásadních faktů. První z faktů je, že samotné hledání uzlů zajišťuje komponenta `DHT`, která je inicializována spolu s `MembershipManagerem`. Dále pak komponenta `DHT`, respektive `dht/Bootstrap.rb` plní objekt repositář `nodeRepository`, nově nalezenými uzly. Z tohoto repositáře jsou vláknem `startAutoConnectingThread`, které je součástí implementace `MembershipManager.rb` vybírány postupně nalezené uzly. V cyklu jsou prováděny kontroly certifikátů prostřednictvím objektu `trustManagement`, který zajišťuje verifikaci uzlů. Pokud je verifikace uzlu s kladným výsledkem, je provedena kontrola, zda-li již není uzel připojen. Pokud ne je provedeno připojení uzlu prostřednictvím objektu `filesystemConnector`, který interně volá externí příkaz `echo`, který je problémem viz. předchozí analýza.

Je-li uzel již připojen, pak je vybrán další uzel z repositáře nalezených uzlů a celý tento mechanismus je opakován. Pokud uzel ovšem objekt `trustManagement` vyhodnotí jako neverifikovaný, pak je provedena metoda `connectToNode`, která interně vyvolá mechanismus na získání veřejného klíče uzlu, ke kterému se bude aktuální uzel připojovat a provede se verifikace znovu, pokud skončí kladně je provedeno připojení uzlu stejným způsobem tedy prostřednictvím objektu `filesystemConnector`. Jestliže verifikace skončí chybou pak je do nekonečna startován mechanismus na získání veřejného klíče uzlu, ke kterému se aktuální uzel chce připojit.

Pro názornost je zde uveden zjednodušený zdrojový kód vlákna, které připojuje nalezené uzly do clusteru.

```
1  def startAutoConnectingThread
2   ExceptionAwareThread.new {
3    loop do
4     @nodeRepository.getAllNodes.each { |node|
5      next if node == @nodeRepository.selfNode
6      if @trustManagement.isVerified?(node.nodeId)
7       if containsDetachedNode(node) == false
8        if node.networkAddress.class == NetworkAddress
9         @filesystemConnector.connect(node.networkAddress)
10      end
11     else
12      connectToNode(node)
13     end
14    }
15   end
16  }
17 end
```

S ohledem k analýze předchozí chyby byl zvolen směr ponechání pouze kostry implementace, která zajišťuje spojení uzlů bez jakékoli kontroly certifikátů a podobných pokročilých mechanismů. Jednoduše řečeno byly zakomentovány v uvedeném zdrojovém kódu řádky 6 a 12.

V tento okamžik byl celý mechanismus hledání a spojení uzlů až na chybu, která byla popsána výše zcela funkční. Pro zajímavost předchozí chyba s voláním externího procesu `echo` byla obejita zavoláním interních funkcí `Ruby` pro operaci se soubory `File.open` a `File.write`. Z výše uvedených faktů vyplývá jednoduchý závěr problém rozpojování uzlů je v mechanismu certifikace a verifikace uzlů pomocí objektu `trustManagement`.

Následovala detailní analýza celého objektu `trustManagement` a jeho pomocných objektů, jejichž implementace je umístěna ve složce `trust`. Prostřednictvím kontrolních ladících výpisů bylo zjištěno, že stěžejní problém se nachází v objektu `authenticationDispatcher`, který si eviduje dva `hashe` `clientNegotiations` a `serverNegotiations` obsahující záznamy jejichž klíče jsou objekty typu `OpenSSL:PKey:RSA`[26], což je objekt reprezentující veřejný klíč nebo privátní klíč.

Vzhledem k nucené aktualizaci linuxové distribuce na verzi `Debian 7` z důvodu budoucího použití `Apache Cassandra ??` a zároveň k aktualizaci na `Ruby 1.9.3`, které musí být nainstalováno vzhledem k využití `CQL3Driver`, byla zjištěna testovacím `Ruby` skriptem nefunkčnost hledání v datové struktuře `hash` podle klíče, který je typu `OpenSSL:PKey:RSA`. Toto zásadní zjištění vede k nefunkčnosti celého mechanismu verifikace uzlů, a proto je třeba navrhnout opravu tohoto problému například jiným typem klíče, pomocí kterého lze vyhledávat v datové struktuře `hash`.

Během analýz ladících výpisů ze `Simple Ruby Directoru`, byl odhalen záznam, který upozorňoval na chybu při serializaci objektu pomocí standardní integrované metody `Ruby marshal.dump`. Tato metoda je pro valnou většinu objektů definovaná uvnitř nich samých s názvem `marshal_dump`. Bohužel objekt typu `OpenSSL:PKey:RSA`, kterého se tato chyba týkala shodou okolností tuto metodu definovanou neměl. Následkem toho byl zajímavý dopad, kdy veřejné klíče, které se posílaly jako objekt prostřednictvím zpráv přes síť nebyly správně serializovány odesílatelem, což vedlo u příjemce k další chybě, kdy nemohl být serializovaný objekt ze zprávy deserializován metodou `marshal.load`. Vzhledem k této chybě nefunguje ve verzi Ruby 1.9.3, kde byla vývojáři upravena knihovna `OpenSSL` plně mechanismus autentizace uzlu, který je implementován v souboru `authenticationDispatcher.rb`.

Řešení naposledy zmíněného problému je najít způsob jak serializovat a deserializovat objekt z knihovny `OpenSSL` a tento způsob implementovat.

1.4.3 Chyba vyvolávající `task tcmi_commd_xx:xxxx blocked for more than 120 seconds`

Analýza této chyby byla velmi krátká vzhledem k tomu, že žádný takový chybový výpis v žádném ze zdrojových kódů projektu `Clondike` nebyl nalezen. Byl proveden průzkum zdrojových kódů samotného jádra a výsledek byl nalezen v souboru `/kernel/hung_task.c`, který implementuje upozorňování na dlouho nepracující (visející) procesy. Je zcela evidentní, že chybová hláška souvisí dle první analýzy v této práci s dlouhým čekáním komponenty `TCMI_COMM`, která čeká kvůli chybě v `Netlink` komunikaci a již zmíněným problémem s blokací operačního systému.

Kapitola 2

Realizace

První část této kapitoly je věnována samotnému návrhu a implementaci řešení vycházející z předchozích analýz chyb a problémů, které jsou s nimi spojeny. V druhé části je uveden postup dokončení aktualizace knihovny Netlink na verzi 3 spolu s implementací chybějící evidence počtu připojených uzlů. Ve třetí části je uvedeno ověřovací měření, které dokládá správnost implementovaného řešení. V poslední části této kapitoly je uveden návrh a implementované řešení ukládání dat z projektu `Clondike` do databáze `Cassandra`[4].

2.1 Návrh a implementace řešení chyby blokace operačního systému

Základem k tvorbě návrhu je předešlá analýza v kapitole 1.4.1, jejíž výsledkem je určitý detekční mechanismus na detekci procesů, které jsou tvořené z procesu `Simple Ruby Directoru`. Na základě takto detekovaných procesů je nutné zamezení kontroly, která je u každého nově vzniklého procesu provedena před migračním mechanismem. Vzhledem k tomu, že celá tato kontrola i tvorba nového procesu je implementována v prostoru jádra, bude i tato implementace muset být vytvořena a integrována do tohoto prostoru.

Na počátku je třeba uvést alespoň zjednodušeně, jak funguje mechanismus vytvoření procesu a kde je umístěno volání inicializace kontroly pro migrační mechanismus. V operačním systému Linux je proces tvořen klonováním procesů konkrétně funkcí `fork`[23]. Tento mechanismus je vyvolán prostřednictvím systémového volání `do_fork`. To znamená, že v okamžiku, kdy je zavolána funkce `fork`, je aktuální (rodičovský) proces pozastaven. Pro nový proces (potomek) jsou alokovány struktury popisující jeho kontext například `task_struct` a do těchto struktur je částečně překopírován obsah kontextu rodičovského procesu.

Nový proces dostane unikátní identifikátor procesu `PID`. Dalším důležitým identifikátorem je `PPID`, což je číslo jeho rodičovského procesu. Jsou nasta-

veny vazby mezi rodičovským procesem a jeho potomkem, pomocí nichž lze dohledat informace o rodokmenu každého z procesů. Interně je toto zajištěno ukazateli na jednu ze strukturu kontextu procesu `task_struct`. Tyto ukazatele jsou uloženy v datové struktuře jádra `list` [19], která implementuje kruhový spojový seznam. Konkrétně jsou ukazatele na spojové seznamy ve struktuře `task_struct` nazvány `children` a `sibling`.

Nový proces je vytvořený a nyní byl spuštěn. V okamžiku spuštění procesu je zavoláno systémové volání, které zavolá funkci `kernel_execve`, což je patchem projektu `Clondike` upraveno na spuštění jiné, ale implementačně podobné funkce `clondike_execve`, která je funkcí typu „`háček`" a tudíž je inicializována komponenta `TCMI_HOOKS`. Jinými slovy je zavolána funkce `tcmi_mighooks_execve`, ve které je při novém procesu vyvolaném běžným způsobem (není to proces vytvořený v průběhu migračním mechanismu) provedena kontrola procesu na migraci prostřednictvím funkce `tcmi_try_npm_on_exec`. Dále je pak zavolána příslušná funkce komponenty `Director` a pak je pomocí `Netlink` zprávy poslán požadavek na kontrolu do uživatelského prostoru.

Při ladícím výpisu stromové struktury procesů, respektive vypsání celého rodokmenu procesů bylo zjištěno, že proces `Simple Ruby Director` nefiguruje jako rodič nově vytvořeného procesu `echo`, který byl vyvolán jako externí příkaz z jazyka `Ruby`. Tento fakt je zvláštní, ale bohužel rodičem tohoto procesu je proces `bash` a nikde dál v rodokmenové linii, ani jako sourozenec není uveden proces `ruby` s příslušným `PID` stejným jako má `Simple Ruby Director`. Na základě tohoto zjištění nemohly být tyto informace vedené jádrem operačního systému, respektive plánovačem procesů použity pro řešení toho problému.

Proto byl zvolen podobný přístup jako zvolili vývojáři jádra, který spočívá v přidání ukazatele `tcmi_parent` do struktury `task_struct`, který bude pro každý proces uchovávat opravdového rodiče, neboli ukazatel na `task_struct` strukturu rodičovského procesu, ze kterého byl doopravdy funkcí `fork` vytvořen. Jinými slovy pomocí těchto ukazatelů `tcmi_parent` bude vytvořen spojový seznam všech procesů. Přidání atributu `tcmi_parent` do struktury `task_struct` je provedeno ve složce zdrojových souborů jádra, konkrétně v souboru `include/linux/sched.h`. Každý nový proces, když vzniká bude mít tento ukazatel nastaven na výchozí hodnotu `NULL`, čili prapůvodní proces `init`, který má `PID = 0` bude mít tento ukazatel s hodnotou `NULL`, což je nastaveno z důvodu korektního ukončení mechanismu procházení spojového seznamu rekurzivním algoritmem. Tato výchozí hodnota je nastavena ve stejném složce jádra v souboru `init_task.h`.

V tuto chvíli je třeba nastavit příslušnou hodnotu výše přidaného ukazatele při tvorbě nového procesu. Toto přiřazení (`child->tcmi_parent = current`) je implementováno ve funkci systémového volání `tcmi_syscall_hooks_in_fork`, která je zavolána mechanismem vytváření procesu v jeho prostřední fázi, čili při klonování nového procesu. Proměnná `current` je ukazatel na `task_struct` aktuálního procesu.

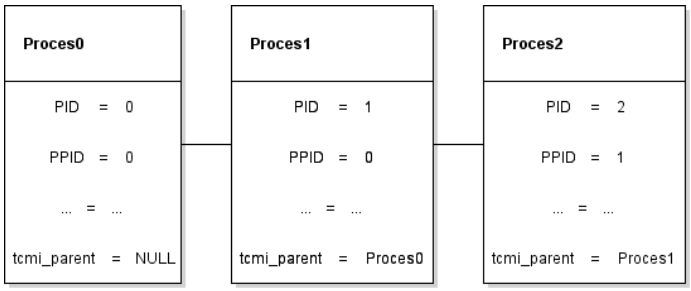
V tento okamžik nastává řešení při ukončování procesů, kdy je třeba tyto

vazby správně modifikovat, aby nedošlo k problému, že ukazatel `tcmi_parent` bude ukazovat na strukturu procesu, který již neexistuje z důvodu toho, že byl právě ukončen. Implementace je provedena ve formě modifikace funkce `tcmi_mighooks_do_exit`, která je volána v době ukončování procesu prostřednictvím systémového volání `do_exit`. Toto řešení nelze provést jednoduchým přepsáním hodnot ukazatelů, z toho důvodu, že se jedná o modifikaci sdílené proměnné. Je tedy zapotřebí použít mechanismu zámků na celý spojový seznam procesů `tasklist_lock` [27], konkrétně je třeba zavolat funkci `write_lock_irq`, která zajistí vypnutí přerušení a zamknutí daného zámku pro operaci zápis.

Pro názornost je zde uvedena implementace modifikace ukazatelů v případě ukončení procesu, pouze přidaná část zmíněné funkce.

```
1 write_lock_irq(&tasklist_lock);
2  for_each_process(child) {
3   if (child->tcmi_parent == current)
4    child->tcmi_parent = current->tcmi_parent;
5  }
6  current->tcmi_parent = NULL;
7 write_unlock_irq(&tasklist_lock);
```

Z ukázky zdrojového kódu je jasně patrné, že nejdříve jsou uzamčeny pro zápis struktury `task_struct` všech procesů a pak je funkcí `for_each_processes` procházen celý spojový seznam se všemi procesy a pokud je nalezen proces, jehož `tcmi_parent` ukazuje na aktuální proces (byl nalezen proces, který je potomkem aktuálního procesu), tak je modifikován jeho ukazatel `tcmi_parent` na stejnou hodnotu jako má aktuální ukončovaný proces. Následně je v cyklu hledán další proces typu potomek aktuálního ukončovaného procesu. Po dokončení cyklu je nastaven ukončovanému procesu ukazatel `tcmi_parent` na hodnotu `NULL` a v posledním kroku jsou odemčeny struktury `task_struct`. Tímto mechanismem je dosaženo správného zapojení vazeb mezi procesy v případě, že jeden z procesů se ukončí. Na obrázku 2.1 je znázorněna vytvořená virtuální datová struktura pomocí ukazatelů `tcmi_parent`.



Obrázek 2.1: TCMI rodokmen - spojový seznam

Předchozí implementace pomůže v identifikaci procesu, který vznikl z jiného, což v tomto případě znamená z procesu `Simple Ruby Director`. Jelikož

v době, kdy vznikne tento proces nelze žádným obecným způsobem zjistit, že je to právě ten proces, který bude považován jako praotce všech potomků, které je potřeba detekovat. Nastává nutnost udělat indikátor `nonmigratable` každého procesu, který bude říkat ano tento proces nebude migrován, nebo může být migrován. Tento indikátor bude zároveň sloužit pro odstranění problému, že doposud všechny procesy jsou kontrolovány na migraci a tato kontrola je prováděna i v době, kdy `Simple Ruby Director` není spuštěný, což vede k chybové návratové hodnotě kontroly. Tato chyba nemá význam na funkčnost projektu `Clondike`, pouze způsobuje chybový výpis v logu `clondike-netlink<err3> Unicast error -111`, jejíž význam je, že spojení s uživatelským `Directorem` nebylo navázáno prostřednictvím protokolu `Netlink`.

Na základě předchozích zjištění je implementován mechanismus samotné detekce s použitím již zmíněného indikátoru `nonmigratable`, jehož výchozí hodnota je 1 (true). Tento indikátor a jeho výchozí hodnota je definována stejným způsobem jako ukazatel `tcmi_parent` s tím rozdílem, že se jedná o celočíselnou proměnnou. Dalším krokem implementace řešení bylo přidání následujících funkcí do samotného `Directoru`.

```
1  int director_check_forked_process(struct task_struct *p){
2   int res = 0;
3   pid_t director_pid = get_director_pid();
4   if (director_pid == 0) return 2; //Director is not connected
5   return director_check_parent(p, director_pid);
6  }
7  EXPORT_SYMBOL(director_check_forked_process);
8
9  int director_check_parent(struct task_struct *p, pid_t find_pid)
       {
10  struct task_struct *tmp = NULL;
11  if (p == NULL) return 0;
12  if (p->pid == 1) return 0;
13  if (p->pid == find_pid) return 1;
14  tmp = p->tcmi_parent;
15 return director_check_parent(tmp, find_pid);
16 }
17
18 void director_disconnect(void){
19  disconnect_director();
20 }
21
22 pid_t director_pid(void) {
23  return get_director_pid();
24 }
```

Hlavní funkcí tohoto mechanismu je `director_check_forked_process` jejíž dostupnost je odkudkoliv ze zdrojového kódu jádra prostřednictvím makra `EXPORT_SYMBOL`. Tato funkce si zjistí `PID` uživatelského `Simple Ruby Directoru`, pokud je tato hodnota rovna 0, pak není `Director` připojen a tudíž celá funkce vrátí návratovou hodnotu 2, která bude mít za následek, že proces nebude kontrolován na migraci.

V opačném případě, kdy je `Director` připojen, nastane rekurzivní volání funkce `director_check_parent`, která dle vstupního parametru, což je vždy ukazatel na strukturu `task_struct` vrátí hodnotu 1 v případě, že aktuální proces má v rodové linii otce, tedy proces `Simple Ruby Directoru`. V ostatních případech vrací hodnotu 0. Rodová linie je dána výše zmíněným spojovým seznamem prostřednictvím ukazatelů `tcmi_parent`. Hloubka rekurzivního volání této funkce je omezena buď hledaným procesem, procesem s `PID = 0` nebo ukazatelem s hodnotou `NULL`.

Dalším krokem k funkčnímu mechanismu je doplnění implementace pomocných funkcí v souboru `comm.c` `get_director_pid(void)` a `disconnect_director`. První slouží pro zjištění `PID` procesu `Simple Ruby Directoru`. Druhá z funkcí slouží v případě ukončení tohoto procesu, kdy nastaví jeho `PID` na hodnotu 0, tím, je zajištěno, že předchozí zmíněná funkce `director_check_forked_process` skončí s návratovou hodnotou 2.

Mechanismus hledání rodičovského procesu `Simple Ruby Directoru` pomocí poslední zmiňované funkce je volán tehdy, kdy je proces vytvořen, ale ještě nedošlo k jeho spuštění. Toto místo je právě funkce systémového volání `tcmi_syscall_hooks_in_fork`, kde je nastavován zmíněný indikátor `nonmigratable` a to následujícím způsobem.

```
1   // Set original parent PID for Clondike NPM check
2  child->tcmi_parent = current;
3  if(director_check_forked_process(child))
4   child->nonmigratable = 1;
5  else child->nonmigratable = 0;
```

I zde je použit zámek `tcmi_parent_lock` z toho důvodu, že volaná funkce `director_check_forked_process` přistupuje k atributu procesu `tcmi_parent`, kde je zapotřebí dodržet konzistenci dat, respektive nesmí dojít během procesu hledání rodičovského uzlu ke změně hodnoty tohoto atributu vlivem ukončení procesu.

Proměnná `child` je ukazatelem na `task_struct` nově vytvořeného ještě nespuštěného procesu. Po jeho spuštění dojde ihned ke kontrole na migraci prostřednictvím funkce `tcmi_try_npm_on_exec`, která interně volá `director_npm_check`, ve které je kontrola tohoto indikátoru ve formě podmínky `if (current->nonmigratable)`, pokud je tento indikátor nastaven na hodnotu 1 je kontrola přerušena s návratovou hodnotou 0 a výpisem do systémového logu, že byla kontrola na migraci přerušena z důvodu příbuznosti s procesem `Simple Ruby Directoru`.

Zmíněná kontrola indikátoru byla přidána po testování provozu projektu `Clondike` i do následujících funkcí.

- `director_task_fork` - informování uživatelského prostoru o vytvoření procesu

```
1  int director_task_exit(struct task_struct *task, int
       exit_code, struct rusage *rusage) {
```

```
2   if (task->nonmigratable) return 0;
3   return task_exitted(task->pid, exit_code, rusage);
4  }
```

- `director_task_exit` - informování uživatelského prostoru o ukončení procesu

```
1  int director_task_fork(struct task_struct *parent, struct
       task_struct *child) {
2   if (child->nonmigratable) return 0;
3   if ( is_director_pid(parent->pid) )
4   return 0;
5   return task_forked(child->pid, parent->pid);
6  }
```

V případě, že dochází k ukončení procesu, neboli je volána funkce systémového volání `tcmi_mighooks_do_exit`, ve které je provedeno již zmiňovaná modifikace ukazatelů vazeb `tcmi_parent`, je zde prováděna kontrola zda ukončovaný proces není procesem `Simple Ruby Directoru`, pokud ano je interní proměnná ukládající jeho `PID` nastavena na hodnotu 0 následující implementací.

```
1  director = director_pid();
2  if(director != 0 && current->pid == director)
3   director_disconnect();
```

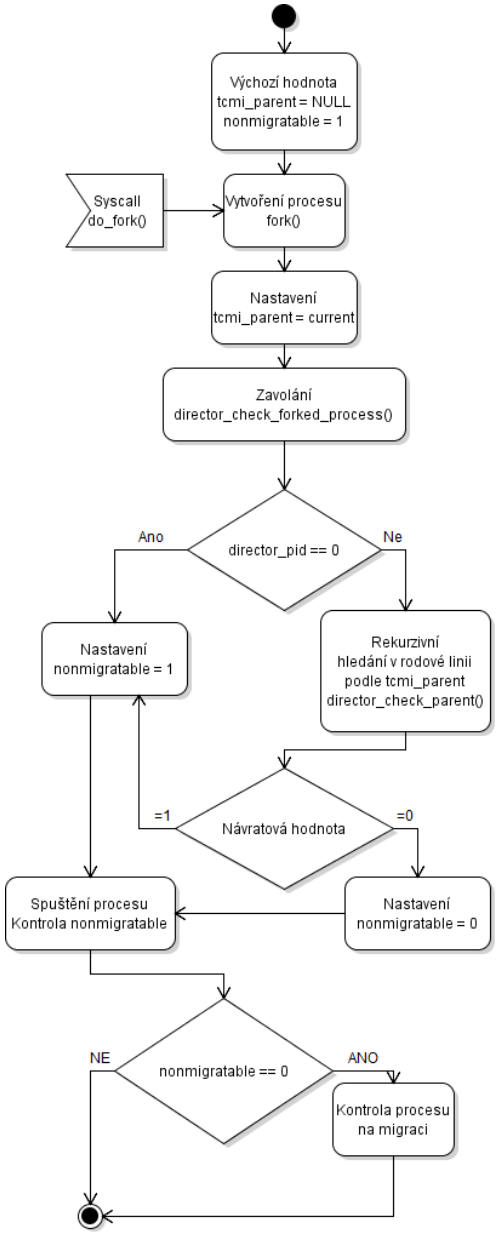
Během vývoje tohoto řešení bylo provedeno několik desítek testů a kompilací celého jádra, které byly nezbytné ať už z důvodu nastavení ladících výpisů, nebo jen k nově přidané testované funkcionalitě. Díky tomuto řešení byla úspěšně odstraněna chyba způsobující blokace operačního systému, především v době, kdy docházelo ke spojování uzlů clusteru. Je fakt, že tato chyba mohla být obejita použitím interní funkce `Ruby`, konkrétně `File.open` a `File.write`. Došlo by k pouhému oddálení problému a chyba by nebyla vyřešena koncepčním způsobem jakým je tento.

Na závěr tohoto vývoje řešení bylo nutné vytvořit nový `patch` jádra, který již obsahuje zmiňované modifikace v souborech `init_task.h` a `sched.h`.

Pro lepší pochopení mechanismu detekce a kontroly je uveden diagram 2.2.

2.2 Návrh a implementace řešení chyby způsobující rozpojení uzlů

Samotný návrh řešení této chyby je dán výsledkem předchozí analýzy v kapitole 1.4.2, ze které vyplývají dva zásadní problémy. Prvním problémem je nefunkčnost prohledávání v datové struktuře `hash`, kde jako index je objekt typu `OpenSSL:PKey:RSA` a druhým problémem je nemožnost úspěšně tento objekt serializovat a následně deserializovat pro přenos po síti.



Obrázek 2.2: Diagram mechanismu detekce a kontroly migrace procesu

Vyhledávání v datové struktuře `hash` lze provádět za pomocí běžných řetězců, což znamená převést objekt `OpenSSL:PKey:RSA` do řetězce. Po detailním nastudování zdrojových kódů bylo zjištěno, že jako index do datové struktury `hash` je použit pouze veřejný klíč. Řešení tedy tkví v převodu klíče za pomocí metody jeho třídy `to_pem`, která veřejný klíč převede do řetězce, se kterým lze nakládat více univerzálněji než se samotným objektem.

Převod je zajišťován nově implementovanou metodou `convertKeyToPEMString` komponenty `trustManagement`. Pro takto převedený klíč byl definován atribut `publicKeyPEM` ve třídách reprezentující různé typy zpráv, které obsahují veřejný klíč uzlu. Důvod zavedení byl prostý, převod je u každé z tříd proveden pouze jednou, nikoli vždy, když je třeba.

Dle dokumentace `Ruby` [26] byl zjištěn fakt, že přetížený konstruktor vytvářející objekt typu `OpenSSL:PKey:RSA` z výše uvedeného řetězce ve formátu `PEM` vytvoří zmiňovaný objekt. Dle tohoto zjištění byla implementována metoda `convertPEMStringToKey` ve stejné komponentě.

Zde je uvedena implementace obou zmiňovaných metod s příslušným ošetřením vstupních parametrů.

```
1 def convertKeyToPEMString(key)
2  return nil if key.nil?
3  return key.to_pem
4 end
5
6 def convertPEMStringToKey(pem)
7  return nil if pem.nil?
8  return OpenSSL::PKey::RSA.new(pem)
9 end
```

Další provedenou změnou v dosavadní implementaci byla úprava metody `sign` komponenty `Identity`, kde je nově použit globální otisk `digest` typu `OpenSSL::Digest::SHA1`. Tento otisk je nově použit i v metodě `verify` této komponenty, která ověřuje podepsaná data, zda-li opravdu byla podepsána příslušným privátním klíčem uzlu, jehož veřejný klíč je pro toto ověření použit.

V implementaci komponenty `AuthenticationDispatcher` byl přidán atribut objektu `trustManagement` právě kvůli použití zmiňovaných transformačních metod pro převod veřejných klíčů. Tento atribut je dále v implementaci této komponenty přidán jako vstupním parametrem k jednotlivým konstruktorům následujících tříd.

- `STSInitialRequestHandler`
- `STSServerChallengeHandler`
- `STSFinalizeHandler`

Jedná se o třídy reprezentující jednotlivé typy zpráv používané v mechanismu autentizace uzlů, respektive jejich příjem a zpracování dat v nich obsa-

žené. Pro představu je zde uveden popis autentizačního procesu se důrazem na místa provedených změn v implementaci.

Autentizační proces začíná vytvoření `NegotiatedSession`, což není nic jiného než objekt obsahující veřejný klíče vzdáleného a aktuálního uzlu spolu s jeho identifikátorem. Celý proces začíná na aktuálním uzlu (inicializační uzel), kde je vygenerováno pseudonáhodné číslo `xValue` a nově je zde převeden veřejný klíč vzdáleného uzlu do formátu `PEM`. Takto převedený klíč je použit jako identifikátor v datové struktuře `hash` pojmenované `clientNegotiations`, kde se uloží tato session. Následně je vytvořena první zpráva `STSInitialRequest`, která obsahuje převedený veřejný klíč aktuálního uzlu, identifikátor lokálního uzlu a `xValue`. Tato zpráva je odeslána vzdálenému uzlu dle jeho identifikátoru `remoteNodeId`.

Na vzdáleném uzlu je inicializační zpráva obsloužena příslušným výše zmíněným handlerem, kde je veřejný klíč z formátu `PEM` vytvořen do podoby objektu zmiňovanou transformační metodou. Jsou extrahována ostatní data ze zprávy včetně `xValue`. Následně je pseudonáhodně vygenerována `yValue` spolu s `proof`. Data k podpisu vzniknou řetězcovým spojením obou hodnot `yValue` a `xValue`. Takto získaná data jsou podepsána privátním klíčem vzdáleného uzlu, čímž vzniká podpis `signature`. Následně je ještě zašifrována hodnota `proof` veřejným klíčem vzdáleného uzlu. Takto vytvořená data jsou uložena do datové struktury `serverNegotiations` typu `hash` s identifikátorem převedeného veřejného klíče uzlu odesílající inicializační zprávu. Poslední fází je vytvoření zprávy typu `STSServerChallenge` obsahující veřejný klíč vzdáleného uzlu, `yValue`, `signature`, zašifrovaný `proof`. Tato zpráva je odeslána zpět k uzlu inicializující autentizační proces jehož identifikátorem je `peerNodeId`.

Po příjmu zprávy na inicializačním uzlu jsou extrahovány všechna data ze právy spolu s vytvořením veřejného klíče do podoby objektu nově implementovanou transformační metodou. Z datové struktury `clientNegotiations` je získána původní hodnota `xValue` k této `NegotiatedSession`, která je identifikovaná zmiňovaným veřejným klíčem ve formátu `PEM`. Pro ověření vzdáleného uzlu jsou spojením přijaté `yValue` a uložené `xValue` vytvořena data k ověření podpisu. Podpis je ověřen nově implementovanou metodou `verifySignature` komponenty `trustManagement`, která interně zavolá metodu `verify` veřejného klíče inicializačního uzlu, jejíž návratová hodnota je buď **true** v případě úspěšného ověření nebo **false** v opačném případě. Pokud je vzdálený uzel tímto ověřen, tak je v `NegotiatedSession` nastaven atribut `confirmed` na **true**. Jsou vytvořena data k podpisu řetězcovým spojením vzájemně prohozených `xValue` a `yValue`, která jsou následně podepsána privátním klíčem inicializačního uzlu a jsou odeslána zprávou typu `STSFinalize` vzdálenému uzlu. V opačném případě, kdy vzdálený uzel nebyl ověřen je nastaven atribut `confirmed` na **false** a proces autentizace je ukončen.

Na vzdáleném uzlu je zpráva typu `STSFinalize` přijata a obsloužena příslušným handlerem, kde je podobným způsobem provedeno ověření posílaného podpisu. V případě, že je inicializační uzel ověřen, pak je nastaven atribut

`confirmed` v `serverNegotiations` na **true** jinak na **false**.

V rámci těchto oprav byla provedena řada malých oprav, které byly realizovány na základě upozorňujících nebo varovných výpisů v logu `Simple Ruby Directoru`. Jednalo se především o nepoužité proměnné. Výsledkem těchto oprav je plně funkční komponenta `trustManagement`, jejíž použití bylo v analýze původně zakomentováno.

Během oprav bylo zjištěno, že k rozpojováním uzlů docházelo vlivem předchozího problému blokace operačního systému, protože `Simple Ruby Director` odesílá zprávy typu `HeartBeat`, kterými zjišťuje stav ostatních uzlů v pravidelných intervalech. Pokud uzel do určité doby od odeslání této zprávy neodpoví, pak je označen jako potenciálně mrtvý. Po další nereakci na `HeartBeat` je inicializováno jeho odpojení. Tento problém byl vyřešen řešením v sekci 2.1.

2.3 Dokončení aktualizace knihovny Netlink na verzi 3

V této části práce je uvedeno dokončení aktualizace komponent `Director API` a `Ruby Director API`. Vzhledem k možnosti koexistence obou knihoven `Netlink v1` a `Netlink v3` [2], které ve své práci uvádí kolega Ing. Michal Salát [6], při kompilaci nebyl vypisován žádný problém.

Situace se změnila v okamžiku, kdy byla z důvodu neposkytování další vývojářské podpory pro první verzi, tato knihovna odebrána z `Makefile` zmíněných komponent.

Bylo třeba najít ekvivalentní náhradu funkcí, které v nové knihovně byly pojmenovány jinak, případně měli odlišné vstupní parametry. Největším rozdílem bylo použití jinak pojmenované struktury soketu `struct nl_sock*` místo původní `struct nl_handle*`. Tato změna byla potřeba promítnou do všech zdrojových souborů, kde byly definovány funkce s těmito parametry.

Další náhrada byla provedena u původní funkce `nl_disable_sequence_check` na novou `nl_socket_disable_seq_check`, která již používá novou výše zmíněnou strukturu. Funkce vypíná kontrolu sekvenčních čísel u zpráv, respektive transakcí.

Funkce `nl_set_buffer_size` nastavující velikost vstupní a výstupní vyrovnávací paměti byla nahrazena jinak definovanou `nl_socket_set_buffer_size`, která též používá novou strukturu soketu.

Dalším krokem byla úprava souboru `extconf.rb` v umístění `Ruby Director API`, který je `Ruby` skriptem a slouží k automatickému vygenerování souboru `Makefile`. Tímto souborem pak lze přeložit a nainstalovat externí knihovnu `directorAPI.so` do repositáře `Ruby`. Tato externí knihovna je pak použita ve skriptech komponenty `Simple Ruby Director`.

Nyní je již aktualizace kompletní a lze do budoucna stahovat novější zdrojové kódy třetí verze od vývojářů do předem vyhrazeného umístění `sources/libnl-3` v repositáři projektu `Clondike` [28].

2.4 Návrh a implementace chybějící evidence počtu připojených uzlů

2.4.1 Návrh

V uživatelském rozhraní operačního systému je počet připojených uzlů doposud získáván ne zcela ideálním způsobem. Tento způsob je za pomocí skriptu jazyka `Bash`, který je spouštěn při vyvolání `prompt`[2]. Tento skript obsahuje volání externích příkazů například `ls`, `cut`, `wc`. Je evidentní, že při jakémkoli stisknutí klávesy *Enter* jsou vždy volány uvedené nástroje.

Tento neduh lze odstranit mnoha způsoby, ale v tomto kontextu, byl zvolen způsob evidence počtu připojených uzlů v souboru speciálního konfiguračního souborového systému `CTLFS`, který je ve výchozím nastavení připojen do adresáře `/clondike`.

Detailněji tato adresářová struktura je popsána v mé bakalářské práci [25]. Stručně pro připomenutí obsahuje dva základní adresáře `ccn` a `pen`. Názvy vycházejí z typů uzlů. Každý tento adresář obsahuje podadresář `nodes`, kde jsou po připojení uzlů příslušné symbolické odkazy spolu s adresáři konkrétně připojených uzlů.

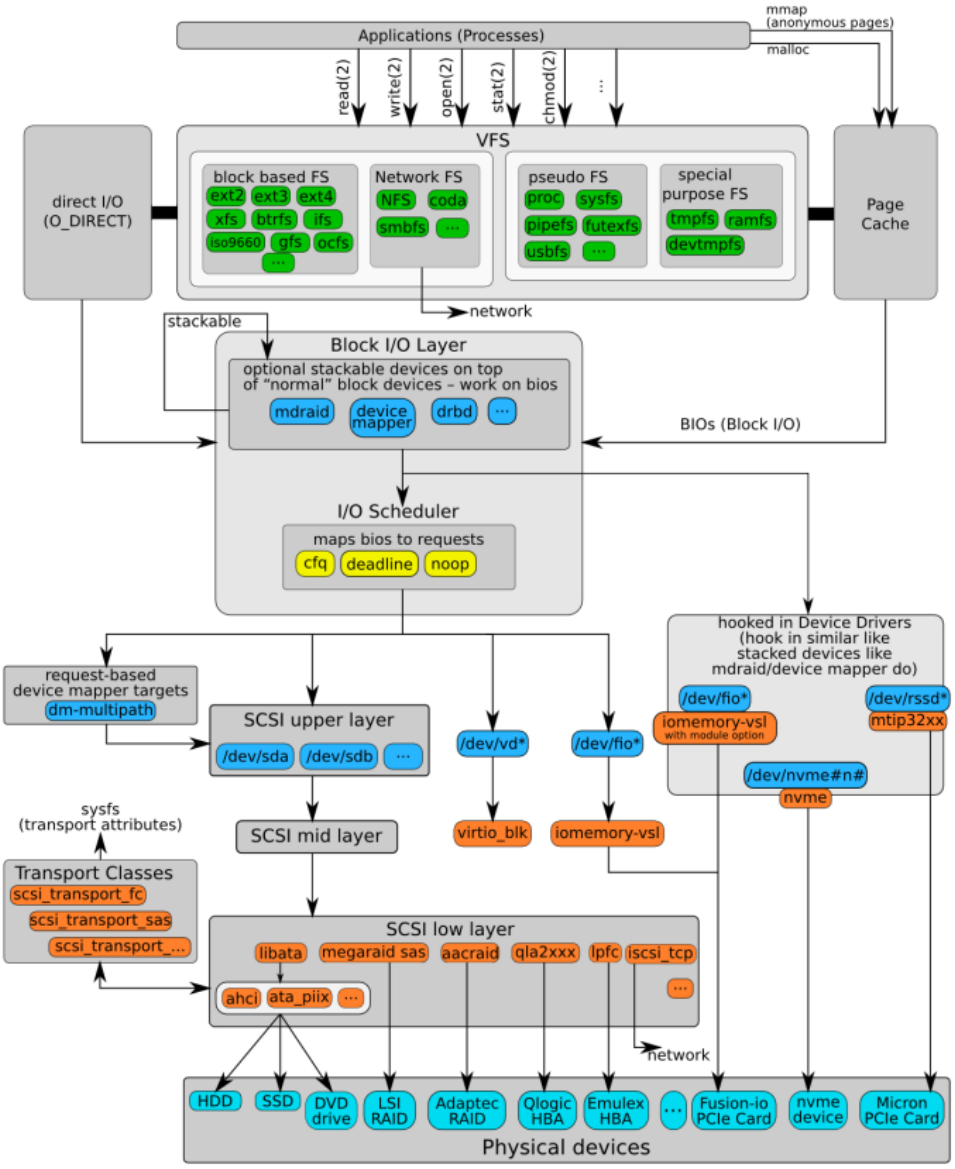
V adresářích `nodes` je vytvořen soubor `count`, který je zpřístupněn pouze pro čtení a obsahuje počet připojených uzlů. Konkrétní typ je dán jeho umístěním v adresářové struktuře.

Návrh byl takto stanoven a teď je na řadě samotná implementace řešení. Hlavní část tohoto řešení bude realizována ve zdrojových kódech komponenty `CTLFS` v prostoru jádra. Z počátku je třeba nastudovat jak se vytváří v této komponentě soubor s požadovanými vlastnostmi. Obecně jádro operačního systému poskytuje jednotné rozhraní `VFS` pro práci se souborovým systémem. Jinými slovy se jedná o abstraktní vrstvu (virtuální souborový systém), ve které jsou definovány základní funkce reprezentující operace se soubory. Ve výsledku je každému uživatelskému procesu poskytováno stejné rozhraní k souborovému systému a důsledkem tohoto nemusí žádný proces zajímat jak jsou konkrétně data ukládána.

Co se týká konkrétních souborových systémů, které jsou pod zmíněnou `VFS` vrstvou, tak jejich implementace spočívá pouze v ovladači, který řídí všechny operace a konkrétní detaily a algoritmy uchovávání dat. Ovladač obsahuje funkce, které jsou následně zaregistrovány do `VFS` vrstvy, ze které jsou pak volány na základě struktury jakou je `super_block` popisující konkrétní souborový systém.

Na obrázku 2.3 je znázorněno rozvrstvení jednotlivých komponent jádra operačního systému v oblasti vstupně výstupních operací, tedy včetně souborových systémů at už klasických používaných na blokových zařízení přes sítové, až po speciální, kterým je i zde použitý `CTLFS`.

[2] **Prompt** (výzva) indikuje, že příkazový řádek je připraven zpracovat další příkaz



Obrázek 2.3: Schéma I/O zásobníku jádra operačního systému Linux v.3x [1]

Zdrojové soubory `CTLFS` obsahují implementaci následujících funkcí volaných přímo pomocí `VFS`:

- `super_operations` - operace na úrovni celého souborového systému
  - `statfs` - poskytuje statistiky souborového systému
- `inode_operations` - operace s `i-nody`
  - `getattr` - Získání atributů o souboru - voláno např funkcí `stat` [3]
  - `readlink` - Načtení symbolického odkazu
  - `follow_link` - Následování symbolického odkazu, funkce zajišťuje vrácení `inode` struktury, kam daný odkaz ukazuje
  - `put_link` - Uvolňuje načtenou strukturu vrácenou funkcí `follow_link`
- `file_operations` - operace se soubory
  - `read` - čtení souboru
  - `write` - zápis souboru
- `dentry_operations` - operace se strukturami `dentry`[4]
  - `d_delete` - funkce je zavolána, když je zrušena poslední reference na tuto strukturu
  - `d_release` - funkce je zavolána pokud opravdu došlo k dealokaci struktury `dentry`

Z těchto uvedených funkcí je pro hlavní potřebu mechanismu evidence počtu připojených uzlů potřebná funkce pro čtení souboru `read`, jejíž implementace je obsažena v `tcmi_CTLFS_file_read`. Tato implementace mimo jiné zavolá čtecí funkci, která je pro konkrétní `CTLFS` soubor zaregistrována. Registrace je provedena při vytváření souboru funkcí `tcmi_ctlfs_genericfile_new`, jejíž jeden ze vstupních parametrů je ukazatel na čtecí funkci `read_method`, která bude provedena v okamžiku operace čtení z tohoto souboru. Tato funkce je volána z dalších funkcí dle typu vytvářeného souboru.

Konkrétně pro účel evidence počtu připojených uzlů je zapotřebí vytvoření souboru obsahující pouze celočíselnou hodnotu, což implementace `CTLFS` umožňuje pomocí funkce `tcmi_ctlfs_intfile_new`, jejíž vstupní parametry jsou následující:

- `parent` - ukazatel na rodičovskou strukturu - adresář, ve kterém je uložen
- `mode` - práva přístupu k souboru

[3] **stat, fstat, lstat, fstatat** - funkce vracející aktuální stav souboru

[4] **dentry** - directory entry - adresářové záznamy

- `object` - ukazatel na objekt, který provádí vytváření souboru
- `read_method` - ukazatel na čtecí funkci
- `write_method` - ukazatel na zapisovací funkci
- `maxlen` - maximální velikost obsažených dat v souboru
- `namefmt` - název souboru

Další analýza zdrojového kódu vede ke komponentě `manager`, která obsahuje implementaci jednotlivých správců migračního procesu. Mimo jiné dva správci zajišťují vytváření adresářové struktury `CTLFS` připojené do `/clondike`. Správce můžeme rozdělit podle funkcionality a typu uzlu následovně:

- **CCNMAN** - správce domovského uzlu zajišťující připojení uzlů typu PEN
- **PENMAN** - správce hostitelského uzlu zajišťující připojení uzlů typu CCN
- CCNMIGMAN - správce domovského uzlu zajišťující migrační proces (emigrování procesu)
- PENMIGMAN - správce hostitelského uzlu zajišťující migrační proces (imigrace procesu + vykonání)

První dva zvýraznění správci v okamžiku inicializace komponenty `TCMI`, která je inicializována při spouštění jádra operačního systému, provedou vytvoření adresářové struktury spolu s vytvořením souborů. Tato adresářová struktura lze potom připojit do námi zvoleného adresáře `/clondike` pomocí příkazu `mount`[5] a lze ji používat běžnými operacemi čtení a zápisu.

Vzhledem k tomu, že správci jsou implementačně do jisté míry podobní, je použit přístup dědičnosti z abstraktní třídy, který známe například z jazyka C++. Jádro operačního systému je implementováno pouze v jazyce C, a proto je tato vlastnost emulována uměle prostřednictvím struktur. Struktury `tcmi_man` a `tcmi_migman` jsou považovány za „abstraktní třídy", tím že obsahují společné atributy pro všechny správce zmíněné konkrétní správce. Jelikož tvorbu souborového systému vyvolávají pouze první dva zmínění správci, pak atribut obsahující počet připojených uzlů `count_connected_nodes` je přidán pouze do první „abstraktní třídy". Datový typ tohoto atributu byl zvolen `atomic_t`, což je celočíselný datový typ umožňující základní atomické operace nastavení hodnoty, inkrement, dekrement a její čtení.

[5] **mount** - příkaz umožňující připojení souborového systému do stromové adresářové struktury

Samotná realizace dědičnosti z „abstraktních tříd" je provedena za pomocí atributu `super`, jehož datový typ je právě „abstraktní struktura" a co je velmi důležité pro funkčnost je pozice definice atributu ve struktuře „konkrétní třídy". Pro názornost je zde uveden obecný příklad implementace.

```
1  struct abstraktni_trida {
2   /** Atributy abstraktni tridy */
3   int global_atribut;
4   ...
5  };
6  struct konkretni_trida {
7   /** Trida ze ktere dedim atribut y*/
8   struct abstraktni_trida super;
9
10  /** Atributy specificke pro konkretni tridu */
11  int specific_atribut;
12  ...
13 };
14 #define ABSTRAKTNI_TRIDA(obj)
15         ((struct abstraktni_trida *)obj)
```

Klíčová pozice atributu `super` umožňuje přistupovat k atributům abstraktní třídy za pomocí přetypování, které je pro jednodušší použití v kódu realizováno makrem `ABSTRAKTNI_TRIDA(obj)`.

```
ccn ..... Adresář pro uzly typu CCN
  mig ..... Adresář pro migraci procesů
    emigrate-ppm-p ..... Soubor pro migraci procesu preemptivně
    migproc ..... Adresář pro migrované procesy
    migrate-home ... Soubor pro konfiguraci migrace na domovký uzel
  nodes ..... Adresář pro jednotlivé připojené uzly
  stop ..... Řídící soubor pro odpojení uzlu
pen ..... Adresář pro uzly typu PEN
  mig ..... Adresář pro migraci procesů
    emigrate-ppm-p ..... Soubor pro migraci procesu preemptivně
    migproc ..... Adresář pro migrované procesy
    migrate-home ... Soubor pro konfiguraci migrace na domovký uzel
  nodes ..... Adresář pro jednotlivé připojené uzly
  stop ..... Řídící soubor pro odpojení uzlu
```

Obrázek 2.4: Adresářová struktura **CTLFS** vytvořená abstraktní třídou `tcmi_man`

2.4.2 Popis implementace

V konkrétní implementaci je tedy „abstraktní třídou" `tcmi_man` vytvořena pro každý typ uzlu (`CCN`/`PEN`) adresářová struktura spolu se soubory. Konkrétně je vytvořena hierarchie na obrázku 2.4. Další adresáře a soubory jsou tvořeny v „konkrétních třídách" reprezentující jednotlivé správce. Tímto přístupem je odstraněna potenciální duplikace zdrojového kódu při implementaci každého správce.

Při tvorbě konkrétního správce je tedy nejdříve zavolána společná část kódu, která následně vyvolá zaregistrované funkce daného správce pro tvorbu adresářů `init_ctlfs_dirs` a souborů `init_ctlfs_files`, ve kterých jsou teprve vytvářeny specifické adresáře a soubory dle typu správce. Jelikož se jedná o stejný soubor pro oba zmiňované správce, tak při této implementaci je použita společná část kódu, nikoliv naposledy zmíněné zaregistrované funkce.

Ve struktuře `tcmi_man` je přidán ukazatel `f_nodes_count`, který nově ukazuje na strukturu popisující vytvářený soubor. Soubor je vytvořen přidáním následujícího kusu kódu do inicializační funkce `tcmi_man_init_ctlfs_files`, která vytváří společné soubory správců.

```
1 if (!(self->f_nodes_count =
2  tcmi_ctlfs_intfile_new(self->d_nodes, TCMI_PERMS_FILE_R,
3  self, tcmi_man_count, NULL,
4  sizeof(int), "count")))
5 goto exit1;
```

Z ukázky kódu jsou patrné vstupní parametry volané funkce `tcmi_ctlfs_intfile_new`, jejichž obecná specifikace je popsána výše, ale zde jen pro vysvětlení konkrétně použitých parametrů.

První parametr `d_nodes` je ukazatel na adresář `nodes`, ve kterém bude soubor umístěn. Druhým parametrem `TCMI_PERMS_FILE_R` je konstanta definující přístupová oprávnění pro čtení souboru, dalším parametrem je objekt, který tento soubor vytváří, v našem případě buď ukazatel na `CCNMAN` nebo `PENMAN`. Čtvrtým v pořadí je ukazatel na čtecí funkci `tcmi_man_count`, která je vyvolána `VFS` subsystémem při operaci čtení souboru. Pátý parametr je nulový, protože zapisovací funkce není třeba v tomto případě. Předposledním parametrem je velikost celočíselného datového typu, který postačuje pro tyto účely a poslední parametr je název souboru.

Korektní odstranění souboru během ukončování komponenty `TCMI` je provedeno zavoláním dvojice funkcí `tcmi_ctlfs_file_unregister`, `tcmi_ctlfs_entry_put`, jejichž parametrem je ukazatel `f_nodes_count`.

Při operaci čtení z tohoto souboru je vyvolána funkce `tcmi_man_count`, jejíž dva vstupní parametry jsou ukazatel na objekt, který vytvářel soubor a vstupně výstupní parametr ukazující na hodnotu, která bude vrácena uživateli ve formě obsahu souboru. Tato funkce je pro oba zmiňované správce společná, ale konkrétní hodnota, která je vrácena závisí na vstupním parametru `obj`.

Následující ukázka implementace této čtecí funkce ukazuje realizaci čtení atributu čítače připojených uzlů a použití přetypování objektu pomocí makra.

```
1 static int tcmi_man_count(void *obj, void *str){
2  int *count = (int*) str;
3  struct tcmi_man *self = TCMI_MAN(obj);
4  *count = atomic_read(&self->count_connected_nodes);
5 return 0;
6 }
```

Soubor je takto korektně vytvořen a dále je nutné inicializovat atribut `count_connected_nodes` na nulovou hodnotu a najít správná místa pro jeho inkrementaci a dekrementaci. Vzhledem k operacím inkrementace a dekrementace, které mohou být prováděny paralelně, jde o situaci, kdy atribut je sdílenou proměnnou a je nutná synchronizace. Synchronizace je provedena pomocí přidaného zámku `sem`, jehož definice je umístěna v abstraktní třídě `tcmi_man`. Inicializace atributu na hodnotu 0 je provedena ve společné funkci `tcmi_man_init`, kam je přidán následující kus kódu.

```
1 sema_init(&self->sem,1);
2 down(&self->sem);
3  atomic_set(&self->count_connected_nodes, 0);
4 up(&self->sem);
```

Samotné umístění inkrementace atributu závisí na typu správce. V případě `PENMAN` je dáno zavoláním zapisovací funkce do souboru `connect`, kdy v tomto okamžiku je zavolána funkce `tcmi_penman_connect`, která zajistí vznik správce `PENMIGMAN` a naváže spojení se vzdáleným uzlem typu `PEN`. Na konci této funkce, kde je úspěšně provedeno připojení vzdáleného uzlu, je přidána inkrementace atributu uvnitř zámku zajišťující výlučný přístup.

```
1 down(&TCMI_MAN(&self)->sem);
2  atomic_inc(&TCMI_MAN(&self)->count_connected_nodes);
3 up(&TCMI_MAN(&self)->sem);
```

V druhém případě u správce `CCNMAN` je implementačně zcela stejné zavolání inkrementace atributu přidáno na konec funkce `tcmi_ccnman_process_sock`, která je zavolána v okamžiku příchozího soketového spojení od vzdáleného uzlu typu `CCN`. V této funkci je provedeno vytvoření správce `CCNMIGMAN` a ověření vzdáleného uzlu. Takto přidané části kódu zajistí správné inkrementace atributů pro každého správce. S dekrementací je to trochu složitější vzhledem k tomu, že správce `CCNMAN`, ani `PENMAN` nezajišťuje odpojení vzdálených uzlů.

Odpojení příslušného vzdáleného uzlu mají na starost konkrétní správci spojení `CCNMIGMAN`, respektive `PENMIGMAN`. Bohužel ti nemají žádný ukazatel na správce, který je vytvořil, a proto byl přidán jeden vstupní parametr `parent` typu ukazatel na strukturu `tcmi_man` do funkcí `tcmi_ccnmigman_new` a `tcmi_penmigman_new`.

Tímto je zajištěna možnost dekrementu atributu podobnou konstrukcí jako je provedena inkrementace jen s rozdílem, že je volána funkce `atomic_dec`. Inkrementace u obou správců `CCNMIGMAN` a `PENMIGMAN` jsou přidány ve funkcích,

jež jsou volány při odstraňování souborů symbolických odkazů z příslušných adresářů `nodes`, konkrétně do funkcí `tcmi_ccnmigman_stop_ctlfs_files` a `tcmi_penmigman_stop_ctlfs_files`.

Celý mechanismus evidence připojených uzlů je zcela funkční a umožnil optimalizovat doposud nešťastné řešení zjišťování počtu připojených uzlů pomocí mnoha externích příkazů při každém vyvolání `promtu`. Na základě této implementace byl upraven skript `bash_prompt.sh`, který nyní obsahuje pouze čtení hodnoty z příslušného souboru `count`. Dále bylo ve skriptu nahrazena sekce, kde je získávána IP adresa uzlu, tato část byla nahrazena za načítání souboru `clondike/ccn/listen`, kam je IP adresa napsána jiným `Bash` skriptem při startu operačního systému.

Tento mechanismus získání a nastavení IP adresy, kde má uzel `CCN` naslouchat, není zcela ideální. Možné řešení do budoucna je konsolidace `Bash` skriptů do implementace `Simple Ruby Directoru`, který bude mít na starost veškerá nastavení a mechanismy spojené s uživatelským prostorem projektu `Clondike`. Vzhledem k rozsahu tato problematika převyšuje rámec této práce.

2.5 Ověřovací měření opraveného systému

Tato část práce je věnována ověření funkčnosti opraveného systému. Již během vývoje opravných řešení byly prováděny testy migračního mechanismu kompilací malých projektů, které obsahovaly jednotky zdrojových souborů. Tyto testy byly střídavě prováděny jak na fyzických strojích, tak ve virtualizovaném prostředí realizované pomocí nástroje `VMware Workstation` [29] na třech strojích. V těchto testech nebyly zjištěny žádné komplikace. Bohužel při pokusu o změření kompilace jádra v2.6.32.5 na fyzických strojích, které probíhalo ve školní laboratoři v závěrečné fázi této práce, byly zjištěny poměrně zásadní chyby, které se projevují při takto velkých úlohách až na fyzickém hardware. Jedná se o chyby typu „race condition", což jsou chyby, které nastávají z důvodu paralelně běžících procesů, které jsou určitým způsobem datově závislé. Během posledních čtrnácti týdnů byla snaha tyto chyby analyzovat a případně odstranit, právě kvůli reálnému měření, jehož výsledky by bylo možné porovnat s výsledky měření prováděné vedoucím práce na starším jádře v2.6.33.1.

Vzhledem k velmi komplexnímu projektu, kterým `Clondike` bezpochyby je, tak samotná analýza byla prováděna obdobným způsobem jako analýzy chyb uvedené v první části této práce, jen s tím rozdílem, že veškeré sledování vývojářských výpisů a práce s nimi, byla prováděna na fyzických strojích ve školní laboratoři. Nalezení příčiny chyby není u tohoto typu jednoduchá záležitost, protože se chyba vyskytuje v náhodných časech a co je horší dle chybových výpisů jádra i při jiných volání funkcí. Tento fakt velmi stěžuje samotné hledání příčiny, a proto v současné chvíli není možné provést rychlou opravu. Dle prvních odhadů analýzy se bude týkat oblasti systémového volání spolu s mechanismem vytváření a ukončování procesů. Z předchozích

zkušeností s aktualizací projektu `Clondike` na nové jádro v3.6.11 a informací, které poskytují vývojáři jádra [30], je možné, že tato chyba je zaviněna mírnou nekompatibilitou zdrojových kódů projektu `Clondike` a samotného jádra, kde bylo od verze 2.6.x provedeno mnoho změn i v této oblasti. Na základě těchto získaných informací je pokus o nalezení příčiny chyby a její oprava časovou náročností nad rámec této práce.

Alespoň kladným aspektem faktu nemožnosti změřit kompilaci jádra na clusteru `Clondike` je zjištění, že opravy provedené v rámci této práce nezpůsobují tuto chybu typu „race condition". Pro toto zjištění bylo v rámci analýzy provedena operace vrácení změn zdrojových souborů do stavu, který odpovídal stavu na počátku této práce. Byl proveden 24 hodinový test a tento typ chyby se projevil na fyzickém stroji. Obecně je chyba způsobena operací dereferencování na ukazatel s nulovou hodnotou `NULL` v různých funkcích. Dále může chyba souviset se známou chybou dle vývojářů této verze jádra, při níž je vypisována tato chybová hláška `BUG: Bad rss-counter state mm:XXXXXX idx:2 val:2` do terminálu.

Z těchto důvodů nebylo měření na fyzických strojích ve školní laboratoři provedeno. Provedení měření na virtualizovaném prostředí postrádá jakýkoli význam z toho důvodu, že vývoj byl prováděn na slabém hardware (notebook Lenovo X200, Intel Dual Core), kde každý z virtuálních strojů měl nastaveny velmi omezené systémové prostředky. Hodnoty vytíženosti `CPU` a celková doba běhu by byla silně závislá na režii virtualizace. A není zaručeno, že by během měření nedošlo k tomuto typu chyby.

2.6 Návrh a implementace ukládání dat do DB Cassandra

2.6.1 Návrh

Samotný návrh této implementace vychází z požadavků, které budou součástí distribuovaného logu migračních transakcí. Tento log bude zajišťovat projekt `Apache Cassandra`, jehož stručný popis je uveden v části 3.3 této práce. Migrační transakce, respektive samotná migrace procesu sebou nese spousta informací, které lze využít k účelům výpočtu důvěryhodnosti jednotlivých výpočetních uzlů clusteru projektu `Clondike`.

Každá migrační transakce musí obsahovat jednoznačný identifikátor v rámci clusteru, identifikátory zúčastněných uzlů a v poslední řadě jednotlivé stavy procesu spolu s časy, kdy ke kterému došlo. Toto jsou základní informace, které je třeba z migračního procesu získat a prostřednictvím nově implementovaného ovladače v komponentě `Simple Ruby Director` zapsat do distribuovaného transakčního logu.

Jednoznačně tento mechanismus ukládání dat z migračního procesu bude implementován napříč celým projektem `Clondike`, neboli jak v prostoru já-

dra, tak v uživatelském prostoru. Celý postup začne detailním nastudováním komponenty TCMI v prostoru jádra, kde je třeba najít požadované informace, které následně budou prostřednictvím již implementovaných zpráv zaslány do uživatelského prostoru, kde dojde k jejich dalšímu zpracování.

Druhá část, tedy implementace v uživatelském prostoru spočívá v nastudování mechanismu příjmu zpráv a extrahování informací v nich uložených. Jako další krok je zjištění možnosti zápisu z Ruby skriptů do distribuovaného logu projektu Apache Cassandra. Pak naprogramovat příslušné Ruby skripty, které budou ve formě připraveného základního rozhraní Clondike <-> Apache Cassandra.

2.6.2 Popis implementace - prostor jádra

Pro začátek je třeba zjistit jak bude migrační transakce jednoznačně identifikována. Po nastudování většiny zdrojových kódů komponenty TCMI lze konstatovat fakt, že jako relativně jednoznačný identifikátor migrační transakce lze použít kombinaci tří informací o migrovaném procesu. První je PID, druhým je název procesu Filename a posledním je Jiffies, což je počet tiků hodin od začátku nabíhání operačního systému. Stejná varianta identifikátoru je použita u názvu checkpoint souboru migrovaného procesu.

Bohužel v současné době není nikde uložena hodnota jiffies pro daný proces vyjma transakčních zprávy tcmi_p_emigrate_msg, která je posílána po vytvoření checkpoint souboru migrovaného procesu na vzdálený uzel typu PEN, kde je využita k načtení zmíněného checkpoint souboru pro extrahování kontextu procesu pro jeho vzdálené vykonání.

Na základě tohoto faktu je analyzován zdrojový kód a je provedeno několik testů, ze kterých vyplývá, že nelze atribut jiffies uložit do struktury tcmi_task, která je alokována funkcí tcmi_shadowtask_new vždy až dojde k rozhodnutí o migraci aktuálně spuštěného procesu. Proto je atribut přidán o úroveň výš čili přímo do task_struct na stejné místo jako byl přidán v předchozí části této práce atribut nonmigratable.

Následně je provedeno přiřazení hodnoty tomuto atributu ve funkci typu „háček" tcmi_mighooks_execve, která je zavolána ihned po spuštění daného procesu. Zjištění hodnoty jiffies je provedeno stejnojmennou funkcí jádra. V tomto okamžiku je tento atribut využit při sestavování názvu checkpoint souboru ve funkci tcmi_taskhelper_checkpoint, která mimo jiné vytváří samotný soubor na disku.

Vzhledem k oddělení atributu ckpt_name, názvu checkpoint souboru, a samotných atributů PID, exec_name ve zprávě tcmi_p_emigrate_msg byla zvolena cesta přidání nového atributu jiffies do struktury této zprávy a tcmi_ppm_p_migr_back_guestreq_procmsg, která je odesílána jako informační zpráva o stavu migrace procesu od hostitelského uzlu PEN k domovskému CCN.

Pro názornost je zde uvedena ukázka struktury zprávy, která je posílána v okamžiku migrace procesu na jiný uzel typu `PEN`.

```
1  struct tcmi_p_emigrate_msg {
2   struct tcmi_msg super;
3   struct {
4    pid_t reply_pid;
5    int32_t size;
6    int32_t exec_name_size;
7    int16_t euid;
8    int16_t egid;
9    int16_t fsuid;
10   int16_t fsgid;
11   unsigned long jif;
12  } pid_and_size  __attribute__((__packed__));
13  char *exec_name;
14  char *ckpt_name;
15 };
```

Tímto přidáním je docíleno poslání všech třech částí jednoznačného identifikátoru migrační transakce hostitelskému uzlu, který je podobným způsobem jako uzel domovský zpracuje a pošle do uživatelského prostoru. Jsou realizovány další kroky, které jsou nezbytné pro funkčnost nově přidaného atributu do zpráv.

Prvním z nich je vyřešení extrahování tohoto atributu ze zprávy. K tomuto kroku vede poměrně dlouhé studium celého mechanismu příjmu zpráv a následné extrahování obsahovaných atributů. Vše je realizováno, jednoduše řečeno, zaregistrovanou funkcí, která je pro každý typ zprávy jiná. Veškerou komunikaci prostřednictvím zpráv zajišťuje část `comm`, která je volána konkrétním správce obecného typu `MIGMAN`. Celý mechanismus je poměrně dost netransparentní a je zde použita určitá míra abstrakce, která není zcela jednoduchá na stručné vysvětlení. Podrobné nastudování a vysvětlení tohoto mechanismu překračuje rámec této práce.

Výsledkem celého studia mechanismu příjmu a odesílání zpráv je fakt, že je vytvořena následující funkce na extrahování přidaného atributu `jiffies`.

```
1 static inline unsigned long tcmi_p_emigrate_msg_ckpt_jif(struct
     tcmi_p_emigrate_msg *self)
2 {
3  return self->pid_and_size.jif;
4 }
```

Tato funkce je volána při odesílání zprávy do uživatelského prostoru ve funkci `tcmi_migcom_immigrate`, která je vyvolána na základě příjmu zprávy typu `tcmi_p_emigrate_msg`. Konkrétně jsou poslány do uživatelského prostoru dvě zprávy. První zpráva je žádost o akceptaci imigrovaného procesu `immigration_request` a druhá zpráva je `immigration_confirmed` potvrzení, že byl imigrovaný proces úspěšně zpracován. U potvrzovací zprávy je již předáván atribut `jiffies` pomocí `guest` úlohy. Této úloze je `jiffies` nastaven při jejím vytváření funkcí, která je v ukázce výše.

Další typ zprávy do uživatelského prostoru je zpráva o chybě v mechanismu migrace, konkrétně se jedná o zprávu `emigration_failed`, jejíž zavolání probíhá v případě chybového stavu migračního mechanismu z funkce `tcmi_migcom_emigrate_ccn_npm`. Jedná se o funkci, která je zaregistrována v operacích správce `CCNMAN`, konkrétně v `ccnman_ops` jako `emigrate_npm`. Tato operace je volána při úspěšném rozhodnutí o migraci procesu.

Zpráva o chybě doposud obsahovala pouze `PID` migrovaného procesu, ale vzhledem k nutnosti identifikovat migrační transakci za pomoci třech atributů jsou i do této zprávy přidány atributy `name` a `jiffies`. Toto přidání znamená dominový efekt, který spočívá v přidáním těchto atributů v podobě vstupních parametrů následujících vzájemně volaných funkcí.

`tcmi_man_emig_npm`

↓

`tcmi_migcom_emigrate_ccn_npm`

Po modifikaci definic zmiňovaných funkcí jsou nové parametry přidány do volání operace `emigrate_npm`.

```
1  int tcmi_man_emig_npm(struct tcmi_man *self, pid_t pid, const
       char *name, unsigned long jiffies, ...)
2  {
3  ...
4  if (self->ops->emigrate_npm)
5  err = self->ops->emigrate_npm(pid, name, jiffies, migman, regs,
       npm_params);
6  ...
7  }
```

Co se týká úpravy implementace na úrovni posílání zpráv mezi uzly, tak se jedná o vše co bylo upraveno. Další nezbytnou úpravou je nutnost informovat uživatelský prostor při kontrole procesu na migraci. To znamená přidat atribut `jiffies` jako parametr funkce `director_npm_check` zajišťující právě tuto kontrolu. Úpravy komponenty `Director` se týkají následujících funkcí:

`director_npm_check` Úpravy interně volaných funkcí `npm_check` a `npm_check_full`. Přidání atributu `jiffies` do následující společné struktury `npm_check_params` a příslušné přiřazení jeho hodnoty v interních funkcích. Následující struktura je u všech typů zpráv obdobná pouze obsahuje jen potřebné atributy.

```
1  struct npm_check_params {
2   /* In params */
3   pid_t pid;
4   uid_t uid;
5   int is_guest;
6   const char* name;
7   int name_length;
8   struct rusage *rusage;
```

```
 9   unsigned long jiffies;
10   /* Params only in full mode */
11   const char __user * const __user * args;
12   const char __user * const __user * envp;
13
14   /* Out params */
15   int decision;
16   int decision_value;
17 };
```

Ve funkci `npm_check_create_request` vytvářející `Netlink` zprávu je přidán atribut `jiffies` zavoláním následující funkce `nla_put_u64`.

```
1  ret = nla_put_u64(skb, DIRECTOR_A_JIFFIES, check_params->jiffies)
       ;
2   if (ret != 0) goto failure;
```

Tato funkce mající tři parametry zajistí přidání neznaménkového celočíselného 64-bitového datového typu, kterým atribut `jiffies` od začátku všech definic je, do příslušně připravené zprávy v `skb` - soketová výstupní vyrovnávací paměť. Druhým parametrem je konstanta `DIRECTOR_A_JIFFIES`, která musí být definovaná na obou stranách jak v prostoru jádra, tak v uživatelském prostoru. Slouží k identifikaci atributu v `Netlink` zprávě a je v tomto případě definována společně s ostatními identifikátory v souboru `msgs.h` ve výčtovém typu `enum`. Třetí parametr je samotná hodnota přidaného atributu `jiffies`. Zcela analogickou úpravou jsou modifikovány funkce `npm_check_full` a `npm_check_full_create_request`.

`director_immigration_request` V rámci této funkce je upravena interně volaná `immigration_request`, kde jsou přidány dva nové parametry `PID` a `jiffies`. V rámci funkce `immigration_request_create_request`, která vytváří, respektive plní zprávu atributy je úprava provedena u atributu `jiffies` stejným voláním jako v předchozím typu zprávy. Atribut `PID` je 32-bitový celočíselný typ, pro který je zavolána funkce `nla_put_32` s identifikátorem atributu `DIRECTOR_A_PID`.

`director_immigration_confirmed` Podobným způsobem je i v interních funkcích této funkce provedena analogická úprava jako v předchozích typech zpráv jen zde je přidán kromě atributu `jiffies` ještě název souboru `name`, u kterého je situace obdobná s rozdílem ve volané funkci `nla_put_string`, která přidává atribut typu řetězec, konkrétně s identifikátorem `DIRECTOR_A_NAME`.

`director_emigration_failed` U této funkce stačí pouze poznamenat fakt, že jsou zde analogicky přidány atributy `name` a `jiffies` podle předchozích úprav.

Takto upravená implementace v prostoru jádra poskytuje navrhované informace uživatelskému prostoru prostřednictvím `Netlink` rozhraní. V další části je popsána implementace z pohledu uživatelského prostoru.

2.6.3 Popis implementace - uživatelský prostor

Na druhé straně je zapotřebí v komponentě `Director API`, která podobným způsobem přijímá a odesílá zprávy přes `Netlink`, příslušně upravit funkce, kde dochází k extrahování atributů zpráv a následnému zavolání dalších funkcí obohacených o přidané atributy. Každý typ zprávy má funkci typu `handler`, která je vyvolána při příjmu určitého typu zprávy. Upravit je třeba následující funkce typu `handler`.

`handle_npm_check` a `handle_npm_check_full` V těchto funkcích je přidána proměnná pro atribut `jiffies`. Interně pomocí funkce `nlmsg_find_attr` je dle daného identifikátoru `DIRECTOR_A_JIFFIES` atribut ze zprávy `req_msg` získán v podobě ukazatele. Následně je pak prostřednictvím funkce `nla_get_u64` získána jeho hodnota jako neznaménkový 64-bitový celočíselný datový typ. Následuje ukázka popisovaného volání funkcí.

```
1 nla = nlmsg_find_attr(nlmsg_hdr(req_msg), sizeof(struct
    genlmsghdr), DIRECTOR_A_JIFFIES);
2 jiffies = nla_get_u64(nla);
```

Po získání tohoto atributu je jeho hodnota předána jako vstupní parametr do funkcí typu `callback`, konkrétně `npm_callback`, respektive `npm_callback_full`, jejichž definice bylo třeba upravit.

`handle_immigration_request` U tohoto typu zprávy jsou přidány atributy `jiffies` a `PID`. První atribut je extrahován stejným způsobem jako v předchozích typech zpráv a druhý je extrahován podle identifikátoru `DIRECTOR_A_PID`, který je s ostatními definován v souboru `msgs.h`. Pro získání hodnoty je použita funkce `nla_get_u32`, která vrací klasický 32-bitový celočíselný datový typ. Hodnoty atributů jsou dále předány jako parametry funkci typu `callback`, konkrétně `immigration_request_callback`, jejíž definici bylo třeba upravit.

`handle_immigration_confirmed` Analogicky jako v předchozích případech je zde provedeno extrahování atributu `jiffies`. U druhého `name` je použit identifikátor `DIRECTOR_A_NAME` a pro získání jeho hodnoty je použita funkce `nla_data`. Jako u předešlých případů je upravena definice volané interně funkce `immigration_confirmed_callback`.

`handle_emigration_failed` U této funkce stačí pouze poznamenat fakt, že jsou zde analogicky přidány atributy `name` a `jiffies` podle předchozích úprav s příslušnou úpravou volané funkce `emigration_failed_callback`.

Většina úprav v této části byla obdobná úpravám v prostoru jádra, s tím rozdílem, že zde se volají jiné funkce, vzhledem k mechanismu extrahování atributů. Soubor `msgs.h` s identifikátory zpráv a atributů je svým obsahem ekvivalentní souboru v prostoru jádra.

Další úprava v uživatelském prostoru spočívá ve změnách v rozhraní mezi jazykem `C` a `Ruby`. Jedná se tedy o rozhraní napsané v jazyce `C`, kde je používána stěžejní knihovna `ruby.h`, která umožňuje využívat speciální funkce, které zprostředkují propojení mezi oběma jazyky. Konkrétně se jedná o komponentu `Ruby Director API`, kde jsou implementovány volané `callback` funkce uvedené v předchozích odstavcích. Přesněji řečeno jsou implementovány zaregistrované funkce na konkrétně volanou `callback` funkci. V následující ukázce kódu jsou provedeny registrace implementovaných funkcí, do jejíchž názvu je přidán prefix **ruby__**.

```
1 register_npm_check_callback(ruby_npm_check_callback);
2 register_npm_check_full_callback(ruby_npm_check_full_callback);
3 register_immigration_request_callback(
      ruby_immigrate_request_callback);
4 register_immigration_confirmed_callback(
      ruby_immigration_confirmed_callback);
5 register_emigration_failed_callback(
      ruby_emigration_failed_callback);
```

Takto zaregistrované funkce realizují spojení jazyka `C` a `Ruby`, jejíž obecný popis je popsán v části 3.2. Konkrétně v tomto případě bylo nutné upravit interně volanou funkci `rb_funcall`, jejíž třetím parametrem je počet vstupních parametrů odpovídající funkce v `Ruby`. Toto číslo bylo ve všech zmíněných funkcích patřičně navýšeno a dalším krokem bylo samotné přidání parametrů, kde je třeba jejich hodnoty správně převést na datový typ, které používá `Ruby`. Pro konkrétní atributy byly převedeny následujícími funkcemi.

- `jiffies` - **ULONG2NUM()**
- `name` - **rb_str_new2()**
- `pid` - **INT2FIX()**

Základní přehled těchto transformačních funkcí je uveden v části 3.2 popisující vzájemné propojení obou jazyků `C` a `Ruby`.

Touto úpravou se dostáváme ke komponentě `Simple Ruby Director`, která využívá předešlé komponenty ve formě externí zkompilované knihovny `directorApi.so`. Hlavním skriptem této komponenty je `Director.rb`, kde jsou zaregistrovány instance ostatních tříd, které jsou v rámci přehlednosti umístěny v oddělených souborech. Celý uživatelský `Netlink` subsystém tj. `Ruby Director API` a `Director API` je inicializován instancí objektu `NetlinkConnector`. V této instanci je možné zaregistrovat interní metody objektu jako `callback` funkce, jejichž úpravy byly popsány v předchozích odstavcích. Samotná registrace zmiňovaných funkcí je uvedena v následující ukázce.

```
1 DirectorNetlinkApi.instance.registerNpmCallback(instance, :
      connectorNpmCallbackFunction)
```

```
2 DirectorNetlinkApi.instance.registerNpmFullCallback(instance, :
    connectorNpmFullCallbackFunction)
3 DirectorNetlinkApi.instance.registerImmigrateRequestCallback(
    instance, :connectorImmigrationRequestCallbackFunction)
4 DirectorNetlinkApi.instance.registerImmigrationConfirmedCallback(
    instance, :connectorImmigrationConfirmedCallbackFunction)
5 DirectorNetlinkApi.instance.registerEmigrationFailedCallback(
    instance, :connectorEmigrationFailedCallbackFunction)
```

Metody objektu v `Ruby`, které se mapují na `callback` funkce mají prefix **connector** a postfix **Function**. Například pokud přijde do uživatelského prostoru zpráva typu `DIRECTOR_CHECK_NPM` je vyvolána následující sekvence volání.

`handle_npm_check` → `npm_callback` → `ruby_npm_check_callback` →

→ `rb_funcall` → `connectorNpmCallbackFunction`

Po zavolání `Ruby` metody typu „**connector**", která musí mít definované vstupní parametry dle definice v komponentě `Ruby Director API`, lze vstupní parametry běžným způsobem zpracovávat.

Další propojení `Simple Ruby Directoru` a distribuovaného transakčního logu projektu `Apache Cassandra` bude zajišťovat `Datastax Ruby Driver`[31], tento ovladač je kompatibilní pouze s `Ruby v1.9.3` a vyšší. Jeho instalaci lze provést následujícími způsoby:

- Balíčkový systém `rubygems` - příkazem `gem install cassandra-driver`
- Stažený Gemfile - příkazem `gem 'cassandra-driver'`

Po instalaci je v adresáři komponenty `Simple Ruby Director` vytvořena nová složka `cassandra`, kde je implementován `Cassandra driver` pro účely projektu `Clondike`. Soubor `CQL3Driver.rb` obsahuje dvě třídy `CQL3Driver` reprezentující samotný ovladač a `TaskRecord` popisující záznam, který bude zapisován do `Cassandra` databáze. Metoda `createRecord` třídy `CQL3Driver`, která vytváří a zapisuje záznam do databáze, je volána právě z příslušných metod s prefixem **connector** třídy `NetlinkConnector` v závislosti na typu přijaté zprávy. Díky přidaným atributům do zpráv, jejichž hodnoty byly získány z jádra operačního systému, lze využít právě pro zápis do distribuovaného logu, respektive databáze projektu `Apache Cassandra` a do budoucna zaznamenávat migrační proces distribuovaným způsobem.

Prozatím je tento ovladač vyvinut pouze jako testovací rozhraní (lze zapisovat do log souborů na disku), které je třeba doladit před samotným propojením s projektem `Apache Cassandra`. Prozatím byla inicializace ovladače zakomentována s ohledem na odstraňování chyb při běhu operačního systému, respektive projektu `Clondike`. Vzhledem k vysokým systémovým nárokům nebyl projekt `Apache Cassandra`, který je napsán v jazyce `Java` s projektem

`Clondike` dlouhodobě testován. Byly provedeny jen základní testy zda-li je funkční ovladač `Datastax Ruby Driver` s databází, rozsáhlejší testy jsou nad rámec této práce.

Dlouhodobé propojení dvou systémů a provedení příslušných testů je směr blízké budoucnosti, kam se projekt `Clondike` zcela jistě chystá.

Kapitola 3

Dokumentace projektu Clondike

Celá tato kapitola je věnována dokumentaci projektu `Clondike`, především v oblasti uživatelského prostoru, z toho důvodu, že nebyl dosud nikým podrobněji dokumentován. Prvním část je stručně věnována protokolu a knihovně `Netlink` spolu se základním popisem použití v tomto projektu. Druhá část je věnována mechanismu propojení jazyka `C` a `Ruby` s velkým důrazem na obecný princip. Je zde vysvětleno jak to celé funguje a jak zdrojovým kódům vlastně rozumět.

3.1 Netlink

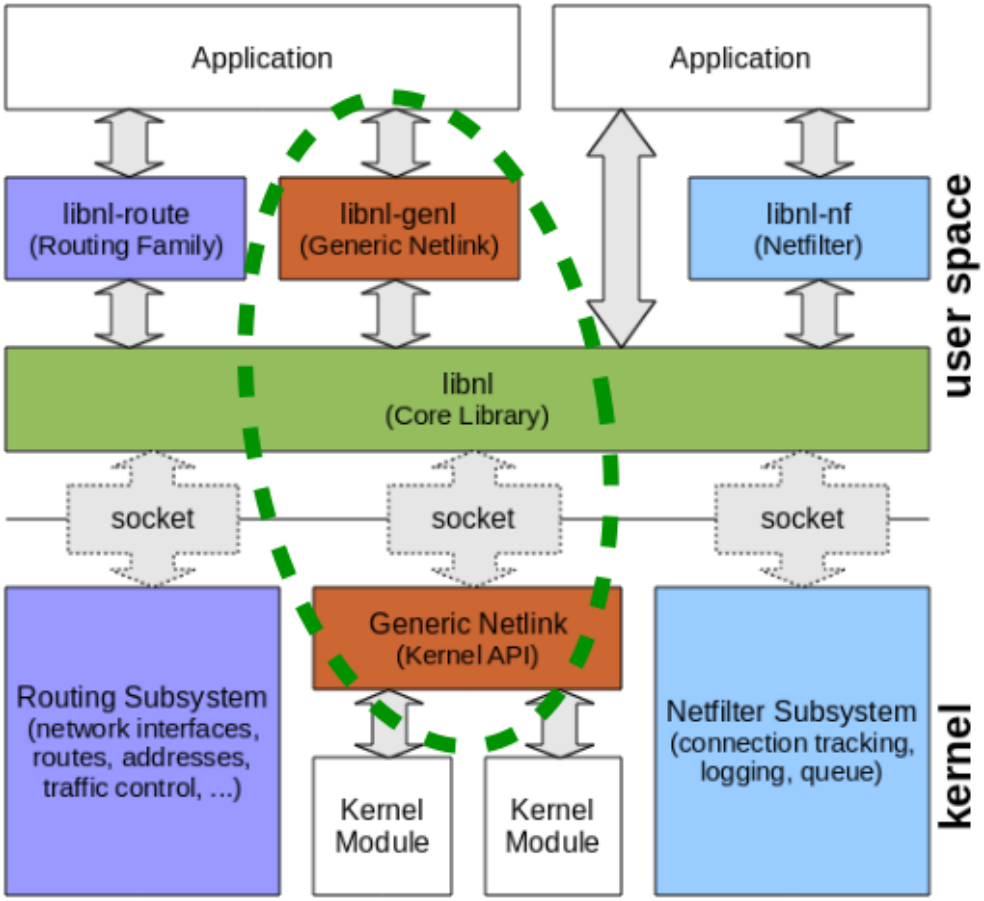
Netlink Protocol Library Suite[2] je sada knihoven poskytující rozhraní `API` pomocí protokolu `Netlink`. Tento protokol poskytuje rozhraní mezi uživatelským prostorem a prostorem jádra. Tento systém si lze představit jako sběrnici, ke které se připojí procesy z uživatelského prostoru nebo z prostoru jádra a komunikují prostřednictvím zpráv na soketové úrovni. Protokol byl navržen tak, aby byl pružnější nástupce `ioctl`[6] a poskytoval především soketovou komunikaci.

Celá tato sada je složena z následujících dílčích knihoven:

- **libnl**
- **libnl-genl**
- libnl-route
- libnl-nf

Pouze první dvě jsou využity pro projekt `Clondike`. První knihovna *libnl* je základní knihovnou implementující základy potřebné pro použití protokolu

[6] **ioctl** - input/output control - funkce manipulující se zásadními parametry zařízení prostřednictvím operací nad speciálními soubory



Obrázek 3.1: Schéma systému `Netlink` s použitými částmi pro projekt `Clondike` [2]

`Netlink`, jako je práce se sokety a zprávami (tvorba, odesílání, příjem, čtení a zápis jejich obsahu). Jedná se o minimalistické jádro `Netlink Protocol Library Suite` v uživatelském prostoru.

Na tomto základě závisí další knihovny včetně *libnl-genl*, což je API pro *generic netlink protokol*, který je použit jak na úrovni jádra, tak v uživatelském prostoru, respektive pro komunikaci mezi oběma klienty nazvaný `Director`.

Komunikace mezi dvěma `Directory` je realizována pomocí transakcí. Každá transakce se skládá ze dvou zpráv. První je vždy žádost a druhá je odpověď, která zároveň slouží jako potvrzení. Potvrzovací mechanismus, který subsystém `Netlink` nabízí není využit. Každá z transakcí má sekvenční číslo, které je po dobu trvání transakce uloženo, konkrétně ve spojovém seznamu `itx`.

Všechny typy zpráv odesílané/přijímané z/do prostoru jádra procházejí následujícím transakčním algoritmem, který je implementován v prostoru jádra.

```
1  int msg_transaction_do(int msg_code, struct msg_transaction_ops*
       ops, void* params, int interruptible) {
2  struct genl_tx tx;
3
4  int res = msg_transaction_request(msg_code, &tx, ops, params,
       interruptible);
5
6  if ( res ) return res;
7
8  return msg_transaction_response(&tx, ops, params);
9  }
```

První funkce `msg_transaction_request` zajistí vytvoření obecné zprávy typu žádost, pak zavolá interně funkci `create_request`, která dle typu zprávy dovytvoří její finální podobu. Následuje krok, kdy se získá sekvenční číslo transakce a uloží se do spojového seznamu k ostatním již běžícím transakcím. Proběhne odeslání zprávy za pomocí funkce `genlmsg_unicast_tx` a v ten okamžik dle návratové hodnoty je buď zahájeno čekání na odpověď, nebo je transakce ukončena.

V případě, že je vše úspěšně odesláno, pak je zahájeno čekání na zprávu typu odpověď prostřednictvím funkce `msg_transaction_response`. Interně je toto provedeno čekáním procesu pomocí funkce `wait_event_timeout` [19]. Proces je probuzen v případě, že přijde jakákoli zpráva typu odpověď funkcí `generic_message_handler`, respektive interně zavolanou `wake_up_all` [19]. Jen ten proces, který nalezne ve spojovém seznamu svou transakci s nastaveným parametrem `done`, pak pokračuje v úspěšnému extrahování atributů z odpovědi a v poslední fázi korektně ukončuje transakci vymazáním jejího záznamu ze spojového seznamu `itx`.

Naopak v uživatelském prostoru je příjem zpráv typu žádost a odesílání zpráv typu odpověď realizován vláknem, které v nekonečné smyčce provádí následující činnost.

```
1  int run_processing_callback(int allow_block) {
2   struct nl_msg *msg = NULL;
3   int res;
4   while (1) {
5    if ( !allow_block && read_would_block(state.sk) )
6    return;
7
8    if ( (res = read_message(state.sk, &msg) ) != 0 ) {
9     printf("Error in message reading: %d\n", res);
10    if ( res == -EINTR ) // Interrupted => finish the thread
11     break;
12    continue;
13   }
14   handle_incoming_message(msg);
15  }
16 }
```

Tato část kódu je prováděna neustále dokola po inicializaci `Simple Ruby Directoru`. V případě, že přijde zpráva je probuzen spící proces ve funkci `rb_thread_wait_fd` a je zavolána funkce `run_processing_callback`.

```
1 static VALUE method_runDirectorNetlinkProcessingLoop(VALUE self)
    {
2         int netlink_fd = get_netlink_fd();
3         while (1) {
4                 run_processing_callback(0); // Do not allow
                      blocking!
5                 rb_thread_wait_fd(netlink_fd);
6 }
```

V této funkci je zahájeno čtení této zprávy `read_message`, které když neskončí chybou, čili jedná se o známý typ zprávy, je zavolána její obsluha `handle_incoming_message`. Pakliže dojde k chybě je zapsána chybová hláška do logu a není volána obsluha příchozí zprávy. V případě ukončování `Simple Ruby Directoru` je vyvoláno přerušení, které ukončí celé vlákno.

Každý typ zprávy má funkci typu `handle`, která je interně volána v případě, kdy přijde daný typ zprávy. Všechny typy zpráv používaný v projektu `Clondike` jsou uvedeny v následujícím seznamu.

- `DIRECTOR_REGISTER_PID` - posílána při registraci `Directoru API` do jádra
- `DIRECTOR_SEND_GENERIC_USER_MESSAGE` - obecná uživatelská zpráva informující uživatelský prostor
- `DIRECTOR_ACK` - potvrzovací zpráva
- `DIRECTOR_CHECK_NPM` - žádost o kontrolu procesu na migraci
- `DIRECTOR_CHECK_FULL_NPM` - žádost o kontrolu procesu na migraci s více atributy
- `DIRECTOR_NODE_CONNECTED` - zpráva o tom, že daný uzel byl připojen
- `DIRECTOR_NODE_DISCONNECTED` - zpráva o tom, že daný uzel byl odpojen
- `DIRECTOR_IMMIGRATION_REQUEST` - zpráva o imigrovaném procesu
- `DIRECTOR_IMMIGRATION_CONFIRMED` - zpráva o úspěšně provedené imigraci
- `DIRECTOR_TASK_EXIT` - zpráva o ukončení procesu
- `DIRECTOR_TASK_FORK` - zpráva o vytvoření procesu
- `DIRECTOR_GENERIC_USER_MESSAGE` - obecná uživatelská zpráva informující prostor jádra

- DIRECTOR_EMIGRATION_FAILED - zpráva o neúspěšné migraci
- DIRECTOR_MIGRATED_HOME - zpráva o migraci procesu zpět na hostitelský uzel
- DIRECTOR_NPM_RESPONSE - odpověď kontroly procesu na migraci
- DIRECTOR_NODE_CONNECT_RESPONSE - odpověď na nově připojený uzel
- DIRECTOR_IMMIGRATION_REQUEST_RESPONSE - odpověď s vyjádřením na imigraci procesu (akceptace/odmítnutí)

Každá z uvedených zpráv používá některé atributy jejichž kompletní seznam a popis je následující.

- DIRECTOR_A_PID - identifikátor procesu
- DIRECTOR_A_REMOTE_PID - identifikátor vzdáleného procesu
- DIRECTOR_A_PPID - identifikátor rodičovského procesu
- DIRECTOR_A_UID - identifikátor uživatele
- DIRECTOR_A_TASK_TYPE - typ procesu (1=guest/0=shadow)
- DIRECTOR_A_NAME - obecný řetězec používaný pro názvy
- DIRECTOR_A_LENGTH - 32-bitová délka
- DIRECTOR_A_INDEX - 32-bitový index
- DIRECTOR_A_SLOT_INDEX - 32-bitový index slotu
- DIRECTOR_A_SLOT_TYPE - typ slotu (1=core, 0=detached)
- DIRECTOR_A_ADDRESS - Adresa vzdáleného uzlu - formát typu řetězec
- DIRECTOR_A_AUTH_DATA - Autentizační data - bezpečnost
- DIRECTOR_A_USER_DATA - Obecný atribut pro uživatelská data
- DIRECTOR_A_ARGS - Pole parametrů procesu
- DIRECTOR_A_ARG - Jeden parametrů procesu
- DIRECTOR_A_ENVS - Pole proměnných prostředí
- DIRECTOR_A_ENV - Jedna proměnná prostředí
- DIRECTOR_A_REASON - 32-bitový atribut důvodu
- DIRECTOR_A_DECISION, - Typ rozhodnutí

- `DIRECTOR_A_DECISION_VALUE` - Hodnota rozhodnutí
- `DIRECTOR_A_EXIT_CODE` - Návratový kód procesu
- `DIRECTOR_A_ERRNO` - Chybový kód
- `DIRECTOR_A_RUSAGE` - Využití uzlu
- `DIRECTOR_A_JIFFIES` - 64-bitový identifikátor uzlu

Veškerá implementace `Netlink` zpráv a mechanismu odesílání/příjmu v prostoru jádra je obsažena v adresáři `sources/kernel_3.6.11/src/director/netlink`. Implementace na úrovni uživatelského prostoru je obsažena v `userspace/director-api` a `userspace/ruby-director-api`. Veškeré cesty jsou uvedeny relativně vůči kořenu `Clondike` repositáře [28].

3.2 Propojení Ruby a C

Samotné propojení dvou jazyků je zajištěno podporou ze strany vývojářů `Ruby`, kteří poskytují rozhraní `Ruby C API` pro jazyk `C`, které je popsáno v hlavičkovém souboru `ruby.h`.

Celý propojovací mechanismus je založen na registraci ukazatelů, kteří ukazují na funkce a metody, které chceme navzájem propojit. Důležitým faktem pro pochopení následujících odstavců je názvosloví, kdy v jazyce `C` jsou funkce a jazyk Ruby používá `metody`. Pro názornost je uveden obecný případ propojovacího mechanismu implementovaný v jazyce `C`.

```
1  #include "ruby.h"
2  #include "my-api.h"
3
4  Init_Api() {
5   VALUE singletonModule;
6   rb_require("singleton");
7   singletonModule = rb_const_get(rb_cObject, rb_intern("Singleton"
        ));
8
9   netlinkApi = rb_define_class("NetlinkApi", rb_cObject);
10  rb_include_module(netlinkApi, singletonModule);
11  rb_funcall(singletonModule, rb_intern("included"), 1, netlinkApi
        );
12  rb_define_method(netlinkApi, "initialize", method_init, 0);
13  rb_define_method(netlinkApi, "registerCallback",
        method_registerCallback, 2);
14 }
15
16 static VALUE method_registerCallback(VALUE self, VALUE
      callbackTarget, VALUE callbackFunc) {
17  rb_iv_set(self, "@CallbackTarget", callbackTarget);
18  rb_iv_set(self, "@CallbackFunction", callbackFunc);
19 return 0;
```

```
20 }
21
22 static VALUE method_init(VALUE self){
23  rb_iv_set(self, "@CallbackTarget", Qnil);
24  rb_iv_set(self, "@CallbackFunction", Qnil);
25  return self;
26 }
27
28 static void ruby_callback(int atribut){
29  VALUE self, selfClass, instanceMethod, callbackTarget,
        callbackMethod;
30  VALUE callResult = Qnil;
31
32  selfClass = rb_const_get(rb_cObject, rb_intern("NetlinkApi"));
33  instanceMethod = rb_intern("instance");
34  self = rb_funcall(selfClass, instanceMethod, 0);
35  callbackMethod = rb_iv_get(self,"@CallbackFunction");
36  callbackTarget = rb_iv_get(self,"@CallbackTarget");
37
38  if ( callbackMethod != Qnil ) {
39   callResult = rb_funcall(callbackTarget, rb_to_id(callbackMethod
        ), 1, INT2FIX(atribut));
40  }
41 }
```

V první funkci `Init_Api` je pro získán `singleton`, což je základní instance `Ruby` modulu. Na řádku 9 je vytvořena třída `NetlinkApi`, která je na řádku 10 přidána do získaného modulu. Na řádku 12 a 13 je definována metoda konstruktoru `initialize` a `registerCallback`, která reprezentuje obecně registrovatelnou metodu jazyka `Ruby`.

Jak je vidět metoda `rb_define_method` spojí definovanou metodu konstruktoru s funkcí v jazyce `C` na řádku 22. Analogicky propojí `Ruby` metodu `registerCallback` s funkcí `method_registerCallback`.

Funkce `method_init` je tedy zavolána v okamžiku, kdy je volána metoda konstruktoru třídy `NetlinkApi` z jazyka `Ruby`. Při této inicializaci dochází k nastavení parametrů instance `@CallbackTarget` a `@CallbackFunction` na výchozí nulovou hodnotu `Qnil`. Typ návratové hodnoty této funkce je `VALUE`, což reprezentuje datový typ ukazatel používaný v `Ruby C API`.

Pokud dojde k volání funkce `ruby_callback` z jazyka `C`, pak je interně získán ukazatel na „objekt třídy `NetlinkApi`" jsou získány hodnoty parametrů `@CallbackTarget` (zaregistrovaný objekt) a `@CallbackFunction` (zaregistrovaná metoda objektu). Po kontrole, zda-li registrovaná metoda objektu existuje, je zavolána pomocí funkce `rb_funcall` mající povinné tři parametry. Prvním je registrovaný objekt, druhým je identifikátor metody registrovaného objektu, třetím je počet parametrů volané metody. Jestliže je počet kladně různý od nuly, pak jsou povinné další parametry, které se předají do volané metody formou vstupních parametrů.

Je nutné vstupní parametry metody překonvertovat pomocí příslušné funkce dle datového typu parametru. V tomto případě je použita funkce `INT2FIX`,

která převádí znaménkový celočíselný typ `integer` na datový typ, který interně používá jazyk `Ruby`.

V případě jiného datového typu v jazyce `C` je zapotřebí použít příslušnou transformační funkci. Pro ukázku je zde uveden základní seznam těchto funkcí.

- INT2NUM() pro **int**
- UINT2NUM() pro **unsigned int**
- LONG2NUM() pro **long**
- ULONG2NUM() pro **unsigned long**
- LL2NUM() pro **long long**
- ULL2NUM() pro **unsigned long long**
- DBL2NUM() pro **double**

Pro opačnou transformaci lze využít některé z následujících funkcí.

- NUM2CHR() pro **char** (funguje pro **unsigned char**)
- NUM2SHORT() pro **short**
- NUM2USHORT() pro **unsigned short**
- NUM2INT() pro **int**
- NUM2UINT() pro **unsigned int**
- NUM2LONG() pro **long**
- NUM2ULONG() pro **unsigned long**
- NUM2LL() pro **long long**
- NUM2ULL() pro **unsigned long long**
- NUM2DBL() pro **double**

Tímto mechanismem je například zavolána registrovaná metoda `connectorCallbackFunction` objektu třídy `MyClass`, jejíž definice je v následující ukázce.

```
1  require "Api"
2  class MyClass
3
4   def initialization(self)
5          NetlinkApi.instance.registerCallback(instance, :
               connectorCallbackFunction)
6   end
7
8   def connectorCallbackFunction (atribut)
9          puts #{atribut}  # Vypis atributu
10  end
11 end
```

V definici třídy `MyClass`, respektive v jejím konstruktoru `initialize` je patrná registrace metody `connectorCallbackFunction`, jejíž zavolání požadujeme při volání funkce jazyka `C ruby_callback`. V tomto ukázkovém příkladu je hodnota parametru této funkce vypsána registrovanou metodou v jazyce `Ruby`.

Mnoho dalších informací o celého mechanismu propojení lze najít ve webovém průvodci `The Definitive Guide to Ruby's C API` [32].

3.3 Apache Cassandra

3.3.1 Historie

Původní projekt `Cassandra` vznikl v roce 2008 ve společnosti Facebook, kde poskytoval uložení uživatelských zpráv. Předchůdci, ze kterých se vývojáři inspirovali byly projekty `Google BigTable` a `DynamoDB` od společnosti Amazon. Po zveřejnění zdrojových kódů se projekt přesunul pod záštitu Apache Foundation. V současné době je `Cassandra` projekt s otevřeným zdrojovým kódem zaštítěná společnostmi Acunu a Datastax. Používají ji například Twitter, eBay nebo Cisco.

3.3.2 Popis

`Cassandra` je distribuovaný `NoSQL` [33] databázový systém vhodný k ukládání a zpracování velkých objemů dat `BigData`[7]. Hlavní myšlenkou systému je jednotná databáze, která je replikovaná napříč množinou uzlů, obsahující velké množství dat.

Právě replikace veškerých dat na množinu uzlů zajišťuje vysokou dostupnost a neexistenci selhání jednoho bodu. Další výhodou je transparentní škálování, kdy je možné napsat kód pro lokální instanci, který bude fungovat v jakémkoli clusteru. `Cassandra` podporuje transakční zpracování (ACID [34]), jak jej známe z relačních databází.

[7] **BigData** - databáze s tak velkými datasety, že je nemožné nebo velice obtížné s nimi manipulovat za pomocí tradičních nástrojů jako jsou například relační databáze

3.3.3 Architektura

Celý systém z hlediska architektury je symetrický, tudíž všechny uzly jsou si navzájem rovny. Data jsou ukládána v několika kopiích napříč celým clusterem. Počet kopií je dán replikačním faktorem pro každý úložný prostoru `keyspace`. Jednotlivé uzly clusteru jsou uspořádány do kruhu. Každý z uzlů je rozdělen na několik virtuálních uzlů, kde jednotlivé virtuální uzly mají na starost určitou část kruhu. Tento fakt umožňuje rychlejší opravu po znovu připojení odpojeného uzlu nebo rychlejší připojení nového uzlu do clusteru, protože jsou data kopírována po menších částech.

Čtení dat Požadavek ke čtení dat může přijít na libovolný uzel, který se v té chvíli stává koordinátorem operace čtení. Koordinátor odešle dotaz na získání potřebných dat od všech uzlů. Po obdržení odpovědí, které nemusí být všechny aktuální, je rozhodnuto v závislosti na úrovni konzistence, která určuje na kolik odpovědí s aktuálně platnými daty bude koordinátor čekat, než je odeslána odpověď na čtecí požadavek zpět klientovi.

Zápis dat Požadavek na zápis lze poslat libovolnému uzlu, který se v okamžiku příjmu tohoto požadavku stává koordinátorem zápisové operace. Tento uzel pošle požadavek na operaci zápisu určitému počtu uzlů, který je dán konfigurací replikačního faktoru. Na těchto uzlech se data zapisují do potvrzovacích logů a zároveň do mezipaměti. Teprve po jejím zaplnění jsou data uložena na pevný disk. Uzel, který data zapíše do potvrzovacího logu a do mezipaměti pošle koordinátorovi odpověď potvrzující zápis dat. Pokud koordinátor obdrží počet odpovědí o zápisu dat, který vyhovuje podmínce dané parametrem konzistence, pošle koordinátor klientovi odpověď na požadavek zápisu o úspěšném provedení.

Konzistence Je vlastnost, která určuje, zda server vrátí pro každý požadavek správný výsledek. Správným výsledkem se rozumí aktuální a platná data. `Cassandra` nabízí několik úrovní konzistence pro čtení a zápis. Úroveň konzistence je možné měnit za běhu až na úroveň jednotlivých dotazů. Pro ukázku je zde uveden základní seznam úrovní pro operaci čtení.

- ONE - Vrátí nejrychlejší odpověď a na pozadí spustí opravné čtení
- TWO - Vrátí dvě nejrychlejší odpovědi ze dvou uzlů mající replikovaná data
- THREE - Vrátí tři nejrychlejší odpovědi ze tří uzlů mající replikovaná data
- QUORUM - Vrátí nejnovější data poté, co odpoví větší polovina uzlů mající replikovaná data

- ...
- ALL - Vrátí nejnovější data poté, co odpoví větší polovina všech uzlů clusteru mající replikovaná data

Konzistence u zápisu je složitější vzhledem k možné situaci, kdy cílový uzel není dostupný. V takovém okamžiku nastává nekonzistence a `Cassandra` obsahuje mechanismy, kterými je tento problém vyřešen. Základní úrovně konzistence pro operaci zápis jsou v následující ukázce.

- ANY - Zápis se musí provést alespoň na jeden uzel, pokud jsou všechny uzly kam se mají replikovat data nedostupné, lze provést zápis do odkladového úložiště. Tato data nebudou dostupná ke čtení dokud alespoň jeden uzel vlastnící replikovaná tato data nebude dostupný.
- ONE - Data musí být zapsána do potvrzovacího logu a paměti alespoň jednoho uzlu, kde mají být data replikována.
- TWO - Data musí být zapsána do potvrzovacího logu a paměti alespoň dvou uzlů, kde mají být data replikována.
- THREE - Data musí být zapsána do potvrzovacího logu a paměti alespoň tří uzlů, kde mají být data replikována.
- QUORUM - Data musí být zapsána do potvrzovacího logu a paměti na větší polovině uzlů mající replikovaná data
- ...
- ALL - Data musí být zapsána do potvrzovacího logu a paměti na všech uzlech mající replikovaná data

Základní odlišnosti od relačních databází Obecně lze říci, že návrh `NoSQL` databází je od navrhování relačních databází naprosto odlišný. Existují dvě hlavní myšlenky NoSQL databází, které jsou zcela protichůdné relačnímu pojetí. Jedná se o denormalizaci a redundanci. Denormalizace znamená, že neexistuje příkaz *join* (ani žádný ekvivalent), který spojuje jednotlivé tabulky. Proto se v `Cassandře` žádná denormalizace neprovádí. Redundance dat vzniká a nijak nevadí. Důvodem je, že cena za redundantní data je mnohem menší než získávání potřebných dat v případě bez redundance.

Využití `Apache Cassandra` v projektu `Clondike` Hlavním důvodem propojení obou těchto projektů je snaha o vytvoření decentralizovaného systému důvěry v projektu `Clondike`. Základní představa je logování veškerých migračních transakcí, které v clusteru probíhají. Účelem této evidence je umožnit uživatelům spustit dotaz, jehož výsledek pomůže při rozhodování, na jaký uzel se má daný proces migrovat. V případně ověření důvěry domovského uzlu, zda-li bude hostitelský uzel imigrovaný proces akceptovat a posléze vykonávat.

3.4 Použitý software

VMware WorkStation 11.0.0 [29] Virtualizační software pro testování více virtuálních počítačů. V tomto případě pro spuštění více počítačů, které mají nainstalovaný Clondike. Umožňuje síťové propojení pomocí virtuálních síťových adaptérů, které poskytnou virtuální lokální sít. Další velmi používanou funkcí je možnost klonování virtuálních strojů (linked clone). Šetří místo na disku a umožňuje sdílet stejné knihovny a soubory virtuálních strojů.

Sublime Text 2 [35] Výkonný textový editor zvýrazňující syntaxi mnoho programovacích i skriptovacích jazyků. Vhodný například pro editaci více řádků najednou a fulltextové vyhledávání v celém projektu, kde výsledek je zobrazen z větším kontextem.

LXR - Linux Cross Referencer [36] [37] Webové stránky poskytující zdrojové kódy jádra OS Linux napříč verzemi a především poskytují pomocí hypertextových odkazů celou strukturu zdrojových kódů. Lze velmi snadno hledat identifikátory funkcí, které jsou zároveň odkazy vedoucí do zdrojových souborů, kde jsou jejich definice či odkud jsou volány. Mezi verzemi jádra lze velmi snadno přepnout, což vede k poměrně dobrému porovnávání jednotlivých implementací v různých verzích jádra.

Linux OS Debian [38] Poměrně hojně rozšířený svobodný operační systém určený jak pro servery, tak pro běžné uživatelské stanice. Clondike byl testován a upravován na verzi stable 7.8.0 (wheezy) 64bit, jen bylo použito jiné jádro.

Inkscape [39] Inkscape je profesionální, kvalitní software pro práci s vektorovou grafikou, který běží na třech známých platformách Windows, Mac OS X a Linux. Je používán k modelování a kreslení profesionály i amatéry po celém světě. Má široké využití v oblasti počítačové grafiky například pro tvorbu ilustrací, ikon, log, diagramů, map, webové grafiky a další. Inkscape nativně podporuje W3C otevřený standard SVG (Scalable Vector Graphics) a je volně šiřitelný jako software s otevřeným zdrojovým kódem.

TeXstudio [40] TeXstudio je multiplatformní integrované prostředí pro psaní a vytváření LaTeX dokumentů. Jeho cíl je jednoduše, gramaticky správně a komfortně napsat LaTeX dokument. Hlavní výhodou je dobré zvýrazňování syntaxe, kontrola gramatiky, integrovaný prohlížeč dokumentů, mnoho průvodců pro tvorbu tabulek, bibliografie, prezentací, uspořádání textu a mnoho dalšího.

Violet UML Editor [41] Violet je jednoduchý multiplatformní UML editor s mnoha benefity například je velmi intuitivní, rychlý na použití a je kompletně zdarma.

Závěr

Projekt `Clondike` se pomocí této práce zbavil mnoha chyb, které omezovaly a v některých případech znemožňovaly jeho samotnou funkcionalitu. První chyba způsobující volání funkcí `tcmi_transaction_timer`, `tcmi_transaction_put` byla úspěšně analyzována v kapitole 1.4.1, vyřešena a otestována v části 2.1.

U chyby způsobující rozpojování uzlů byla provedena důkladná analýza, která je uvedena v kapitole 1.4.2. Návrh a samotná implementace řešení byly úspěšně provedeny a otestovány. Jejich detailní popis je uveden v části 2.2.

Výsledek analýzy chyby `task tcmi_commd_xx:xxxx blocked for more than 120 seconds` ukázal, že tato chyba přímo souvisí s příčinou a důsledky prvně zmíněné chyby. Řešení a testování bylo úspěšně provedeno v rámci implementace prvně uvedené chyby.

Ověřovací měření nemohlo být provedeno z důvodů nalezení nové skryté chyby typu „**race condition**". Podrobnější popis je uveden v části 2.5, kde byla provedena i základní analýza. Řešení této chyby je nad rámec této práce, svým rozsahem a složitostí odpovídá dizertační práci, ve které bude uvedeno.

Úspěšně byl proveden návrh a samotná implementace mechanismu ukládání klíčových parametrů migrace z jádra do databáze projektu `Apache Cassandra`. Byla úspěšně implementována a otestována základní verze ovladače pro přístup k databázi. Této problematice je věnována kapitola 2.6.

Bylo provedeno rozsáhlé studium vnitřních mechanismů a algoritmů, za pomocí kterého by nebylo možno příčinu těchto chyb zjistit a následně navrhnout opravné řešení. Podstatná část tohoto studia byla zdokumentována v poslední kapitole 3.

Během celé práce se vyskytly mnohdy neřešitelně vypadající problémy, ale s velkou dávkou trpělivosti a detailního studia principů, které se týkaly především oblasti vytváření a spouštění procesů, se podařilo splnit většinu vytyčených cílů této práce. Bylo třeba použít metody reverzního inženýrství, které jsou časově náročné. Obecně kompilace jádra operačního systému byla velmi podstatnou operací, která byla provedena nespočetněkrát.

Samotné ladění chyb a implementovaného řešení představovalo mnoho chybových stavů od známého chybového výpisu `kernel panic` až po uváznutí celého systému vlivem špatně navržené synchronizace pomocí mechanismů zámků. Hledání příčin chyb a s tím spojené hodiny studia zdrojových kódů nejenom projektu `Clondike`, ale i samotného jádra, bylo velkou zkušeností do světa vývoje jádra operačního systému.

Výsledkem této práce je více zdokumentovaný projekt s menším počtem dosud objevených chyb a se zjištěním zcela nové chyby, která byla poněkud skrytá, ale její význam a důsledek byl popsán pro budoucí vývoj projektu.

Budoucí směr vývoje projektu Clondike

Tato část obsahuje itinerář nejdůležitějších úkolů budoucího vývoje projektu seřazené podle priority.

Chyba typu „race condition” Nejprioritnější je detailnější analýza příčiny a opravení zjištěné chyby v kapitole 2.5, která vzniká v náhodných časech a při různém volání. Rozsahem a složitostí problému nalezení řešení spadá spíše do tématu dizertační práce.

Simple Ruby Director Jedna z podstatných úprav je v uživatelském prostoru tato komponenta, jejíž implementace není zrovna optimální v mnoha ohledech. Konsolidace konfiguračních souborů do jediného je rozhodně potřebné řešení spolu s eliminací řídících `Bash` skriptů. Funkcionalitu těchto skriptů lze přepsat z velké části do jazyka `Ruby`.

Propojení Cassandra Použít řešení vytvořené v této práci pro samotné propojení s distribuovanou databází projektu `Apache Cassandra`

Cache Clondike File System (CCFS) Analýza a oprava chyb této komponenty pro efektivnější práci se sdíleným souborovým systémem při mechanismu migrace.

Portace na ARM Velice zajímavou myšlenkou je budoucí podpora další architektury například ARM, respektive platformy Android OS.

Literatura

[1] Budilovsky, E.: *Kernel Based Mechanisms for High Performance I/O*. Diplomová práce, Tel Aviv University, Tel Aviv, 2013. Dostupné z: `http://www.tau.ac.il/~stoledo/BudilovskyMScThesis.pdf`

[2] Netlink Protocol Library Suite. 2015. Dostupné z: `http://www.infradead.org/~tgr/libnl/`

[3] Parallel Computing Group. 2011. Dostupné z: `https://sites.google.com/a/fit.cvut.cz/pcg/structure/clondike`

[4] Apache Cassandra. 2015. Dostupné z: `http://cassandra.apache.org/`

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

[6] Salát, M.: *Rešerše kešovacích mechanizmů v systému Clondike*. Diplomová práce, České vysoké učení technické v Praze, Fakulta informačních technologií, Praha, 2014.

[7] Šťáva, M.: *Overlapping Non-dedicated Clusters Architecture*. Disertační práce, České Vysoké Učení Technické v Praze, 2012.

[8] Vahalia, U.: *UNIX internals*. Prentice-Hall, vyd. 1. vydání, 1996.

[9] Stallings, W.: *Operating systems*. Pearson Prentice Hall, páté vydání, c2005.

[10] The UNIX system today. 2000. Dostupné z: `http://www.unix.org/`

[11] The Linux Kernel Archives. 2013. Dostupné z: `https://www.kernel.org`

[12] Ruby Programming Language. 2001. Dostupné z: http://www.ruby-lang.org

[13] Fulton, H. E.: *Ruby*. Brno: Zoner Press, vyd. 1. vydání, 2009, ISBN 978-80-7413-018-2.

[14] Herout, P.: *Učebnice jazyka C*. České Budějovice: Kopp, čtvrté vydání, 2008, ISBN 978-80-7232-367-8.

[15] Herout, P.: *Učebnice jazyka C*. České Budějovice: Kopp, 6 vydání, 2009, ISBN 978-80-7232-383-8.

[16] Tanenbaum, A. S.: *Computer networks*. Prentice-Hall, čtvrté vydání, c2003.

[17] *RFC 791 Internet Protocol*. 1981. Dostupné z: http://tools.ietf.org/html/rfc791

[18] Graf, T.: Netlink Library (libnl). 2011. Dostupné z: http://www.infradead.org/~tgr/libnl/doc/core.html

[19] Corbet, J.: *Linux device drivers*. Sebastopol: O'Reilly, třetí vydání, 2005.

[20] Techopedia. 2015. Dostupné z: http://www.techopedia.com/definition/26897/linux-console-terminal

[21] RFC 4648. 2006. Dostupné z: https://www.ietf.org/rfc/rfc4648

[22] RFC 1422. 1993. Dostupné z: https://www.ietf.org/rfc/rfc1422

[23] Linux Programmer's Manual. 2015. Dostupné z: http://man7.org/linux/man-pages/

[24] GNU Bash. 2014, [software]. Dostupné z: https://www.gnu.org/software/bash/

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

[26] API documentation for Ruby 1.9.3. 2015. Dostupné z: http://ruby-doc.org/core-1.9.3/

[27] Bovet, D. P.; Cesati, M.: *Understanding the Linux Kernel*. O Reilly, třetí vydání, 2005. Dostupné z: http://www.makelinux.net/books/ulk3/

[28] Projekt Clondike studentů FIT a FEL CVUT v Praze. 2015. Dostupné z: https://github.com/FIT-CVUT/Clondike

[29] VMware WorkStation 11.0.0. 2015, [software]. Dostupné z: http://www.vmware.com/cz/products/workstation/

[30] Kernel Bug Tracker. 2015. Dostupné z: https://bugzilla.kernel.org/

[31] Datastax Ruby Driver for Apache Cassandra. 2015. Dostupné z: https://github.com/datastax/ruby-driver

[32] Anselm, M.: The Definitive Guide to Ruby's C API. 2015. Dostupné z: http://silverhammermba.github.io/emberb/

[33] NoSQL. 2015. Dostupné z: http://nosql-database.org/

[34] Teorie databází: Transakce. 2015. Dostupné z: http://www.manualy.net/article.php?articleID=8

[35] Sublime Text. 2012, [software]. Dostupné z: http://www.sublimetext.com/

[36] Linux Cross Referencer. 1995, [software]. Dostupné z: http://lxr.linux.no

[37] Linux Cross Reference. 2012, [software]. Dostupné z: http://lxr.free-electrons.com/

[38] Debian. 1997, [software]. Dostupné z: http://www.debian.org/

[39] Inkscape v0.91. 2015, [software]. Dostupné z: https://inkscape.org

[40] TeXstudio v2.9.4. 2015, [software]. Dostupné z: http://texstudio.sourceforge.net/

[41] Violet UML editor v2.0.0. 2015, [software]. Dostupné z: http://alexdp.free.fr/violetumleditor/page.php

Příloha **A**

Seznam použitých zkratek

`ACID` - **A**tomicity, **C**onsistency, **I**solation, **D**urability - požadavky na databázové transakce [34]

`CCN` - **C**luster **C**ore **N**ode - Domovský uzel

`PEM` - **P**rivacy-**E**nhanced **M**ail je internetový standard, který je používán v oblasti elektronické pošty a certifikátů

`PEN` - **P**rocess **E**xecution **N**ode - Hostitelský uzel

`UML` - **U**nified **M**odeling **L**anguage - grafický jazyk pro vizualizaci, specifikaci, navrhování a dokumentaci systémů.

`VFS` - **V**irtual **F**ile **S**ystem - Abstraktní vrstva definující obecný souborový systém

Příloha **B**

Obsah přiloženého CD

- `diplomova_prace` ..... Zdrojové soubory této práce
- `clondike_repository` ..... Oficiální repozitář Clondike
  - `backup` ..... Záloha souborů z původního kořene repositáře
  - `devel` ..... Adresář pro vývoj
    - `measurements` ..... Testovací soubory pro měření
    - `test` ..... Zdrojové soubory pro testy jednotlivých komponent
  - `doc` ..... Automaticky generovaná dokumentace (doxygen)
  - `etc` ..... Init skripty pro NPFS a Clondike
  - `patches` ..... Záplaty pro jednotlivá jádra
  - `root` .... Adresář pro uživatele root s konfiguračním souborem a npfs
  - `scripts` ..... Pomocné skripty
    - `devel` ..... Skripty jen pro vývoj
  - `sources` .. Zdrojové soubory kernel části Clondike a Netlink knihovna
  - `userspace` ..... Zdrojové soubory C-API a Simple Ruby Director
    - `director-api` ..... Implementace pro komunikaci přes Netlink
    - `ruby-director-api` ..... Rozhraní mezi Ruby a jazykem C
    - `simple-ruby-director` ..... Ruby skripty uživatelského direktoru
  - `INSTALL` ..... Instalační návod pro Clondike
  - `LICENSE` ..... Licence GNU
  - `README` ..... Popis struktury celého repozitáře