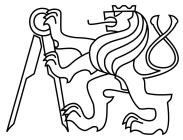
ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ



ZADÁNÍ DIPLOMOVÉ PRÁCE

Název:	Simulátor jádra operačního systému
Student:	Bc. Jan Friedl
Vedoucí:	Ing. Josef Gattermayer
Studijní program:	Informatika
Studijní obor:	Počítačové systémy a sítě
Katedra:	Katedra počítačových systémů
Platnost zadání:	Do konce letního semestru 2016/17

Pokyny pro vypracování

Cílem práce je vytvořit simulátor rozšíření jádra operačního systému Clondike pro komunikaci s knihovnami Clondike v uživatelském prostoru.

- Nastudujte komunikaci jednotlivých komponent systému Clondike.
- Navrhněte, na kterou komponentu systému provést napojení simulátoru tak, aby mohl tento upravený systém běžet bez změny jádra operačního systému. Migrace procesů bude pouze simulována.
- Implementujte do simulátoru možnost změny chování pomocí parametrů. Rychlost výpočtu, chybovost a další parametry po dohodě s vedoucím práce.
- Celé řešení zdokumentujte.
- Proveďte měření na platformě IBM Bluemix, kterým verifikujete správnost implementace.

Seznam odborné literatury

Dodá vedoucí práce.

L.S.

prof. Ing. Róbert Lórencz, CSc.
vedoucí katedry

prof. Ing. Pavel Tvrdík, CSc.
děkan

V Praze dne 11. února 2016

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

Diplomová práce

Simulátor jádra operačního systému

Bc. Jan Friedl

Vedoucí práce: Ing. Josef Gattermayer

10. května 2016

Poděkování

Rád bych poděkoval Ing. Josefu Gattermayerovi za vedení této práce a za vedení celého projektu Clondike. Obrovské poděkování patří také rodičům Bc. Věře Friedlové a Ing. Janu Friedlovi za jejich podporu při studiu, při psaní této práce i v běžném životě. Dále bych chtěl poděkovat Bc. Matěji Kuntemu, Bc. Tomášovi Němcovi a Bc. Tomášovi Vicherovi za konzultace a BS. Johanně Katchen PhD. za anglické konzultace k této práci.

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 10. května 2016 .....

České vysoké učení technické v Praze
Fakulta informačních technologií

Odkaz na tuto práci

Friedl, Jan. *Simulátor jádra operačního systému.* Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2016.

Abstrakt

Diplomová práce se zabývá úpravou záplaty Linuxového jádra Clondike verze 3.18.21 a návrhem a implementací simulátoru jádra Clondike. Díky simulátoru bude možné snadněji provádět vývoj uživatelského prostoru systému Clondike a nespoléhat se na dosud ne zcela funkční a problematickou část prostoru jádra. Výsledkem práce je funkční simulátor jádra Clondike, jehož vlastnosti byly prověřeny měřením v závěrečné kapitole.

Klíčová slova Clondike, Peer-to-peer, Nededikovaný cluster, Linux, Jádro, Simulátor, Cassandra, Docker

Abstract

This Diploma thesis deals with the adjustment of Linux kernel 3.18.21 of the project Clondike. It also deals with the design and implementation of Clondike kernel simulator. Thanks to the simulator it is possible to make the development of the user space part of the Clondike system easier and not to rely on the problematic kernel space part. The result of the thesis is a fully functional program. The properties of the simulator were checked in the measurements, which were essentials of the final chapter.

Keywords Clondike, Peer-to-peer, Non-dedicated cluster, Linux, Kernel, Simulator, Cassandra, Docker

Obsah

Seznam obrázků

Seznam tabulek

Úvod

Tato diplomová práce se zabývá projektem Clondike.

V úvodu práce je představen projekt Clondike a je zhodnocen jeho aktuální stav po implementaci diplomové práce Ing. Zdeňka Nového [1].

Druhá kapitola se věnuje úpravě záplaty linuxového jádra verze 3.18.21 pro projekt Clondike. Během této úpravy bylo nalezeno velké množství chyb, jejichž opravě se tato kapitola věnuje.

Následující kapitola se zabývá návrhem a vytvořením simulátoru jádra Clondike. V úvodu této kapitoly jsou popsány komponenty, které bude nutné simulovat. V další části kapitoly je uveden návrh komunikačního protokolu pro komunikaci mezi uzly, návrh komunikace mezi simulátorem a uživatelským prostorem Clondike a návrh uživatelského rozhraní pro komunikaci mezi uživatelem a simulátorem. Třetí část kapitoly se zabývá implementací simulátoru jádra Clondike podle předchozího návrhu. V dalších dvou podkapitolách jsou popsány úpravy komponenty Simple Ruby Director a knihovny Director API běžících v uživatelském prostoru operačního systému. Poslední část kapitoly se věnuje popisu vytvoření Docker kontejneru se simulátorem jádra Clondike.

Poslední kapitola se zabývá měřením charakteristik simulátoru jádra Clondike. V první části je popsán návrh měření a popis měřícího prostředí. V další podkapitole je provedeno měření a zpracování výsledků z těchto měření. Ve třetí části této kapitoly jsou vyhodnoceny výsledky těchto měření.

Kapitola 1

Analýza a stav projektu

1.1 Popis projektu Clondike

Clondike je cluster nededikovaných počítačů s upraveným linuxovým operačním systémem. Název vychází z anglické zkratky slov „**CL**uster **O**f **N**on-**D**edicated **I**nteroperating **KE**rnels“, což můžeme volně přeložit jako cluster nededikovaných kooperujících jader.

Clondike využívá bežné počítače s upraveným operačním systémem Linux, aby umožnil ostatním uživatelům v clusteru zvýšit výpočetní výkon svých strojů. Unikátnost tohoto systému je v tom, že využívá pouze aktuálně nezatížené uzly v clusteru, takže neobtěžuje jiné uživatele aktivně využívající jiné uzly.

Systém je navržen tak, aby jednotlivé uzly clusteru byly nezávislé a v clusteru neexistovala žádná hierarchie. K tomu je využita architektura peer-to-peer [2], kdy je každý uzel připojen přímo ke všem ostatním uzlům clusteru (nebo podmnožině uzlů). Díky této nezávislosti je cluster odolný vůči výpadkům kteréhokoliv uzlu, zároveň je každému uzlu umožněno nejen využívat výpočetní sílu clusteru, ale také svoji výpočetní sílu sdílet.

1.2 Definice základních pojmů

V této podkapitole jsou uvedeny definice pojmů, které budou použity v této diplomové práci. U některých anglických pojmů není uveden jejich překlad do češtiny, protože pro ně český ekvivalent neexistuje, je nepřesný nebo se nepoužívá.

Cluster Cluster je množina počítačů propojená mezi sebou počítačovou sítí, které mezi sebou spolupracují. Navenek se mohou tvářit jako jeden počítač. Obvykle se počítače seskupují do clusteru za účelem zvýšení výpočení síly.

Cluster nededikovaných počítačů Jedná se o typ clusteru, kde nejsou jednotlivé uzly plně vlastněny clusterem a jejich uživatelé/administrátoři nad nimi mají stále plnou moci. Přesto mohou být součástí clusteru a sdílet svůj výpočetní výkon [3].

Uzel Počítač, který je připojený do clusteru a sdílí s ním svůj výpočetní výkon.

Linux Linux je operační systému, který vyvinul Linus Torvalds s pomocí dalších programátorů. Původně byl vyvinut pro platformu $x86$, později byla doplněna podpora pro mnoho dalších architektur [4].

Kernel Jádro operačního systému. Stará se o přidělování procesorového času procesům, správu operační paměti, přístup ke vstupně/výstupním zařízením a ke správě oprávnění.

Peer-to-peer Jedná se o propojovací architekturu, kde je každý uzel připojený přímo k jinému uzlu nebo jiným uzlům. Díky tomuto propojení se stává systém odolný vůči výpadkům jednoho nebo více jiných uzlů [3].

Proces Proces je instance programu, který běží v počítači [5].

Migrace procesu Technika přesunutí procesu na jiný uzel v clusteru, kde bude migrovaný proces vykonán. Clondike rozlišuje dva typy migrace – preemptivní a nepreemptivní.

Preemptivní migrace procesu Typ migrace procesu, kdy je o samotné migraci rozhodnuto v jakémkoliv čase jeho běhu. Jedná se o velice silný a flexibilní mechanismus, který nám umožní přesunout již běžící proces na jiný uzel v případě, že se uživatel aktuálně nevyužívaného uzlu rozhodne opět začít využívat svoji pracovní stanici a již bude potřebovat její výkon.

Nepreemptivní migrace procesu Typ migrace procesu, kdy je o samotné migraci na jiný uzel rozhodnuto na začátku běhu procesu. Proces pak musí na vzdáleném uzlu doběhnout a nelze ho odmigrovat jinam, tak jako je tomu u preemptivní migrace.

Broadcast Jedná se o způsob síťového vysílání, kdy zaslaný paket obdrží všechna zařízení v síti.

IP adresa Identifikátor zařízení v síti. Projekt Clondike používá pouze IPv4 adresy podle stejnojmenného protokolu.

Patch Záplata, anglicky patch, je soubor změn daného projektu, jehož aplikací na určitou verzi projektu získáme jeho aktualizaci.

Checkpoint Aktuální obraz migrovaného procesu, který je uložen v souboru a slouží k přenosu aktuálního stavu procesu na jiný uzel při migraci procesu.

Platform as a Service – PaaS Platforma jako služba. Jedná se o typ clusteru, který je zákazníkům poskytován jako platforma pro vývoj jejich aplikací prostřednictvím online nástrojů [6].

Git Distribuovaný systém správy verzí [7].

Inode Datová struktura uchovávající data o souborech a adresářích [8].

1.3 Popis jednotlivých komponent

Operační systém je z uživatelského hlediska rozdělen na dvě části – prostor jádra (Kernel space) a uživatelský prostor (User space). Projekt Clondike zasahuje do obou těchto částí a v každé z těchto částí obsahuje několik komponent.

- **Prostor jádra** – upravené linuxové jádro, do kterého je zaintegrována hlavní funkcionalita projektu Clondike. Zdrojové soubory Clondike pro prostor jádra obsahují tyto moduly:
  - **KKC** – Kernel-To-Kernel Communication library.
  - **TCMI** – Task Checkpointing and Migration Infrastructure.
  - **ProxyFS**
  - **Director**
  - **CCFS** – Cache Clondike File System
- **Uživatelský prostor**
  - **Director-api**
  - **Ruby-director-api**
  - **Simple-ruby-director**
  - **Npfs** – Network Pipe File System

1.3.1 KKC – Kernel-To-Kernel Communication library

Jedná se o komunikační knihovnu, která slouží ke komunikaci mezi jádry operačních systémů v Clondike clusteru. Knihovna je architektonicky nezávislá, je možné přidat podporu pro jiné architektury než pouze aktuálně podporovanou *x*86. Je pamatováno na použití různých síťových protokolů, aktuálně implementovaný je pouze protokol TCP [9].

1.3.2 TCMI – Task Checkpointing and Migration Infrastructure

Tato nejdůležitější komponenta systému Clondike se stará o celý migrační mechanismus. Aktivně využívá ostatních komponent jádra k řízení migrace, práci s checkpoint soubory a ke komunikaci s uživatelským prostorem nebo jinými uzly v cloudu.

1.3.3 ProxyFS

Proxy File System zpřístupňuje speciální soubory souborového systému, například terminály, roury nebo sokety, které není možné zpřístupnit pomocí běžného síťového souborového sytému. Jedná se o virtuální souborový systém, který pomocí komponenty *KKC* zpřístupňuje tyto speciální soubory pro práci při migracích procesů.

1.3.4 Director

Komponenta, která se stará o komunikaci s uživatelským prostorem pomocí protokolu Netlink [10]. Stará se o zasílání požadavků o nových procesech, požadavků na migraci, přijímá a odesílá HeartBeat pakety generované v uživatelském prostoru.

1.3.5 CCFS – Cache Clondike File System

Speciální typ souborového systému implementovaného pro projekt Clondike. Slouží k laboratornímu měření výkonnosti systému Clondike.

1.3.6 Director API

Knihovna napsaná v jazyce C, která slouží pro komunikaci s jádrem operačního systému pomocí protokolu Netlink [10].

1.3.7 Ruby Director API

Knihovna napsaná v jazyce C poskytující rozhraní pro volání funkcí z knihovny Director API pomocí jazyka Ruby. Knihovnu využívá Simple Ruby Director pro komunikaci s jadernou částí projektu Clondike.

1.3.8 Simple Ruby Director

Řídící část systému Clondike, která se stará o objevování, ověřování a připojování nových uzlů v clusteru. Obsahuje Load Balancer, který rozhoduje o tom, jestli a kam se bude jakýkoliv proces migrovat.

1.3.9 Npfs – Network Pipe File System

Server, který protokolem 9p [11] sdílí souborový systém pomocí Network File System protokolu [12].

1.4 Stav projektu před realizací

Na začátku této práce byly nastudovány práce Ing. Zdeňka Nového [1], Ing. Pavla Tvrdíka [2] a Ing. Jiřího Rákosníka [13]. Projekt se nacházel ve funkčním stavu v podobě virtuálního stroje s jádrem 3.6.11 a jádrem 3.18.21, které mělo být také funkční. Později bylo zjištěno, že jádro 3.18.21 funkční není. Po diskuzi s vedoucím práce bylo rozhodnuto začít pracovat na úpravách jádra 3.18.21 tak, aby bylo upraveno do funkčního stavu. Tyto úpravy popisuje kapitola 2.

Kapitola 2

Úprava patche pro nejnovější podporované jádro

2.1 Příprava

Od kolegů pracujících na projektu Clondike v minulosti byla obdržena kopie virtuálního serveru s nainstalovaným jádrem 3.6.11 a 3.18.21. Při otestování kompilace jádra se záplatou systému Clondike pro verzi 3.18.21 bylo zjištěno, že se jádro zkompiluje úspěšně, ale při instalaci se nainstaluje úplně jiná verze jádra. Postupným hledáním chyby bylo odhaleno, že v zavaděči operačního systému se jádro tváří jako verze 3.18.21, ve skutečnosti se ale zavádí jádro 3.6.11.

Chyba byla nalezena v souboru `/usr/src/linux-3.18.21/clondike-/src/Makefile`. Proměnná KDIR obsahovala špatnou hodnotu, která odkazovala ke zdrojovým souborům jádra `/usr/src/linux-3.6.11`.

Po opravě na správnou cestu `/usr/src/linux-3.18.21` proběhla kompilace i instalace jádra v pořádku. Tato chyba byla nejspíš způsobena zkopírováním souborů jádra 3.6.11 na začátku implementace jádra 3.18.21.

Po opravení chyby s instalací jádra systém úspěšně zavedl jádro 3.18.21 a bylo možné otestovat jeho funkcionalitu. Ihned po spuštění Simple Ruby Directoru a jeho pokusu o navázání spojení s jádrem pomocí protokolu Netlink došlo k chybě.

S vedoucím práce bylo dohodnuto začít pracovat na opravě jádra 3.18.21, kterou se zabývá tato kapitola.

2.1.1 Postup úprav

Před prvními úpravami jádra byla zkontrolována aktuálnost zdrojových souborů v oficiálním repozitáři Clondike [14] pomocí čisté instalace jádra Clondike

aplikovaného na vanilla jádro [1]. V repozitáři se zdrojové soubory pro jádro 3.18.21 nacházejí ve větvi patch_kernel_3.18.21.

Z oficiálního repozitáře Linuxu [4] bylo staženo vanilla jádro 3.18.21, na nějž byl aplikována záplata clondike_kernel_3.18.21.patch. Do adresáře `clondike/src` [2] byly nakopírovány zdrojové soubory Clondike jádra. Poté byla upravena proměnná KDIR v souboru `src/clondike/Makefile` a hodnota byla nastavena na správnou cestu tak, aby nedošlo k podobným problémům, které byly popsány v úvodu této kapitoly. Příkazem **make menuconfig** se provedla konfigurace jádra. Do kompilace byly zahrnuty tyto položky:

- Clondike -> Enable TCMI
- Clondike -> Enable TCMI CCN support
- Clondike -> Enable TCMI PEN support
- Clondike -> Enable TCMI debug messages
- Clondike -> File systems -> Network File Systems -> NFS client support
- Clondike -> File systems -> Network File Systems -> Plan 9 Resource Sharing Support (9P2000)
- Clondike -> Networking Support -> Plan 9 Resource sharing Support (9P2000)

Pomocí příkazu **make -j4** se spustila kompilace. Jádro se úspěšně zkompilovalo a příkazem **make install** také úspěšně nainstalovalo. Zavedení tohoto jádra proběhlo v pořádku, ale při pokusu o spuštění Simple Ruby Directoru došlo k chybě při navazování spojení s jádrem pomocí protokolu Netlink. Na závěr byla provedena kontrola, jestli jsou soubory jádra z repozitáře shodné se soubory jádra, které byly na obdrženém virtuálním stroji. Soubory byly shodné. Tímto bylo ověřeno, že stav repozitáře je sice aktuální, ale nefunkční a je nutné pokračovat hledáním chyb a jejich opravou.

2.2 Úpravy jádra Clondike

2.2.1 Nastavení logování

Pro účely ladění byly zapnuty ladící výpisy jak v Clondike jádře, tak v Simple Ruby Directoru. V Clondike jádře je možné zapnout několik úrovní logování. Nastavení úrovně logování se provede v souboru `clondike/src/Makefile`,

[1] Vanilla jádro je čisté linuxové jádro stažené z oficiálního repozitáře Linuxu [4], ve kterém nebyly provedenu žádné úpravy.

[2] Všechny relativní cesty v této kapitole jsou relativní vzhledem k adresáři se zdrojovými soubory jádra.

kde se v proměnné EXTRA_CFLAGS specifikuje úroveň ladění pro jednotlivé kategorie výpisů. V našem případě byla nastavena úroveň logování u všech kategorií na hodnotu 4, tj. nejpodrobnější výpisy.

```
EXTRA_CFLAGS += -DMDBG_CRIT=4 -DMDBG_ERR=4 \
                -DMDBG_WARN=4 -DMDBG_INFO=4
```

V Simple Ruby Directoru lze zapnout pouze jednu úroveň ladění. To se provede v souboru ClondikeInit.rb v adresáři `userspace/simple-ruby-director`, kde je potřeba odkomentovat řádek:

```
$log.level = Logger::DEBUG;
```

Po těchto úpravách logování, zkompilování a nainstalování Clondike jádra je k dispozici systém, který je připravený k ladění.

2.2.2 Oprava připojení k netlink family

Po startu operačního systému a spuštění Simple Ruby Directoru byla vypsána následující chyba.

```
-----------------------   BEGIN NETLINK MESSAGE ---------------
  [HEADER] 16 octets
    .nlmsg_len = 48
    .nlmsg_type = 2 <ERROR>
    .nlmsg_flags = 0 <>
    .nlmsg_seq = 1452158563
    .nlmsg_pid = 5936
  [ERRORMSG] 20 octets
    .error = -95 "Operation not supported"
  [ORIGINAL MESSAGE] 16 octets
    .nlmsg_len = 16
    .nlmsg_type = 24 <0x18>
    .nlmsg_flags = 5 <REQUEST,ACK>
    .nlmsg_seq = 1452158563
    .nlmsg_pid = 5936
------------------------  END NETLINK MESSAGE   ---------------
```

Tato chyba nastala při čtení Netlinkové zprávy a dokumentace se o ní nezmiňuje. Ze zdrojového kódu bylo zjištěno, že chyba nastala ještě dříve a je způsobena nesprávným připojením Simple Ruby Directoru ke Clondike jádru.

Zkoumáním zdrojových kódu jádra bylo zjištěno, že ve funkci `init_director_comm()` došlo k úpravám zdrojového kódu a k registraci netlink family se používají multicast skupiny. Použití multicast skupin nemůže fungovat, protože uživatelský prostor Clondike s nimi nepočítá a ani použití multicastu není potřeba. V této funkci bylo nahrazeno volání funkce `genl_register_family()` funkcí `genl_register_family_with_ops()`, jíž je jako první pa-

Obrázek 2.1: Inicializační parametry pro registraci netlink family

```
static struct genl_family director_gnl_family = {
    .id = GENL_ID_GENERATE,
    .hdrsize = 0,
    .name = "DIRECTORCHNL",
    .version = 1,
    .maxattr = DIRECTOR_ATTR_MAX
};

static struct genl_ops register_family_ops[] = {
    {
        .cmd = DIRECTOR_REGISTER_PID,
        .flags = 0,
        .policy = director_genl_policy,
        .doit = register_pid_handler,
        .dumpit = NULL,
    },
    {
        .cmd = DIRECTOR_SEND_GENERIC_USER_MESSAGE,
        .flags = 0,
        .policy = director_genl_policy,
        .doit = handle_send_generic_user_message,
        .dumpit = NULL,
    },
    ...
};
```

rametr předána struktura `genl_family` a jako druhý parametr pole struktur `genl_ops`, viz. zdrojový kód 2.1.

Bylo smazáno nesprávné použití funkce `genl_register_family_with_ops_groups()`. Dále byla smazána volání funkce `genlmsg_register_tx_ops()`, která dříve registrovala jednotlivé operace podle druhu zprávy. Nyní je třeba tyto operace zaregistrovat při registraci netlink family, protože aktuální jádro nepodporuje jejich registraci později. Výsledná funkce se velmi zjednodušila a je znázorněna v kódu 2.2. Ve funkci `destroy_director_comm()` bylo smazáno volání funkce `kfree(check_npm_ops_ref)`, protože její parametr, proměnná `check_npm_ops_ref`, byl už z jádra také odstraněn.

V souboru `director/netlink/genl_ext.c` byla odstraněna funkce `genlmsg_register_tx_ops()`, která již není potřeba. Současně byla odstraněna její deklarace z hlavičkového souboru `director/netlink/genl_ext.h`.

Po těchto úpravách modulu Director v jádře Clondike se Simple Ruby Director úspěšně připojil k jádru. Úspěšné bylo také vytvoření spojení mezi dvěma uzly v clusteru.

Obrázek 2.2: Registrace netlink family

```
int init_director_comm(void) {
    int ret;

    minfo(INFO4,"initializing director")
    ret = genl_register_family_with_ops(&director_gnl_family,
        register_family_ops);
    if (ret != 0)
        return ret;

    minfo(INFO3, "Director comm component initialized");

    return 0;
}
```

2.2.3 Oprava vytváření checkpoint souboru

Při pokusu o migraci procesu se vyskytla chyba. Jádro Clondike nebylo schopné úspěšně vytvořit checkpoint soubor, protože nastala chyba při dumpu paměti [3]. Při dumpu bylo nalezeno 18 oblastí paměti, které se úspěšně uložily do checkpoint souboru, ale správce paměti signalizoval 19 dostupných oblastí. Byl proveden experiment s migrováním stejného procesu s jádrem 3.6.11, kde bylo úspešně detekováno a exportováno 18 stránek paměti. Ze speciálního souborového systemu `/proc` bylo výpisem souboru `maps`, viz. výpis 2.3, zjištěno, že v novém linuxovém jádře přibyla oblast paměti s názvem `[vvar]` [4]. Dump této paměti se nepovedl, protože v Clondike jádře je zakázán dump procesů, které mají ve svém adresním prostoru stránky s příznakem VM_IO (vstupně-výstupní zařízení) nebo VM_SHARED (sdílená paměť). Právě stránka `[vvar]` má nastaven příznak VM_IO. Proto byl kód upraven tak, aby umožnil exportování stránky `[vvar]`, přestože má nastaven příznak VM_IO.

2.2.4 Oprava použití parametru regs

Po opravení chyby s exportem stránek paměti se při dalším pokusu o migraci procesu objevila nová chyba typu „Segmentation Fault“. Problém byl zjištěn ve funkci `do_execve_common()` v souboru `fs/exec.c`. Z této funkce byl v novém jádře odstraněn parametr `regs`, obsahující aktuální stav registrů procesoru. Tento parametr se dále předával jako parametr při volání clondike háčku

[3] Uložení obsahu paměti na disk.

[4] Speciální stránka v paměti uživatelského prostoru, která zpřístupňuje pouze pro čtení některé proměnné z prostoru jádra. [15]

Obrázek 2.3: Výpis oblastí virtuální paměti procesu (zkráceno)

```
...
7f0e35e75000-7f0e35e76000 r--p 00020000 08:01 122698
/lib/x86_64-linux-gnu/ld-2.19.so
7f0e35e76000-7f0e35e77000 rw-p 00021000 08:01 122698
/lib/x86_64-linux-gnu/ld-2.19.so
7f0e35e77000-7f0e35e78000 rw-p 00000000 00:00 0
7ffdd1ec2000-7ffdd1ee3000 rw-p 00000000 00:00 0           [stack]
7ffdd1f26000-7ffdd1f28000 r-xp 00000000 00:00 0           [vdso]
7ffdd1f28000-7ffdd1f2a000 r--p 00000000 00:00 0           [vvar]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0
[vsyscall]
```

`execve` [5]. Parametr `regs` byl nahrazen hodnotou `NULL`, což nebylo zohledněno v dalších funkcích, které neočekávaly hodnotu `NULL` tohoto parametru. Bylo zjištěno, že parametr `regs` se používá pouze v souboru `clondike/src/tcmi/ckpt/tcmi_ckptcom.c` ve funkci `tcmi_ckptcom_checkpoint()`, která vytváří checkpoint soubor pro migraci, a ve funkci `tcmi_ckptcom_restart()`, která naopak checkpoint soubor rozbaluje. V těchto funkcích bylo zakomentováno zapisování obsahu registrů do souboru, respektive jejich načítání ze souboru zpět do struktury `regs`.

Po opravení této chyby se úspěšně provedla kompilace a instalace jádra.

2.2.5 Oprava chyby Kernel Oops

Po spuštění Simple Ruby Directoru a pokusu o migraci migrovaný proces úspěšně emigroval na jiný uzel, kde došlo k závažné chybě a běh systému byl přerušen chybou „kernel oops“ [16]. Výpis této chyby je na obrázku 2.4.

Z obsahu registru `RIP` bylo zjištěno, že program spadnul ve funkci `path_init()` na instrukci $0x99$. Tato funkce se nachází v souboru `fs/namei.c`. Pomocí programu gdb bylo zjištěno, že problém vznikl na řádku 1857 v souboru `fs/namei.c`, viz. obrázek 2.5.

Z výpisu programu gdb je vidět, že problém vznikl při porovnávání jednoho znaku z řetězce `name` a ASCII znaku „/“. Další analýzou kódu bylo zjištěno, že tento problém vznikl ve funkci `tcmi_ckptcom_restart()` z důvodu nekompatibility datových typů při volání funkce `do_execve()`. Funkce `do_execve()` má v aktuálním jádře změněn datový typ parametru `filename`. V předchozím jádře byl parametr typu `const char *`, v aktuálním jádře je typu `struct filename`. Vytvoření a inicializaci proměnné typu `struct filename` a správné volání funkce `do_execve()` znázorňuje úsek kódu 2.6.

[5] Háčky v Clondike jádře umožňují jednoduchým způsobem vložit volání funkcí jaderné části Clondike z čistého linuxového jádra.

Obrázek 2.4: Výpis chyby kernel oops (zkráceno)

```
kernel: general protection fault: 0000 [#1] SMP
kernel: RIP: 0010:[<ffffffff811516d9>]  [<ffffffff811516d9>]
  path_init+0x99/0x420
kernel: RSP: 0018:ffff880078b37148  EFLAGS: 00010246
kernel: RAX: 00000000000000b4 RBX: ffff880078b37228
  RCX: ffff880078b37228
kernel: RDX: 0000000000000051 RSI: 62696c2f7273752f
  RDI: 00000000ffffff9c
...
kernel: [<ffffffff811485bc>] ? get_empty_filp+0xac/0x160
kernel: [<ffffffff81154d91>] ? path_openat+0x91/0x650
kernel: [<ffffffff810b6abf>] ? wake_up_klogd+0x2f/0x40
kernel: [<ffffffff810b7b45>] ? vprintk_emit+0x225/0x470
kernel: [<ffffffff811554be>] ? do_filp_open+0x4e/0xc0
kernel: [<ffffffff81b49924>] ? printk+0x4c/0x51
kernel: [<ffffffff8114b946>] ? do_open_exec+0x26/0xe0
kernel: [<ffffffff8114d7da>] ? do_execve_common+0x1ea/0x5b0
kernel: [<ffffffff819367fa>] ? tcmi_ckptcom_restart+0x1c2a/0x1d90
...
```

Obrázek 2.5: Nalezení chyby v programu gdb podle čísla instrukce

```
(gdb) list *(path_init+0x99)
0x2249 is in path_init (fs/namei.c:1857).
1852        }
1853
1854        nd->root.mnt = NULL;
1855
1856        nd->m_seq = read_seqbegin(&mount_lock);
1857        if (*name=='/') {
1858            if (flags & LOOKUP_RCU) {
1859                rcu_read_lock();
1860                nd->seq = set_root_rcu(nd);
1861            } else {
```

Po úspěšné kompilaci byla vyzkoušena migrace procesu na jiný uzel clusteru. Emigrace procesu proběhla v pořádku, ale nastala další chyba, která se projevila při spouštění imigrovaného procesu na vzdáleném uzlu.

2.2.6 Oprava protokolu Plan 9

Imigrovaný proces se spouští ze souborového systému původního uzlu a je připojen pomocí síťového souborového systému Plan 9. Proces je spouštěn v takzvaném „chroot režimu“, při kterém je procesu nastaven virtuální kořenový adresář, který se nachází uvnitř skutečného souborového systému počítače. V případě projektu Clondike se souborový systém původního uzlu připojuje do adresáře `/mnt/clondike/<IP_adresa>-<INDEX_CCN>-<INDEX_PEN>`.

Obrázek 2.6: Úprava předávání parametru do funkce do_execve()

```
int tcmi_ckptcom_restart(struct linux_binprm *bprm,
    struct pt_regs *regs)
{
    ...
    struct filename * execve_filename;
    execve_filename = getname_kernel(params->file_name);


    exec_result = do_execve(execve_filename,
        (const char __user * const __user *)params->args,
        (const char __user * const __user *)params->envp);
    ...
}
```

Ten samý adresář se později použije jako domovský kořenový adresář pro imigrovaný proces.

Při spuštění imigrovaného procesu na vzdáleném uzlu došlo k chybě při vyhledávání inode, který obsahuje spouštěný program. K problému došlu v důsledku toho, že v nové verzi jádra byla u souborového systému Plan 9 vypuštěna podpora symbolických odkazů, které byly v souborové cestě testovacího migrovaného procesu – spouštěn byl process cc1, jehož absolutní cesta je `/usr/lib/gcc/x86_64-linux-gnu/4.4.6/cc1` a adresář 4.4.6 je symbolický link na adresář 4.4 ve stejném adresáři.

Bylo vyzkoušeno převzetí zdrojových kódu pro protokol Plan 9 z původní verze jádra, ale vzhledem k velkému počtu změn bylo od tohoto řešení upuštěno. Proto bylo rozhodnuto o použití jiného Plan 9 serveru, který není obsažen v jádře a funguje přímo v uživatelském prostoru bez nutnosti zásahu do jádra operačního systému. Dle dokumentace Linuxu k protokolu Plan 9 [17] pouze verze protokolu 9P2000.L nativně podporuje symbolické odkazy. V této dokumentaci je uveden odkaz na webovou stránku [18] obsahující všechny známé implementace protokolu Plan 9. Pouze Diod server [19] implementuje verzi protokolu 9P2000.L, proto bylo rozhodnuto o použití této implementace.

Instalace Diod serveru je jednoduchá, stačí pouze zkompilovat zdrojové kódy. Diod Server byl poté spuštěn v instalačním adresáři následujícím příkazem:

```
./diod -e / -n
```

Parametr **-e** znamená exportovaný adresář, v našem případě kořenový adresář, a parametr **-n** vypne autentizaci uživatele při připojování k serveru.

Dále bylo nutné upravit parametry použité při připojování síťového sou-

borového systému na začátku migrace. Parametry pro připojení se definují v Simple Ruby Directoru v souboru **ClondikeInit.rb**. Původní (zakomentovaný) a nový řádek s parametry připojení vypadá takto:

```
def listen( configuration )
  ...
  #File.write("#{mounterdir}fs-mount-options",
    'aname=/,uname=root,port=5577')
  File.write("#{mounterdir}fs-mount-options",
    'aname=/,uname=root,version=9p2000.L')
  ...
end
```

Po zapracování výše zmíněných úprav byla opět otestována migrace. Proces úspěšně emigroval na jiný uzel, kde správně proběhlo jeho vytvoření a obnova z checkpointu, ale po spuštění došlo k chybě „Segmentation Fault“. Po domluvě s vedoucím bylo rozhodnuto ukončit opravu jádra Clondike, protože se po celou dobu objevovala jedna chyba za druhou a výsledek opravy byl vzhledem k časové náročnosti nejistý.

2.3 Výsledek opravy jádra

Během opravy jádra se našlo velké množství chyb, jejichž odhalení zabralo velmi mnoho času. Opravení chyb znamenalo většinou pouze malé změny v jaderném kódu, při problému s Plan 9 souborovým systémem byly provedeny změny i v Uživatelském prostoru. Zatím poslední objevená chyba zůstala neopravena a její opravou se bude zabývat jiná práce. Lze se domnívat a určitě bude nutné i ověřit, jestli není chyba způsobena špatnou prací s pamětí při jejím exportu do checkpoint souboru, respektive jejím importu zpět do systému nebo problematickou stránkou [vvar] v paměti, viz. sekce 2.2.3.

Kapitola 3

Návrh a implementace simulátoru jádra Clondike

Vytvoření simulátoru jádra Clondike je hlavní náplní této diplomové práce. Simulátor má sloužit především ke snadnějšímu vývoji komponent Clondike běžících v uživatelském prostoru operačního systému. Také bude sloužit k otestování plánovače migrací, který je součástí Simple Ruby Directoru.

Simulátor bude vytvořen, protože vývoj a aktualizace jádra Clondike je značně složitý a zdlouhavý a nezřídka v jádře nastávají různé chyby, jako je tomu například u nejnovějšího jádra 3.18.21.

V této kapitole je popsána architektura jádra Clondike, popis komunikace jádra s uživatelským prostorem a jinými uzly. Dále je proveden návrh simulátoru, který vzešel z důkladného studia fungování jádra Clondike, a poté implementace simulátoru.

3.1 Popis komponent jádra

Jak již bylo uvedeno v kapitole 1.3, jádro Clondike se skládá z těchto komponent:

- **KKC** – Kernel-To-Kernel Communication library.
- **TCMI** – Task Checkpointing and Migration Infrastructure.
- **ProxyFS**
- **Director**
- **CCFS** – Cache Clondike File System

Komponentu TCMI je možné rozdělit na několik modulů:

- **CKPT** – Modul zajišťující práci s checkpoint soubory. V checkpoint souboru jsou uloženy informace o aktuálním migrovaném procesu včetně jeho aktuálního stavu neboli kontextu. Checkpoint soubor se ukládá před migrací do domovského adresáře uživatele clondike. Název checkpoint souboru se skládá z názvu procesu, PID a hodnoty jiffies.
- **COMM** – Modul zajišťující komunikaci mezi uzly clusteru Clondike. Pro zasílání zpráv používá komponentu KKC Clondike jádra.
- **CTLFS** – Control File System. Modul implementující virtuální souborový systém, který se používá pro komunikaci mezi prostorem jádra a uživatelským prostorem. Pomocí tohoto souborového systému se řídí připojování a odpojování uzlů, řídí migrace procesů nebo se zjišťuje aktuální stav daného uzlu. CTLFS souborový systém je připojen do adresáře `/clondike` a jeho připojení je realizováno při startu systému podle definice v souboru `/etc/fstab`. Adresářová struktura CTLFS souborového systému je na obrázku 3.1.
- **LIB** – Knihovna poskytující funkce a struktury pro uložení a práci se sokety komponenty KKC, frontami, sloty.
- **Manager** – Modul, který se stará o řízení průběhu migrace mezi domovským a vzdáleným uzlem.
- **Migration** – Modul zajišťující emigraci procesů na jiné uzly nebo přijímání imigrovaných procesů od jiných uzlů. Také se stará o připojení síťových souborových systémů před začátkem migrace. V tomto modulu jsou mimo jiné implementovány háčky, které jsou integrovány v linuxovém jádře a zajišťují jeho propojení s komponentami Clondike.
- **Syscall** – Modul implementující různá systémová volání potřebné pro migraci.
- **Task** – Modul poskytující funkce pro reprezentaci procesů při migracích.

3.1.1 Komunikace v rámci uzlu clusteru Clondike

Popis nejdůležitějších komponent, které mezi sebou komunikují v rámci jednoho uzlu, je znázorněn na obrázku 3.2. Jádro Clondike komunikuje skrz modul Director komponenty TCMI pomocí protokolu Netlink s uživatelským prostorem, kde Netlink komunikaci zajišťuje komponenta Director API. Druhým způsobem, kterým Clondike jádro komunikuje s uživatelským prostorem, je virtuální souborový systém CTLFS. S tímto souborovým systémem komunikuje převážně komponenta Simple Ruby Director, ale číst, resp. zapisovat do něj může jakýkoliv proces v uživatelském prostoru, pokud má dostatečná oprávnění.

Obrázek 3.1: Adresářová struktura CTLFS souborového systému

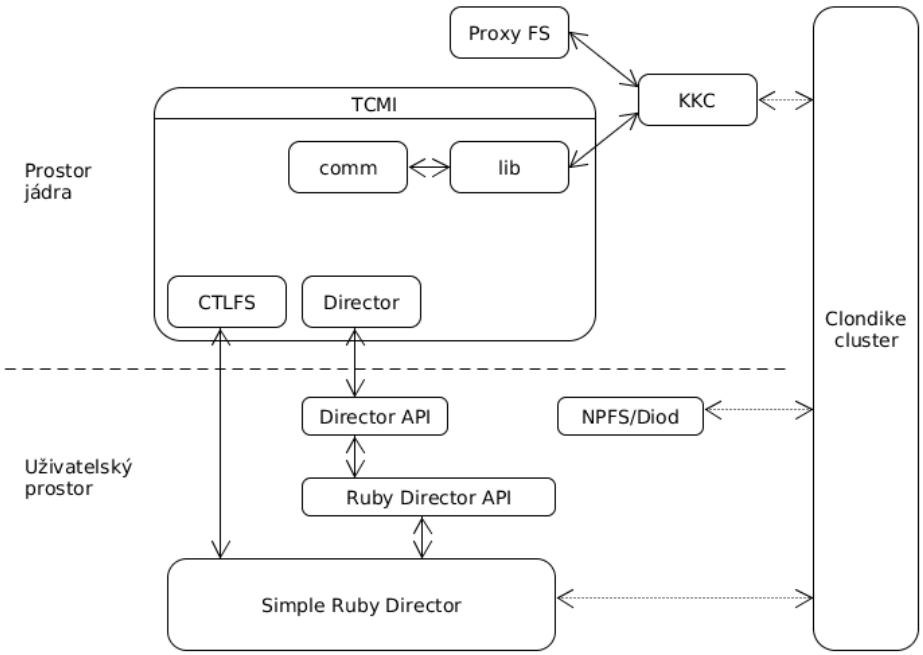
```
/clondike ..... Kořenový adresář CTLFS
├── ccn ..... Adresář pro uzly typu CCN
│   ├── listen ..... Soubor pro nastavení naslouchání
│   ├── listening-on ..... Seznam naslouchajících soketů
│   │   └── listen-00 ..... Adresář s informacemi o naslouchajícím soketu
│   │       ├── archname ..... Typ komunikačního protokolu
│   │       └── sockname ..... IP adresa a port naslouchajícího soketu
│   ├── mig ..... Adresář s informacemi o migraci
│   ├── mounter ..... Řízení připojení síťových souborových systémů
│   │   ├── fs-mount ..... Typ síťového souborového systému
│   │   ├── fs-mount-device ..... IP adresa vzdáleného uzlu
│   │   └── fs-mount-options ..... Parametry pro připojení
│   ├── nodes ..... Adresář se seznamem připojených uzlů
│   │   └── count ..... Počet připojených CCN uzlů
│   ├── stop ..... Soubor pro odpojení uzlu
│   ├── stop-listen-all ..... Soubor pro zrušení všech naslouchání
│   └── stop-listen-one ..... Soubor pro zrušení jednoho naslouchání
└── pen ..... Adresář pro uzly typu PEN
    ├── connect ..... Soubor pro řízení připojení k jinému uzlu
    ├── mig ..... Adresář s informacemi o migraci
    └── nodes ..... Adresář se seznamem připojených uzlů
        ├── 0 ..... Adresář reprezentující připojený uzel
        │   └── connections ..... Adresář se seznamem připojení k uzlu
        │       └── ctrlconn ..... Adresář s informacemi o připojení
        │           ├── archname ..... Typ komunikačního protokolu
        │           ├── peername ..... IP adresa a port vzdáleného uzlu
        │           └── sockname ..... IP adresa a port místního uzlu
        └── count ..... Počet připojených PEN uzlů
```

3.1.2 Komunikace mezi uzly v clusteru Clondike

Komunikaci mezi uzly clusteru znázorňuje obrázek 3.3. Mezi dvěma uzly probíhá komunikace třemi různými způsoby:

- **KKC** – Tato jaderná komponenta slouží ke komunikaci mezi jádry Clondike. Pomocí prokolu TCP na portu 54321 standartně poslouchá komponenta TCMI, na portu 1112 protokolu TCP poslochá ProxyFS souborový systém.
- **Plan 9** – Pomocí protokolu Plan 9 je sdílen souborový systém domovského uzlu, který si jiný uzel v případě imigrace procesu připojí. Vzdálený uzel si připojí tento souborový systém do adresáře `/mnt/clondike/<IP_adresa>-<INDEX_CCN>-<INDEX_PEN>`. Tento adresář je poté

Obrázek 3.2: Architektura uzlu v clusteru Clondike



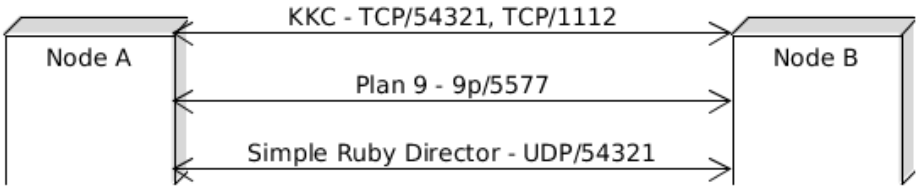
u migrovaného procesu nastaven jako kořenový adresář souborového systému, díky čemuž může migrovaný proces přistupovat k souborovému systému naprosto totožně, jako kdyby běžel na svém domovském uzlu.

- **Simple Ruby Director** – Simple Ruby Director komunikuje s ostatními uzly clusteru pomocí protokolu UDP, typicky na portu 54321. Simple Ruby Director komunikuje s ostatními uzly především za účelem nalezení dalších uzlů v clusteru. K tomutu účelu se používá technologie DHT (Distributed hash table – Distribuovaná hashovací tabulka). Touto technologií a její integrací do Clondike se zabývá diplomová práce Ing. Pavla Tvrdíka [2]. Dále se komunikace na této úrovni používá k zasílání HeartBeat zpráv a informací o aktuálním vytížení uzlu.

3.1.3 Identifikace uzlů v rámci clusteru

Každý uzel v clusteru musí být jednoznačně identifikován. Pro identifikaci uzlů v rámci clusteru používá Clondike RSA klíče a certifikáty. Každý uzel clusteru je identifikován pomocí jeho veřejného klíče a vzájemná identita uzlů je vždy při spojení ověřena pomocí certifikátů.

Obrázek 3.3: Komunikace mezi uzly v clusteru Clondike



3.1.4 Proces spojení uzlů v clusteru

Simple Ruby Director začne ihned po startu objevovat jiné uzly pomocí DHT. Každý uzel může mít v konfiguračním souboru Simple Ruby Directoru (clondike.conf) uvedenu jednu nebo více adres jiných uzlů. Obecně lze říci, že uzel bude do clusteru připojen v případě, že má v konfiguračním souboru adresu alespoň jednoho uzlu v clusteru nebo alespoň jeden jiný uzel v clusteru zná adresu tohoto uzlu.

V případě, že se Simple Ruby Director připojí k jinému uzlu v clusteru, pošle Simple Ruby Director požadavek jádru Clondike, aby se pomocí komponenty KKC připojil k druhému uzlu. Tento požadavek je předán prostřednictvím CTLFS souborového systému zápisem do souboru `/clondike/pen/connect`. Pokud druhý uzel toto připojení přijme, informuje o tom Simple Ruby Director zasláním Netlinkové zprávy. Ten samý požadavek o připojení předá Simple Ruby Director na druhém uzlu svému jádru, které se taktéž pokusí o připojení.

Pokud se uzel úspěšně připojí k jinému uzlu, zvýší se hodnota počtu připojených uzlů v CTLFS souborovém systému v souboru `/clondike/pen/nodes/count` o jedna. Pokud se k uzlu připojí jiný uzel, zvýší se o jedna hodnota v souboru `/clondike/ccn/nodes/count`.

3.1.5 Zasílání HeartBeat zpráv

Každý uzel pravidelně kontroluje, jestli jsou ostatní k němu připojené uzly „živé“. K tomuto účelu slouží HeartBeat zprávy. Každý uzel v pravidelném časovém intervalu zasílá HeartBeat zprávy všem připojeným uzlům. Tento časový interval je aktuálně nastaven na 10 sekund. Pokud některý uzel neobdrží od některého z připojených uzlů HeartBeat zprávu více než jednu minutu, je tento uzel prohlášen za mrtvý a je odpojen. HeartBeat zprávy jsou zasílány dvěma způsoby:

- Simple Ruby Director vygeneruje HeartBeat zprávu, která je protokolem UDP zaslána postupně všem připojeným uzlům.

- Simple Ruby Director vygeneruje HeartBeat zprávu, která je pomocí protokolu Netlink poslána do jádra Clondike spolu s informací, kterému uzlu má být zpráva poslána. Jádro Clondike následně prostřednictvím komponenty KKC odešle zprávu vzdálenému uzlu.

  Pokud Clondike jádro přijme prostřednictvím komponenty KKC HeartBeat zprávu, přepošle ji pomocí komponenty Director protokolem Netlink Simple Ruby Directoru spolu s informací, od kterého uzlu zprávu obdržel.

Díky zasílání HeartBeat zpráv těmito dvěma kanály je zaručena kontrola navázaných spojení jak na straně uživatelského prostoru, tak na straně jádra.

Obsahem HeartBeat zprávy je identifikátor uzlu, tedy jeho veřejný RSA klíč.

Pokud Simple Ruby Director neobdrží od nějakého uzlu HeartBeat zprávu více než jednu minutu, je provedeno jeho odpojení jak v uživatelském prostoru, tak v prostoru jádra. Simple Ruby Director provede odpojení uzlu v jaderné části tak, že zapíše do souboru `/clondike/pen/nodes/<ID_UZLU>/stop` v CTLFS souborovém systému hodnotu 1. Jádro následně odpojí daný uzel a zruší adresářovou strukturu uzlu v CTLFS souborovém systému `/clondike/pen/nodes/<ID_UZLU>`.

3.1.6 Popis nepreemptivní migrace procesu

Nepreemptivní migrace procesu je typ migrace, kdy je o samotné migraci rozhodnuto na začátku běhu procesu. O tom zda, případně kam se bude migrovat, rozhoduje Simple Ruby Director podle nastavených pravidel a na základě informací o vytížení uzlů v clusteru.

Samotná migrace začíná vytvářením procesu v jádře operačního systému:

1. Během vytváření procesu v jádře operačního systému je ve funkci `do_execve()` háček, díky kterému se předá řízení vytváření procesu komponentě TCMI.

2. Komponenta TCMI zašle Netlinkovou zprávu typu `DIRECTOR_CHECK_NPM` Simple Ruby Directoru, který zkontroluje, jestli má být provedena nepreemptivní migrace. Pokud potřebuje k rozhodnutí více informací, zašle zpět odpověď typu `DIRECTOR_NPM_RESPONSE` s informací `REQUIRE_ARGS_AND_ENVP`. Tuto informaci Simple Ruby Director vždy vyžaduje.

3. TCMI zašle Simple Ruby Directoru zprávu typu `DIRECTOR_CHECK_FULL_NPM`.

4. Simple Ruby Director rozhodne o tom, zda se bude migrovat. Pokud je nastavena proměnná prostředí EMIG na hodnotu 1, proces se bude

vždy pokoušet o migraci. Dále je zkontrolován soubor **migration.conf** obsahující seznam programů, které lze migrovat. Následně Load Balancer (vyvažovač zátěže) rozhodne o tom, na který uzel se bude program migrovat a zašle tuto informaci jádru.

5. Jádro operačního systému zašle pomocí komponenty KKC žádost o migraci procesu druhému uzlu. Tento uzel zkontroluje, jestli je migrace na něj možná, a informuje o tom původní uzel.

6. V případě pozitivní odpovědi vytvoří jádro checkpoint soubor, který uloží na disk, a proces dočasně uspí. Druhý uzel následně informuje, který soubor obsahuje popis procesu, který má uzel vykonat. V případě zamítavé odpovědi o migraci vykoná proces sám.

7. Hostující proces si připojí pomocí protokolu 9P souborý systém prvního uzlu a načte si z něj checkpoint soubor. Následně se pokusí proces vykonat. Těsne před skončením procesu vytvoří checkpoint soubor s aktuálním stavem procesu a zašle jeho název prvnímu uzlu.

8. První uzel načte checkpoint soubor do původního uspaného procesu a proces nechá standardně doběhnout. Tím je migrace hotova.

3.2 Návrh simulátoru

Dle obrázku 3.2 byly identifikovány komponenty, které bude nutné simulovat. Jedná se o tyto části jádra Clondike:

- Komponenta KKC,
- Modul Director – Netlink komunikace,
- Modul CTLFS.

Mimo komponenty jádra bude nutné ještě simulovat spouštění procesů. V Clondike jádře je každý spuštěný proces odchycen háčkem ve funkci `do_execve_common()`, jehož jméno se poté předává uživatelskému prostoru pomocí protokolu Netlink. V požadavcích práce je, aby simulátor žádným způsobem nemodifikoval jádro operačního systému, proto bude nutné emulovat i spuštění procesů tak, aby jméno spuštěného procesu zachytil simulátor.

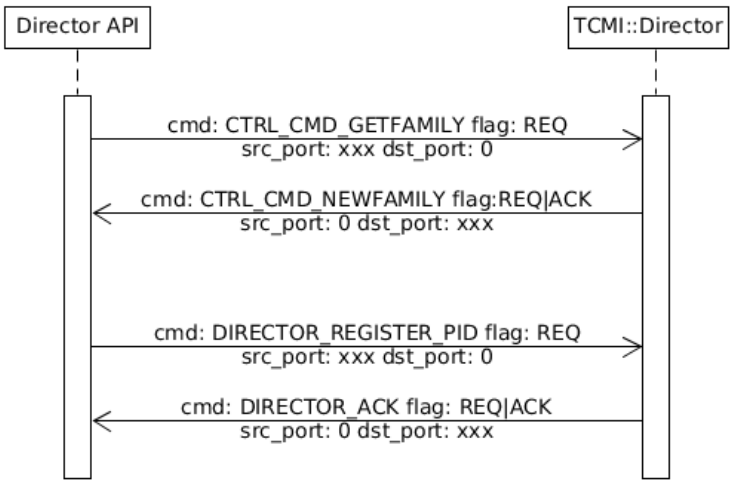
Ostatní komponenty jádra není nutné simulovat, protože jejich činnost souvisí pouze s prací samotného jádra.

3.2.1 Komunikace mezi simulátorem a uživatelským prostorem Clondike

Jádro Clondike komunikuje s uživatelským prostorem pomocí protokolu Netlink. Na straně uživatelského prostoru zprostředkovává komunikaci s uživatelským prostorem knihovna Director API, se kterou bude simulátor komunikovat.

Navázání komunikace mezi Director API a modulem Director v Clondike jádře ilustruje diagram na obrázku 3.4. Director API začíná navazování komunikace s jádrem zasláním zprávy typu `CTRL_CMD_NEWFAMILY`, kterou zašle na port 0 protokolu Netlink. Port 0 je v protokolu Netlink rezervován pro jádro operačního systému, proto zpráva dorazí na správné místo. V simulátoru bude muset být Netlink port, na kterém bude simulátor naslouchat, změněn. Z tohoto důvodu bude muset být proveden i zásah do knihovny Director API, ve které se bude muset nastavit cílový port na port, na kterém naslouchá simulátor.

Obrázek 3.4: Navázání Netlink spojení mezi jádrem a uživatelským prostorem



Komunikaci mezi Director API a jádrem operačního systému lze rozdělit na dva typy podle toho, kdo je iniciátorem komunikaci. V případě, že je iniciátorem zprávy Director API, budou zaslány celkem dvě zprávy:

1. Zpráva jádru s určitým požadavkem, příznak zprávy (flag) nastavený na `REQUEST`.

2. Odpověď jádra konkrétním typem zprávy dle požadavku s příznakem zprávy nastaveným na `REQUEST|ACK`. Jednoznačný typ odpovědi podle požadavku určuje tabulka 3.1.

Tabulka 3.1: Mapování typů odpovědí dle požadavků zpráv iniciovaných Director API.

Požadavek	Odpověď
DIRECTOR_REGISTER_PID	DIRECTOR_ACK
DIRECTOR_SEND_GENERIC_USER_MESSAGE	DIRECTOR_ACK

Tabulka 3.2: Mapování typů odpovědí dle požadavků zpráv iniciovaných jádrem.

Požadavek	Odpověď
DIRECTOR_CHECK_NPM	DIRECTOR_NPM_RESPONSE
DIRECTOR_CHECK_FULL_NPM	DIRECTOR_NPM_RESPONSE
DIRECTOR_NODE_CONNECTED	DIRECTOR_NODE_CONNECT_RESPONSE
DIRECTOR_NODE_DISCONNECTED	DIRECTOR_ACK
DIRECTOR_IMMIGRATION_REQUEST	DIRECTOR_IMMIGRATION_REQUEST_RESPONSE
DIRECTOR_IMMIGRATION_CONFIRMED	DIRECTOR_ACK
DIRECTOR_TASK_EXIT	DIRECTOR_ACK
DIRECTOR_TASK_FORK	DIRECTOR_ACK
DIRECTOR_MIGRATED_HOME	DIRECTOR_ACK
DIRECTOR_EMIGRATION_FAILED	DIRECTOR_ACK
DIRECTOR_GENERIC_USER_MESSAGE	DIRECTOR_ACK

V případě, že je iniciátorem zprávy jádro Clondike, budou protokolem Netlink zaslány celkem tři zprávy:

1. Zpráva Director API s určitým požadavkem, příznak zprávy nastavený na `REQUEST`.

2. Odpověď jádru konkrétním typem zprávy dle požadavku s příznakem zprávy nastaveným na `REQUEST|ACK`. Jednoznačný typ odpovědi podle požadavku určuje tabulka 3.2.

3. Zpráva typu `ERROR` do uživatelského prostoru knihovně Director API. Zpráva typu `ERROR` je vrácena i v případě, že nedošlo k žádné chybě. V tomto případě je chybový kód zprávy nastaven na hodnotu 0 reprezentující `SUCCESS` (úspěch).

Zjednodušeně lze říci, že poslední slovo v komunikaci při výměně zpráv má vždy jádro operačního systému.

Každá Netlink zpráva může obsahovat libovolné množství argumentů. Přehled podporovaných argumentů a jejich popis je uveden v následujícím seznamu:

- `DIRECTOR_A_PID` – identifikátor procesu,
- `DIRECTOR_A_REMOTE_PID` – identifikátor procesu na vzdáleném uzlu,
- `DIRECTOR_A_PPID` – identifikátor rodičovského procesu,
- `DIRECTOR_A_UID` – idenfikátor uživatele,
- `DIRECTOR_A_TASK_TYPE` – typ procesu (1=guest, 0=shadow),

- `DIRECTOR_A_NAME` – název migrovaného procesu,
- `DIRECTOR_A_LENGTH` – délka názvu procesu,
- `DIRECTOR_A_INDEX` – index,
- `DIRECTOR_A_SLOT_INDEX` – index slotu vzdáleného uzlu,
- `DIRECTOR_A_SLOT_TYPE` – typ slotu (1=core, 0=detached),
- `DIRECTOR_A_ADDRESS` – adresa vzdáleného uzlu,
- `DIRECTOR_A_AUTH_DATA` – autorizační data,
- `DIRECTOR_A_USER_DATA` – obecná uživatelská data,
- `DIRECTOR_A_ARGS` – argumenty procesu – obálka vnořených argumentů,
- `DIRECTOR_A_ARG` – argument procesu – vnořený argument,
- `DIRECTOR_A_ENVS` – proměnné prostředí procesu – obálka vnořených argumentů,
- `DIRECTOR_A_ENV` – proměnná prostředí procesu – vnořený argument,
- `DIRECTOR_A_REASON` – důvod,
- `DIRECTOR_A_DECISION` – typ rozhodnutí,
- `DIRECTOR_A_DECISION_VALUE` – hodnota rozhodnutí,
- `DIRECTOR_A_EXIT_CODE` – návratový kód na vzdáleném uzlu,
- `DIRECTOR_A_ERRNO` – chybový kód na vzdáleném uzlu,
- `DIRECTOR_A_RUSAGE` – využití uzlu,
- `DIRECTOR_A_JIFFIES` – hodnota jiffies.

Argumenty `DIRECTOR_A_ARGS` a `DIRECTOR_A_ENVS` slouží pro uvození speciálních typů argumentů – vnořených argumentů. Vnořené argumenty se používají například k přenosu obsahu pole. V případě Netlinkových zpráv v Clondike se používají k přenosu pole argumentů procesu a pole proměnných prostředí procesu.

V simulátoru budou implementovány všechny podporované zprávy s výjimkou `DIRECTOR_MIGRATED_HOME`. Tato zpráva slouží pouze k preemtivní migraci, která aktuálně není podporována ani Clondike jádrem, ani Simple Ruby Directorem. Z tohoto důvodu nebude implementována ani v simulátoru, protože by nenašla využití.

Dále bude změněn typ odpovědi jádra v případě, kdy je iniciátorem zprávy jádro. V tomto případě nebude posílána zpráva typu `ERROR` vždy, ale pouze v případě, že skutečně dojde k nějaké chybě. V případě úspěchu bude zasílána zpráva typu ACK. K této změně bylo přistoupeno proto, že v ladících výpisech se zpráva typu `ERROR` vyskytuje velice často a je obtížné skriptem správně filtrovat skutečné chyby od pouhého potvrzení přijetí zprávy.

3.2.2 Komunikace mezi simulátory

V Clondike jádře zajišťuje komunikaci s jiným uzlem komponenta KKC. Stejné pojmenování bude zvoleno i u simulátoru. Komponenta KKC bude ke komunikaci s jinými uzly využívat sokety. V komponentě KKC jádra Clondike je komunikace zajišťována protokolem TCP. Port, na kterém Clondike jádro, resp. simulátor, naslouchá, je nastavován pomocí virtuálního souborového systému CTLFS. Ačkoliv port může být jakýkoliv, v Clondike se typicky používá port 54321.

Komponenta KKC v simulátoru bude vytvářet pro spojení s jiným uzlem dva sokety, stejně jako v jádře Clondike. Tyto dva sokety budou reprezentovat spojení CCN -> PEN resp. PEN -> CCN. Spojení typu CCN -> PEN bude využíváno pro přijímání požadavků na migraci, zatímco spojení PEN -> CCN bude využíváno k zasílání žádostí o migraci procesů a zasílání HeartBeat zpráv. Zde zůstane zachována funkcionalita stejně jako u Clondike jádra.

3.2.3 Virtuální souborový systém CTLFS

Virtuální souborový systém CTLFS je kromě Netlinkových zpráv druhý způsob, jakým komunikuje Simple Ruby Director s jádrem Clondike.

Implementace CTLFS souborového systému v simulátoru se od implementace v Clondike jádře liší zásadním způsobem. V jádře Clondike bylo možné registrovat funkce, které se zavolaly v případě zápisu do nějakého souboru. Například pokud Simle Ruby Director objevil nějaký nový uzel a úspěšně s ním navázal komunikaci, příkazem

```
echo "tcp:172.16.168.2:54321" > /clondike/pen/connect
```

zapsal do souboru **/clondike/pen/connect** adresu uzlu, se kterým má jádro navázat spojení (PEN -> CCN). Zápis do tohoto souboru vyvolal v Clondike jádře v modulu CTLFS komponenty TCMI akci a byla spuštěna funkce, která začala navazovat připojení s jiným uzlem. V simulátoru není možné zasahovat do jádra operačního systému, proto bude CTLFS souborový systém reprezentován soubory na standartním souborovém systému. Simulátor bude v pravidelných intervalech kontrolovat soubory CTLFS souborového systému určené pro zápis a provádět příslušné akce (například připojení k jinému uzlu).

CTLFS souborý systém v simulátoru nebude podporovat některé operace. Bude například ignorovat zápis do souborů v adresáři `/clondike/ccn/mounter`, kterými se řídí připojení vzdáleného souborového systému na začátku migrace. Vzdálené souborové systémy se nebudou připojovat, proto může být tato funcionalita CTLFS ignorována.

3.2.4 Spuštění migrace – uživatelský vstup

Protože simulátor nebude pracovat v prostoru jádra, nýbrž v uživatelském prostoru, je nutné vyřešit problém, jakým způsobem informovat simulátor o novém procesu, aby ověřil možnosti jeho migrace. Bylo zvoleno řešení pomocí pojmenované roury (Named Pipe) [6]. Kdykoliv bude chtít uživatel nějaký proces virtuálně migrovat, bude stačit zapsat název programu do pojmenované roury, kterou bude simulátor v pravidelných intervalech kontrolovat.

Společně s názvem souboru potřebuje program zjistit také hodnotu jiffies. Hodnota jiffies představuje počet tiků procesoru od startu systému. Tato hodnota je v jádře operačního systému dostupná přes stejnojmennou proměnnou, v uživatelském prostoru ale dostupná není [20]. Proto bude hodnota jiffies nahrazena časem od zavedení systému (uptime), který bude získán ze souboru **/proc/uptime**. Hodnota jiffies se používá jako část identifikátoru procesu v rámci clusteru Clondike. Její nahrazení časem od startu systému s rozlišením v řádu milisekund by nemělo představovat žádný problém.

3.3 Implementace simulátoru

Simulátor je implementován v jazycích C a C++. Jazyk C++ byl použit kvůli možnosti jednoduché práci s datovými kontejnery – v simulátoru je použit Vector jako dynamické pole pro správu migrovaných procesů.

3.3.1 Jádro simulátoru

Jádro simulátoru je tvořeno dvěma správci procesů:

- emig_processes – Správce lokálních procesů a procesů, které se budou migrovat nebo jsou již v procesu migrace.
- imig_processes – Správce imigrovaných procesů.

Proces je v rámci simulátoru reprezentován strukturou `struct mig_process`. Její definice je na obrázku 3.5. Každý proces ve správci procesů `emig_processes` má své PID, které je v rámci tohoto správce unikátní. Ve správci procesů `imig_processes` se můžou objevit procesy, které mají stejné PID. Z tohoto důvodu se k identifikaci procesů ve správci `imig_processes` používá

[6] Pojmenovaná roura – fronta reprezentovaná souborem v souborovém systému.

kombinace `PID` a `peer_index`, který reprezentuje index uzlu, ze kterého proces pochází.

Stav procesu reprezentuje proměnná `migration_state`, která může nabývat následujících stavů:

- `MIG_PROCESS_PREPARED` – Nově vytvořený proces čekající na rozhodnutí, zda bude migrován nebo ne.
- `MIG_PROCESS_NEW` – Proces schválen lokálně k migraci.
- `MIG_PROCESS_REQUEST` – Proces byl odeslán na vzdálený uzel s dotazem, jestli na něj může být migrován.
- `MIG_PROCESS_CONFIRMED` – Migrace procesu byla schválena na domovském i vzdáleném uzlu. Čeká na vykonání migrace.
- `MIG_PROCESS_CONFIRMED_SEND` – Proces byl schválen k migraci na vzdáleném uzlu.
- `MIG_PROCESS_DENIED` – Migrace byla zamítnuta, proces bude vykonán lokálně.
- `MIG_PROCESS_BEGIN` – Proces emigroval na vzdálený uzel, kde bude vykonán.
- `MIG_PROCESS_END` – Proces byl úspěšně dokončen na vzdáleném uzlu a výsledek byl vrácen zpět na domovský uzel.
- `MIG_PROCESS_WORKING` – Proces je aktuálně vykonáván.
- `MIG_PROCESS_WORKING_LOCALY` – Proces je aktuálně vykonáván na domovském uzlu.
- `MIG_PROCESS_CLEAN` – Proces již byl dokončen, čeká na odstranění ze seznamu procesů.

Správci procesů nepoužívají všechny tyto stavy, některé z nich jsou určené pouze pro správce `emig_processes`, některé pouze pro správce `imig_processes`.

Procesy jsou ze správce procesů odstraňovány poté, co jim je nastaven stav `MIG_PROCESS_CLEAN`. Protože odstraňování procesů ze správce je časově složité, neprovádí se kontrola procesů na tento stav pravidelně, ale pouze tehdy, je-li některému z procesů nastaven stav `MIG_PROCESS_CLEAN`.

Obrázek 3.5: Stuktura procesu v simulátoru

```
struct mig_process {
    int pid;
    int remote_pid;
    char name[256];
    int uid;
    int peer_index;
    int migration_state;
    int return_code;
    uint64_t jiffies;
    unsigned int sequence_number;
};
```

3.3.2 Vstup procesů do simulátoru – pojmenovaná roura

Vstupním bodem simulátoru je pojmenovaná roura reprezentovaná souborem `/var/run/clondike.pipe`. Zápisem do této roury je poslán do simulátoru proces, který se má pokusit o migraci. Každý řádek vložený do roury reprezentuje jeden proces. První slovo v řádku obsahuje název procesu, který se má migrovat, další slova jsou jednotlivé parametry procesu.

Pojmenovaná roura je vytvořena simulátorem při jeho startu s unixovým oprávněním 666, jakýkoliv uživatel počítače tedy může do této roury zapisovat a pokusit se tím o migraci procesu. Roura je otevřena pomocí funkce `open()` s příznaky `O_RDWR` a `O_NONBLOCK`. Díky těmto příznakům je možné držet rouru stále otevřenou a pravidelně zjišťovat, jestli se v ní neobjevila nová data.

Kontrola dat v rouře je prováděna v pravidelných intervalech ve funkci `try_read_fifo()` funkcí `select()`. V případě, že jsou v rouře nějaká data, jsou po řádcích postupně načtena funkcí `getline()`. Poté je zavolána funkce `netlink_send_npm_check_full()`, která pošle do Simple Ruby Directoru požadavek na kontrolu migrace. Toto volání je asynchronní, na výsledek se neček á. Následně je zavolána funkce `emig_process_put()`, která vloží nově vzniklý proces do správce procesů `emig_processes`.

Při testování simulátoru bylo zjištěno, že použití pojmenované roury jako vstupního bodu simulátoru není vhodné. Během okamžiku je možné do roury zapsat tisíce požadavků na migraci, protože z pohledu uživatele, který požaduje migraci procesu, je migrace dokončena v okamžiku dokončení zápisu do pojmenované roury a ne v okamžiku doběhnutí simulovaného procesu. Z tohoto důvodu byla implementace vstupního bodu změněna.

3.3.3 Vstup procesů do simulátoru – unixový soket

Pro náhradu nevyhovující pojmenované roury pro vstup procesů do simulátoru byly vybrány unixové sokety.

Unixové sokety jsou nástroj pro meziprocesovou komunikaci v rámci operačního systému [21]. Práce s nimi je podobná práci se síťovými sokety.

Změny simulátoru Clondike pro použití unixových soketů proběhly především v souboru **fifo_reader.c**. Ve funkci `init_process_reader()` je otevřen unixový soket. Cesta k soketu je reprezentována makrem `UNIX_SOCKET_PATH`, tato cesta je nastavena do souboru `/var/run/clondike.sock`. Funkce `try_read_fifo()` byla přejmenována na funkci `try_read_processes()`. V této funkci je pomocí funkce `select()` prováděna kontrola unix soketu, jestli na něm nečekají požadavky na nová spojení. Nová spojení jsou přijata funkcí `accept()`, která vrací nově vytvořený soket, na kterém bude probíhat komunikace s klientem. Pokud je akceptováno nové připojení, je z něj funkcí `read_all()` přečtena zpráva. První 4 bajty zprávy obsahují její celkovou délku, následuje obsah zprávy. Obsahem zprávy je název procesu a seznam parametrů procesu, vše je odděleno mezerami. Po úspěšném přijetí zprávy je funkcí `netlink_send_npm_check_full()` poslán tento proces do Simple Ruby Directoru a funkcí `emig_process_put()` je vytvořen nový proces ve správci procesů k emigraci. Spolu s informacemi o procesu je uložen i soket, ze kterého byl tento požadavek přijat.

Po dokončení procesu je provedeno jeho ukončení voláním funkce `send_process_exit()`, která zašle klientovi zprávu s návratovým kódem procesu. Následně je soket uzavřen.

Klientská část pro komunikaci s unixovám soketem je jednoduchý program nazvaný clondike_client. Tento program je spouštěn s parametry obsahujícími název programu, který se má migrovat, a všechny jeho parametry. Clondike_client se po spuštění připojí na unixový soket `/var/run/clondike.sock` a pošle na něj zprávu obsahující název programu a všechny jeho parametry. Následně zavolá blokující funkci `read()`, kterou přijme od simulátoru zprávu o dokončení migrace procesu a jeho návratový kód. Po přečtení této zprávy je soket uzavřen a program clondike_client ukončen.

3.3.4 Komunikace mezi simulátory

Komunikaci mezi simulátory zajišťuje komponenta KKC. Název byl zvolen záměrně stejný, jako je používán v jádře Clondike, aby byl jednoznačně rozpoznán jeho účel.

Pro komunikaci mezi simulátory se používají sokety. Aktuálně je jak v Clondike jádře, tak v simulátoru, používán pouze protokol TCP.

Každý simulátor naslouchá na adrese a portu, kterou mu sdělí Simple Ruby Director pomocí CTLFS souborového systému. Mezi simulátorem a k němu připojeným uzlem jsou otevřena dvě spojení. Jedno spojení je vytvořeno, když se uzel připojí ke vzdálenému uzlu. Toto spojení je využíváno ke komunikaci směrem ke vzdálenému uzlu. Druhé připojení je vytvořeno, když se vzdálený uzel připojí k domovskému uzlu. Přes toto spojení čte simulátor zprávy od vzdálených uzlů.

Obrázek 3.6: Struktura KKC zpráv v simulátoru

```
struct kkc_message_header {
    uint16_t len;
    uint16_t type;
};

struct kkc_attr_header {
    uint16_t len;
    uint16_t type;
};

struct kkc_message{
    struct kkc_message_header hdr;
    char data[MAX_DATA_LEN];
};

struct kkc_attr{
    struct kkc_attr_header hdr;
    char data[MAX_DATA_LEN];
};
```

Simulátor pravidelně kontroluje všechny sokety, kterými jsou k němu připojeny vzdálené uzly a soket, na kterém přijímá nová spojení. Kontrola se provádí ve funkci `try_receive_ccn()`, která pomocí funkce `select()` zjišťuje, jestli na soketech nečekají nějaké zprávy.

Pro komunikaci byl vytvořen vlastní komunikační protokol fungující na principu zasílání zpráv. Zpráva je reprezentována strukturou `kkc_message` a obsahuje hlavičku a data. Hlavičku zprávy představuje struktura `kkc_message_header` s atributy `len` – délka zprávy a `type` – typ zprávy. Datová část zprávy obsahuje libovolný počet atributů `kkc_attr`. Atribut obsahuje dvě položky – strukturu `kkc_attr_header` obsahující hlavičku atributu a data atributu. Hlavička atributu `kkc_attr_header` obsahuje dvě položky – délka atributu `len` a typ atributu `type`, viz. obrázek 3.6.

Mezi simulátory se používají následující typy zpráv s konkrétními parametry:

- `KKC_EMIG_REQUEST` – Požadavek na migraci procesu na vzdálený uzel. Zpráva obsahuje atributy:
  - `ATTR_PID` – `PID` procesu na domovském uzlu.
  - `ATTR_UID` – `UID` uživatele na domovském uzlu.
  - `ATTR_NAME` – Název migrovaného procesu.
  - `ATTR_JIFFIES` – Hodnota jiffies.

- `KKC_EMIG_REQUEST_RESPONSE` – Odpověď na požadavek migrace procesu. Zpráva obsahuje atributy:
  - `ATTR_PID` – `PID` procesu na domovském uzlu.
  - `ATTR_DECISION` – Rozhodnutí, zda je možné na tento uzel migrovat.
- `KKC_EMIG_BEGIN` – Pokyn vzdálenému uzlu k začátku migrace. Zpráva obsahuje stejné atributy jako `KKC_EMIG_REQUEST` kromě atributu `ATTR_JIFFIES`.
- `KKC_EMIG_DONE` – Oznámení domovskému uzlu o dokončení migrovaného procesu a výsledek migrace. Zpráva obsahuje tyto atributy:
  - `ATTR_PID` – `PID` procesu na domovském uzlu.
  - `ATTR_RETURN_CODE` – Návratový kód migrovaného procesu.
- `KKC_GENERIC_USER_MESSAGE` – Zpráva obsahující data, která chce Simple Ruby Director poslat vzdálenému uzlu pomocí `KKC`. V aktuální implementaci Simple Ruby Directoru se používá pouze k zasílání HeartBeat paketů. Zpráva obsahuje tyto atributy:
  - `ATTR_SLOT_TYPE` – Je použita hodnota obdržená v Netlink zprávě.
  - `ATTR_DATA_LEN` – Délka dat.
  - `ATTR_DATA` – Data obdržená v Netlink zprávě.

3.3.5 Implementace Netlink komunikace

Pro snadnější práci s protokolem Netlink byla použita knihovna libnl (Netlink Protocol Library Suite) [10]. Knihovna libnl se skládá z těchto částí:

- libnl,
- libnl-route,
- libnl-genl,
- libnl-nf.

V této práci jsou použity pouze části libnl a libnl-genl. Libnl je obecná knihovna pro práci s Netlink protokolem, libnl-genl je API pro Generic Netlink Protocol, rozšíření obecné verze libnl knihovny.

Protokol Netlink je navržen především pro komunikaci mezi jádrem operačního systému a uživatelskými aplikacemi. Protokol také umožňuje komunikaci mezi aplikacemi v uživatelském prostoru, proto není problém pomocí něj propojit simulátor s Director API.

Netlink protokol je založen na komunikaci pomocí soketů. Pro adresování používá rodinu adres `AF_NETLINK`. Pro adresu procesu se používají porty – 32-bitová neznaménková čísla. Port 0 je typicky vyhrazen jádru operačního systému.

Na port 0 zasílá všechny své zprávy Director API při komunikaci s jádrem Clondike. Tento port nemůže být přiřazen simulátoru, proto mu byl přidělen port 111111111. V knihovně Director API byla provedena změna a místo výchozího cílového portu pro všechny zprávy byl nastaven zmíněný port 111111111. Nastavení bylo provedeno v souboru **director-api.c**, kde byl do inicializace komunikace přidán následující řádek:

```
nl_socket_set_peer_port(sk, 111111111);
```

Toto je jediná úprava uživatelských skriptů Clondike, která je nutná pro běh simulátoru a není kompatibilní s jádrem Clondike.

Netlink komunikace je v simulátoru inicializována v souboru **netlink_message.c**. Metoda `init_netlink()` v tomto souboru inicializuje Netlink komunikaci a poté čeká na připojení z uživatelských skriptů. Inicializace komunikace je zobrazena na obrázku 3.4. Změněn je cílový port zpráv zaslaných Director API, který je nastaven na port simulátoru 111111111. Z hlavičky první obdržené zprávy simulátor zjistí port, na kterém komunikuje knihovna Director API. Na tento port bude od této doby zasílat všechny zprávy. V této funkci je také zaregistrována callback funkce `netlink_callback_message()`, která je automaticky volána po přijmutí zprávy.

V tomto souboru se také nachází funkce `try_netlink_receive()`. Tato funkce obsahuje volání funkce `select()`, která zjistí, jestli na file descriptoru pro Netlink spojení nečekají nějaké zprávy. Pokud ano, tak jsou tyto zprávy přijmuty voláním funkce `receive_netlink_message()`. Funkce `receive_netlink_message()` přijme zprávu voláním funkce `nl_recvmsgs_default()` z knihovny libnl. Zpráva je zpracována zaregistrovanou callback funkcí `netlink_callback_message()`. Po zpracování příchozí zprávy je zaslána odpověď – potvrzující zpráva typu ACK.

Každá přijatá zpráva je zpracována funkcí `netlink_callback_message()`. V této funkci je na základě typu zprávy vykonána požadovaná akce. Simulátor podporuje pro komunikaci s Director API následující typy zpráv:

- `CTRL_CMD_GETFAMILY` – Inicializační zpráva od Director API. Z hlavičky této zprávy se získává port, na který se budou všechny ostatní zprávy posílat. Obsahem této zprávy je family_id, které je později obsaženo v hlavičce všech zpráv. Z hlediska simulátoru není family_id důležité, to je důležité pouze v rámci jádra operačního systému, které podle family_id určuje, kterému procesu v jádře má danou zprávu doručit.

- `DIRECTOR_REGISTER_PID` – V simulátoru slouží pouze k ověření, že komunikuje stále se stejným Simple Ruby Directorem. Jeho PID je shodný s portem, na kterém přijímá Director API zprávy. Tato zpráva je z hlediska simulátoru nedůležitá, ale je zachována kvůli kompatibilitě s Director API.

- `DIRECTOR_NPM_RESPONSE` – Tato zpráva je odpovědí na zprávu s dotazem na migraci procesu. Obsahem této zprávy jsou argumenty `DIRECTOR_A_DECISION` a `DIRECTOR_A_DECISION_VALUE`. Atribut `DIRECTOR_A_DECISION` může nabývat jedné z těchto hodnot:

  - `DO_NOT_MIGRATE` – Proces nebude migrován.
  - `MIGRATE` – Proces bude migrován.
  - `MIGRATE_BACK` – Aktuálně nepodporovaná hodnota, slouží pouze pro preemptivní migraci.
  - `REQUIRE_ARGS_AND_ENVP` – Pro rozhodnutí o migraci je nutné dodat argumenty procesu a proměnné prostředí.

  Pokud je obsahem argumentu `DIRECTOR_A_DECISION` hodnota `MIGRATE`, je v argumentu `DIRECTOR_A_DECISION_VALUE` index uzlu, na který se má proces migrovat.

  V případě, že bude proces migrován, je změněn jeho stav ve frontě emigrujících procesů na `MIG_PROCESS_NEW`. Zároveň je u tohoto procesu nastaven index uzlu, na který má emigrovat.

  Pokud je rozhodnutí o migraci záporné, je změněn stav procesu ve frontě emigrujících procesů na `MIG_PROCESS_DENIED`.

- `DIRECTOR_IMMIGRATION_REQUEST_RESPONSE` – Odpověď na zprávu s žádostí o imigraci procesu. Obsah zprávy je totožný se zprávou typu `DIRECTOR_NPM_RESPONSE`.

  V případě, že je imigrace procesu schválena, je změněn jeho stav ve frontě imigrovaných procesů na `MIG_PROCESS_CONFIRMED`.

  V případě zamítavého rozhodnutí ze strany Simple Ruby Directoru je stav procesu nastaven na `MIG_PROCESS_DENIED`.

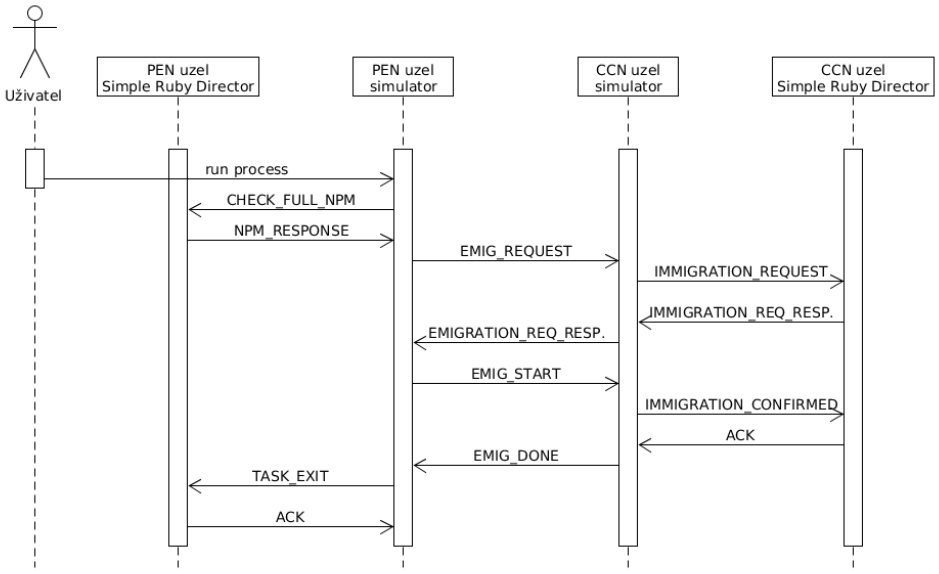
- `DIRECTOR_SEND_GENERIC_USER_MESSAGE` – Tato zpráva je žádost Simple Ruby Directoru o zaslání zprávy vzdálenému uzlu prostřednictvím komponenty KKC. Obsahem zprávy jsou data, která se mají zaslat, a index uzlu, na který se mají data poslat.

Aktuálně tato zpráva slouží pouze k zasílání HeartBeat zpráv. Obsahem zprávy je serializovaná instance [7] třídy HeartBeatRubySocketMessage, jejímž obsahem je identifikátor uzlu.

- `DIRECTOR_NODE_CONNECT_RESPONSE` – Tento typ zprávy je v systému Clondike aktuálně nevyužívaný. Je připravený pro ověřování identity připojených uzlů.
- `DIRECTOR_ACK` – Touto zprávou potvrzuje Simple Ruby Director přijetí některých zpráv. Nevyvolává žádnou akci.

Sekvenční diagram na obrázku 3.7 zobrazuje komunikaci mezi dvěma uzly v clusteru Clondike při migraci úlohy.

Obrázek 3.7: Sekvenční diagram zasílání zpráv při migraci úlohy v clusteru Clondike



3.3.6 Simulace práce migrovaného procesu

Protože simulátor nepracuje se skutečnými procesy, je nutné v něm provádění procesů simulovat.

Simulace práce procesu začíná, kdykoliv se dostane proces ve frontě procesů do stavu `MIG_PROCESS_BEGIN` na vzdáleném uzlu nebo do stavu `MIG_PROCESS_DENIED` na místním uzlu.

[7] Instance třídy, která je převedena do formy vhodné k přenosu sdíleným médiem nebo k uložení na disk. Později je možné tuto informaci deserializovat za účelem obnovení původního stavu objektu.

Simulace práce je implementována v souboru **worker.c**. Vstupním bodem simulátoru práce procesu je funkce `fork_and_work()`. V této funkci se vytvoří nové vlákno, které je reprezentováno funkcí `work()`. Po spuštění tohoto vlákna je stav procesu přepnut do stavu `MIG_PROCESS_WORKING`.

V simulátoru jsou implementovány 3 různé možnosti simulace práce:

- Uspání procesu na náhodně dlouhou dobu. Maximální doba spánku je určena makrem `MAX_WORKING_TIME`. Minimální doba spánku je 0 sekund. Tato simulace je vhodná pro vývoj simulátoru, kdy není třeba zatěžovat procesor jakýmikoliv výpočty.

- Nalezení n-tého prvočísla. Pomocí jednoduchého algoritmu je hledáno n-té prvočíslo. Počet hledaných prvočísel je určen funkcí `get_hash_name()`, viz. sekce 3.3.6.1. Tato metoda hledání prvočísla je vhodná k většímu testování simulátoru, protože zatěžuje procesor výpočtem a lze tak simulovat skutečnou práci. Nevýhodou této metody je fakt, že nelze jednoduše měřit náročnost výpočtu, protože s každým nalezeným prvočíslem roste náročnost nalezení dalšího prvočísla. Proto je tato metoda nevhodná pro měření, viz. kapitola 4.

- Počítání hašů. Tato metoda počítá haše pomocí algoritmu `MD5` [22]. Vstupem algoritmu `MD5` je název migrovaného procesu včetně jeho parametrů. Počet počítaných hašů je určen funkcí `get_hash_name()`, viz. sekce 3.3.6.1. Metoda byla použita pro účely měření v kapitole 4, protože lze podle počtu spočítaných hašů za sekundu měřit aktuální vytížení uzlu.

3.3.6.1 Určení náročnosti procesu

Funkce `get_hash_name()` slouží k určení náročnosti simulované práce. Obecně tato funkce vrací jedno číslo na základě vstupních parametrů. Vstupní paremetry jsou:

- Název migrovaného procesu včetně jeho parametrů.
- Minimální hodnota vráceného čísla – **range_min**.
- Maximální hodnota vráceného čísla – **range_max**.

Výsledek – číslo, které tato funkce vrací, je počítán podle vzorce:

$$výsledek = (acc \bmod (range\_max - range\_min)) + range\_min$$

kde hodnota *acc* je součet `ASCII` hodnot [23] všech písmen názvu migrovaného procesu vynásobený rozdílem $range\_max - range\_min$.

3.3.7 Spuštění migrace procesu

Popis začátku migrace procesu v simulátoru je popsán v sekci 3.3.2. Z pohledu uživatele se migrace spouští příkazem `clondike`, jehož parametry jsou název spouštěného procesu a parametry spouštěného procesu. Příkaz k migrace může vypadat takto:

```
clondike gcc main.c -o my_program
```

Příkaz clondike je bashový skript umístěný v adresáři `/usr/bin`. Skript po spuštění vykoná tyto operace:

1. Úprava proměnné `PATH` [8] – na první místo v hodnotě této proměnné je přidána cesta k adresáři `/usr/fake_bin`. Proměnná je následně exportována, aby byla dostupná ve všech dalších shelech spuštěných z tohoto shelu.

2. Je spuštěn program uvedený jako první parametr příkazu `clondike`. Všechny další parametry příkazu `clondike` budou parametry nově spuštěného programu.

Adresář `/usr/fake_bin` obsahuje skript **fake_bin**, který spustí program clondike_client s parametry obsahujícími název procesu a jeho parametry. Zároveň tento adresář obsahuje symbolické odkazy na skript **fake_bin**. Název symbolických linků je shodný s názvy procesů, které chceme migrovat. Pokud je požadavkem migrovat například procesy **gcc**, je v adresáři `/usr/fake_bin` symbolický odkaz **gcc** odkazující na skript **fake_bin**. Díky tomuto nastavení bude program spouštěný příkazem **clondike** vyhledáván nejdříve v tomto adresáři a pokud je nalezen, bude spuštěn skript **fake_bin**, který spustí pokus o jeho migraci.

Bude-li spuštěn výše uvedený příklad na migraci, bude nejdříve hledán program **gcc** v adresáři `/usr/fake_bin`, kde bude také nalezen. Protože program **gcc** v tomto adresáři je symbolický odkaz na skript **fake_bin**, bude tento skript spuštěn a následně se provede pokus o migraci. Pokud není spustitelný program v tomto adresáři nalezen, provádí se hledání v dalších adresářích uvedených v proměnné `PATH` a spustí se skutečné procesy.

Díky tomuto mechanismu s uvedením falešných procesů v adresáři `/usr/fake_bin` je umožněno spouštět například kompilace zdrojových souborů příkazem **make** a pokoušet se o migraci pouze některých programů, například **gcc**, zatímco potenciálně spoustěné bashové skripty se skutečně provedou.

[8] Proměnná `PATH` obsahuje názvy adresářů oddělené dvojtečkami, ve kterých jsou v operačním systému umístěny spustitelné soubory [24].

3.4 Úpravy Simple Ruby Directoru

V Simple Ruby Directoru byla příprava na integraci databáze Cassandra. Tato databáze má v projektu Clondike sloužit k ukládání distribuovaného logu transakcí. Dle tohoto logu transakcí je možné zjistit důvěryhodnost libovolného uzlu v clusteru Clondike podle provedených transakcí. Transakcí jsou myšleny tyto operace:

- emigrace úlohy na vzdálená uzel,
- nepovedená emigrace úlohy na vzdálený uzel,
- přijmutí požadavku na imigraci úlohy na vzdáleném uzlu,
- potvrzení nebo zamítnutí požadavku na imigraci na vzdáleném uzlu.

3.4.1 Databáze Cassandra

Cassandra je distribuovaná NoSQL databáze se širokými sloupci (Wide column) typu klíč – hodnota. Jedná se o distribuovanou databázi replikovanou napříč množinou uzlů, která je vhodná k ukládání a zpracování velkého množství dat. Pro uložení dat používá tabulky, jednotlivé záznamy jsou uloženy v řádcích. Na rozdíl od relačních databází mohou být řádky jedné tabulky rozdílné [1].

3.4.1.1 Vlastnosti databáze Cassandra

Neexistence bodu selhání Databáze Cassandra je odolná vůči selhání jakéhokoliv uzlu. Data jsou rozprostřena mezi množinu všech uzlů. Zároveň jsou všechna data replikována. Počet udržovaných kopií těchto dat je dán replikačním faktorem. Replikační faktor lze měnit i za běhu databáze [25].

Škálovatelnost Je velice jednoduché přidávat a odebírat uzly clusteru databází Cassandra. Z programátorského pohledu je naprosto nepodstatné, jestli je v clusteru 1 nebo 1000 uzlů, data jsou vždy posílána na jeden uzel, o jejich replikaci se stará sama databáze.

Cassandra Query Language Jedná se o dotazovací jazyk databáze Cassandra, zkráceně CQL. CQL je velmi podobný jazyku SQL (Structured Query Language).

Denormalizovaná data Obecně v NoSQL databázích neexistuje příkaz join, který spojuje více tabulek dohromady, ani žádná jeho alternativa. Proto je v těchto databázích běžná redundance dat. Nároky na získání chybějících dat jsou mnohem větší než nároky na udržování redundantních dat. Ze stejného důvodu je zároveň nevhodná normalizace dat [1].

3.4.1.2 Implementace databáze v Simple Ruby Directoru

V Simple Ruby Directoru se zdrojové kódy pro práci s databází Cassandra nachází v souboru **cassandra/CQL3Driver.rb**. V tomto souboru se nachází třída CQL3Driver, která se stará o připojení k databázi Cassandra, vytvoření tabulky `task_timeline` pro zápis transakcí v případě, že tabulka neexistuje, a vkládání záznamů do databáze. Pro práci s databází je použit ovladač `cassandra-driver`. Ovladač je nutné nainstalovat příkazem:

```
gem install cassandra-driver -v 3.0.0.rc.2
```

V konstruktoru třídy CQL3Driver byla přidána reference na objekt `configuration` třídy ConfigurationParser. Díky této změně je možné specifikovat IP adresy uzlů databáze Cassandra v konfiguračním souboru Simple Ruby Directoru **clondike.conf**. Před touto úpravou byly IP adresy uzlů databáze Cassandra napsány přímo ve zdrojovém kódu. Dále bylo v kontruktoru nahrazeno volání metody `Cassandra.connect()` za metodu `Cassandra.cluster()`.

Do třídy CQL3Driver byla přidána metoda `getCassandraNodes()`, která z objektu `configuration` získá IP adresy uzlů databáze Cassandra.

V metodě `createTableIfNotExists()` byla upravena definice tabulky `task_timeline`, která se v databázi vytváří v případě, že neexistuje. Změněn byl typ sloupce `id_task` z UUID na VARCHAR. Tato změna byla provedena, protože k identifikaci tasku se používá řetězec tří hodnota oddělených dvojtečkou – `název_úlohy:pid:jiffies`.

V metodě `createRecord()` bylo změněno volání metody `cluster.execute` na `@session.execute_async`, protože metoda `cluster.execute` v novém ovladači neexistuje.

V souboru **NetlinkConnector.rb** byla opravena chyba syntaxe po odkomentování řádků s voláním `@cql3Driver.createRecord()`.

Ve souborech **trust/Identity.rb**, **ExecutionTimePredictor.rb**, **FilesystemConnector.rb**, **Interconnection.rb**, **LoadBalancer.rb**, **PersistentIdSequence.rb** byla nahrazena zastaralá metoda `File.exists?()` metodou `File.exist?()`.

V souboru **Director.rb** byl odkomentován řádek s vytvářením instance CQL3Driveru a objektu byla předána reference na objekt `configuration`.

3.5 Úpravy Director API

V knihovně Director API byly provedeny nepatrné změny především z důvodu lepší čitelnosti kódu.

Nejdůležitější změnou byla úprava přijímání zpráv protokolem Netlink. Přijímání Netlink zpráv se provádí voláním blokující funkce `read_message()`. Tato funkce byla volána zároveň ze dvou míst v kódu:

- Z funkce `send_user_message()` po zaslání Netlink zprávy typu `DIRECTOR_SEND_GENERIC_USER_MESSAGE`.
- Z funkce `run_processing_callback()`, která v nekonečné smyčce provádí čtení a zpracování Netlink zpráv.

V obou částech programu je vždy po přijmutí zprávy volána funkce `handle_incoming_message()`, která přijatou zprávu zpracuje. Přijímání Netlink zpráv ve funkci `send_user_message()` probíhá do doby, než je přijata zpráva typu `DIRECTOR_ACK`. Protože se neprovádí žádná návazná akce po přijetí zprávy tohoto typu, je možné přijímání zpráv z funkce `send_user_message()` kompletně odstranit a ponechat tuto práci pouze ve funkci `run_processing_callback()`, která je k tomu určená.

Další úpravou bylo vytvoření handleru na zprávy typu `DIRECTOR_ACK`. Původně tyto zprávy nebyly knihovnou Director API přijímány, protože jádro potvrzovalo přijetí zprávy zprávou typu `ERROR`. Byl vytvořen soubor **ack.c** s funkcí `handle_ack()`. V této funkci je možné zprávu zpracovat. Aktuálně tato funkce nedělá nic, protože není nutné se zprávou typu `DIRECTOR_ACK` nijak pracovat.

V souboru **director-api.c** byla do funkce `initialize_director_api()` přidána registrace funkce `handle_ack()`. Ve funkci `is_ack_message()` byla upravena podmínka návratové hodnoty funkce. Nově funkce vrací nenulovou hodnotu, pokud je response_code zprávy roven hodnotě 0 nebo pokud je zpráva typu `DIRECTOR_ACK`:

```
return get_message_response_code(msg) == 0 || \
  cmd == DIRECTOR_ACK;
```

Všechny tyto úpravy v knihovně Director API byly provedeny tak, aby byla zachována kompatibilita s Clondike jádrem, které potvrzuje Netlink zprávy zprávou typu `ERROR`.

3.6 Vytvoření kontejneru se simulátorem jádra Clondike

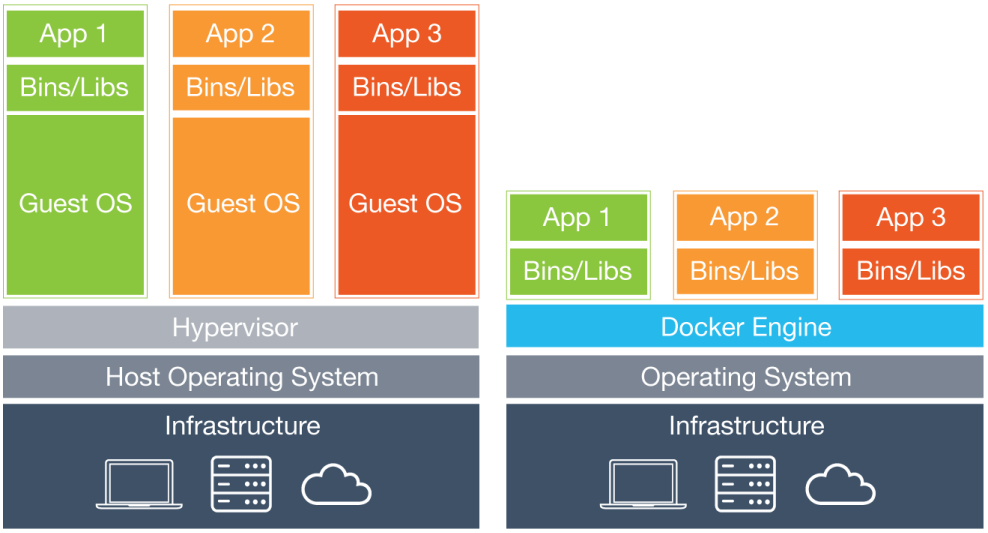
Pro účely měření (viz kapitola 4) a snadného vytvoření testovacího clusteru s velkým počtem uzlů bylo rozhodnuto o použití nástroje Docker.

3.6.1 Docker

Docker je open–source nástroj, umožňující rychlou a efektivní práci s kontejnerovou virtualizací. Virtualizace pomocí kontejnerů je hodně podobná klasické virtualizaci pomocí virtuálních strojů, má však svá specifika. Rozdíl mezi

těmito druhy virtualizace ukazuje obrázek 3.8. Kontejnery narozdíl od virtuálních strojů nevyužívají virtualizovaný hardware a nemají vlastní operační systém, namísto toho sdílí s ostatními kontejnery jádro operačního systému. Běží samostatně jako izolované procesy v uživatelském prostoru operačního systému [26].

Obrázek 3.8: Diagram architektury virtuálních strojů (vlevo) a Docker kontejnerů (vpravo)[26]



Šablona, ze které lze vytvořit Docker kontejner, se nazývá Docker obraz (Docker image). Získání Docker obrazu je možné několika způsoby:

- Pomocí předpisu v souboru Dockerfile – Dockerfile je soubor, jakási kuchařka, obsahující instrukce, pomocí kterých bude vytvořen Docker obraz budoucího kontejneru. Jedná se o velmi efektivní způsob vytvoření Docker obrazu, protože je možné předpis kdykoliv jednoduše modifikovat. Tento způsob vytvoření Docker obrazu je použit i v této diplomové práci.

- Vytvořením Docker obrazu z již běžícího kontejneru – tento způsob je možné použít, pokud máme nějaký běžící kontejner, v němž bylo uděláno mnoho změn a chceme vytvořit jeho obraz jako zálohu pro pozdější použití nebo chceme použít jeho kopii pro běh více instancí stejného kontejneru.

- Stažením již existujícího Docker obrazu z veřejného nebo soukromého repozitáře – Pokud byl nějaký Docker obraz již vytvořen, je možné jej sdílet pomocí veřejných repozitářů. Soukromé repozitáře slouží pro sdílení Docker obrazů mezi uzavřenou skupinou lidí nebo pro uchování vlastních Docker obrazů vývojáře. V repozitářích je možné najít a stáhnout

již vytvořené Docker obrazy, díky čemuž může být ušetřen čas při jejich tvorbě. Docker provozuje svůj vlastní veřejný i soukromý repozitář. Odkaz na tento repozitář lze najít v literatuře [27].

3.6.2 Vytvoření Clondike Docker obrazu

Pro Clondike Docker obraz byl zvolen způsob vytvoření pomocí Dockerfile. Jako bázový obraz [9] byl zvolen obraz operačního systému Debian. Úkony nutné pro vytvoření Clondike Docker obrazu lze shrnout pomocí těchto kroků:

- Instalace potřebných programů pro běh simulátoru a uživatelských ruby skriptů.
- Kompilace zdrojových kódů simulátoru, Director API a Ruby Director API.
- Úprava konfiguračních souborů Simple Ruby Directoru.
- Nastavení akcí při spuštění kontejneru – spuštění simulátoru jádra a Simple Ruby Directoru.

Na základě těchto kroků byl vytvořen Dockerfile, ze kterého byl pomocí příkazu

```
docker build -t kernel_simulator .
```

vytvořen Docker obraz simulátoru. Spuštění kontejneru podle tohoto obrazu se provede příkazem:

```
docker run --privileged -it kernel_simulator 172.17.0.5 \
  172.17.0.2:54321
```

První parametr při spuštění kontejneru je IP adresa libovolného uzlu databáze Cassandra, druhý parametr je IP adresa a port libovolného uzlu v clusteru Clondike. Parametr `--privileged` spouští kontejner v privilegovaném režimu operačního systému. Tento režim je nutný, aby bylo možné používat komunikaci protokolem Netlink.

3.6.3 Spuštění databáze Cassandra

Pro běh systému Clondike je zapotřebí databáze Cassandra. Zde byly také využity výhody technologie Docker a tato databáze byla spuštěna v kontejnerech.

[9] Základní obraz, ze kterého je odvozen kontejner.

Použit byl oficiální Cassandra obraz `library/cassandra` z repozitáře Docker Hub [27]. Tento obraz lze použít i k vytvoření clusteru Cassandra. Cluster se vytvoří automaticky při startu kontejneru po specifikování IP adres již běžících kontejnerů s databází Cassandra. K tomuto účelu slouží proměnné prostředí `CASSANDRA_SEEDS` a `CASSANDRA_BROADCAST_ADDRESS`, které se uvedou v parametrech příkazu docker run při spuštění kontejneru. Adresy běžících uzlů se uvedou v parametru `CASSANDRA_SEEDS`. Pokud se jedná o první uzel clusteru, tento parametr se neuvádí. Pokud se uzly clusteru Cassandra spouští v různých sítích, je nutné použít parametr `CASSANDRA_BROADCAST_ADDRESS`, který musí obsahovat IP adresu daného serveru, pomocí které bude uzel komunikovat s jinými uzly clusteru. Výsledný příkaz na spuštění kontejneru Cassandra s uzly ve stejné síti vypadá takto:

```
docker run --privileged -d -e CASSANDRA_SEEDS='172.17.0.2' \
  library/cassandra:3.3
```

Kapitola 4

Měření

V této kapitole budou provedena měření, která prověří vlastnosti simulátoru jádra Clondike.

4.1 Návrh měření

Kompilace zdrojového kódu je netriviální úloha, která potřebuje mnoho výpočetních zdrojů. Jak rostou softwarové projekty, roste i čas potřebný k jejich sestavení a ladění na samostatném počítači se stává více a více náročné na výpočetní čas. Jednou z těchto netriviálních úloh je i kompilace jádra Linuxu. Linuxové jádro je velmi komplexní a skládá se z mnoha částí s velkými možnostmi konfigurace [3]. Z tohoto důvodu byla vybrána kompilace linuxového jádra jako úloha pro změření vlastností simulátoru jádra Clondike. Tato kompilace bude pouze simulovaná, protože simulátor jádra Clondike migrované procesy pouze simuluje.

4.1.1 Vývojová platforma

Vývoj simulátoru jádra Clondike byl prováděn na počítači s operačním systémem Debian GNU Linux a jádrem Linux verze 3.16.7. Konfigurace počítače obsahovala procesor Intel Core i7-4600U (2 jádra, 2 vlákna na jádro, 4 MB keš, 2.10 GHz), 8 GB DDR3 operační paměti, 256 GB SSD disk. S touto konfigurací bylo možné spustit nejvýše 6 kontejnerů (uzlů) Clondike a 2 kontejnery s databází Cassandra.

Experimentálně bylo zjištěno, že při kompilaci linuxového jádra pomocí simulátoru jádra Clondike dochází již při 4 spuštěných procesech kompilátoru `gcc` k naprostému vytížení procesoru. Kvůli tomuto vytížení nefungovala zcela správně virtuální síť mezi kontejnery a docházelo k zahazování paketů. Nefunkční síť poté způsobovala problémy celému vývojovému clusteru, protože došlo k nedoručování heartbeat paketů, které vedlo k násilnému odpojování uzlů clusteru a postupně k rozpojení celého clusteru.

Problém s vysokou zátěží procesoru při vývoji simulátoru vedl k implementaci nového způsobu simulování práce migrovaného procesu. Tato implementace je popsána v sekci 3.3.6. Nově bylo pro účely simulace práce použito uspání vlákna na určitou dobu místo původního počítání x-tého prvočísla. Díky tomu nedocházelo k vytížení procesoru simulovanou prací při migraci procesů.

Po této úpravě bylo možné spustit na vývojové platformě cluster s 15 kontejnery simulátoru jádra Clondike a 2 kontejnery s databází Cassandra. Při tomto množství uzlů byl procesor vývojového počítače plně vytížen především při navazování zabezpečeného připojení mezi uzly na úrovni Simple Ruby Directoru. Po propojení všech uzlů mezi sebou zátěž procesoru poklesla, ale docházelo k problémům se sítí a problémům s občasnou nedostupností některých uzlů. Protože ke stejným problémům s chvilkovou nedostupností sítě dochází na testovacím počítači i při použití virtuální sítě mezi virtuálními počítači, lze se domnívat, že jsou způsobeny na straně jádra operačního systému vývojového počítače.

Pro účely testování je dostatečné použití 4 kontejnerů se simulátorem jádra Clondike, proto tyto problémy nebyly dále řešeny.

4.1.2 Měřící platforma

Vedoucím práce bylo navrženo provést měření v cloudové platformě IBM Bluemix.

4.1.2.1 IBM Bluemix

IBM Bluemix je cloudová platforma (PaaS – Platform as a Service) vyvinutá firmou IBM. Podporuje mnoho programovacích jazyků (Java, Node.js, Go, PHP, Python, Ruby) [28] a vývojářské nástroje pro sestavení, běh, nasazování a správu aplikací. IBM Bluemix je založen na technologii Cloud Foundry [29] a běží na technologii od firmy SoftLayer [30].

IBM Bluemix je placená služba, poskytuje ale zdarma zkušební 30-denní licenci na vyzkoušení všech služeb. Jediné omezení je dostupnost pouze 2 GB operační paměti. Toto omezení nepředstavuje problém, pokud nejsou spouštěny kontejnery s databází Cassandra. Proto bylo rozhodnuto o použití hostované databáze Cassandra u společnosti Instacluster [31]. Tato společnost poskytuje zdarma zkušební 30-denní přístup k databázi Cassandra.

Zkombinováním zkušebních verzí IBM Bluemix a Instacluster Cassandra lze vytvořit profesionální prostředí pro práci s kontejnery a databází Cassandra.

Po studiu dokumentace IBM Bluemix vyšlo najevo, že tato platforma nepodporuje spuštění kontejnerů v privilegovaném režimu a tudíž není možné tuto platformu použít pro spuštění kontejnerů simulátoru jádra Clondike.

Po diskuzi s vedoucím práce bylo rozhodnuto využít pro měření služeb Amazon Web Services, které umožňují spuštění kontejnerů v privilegovaném režimu [32].

4.1.2.2 Amazon Web Services

Amazon Web Services je kolekce cloudových služeb. Obsahuje mnoho na sobě nezávislých služeb, které je možné jednoduchým způsobem objednávat online. Popis některých služeb obsahuje následující seznam:

- **EC2 – Elastic Computer Cloud** – Služba virtuálních strojů. Pomocí této služby je možné vytvořit libovolný virtuální stroj různé konfigurace. V nabídce jsou malé servery s nízkým výkonem a velkou kapacitou disku až po velké servery s vysokým výpočetním výkonem. Pomocí této služby je možné pracovat s Docker kontejnery.
- **CloudFront** – tato služba poskytuje službu CDN – Content Delivery Network. Jedná se o službu, která zajišťuje dostupnost dat tak, aby byly dostupné vždy a všude. Využívá k tomu geograficky oddělená datacentra po celém světě.
- **S3 – Simple Storage Service** – Tato služba poskytuje úložný prostor pro uchovávání dat v cloudu. Virtuální servery zde mají uloženy obrazy svých disků, CDN zde má uložena data, která poskytuje.
- **SimpleDB** – Tato služba poskytuje jednoduchou NoSQL databázi na principu klíč – hodnota. SimpleDB umožňuje uložit data, najít data a zase je přečíst a to na základě klíče.
- **RDS – Relation Database Service** – Tato služba poskytuje škálovatelné relační databázové servery s nabídkou databází MySQL, Oracle, SQL Server, PostgreSQL.
- **SES – Simple Email Service** – Služba, která se stará o posílání emailů. Umožňuje vytvářet statistiky úspěšnosti doručování emailů.
- **Elastic Load Balancing** – Tato služba slouží k zajištění horizontálního škálování (rozdělení zátěže na více serverů). Jejím úkolem je přesměrovávat požadavky uživatelů na různé servery dle jejich zatížení.
- **Mechanical Turk** – Díky této službě je možné si najmout lidskou práci k vykonávání úloh, kde je zapotřebí lidského intelektu.

Výhodou Amazon Web Services je to, že zákazník platí pouze za to, co skutečně využil. U virtuálních serverů je to například strojový čas, přenesená data po síti, disková kapacita apod. Zákazník tedy neplatí za virtuální stroj,

pokud je vypnutý, neplatí za veřejnou adresu, pokud zrovna žádnou nemá přiřazenou.

Pro účely měření byla použita služba EC2. Pomocí této služby byl vytvořen virtuální počítač s optimalizací na výkon. Vybrána byla sestava c4.4xlarge s následující konfigurací:

- procesor Intel 2.4 GHz Xeon E5-2676 v3 – 16 virtuálních procesorových jader,
- 30 GB operační paměti,
- 8 GB síťový disk v úložišti S3 s vysokou dostupností.

Cena za hodinu provozu tohoto virtuálního počítače je $0.838 USD.

Jako operační systém byl zvolen Amazon Linux AMI (Amazon Machine Image). Jedná se o linuxový operační systém založený na distribuci Red Hat Enterprise Linux [33]. Pro potřeby měření byl na operační systém nainstalován nástroj Docker a program Git.

4.1.3 Testovací úloha

Jako testovací úloha pro měření byla zvolena kompilace linuxového jádra. Byla vybrána verze jádra 3.18.31. Pro kompilaci bylo zvoleno jádro v základní konfiguraci. Vygenerování základní konfigurace jádra se provede příkazem

```
make defconfig
```

Kompilace se spustí příkazem **make**. Přepínačem -j je možné specifikovat, kolik se má spustit paralelně běžících úloh kompilace. Pro 4 paralelně běžící úlohy bude příkaz **make** vypadat takto:

```
make -j 4
```

Čas běhu kompilace se měří příkazem `time`. Z výstupu příkazu bude k měření použita hodnota `real`.

Kompilace linuxového jádra je komplexní úloha, při které se nespouští pouze procesy kompilátoru, ale také mnoho jiných programů a skriptů. Některé skripty například v průběhu kompilace kontrolují, jestli se podařilo během kompilace úspěšně zkompilovat různé závislé soubory. Některé programy, které se mají spustit, vzniknou až v průběhu kompilace jádra tím, že jsou také zkompilovány. Tyto dočasné programy jsou zpravidla po skončení jejich běhu nebo po dokončení kompilace jádra smazány.

Aby bylo možné provést simulovanou kompilaci jádra, bylo nutné provést určité úpravy systému tak, aby se tato simulace dala provést. Největší problém představují dočasné programy zkompilované v průběhu kompilace jádra,

které při simulované kompilaci vzniknout nemůžou (simulátor jádra Clondike procesy pouze simuluje).

Všechny pomocné programy vzniklé při kompilaci jádra jsou nejpozději na konci kompilace smazány. Aby bylo zabráněno smazání těchto souborů, byl v operačním systému kontejneru, na kterém se bude spouštět simulovaná kompilace, upraven příkaz **rm**, který slouží k mazání souborů a složek. Příkaz se nachází ve složce `/bin` a byl přejmenován na příkaz **rm.old**. Byl vytvořen nový příkaz **rm**, který je reprezentován skriptem a který po zavolání pouze provede zápis všech mazaných souborů a adresářů do souboru `/tmp/rm.out`. Následně se spustí kompilace linuxového jádra příkazem **make**. Díky předchozí úpravě příkazu **rm** zůstanou po skončení kompilace všechny dočasné programy vzniklé v průběhu kompilace zachovány.

Aby bylo možné spustit simulovanou kompilaci, je nutné provést poslední úpravu. V případě opakovaného spuštění příkazu **make** by kompilace neproběhla, protože by příkaz **make** poznal, že nedošlo k žádné úpravě zdrojových souborů. Pomocí příkazu **touch** byly modifikovány časy poslední úpravy a posledního přístupu ke všem hlavičkovým souborům linuxového jádra. Tyto soubory byly nalezeny příkazem `find`:

```
find /usr/src/linux-3.18.31 -name *.h -exec touch {} \;
```

Pomocí těchto úprav byla připravena testovací úloha vhodná pro opakovatelná měření.

4.2 Provedení měření

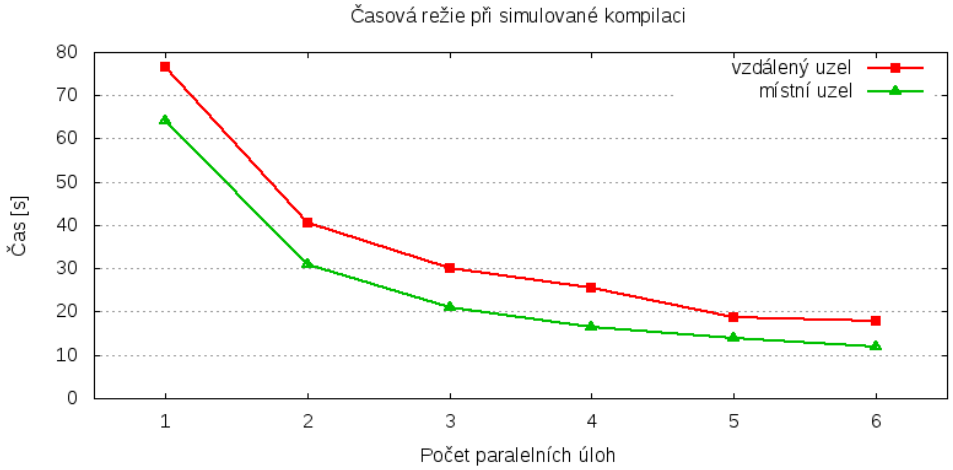
4.2.1 Časová režie při komunikaci

Jako první test bylo provedeno měření časové režie při simulované kompilaci linuxového jádra na místním uzlu a jednom vzdáleném uzlu. Měřený parametr je počet paralelně běžících úloh kompilace. Výstupem každého měření je celkový čas běhu kompilace. Byla provedena celkem 3 měření pro každý měřený parametr. Výsledkem je aritmetický průměr časů těchto 3 měření. Simulovaná práce pro toto měření bylo počítání MD5 haše kompilačního příkazu. Počet počítaných hašů pro toto měření je v rozmezí 5000 – 15000.

Výsledek tohoto měření je v grafu 4.1.

Z naměřených výsledků je patrné, že se vzrůstajícím počtem paralelních úloh klesá čas potřebný ke kompilaci jádra. Zároveň je vidět, že kompilace na místním uzlu je rychlejší než na vzdáleném uzlu. Toto zpoždění lze nejspíše přisuzovat komunikační režii při komunikaci mezi uzly. Z grafu je také patrné, že při vzrůstajícím počtu paralelních úloh klesá hodnota rozdílu času kompilace na místním a vzdáleném uzlem – komunikační režie.

Obrázek 4.1: Graf časové režie při komunikaci mezi uzly

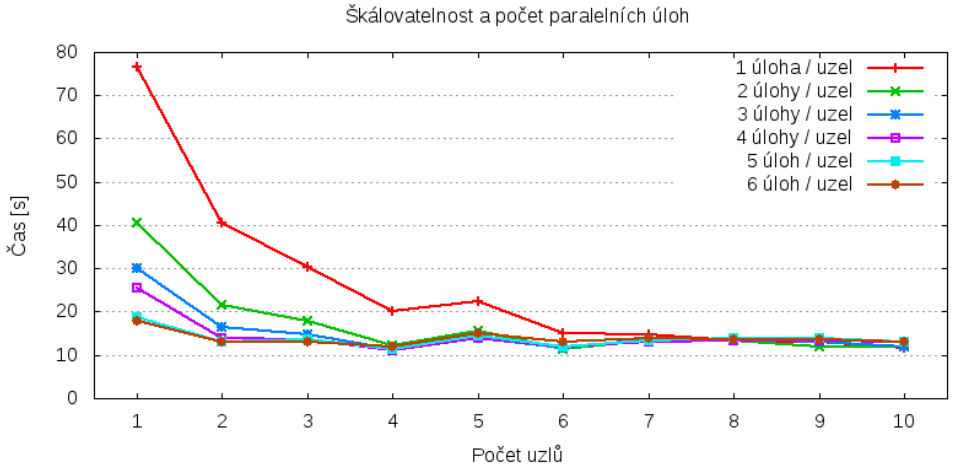


4.2.2 Škálovatelnost

V tomto testu byla měřena škálovatelnost. Cílem je zjistit, pro jaký počet paralelních úloh dosahuje simulátor nejlepších výsledků. Měřen bude čas simulované kompilace. Měření bylo provedeno na 1 až 10 uzlech, 11. uzel spouštěl kompilaci. Na každém uzlu bylo spouštěno 1 – 6 paralelních úloh. Výsledkem každého měření je čas běhu kompilace. Celkem proběhla 3 měření pro všechny kombinace počtu uzlů a počtu paralelních úloh. Výsledkem každého měření je aritmetický průměr časů ze všech 3 měření. Simulovaná práce pro toto měření byla shodná s měřením v sekci 4.2.1.

Výsledek tohoto měření je v grafu 4.2.

Obrázek 4.2: Graf škálovatelnosti



Podle [34] definujeme **zrychlení** S testovací úlohy jako poměr času běhu kompilace na 1 uzlu a času běhu kompilace na p uzlech:

$$S(n,p) = \frac{T_0(n)}{T(n,p)}$$

Výsledek ukazuje, že se 2 uzly dosahuje simulátor zrychlení 1.90 pro 1 úlohu, 1.88 pro 2 paralelní úlohy a 1.83 pro 3 a 4 paralelní úlohy. Pro 5 a 6 paralelních úloh dosahuje zrychlení hodnoty přibližně 1.38.

Podle [34] definujeme **efektivnost** E testovací úlohy jako poměr zrychlení a počtu uzlů p:

$$E(n,p) = \frac{S(n,p)}{p}$$

Pro 2 uzly s 1 úlohou na uzel dosahuje simulátor efektivnost 95%, pro 4 paralelní úlohy 91%. Pro 5 a 6 paralelních úloh dosahuje efektivnost pouze 69%.

Zrychlení pro 6 procesů se zastaví na hodnotě 1.6 pro 4 uzly, efektivnost dosahuje 26%. Od 4 uzlů a více se kompilační čas s 2 a více úlohami na uzel nezvyšuje, tudíž klesá zrychlení a efektivnost.

Z naměřeného grafu je zřejmé, že nejlepšího výsledku se dosáhne se 4 uzly, nezáleží přitom na počtu úloh na uzel. Pouze při kompilaci s 1 úlohou na uzel dosahuje na 4 uzlech horších výsledků.

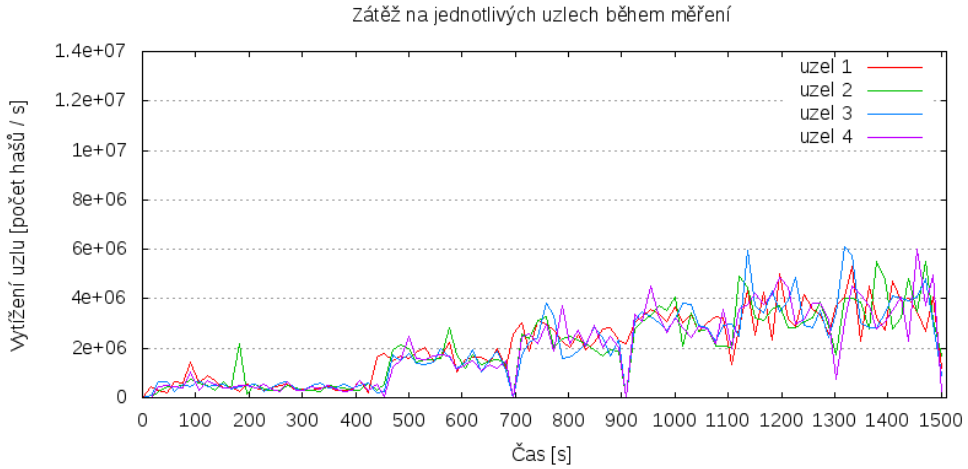
4.2.3 Rozdělení zátěže

Tento test je zaměřen na měření rozdělení zátěže. Měřit se bude zátěž na jednotlivých uzlech s cílem určit, jestli je zátěž rozložena rovnoměrně mezi všechny uzly. Protože není možné zjistit aktuální zátěž uzlu z operačního systému (aktuální zátěž kontejneru je shodná s aktuální zátěží hostitelského počítače), bude zátěž reprezentována počtem spočítaných hašů za sekundu. Simulování práce u tohoto testu bude provedeno stejně jako u předchozích měření. Aby bylo možné lépe porovnat zátěž na jednotlivých serverech, byly zvýšeny minimální a maximální hodnoty počtu počítaných hašů – spodní mez pro tento test je 60000 hašů, horní mez 120000 hašů. Zvýšení počtu počítaných hašů má za následek delší dobu běhu simulované kompilace, tím pádem také více dat k porovnání. Měření bude probíhat se 4 uzly na kompilování, 5. uzel bude spouštět kompilaci. Spouštěno bude 1 až 6 paralelních úloh na uzel.

Výsledek tohoto měření je v grafu 4.3.

Graf zobrazuje zátěž všech uzlů během měření. Během tohoto měření byla monitorována zátěž hostitelského počítače. Zátěž byla monitorována příkazem `uptime` a uvažována byla hodnota „load average 1 min.“ (průměrná zátěž za minutu). Během měření zátěž nepřesáhla hodnotu 16, která v případě 16 jader procesoru znamená 100% vytížení. Všechny uzly měly tedy během měření dostatek výpočetních prostředků pro výpočty.

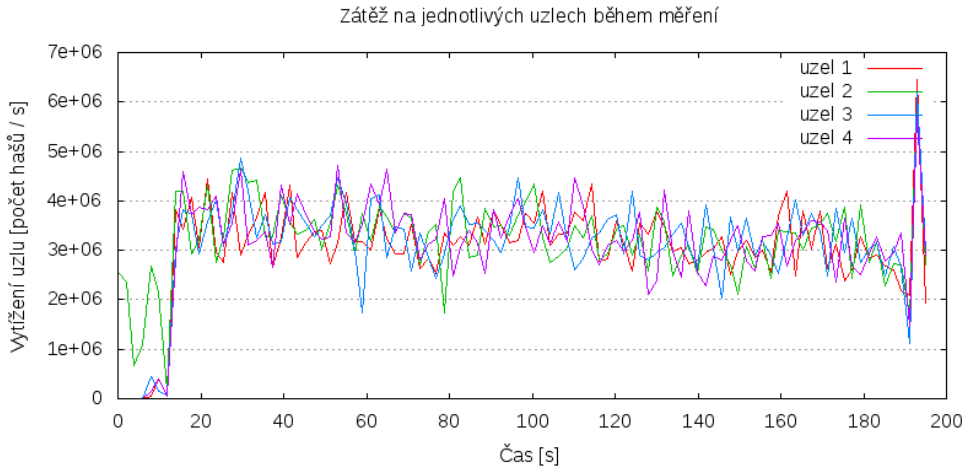
Obrázek 4.3: Měření distribuce zátěže mezi jednotlivé uzly, 1 – 6 úloh na uzel



Z výsledků je patrné, že zátěž je rozdělována rovnoměrně mezi všechny uzly – jednotlivé čáry grafu se překrývají. Nejvyšší zátěž, kterou byl schopen uzel dosáhnout, bylo přibližně 6000000 hašů za sekundu. Této zátěže uzly dosáhly při 5 a 6 úlohách na uzel.

V příloze této práce C.1 je zobrazen podobný graf rozložení zátěže, který zobrazuje stejné měření, ale s 6 pracujícími uzly.

Obrázek 4.4: Měření distribuce zátěže mezi jednotlivé uzly – 4 úlohy na uzel



Graf 4.4 zobrazuje zátěž uzlů při simulované kompilaci jádra. Kompilace byla spuštěna se 4 úlohami na uzel. Z grafu je patrné, že na začátku kompilace, prvních 5 sekund, pracuje pouze jeden uzel. Během této doby se kompilují pomocné programy pro kompilaci linuxového jádra. Následně je vidět

rovnoměrné rozprostření zátěže mezi všechny uzly – čáry grafu se překrývají. Na konci kompilace je vidět prudké zvýšení zátěže všech uzlů a následné dokončení kompilace.

4.3 Interpretace výsledků

Byla provedena 3 různá měření. Z měření komunikačního zpoždění je možné říci, že simulovaná kompilace na 1 vzdáleném uzlu je pomalejší než kompilace na místním uzlu. Tento výsledek byl očekávaný, protože při simulované kompilaci na vzdáleném uzlu vzniká při síťové komunikaci mezi uzly zpoždění, které se nutně musí promítnout do času běhu kompilace.

V testu škálovatelnosti bylo změřeno, že lze počet uzlů efektivně navyšovat až do 4 uzlů. Od 5 a více uzlů již nedochází ke zlepšení času kompilace. Je možné se domnívat, že v případě delšího běhu simulované práce by byla škálovatelnost vyšší, toto ale nebylo testováno.

V posledním testu bylo potvrzeno, že je práce při simulované kompilaci rovnoměrně rozložena mezi výpočetní uzly.

Závěr

Cílem práce bylo navrhnout a implementovat simulátor jádra Clondike. Tohoto cíle bylo dosaženo. Součástí práce je také úprava záplaty linuxového jádra 3.18.21 pro projekt Clondike.

V úvodu byla nastudována práce Ing. Zdeňka Nového [1], ze které bylo vycházeno při následných úpravách linuxového jádra. Po rozsáhlém studiu fungování linuxového jádra a jaderné části projektu Clondike bylo opraveno množství kritických chyb, se kterými nebylo možné jádro používat.

Hlavní část této práce byla věnována návrhu a implementaci simulátoru jádra Clondike. Ze studia fungování jádra Clondike vzešel návrh simulátoru, který se zaměřuje na simulaci všech komponent jádra Clondike nutných pro běh simulátoru. Simulátor byl implementován tak, aby nebylo nutné provádět změny v uživatelském prostoru systému Clondike, které by mohly způsobit nekompatibilitu s jádrem Clondike. Toho se podařilo dosáhnout s jednou výjimkou, kdy bylo nutné nastavit v knihovně Director API port pro komunikaci se simulátorem pomocí protokolu Netlink.

Pro účely snadného použití a vývoje byl vytvořen Dockerfile s instrukcemi pro vytvoření Docker kontejneru. Díky použití Docker kontejnerů mohou vývojáři Simple Ruby Directoru jednoduše pracovat se simulátorem a velmi rychle si mohou vytvořit libovolně velký cluster pro testování.

Závěrečná část práce se zabývá měřením vlastností simulátoru jádra Clondike. V prvním měření je měřeno komunikační zpoždění při kompilaci jádra na místním uzlu a na vzdáleném uzlu. Další měření zabývající se škálovatelností simulátoru ukázalo, že je simulátor dobře škálovatelný až do počtu 4 uzlů. Z průběhu měření se lze domnívat, že výsledky byly ovlivněny během simulátoru a jednotlivých uzlů na virtuálním stroji. Poslední měření ukázalo, že při simulované kompilaci linuxového jádra je rozdělování zátěže mezi jednotlivé uzly spravedlivé.

Budoucí práce

Využití transakčních záznamů v databázi Cassandra V této práci bylo provedeno dokončení napojení Simple Ruby Directoru na databázi Cassandra, do které se ukládá distribuovaný log transakcí. V další práci by bylo vhodné upravit Simple Ruby Director tak, aby při výběru uzlu k migraci bral ohled na spolehlivost daného uzlu v clusteru Clondike.

Úpravy jádra Clondike V aktuálním jádře Clondike verze 3.18.21 se stále objevuje kritická chyba, které znemožňuje jeho používání. Oprava této chyby bude relativně náročná a pro její opravení bude nutné velmi podrobně nastudovat fungování linuxového jádra při práci s pamětí.

Portace na linuxové jádro 4.X Aktuální verze Clondike jádra 3.18.21 již není nejnovější a bylo by vhodné provést aktualizaci na verzi linuxového jádra $4.X$.

Výměna 9P protokolu Analyzovat použití protokolu 9P a navrhnout využití jiného protokolu pro připojení vzdáleného souborového systému. V této práci byl původně používaný NPFS server nahrazen serverem Diod. Bude nutné analyzovat, jestli tato změna byla vhodná a jestli se nenabízí lepší použití jiného síťového protokolu pro připojení vzdáleného souborového systému.

Literatura

[1] Nový, Z.: Implementace škodné do projektu Clondike. 2016.

[2] Tvrdík, P.: Implementace BitTorrent discovery protokolu do Clondike. 2014.

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

[4] The Linux Foundation: The Linux Kernel. [online], [cit. 2016-05-01]. Dostupné z: `https://www.kernel.org/`

[5] Margaret Rouse: Process Definition. [online], 2005, [cit. 2016-05-01]. Dostupné z: `http://whatis.techtarget.com/definition/process`

[6] Lawton, G.: Developing software online with platform-as-a-service technology. *Computer*, ročník 41, č. 6, 2008: s. 13–15.

[7] Git-users: Git. [online], [cit. 2016-05-01]. Dostupné z: `https://git-scm.com/`

[8] Project, T. L. I.: Inode Definition. 2006, [cit. 2016-05-01]. Dostupné z: `http://www.linfo.org/inode.html`

[9] Postel, J.: Transmission Control Protocol. STD 7, RFC Editor, September 1981. Dostupné z: `http://www.rfc-editor.org/rfc/rfc793.txt`

[10] Thomas Graf: Netlink Library. [online], 2011, [cit. 2016-05-01]. Dostupné z: `https://www.infradead.org/~tgr/libnl/doc/core.html`

[11] Alcatel-Lucent: Plan 9 from Bell Labs. [online], [cit. 2016-05-01]. Dostupné z: `http://plan9.bell-labs.com/plan9/about.html`

[12] B. Pawlowski and Ch. Juszczak and P. Staubach and C. Smith and D. Lebel and David Hitz : NFS version 3, Design and Implementation. [online], [cit. 2016-05-01]. Dostupné z: https://www.usenix.org/legacy/publications/library/proceedings/bos94/full_papers/pawlowski.ps

[13] Rákosník, J.: Úpravy Clondike pro představení open source komunitě. 2013.

[14] (PCG), P. C. G.: FIT-CVUT/clondike. 2012, [cit. 2016-05-01]. Dostupné z: https://github.com/FIT-CVUT/clondike

[15] Daniel Pierre Bovet: Implementing virtual system calls. [online], 2014, [cit. 2016-05-01]. Dostupné z: https://lwn.net/Articles/615809/

[16] Surya Prabhakar: Understanding a Kernel Oops! [online], 2011, [cit. 2016-05-01]. Dostupné z: http://opensourceforu.efytimes.com/2011/01/understanding-a-kernel-oops/

[17] The Linux Foundation: Plan 9 Resource Sharing for Linux. [online], [cit. 2016-05-01]. Dostupné z: https://www.kernel.org/doc/Documentation/filesystems/9p.txt

[18] The cat-v.org Community: 9P Implementations. [online], [cit. 2016-05-01]. Dostupné z: http://9p.cat-v.org/implementations

[19] V9fs-users: Distributed I/O Daemon - a 9P file server. [online], [cit. 2016-05-01]. Dostupné z: https://github.com/chaos/diod

[20] Love, R.; Are, S. H. W.; Linus, A. C.; aj.: *Linux kernel development second edition*. Pearson Education, USA, 2005.

[21] Robert Watson: Unix domain sockets vs. internet sockets. [online], [cit. 2016-05-01]. Dostupné z: http://lists.freebsd.org/pipermail/freebsd-performance/2005-February/001143.html

[22] Rivest, R.: The MD5 message-digest algorithm. 1992.

[23] Cerf, V. G.: ASCII format for network interchange. 1969.

[24] Josey, A.; Cragun, D.; Stoughton, N.; aj.: The Open Group Base Specifications Issue 6–IEEE Std 1003.1. *The IEEE and The Open Group*, ročník 20, č. 6, 2004.

[25] The Apache Software Foundation.: The Apache Cassandra Project. [online], [cit. 2016-05-01]. Dostupné z: http://cassandra.apache.org/

[26] Docker, Inc.: What is Docker? [online], [cit. 2016-05-01]. Dostupné z: https://www.docker.com/what-docker

[27] Docker, Inc.: Docker Hub. [online], [cit. 2016-05-01]. Dostupné z: https://hub.docker.com

[28] Cloud Foundry Documentation: Buildpacks. [online], 2016, [cit. 2016-05-01]. Dostupné z: http://docs.cloudfoundry.org/buildpacks/?cm_mc_uid=98262923273014508539884&cm_mc_sid_50200000=1462106814

[29] Charlie Dai: The Cloud Foundry Foundation: The Key Driver Of A Breakthrough In PaaS Adoption. [online], 2014, [cit. 2016-05-01]. Dostupné z: http://blogs.forrester.com/charlie_dai/14-12-10-the_cloud_foundry_foundation_the_key_driver_of_a_breakthrough_in_paas_adoption

[30] SoftLayer Technologies. Inc.: About SoftLayer. [online], [cit. 2016-05-01]. Dostupné z: http://www.softlayer.com/about-softlayer

[31] Instaclustr, inc.: Instaclustr. [online], [cit. 2016-05-01]. Dostupné z: http://www.instaclustr.com/

[32] Amazon Web Services, Inc. or its affiliates: Task Definition Parameters. [online], 2016, [cit. 2016-05-01]. Dostupné z: http://docs.aws.amazon.com/AmazonECS/latest/developerguide/task_definition_parameters.html

[33] Amazon Web Services, Inc. or its affiliates: Amazon Linux AMI. [online], [cit. 2016-05-01]. Dostupné z: http://aws.amazon.com/amazon-linux-ami/

[34] Tvrdík, P.: Paralelní systémy a algoritmy. Nakladatelství ČVUT, 2000.

Příloha A

Seznam použitých zkratek

9P Plan 9 Filesystem Protocol

API Application Programming Interface

ASCII American Standard Code for Information Interchange

BASH Bourne Again SHell

CCFS Cache Control File System

CCN Core Cluster Node

CDN Content Delivery Network

CPU Central Processing Unit

CQL Cassandra Query Language

CTLFS Controller Filesystem

DHT Distributed Hash Table

FS File System

IP Internet Protocol

KKC Kernel-To-Kernel Communication library

MD5 Message Digest 5 algorithm

NFS Network File System

NoSQL Non SQL

NPFS Named Pipe File System

PaaS Platform as a Service

PEN Process Execution Node

PID Process Identifier

RAM Random-access Memory

SQL Structured Query Language

TCMI Task Checkpointing and Migration Infrastructure

TCP Transmission Control Protocol

UDP User Datagram Protocol

UID User Identifier

Příloha **B**

Obsah přiloženého CD

- `readme.txt` — stručný popis obsahu CD
- `clondike_repository` — oficiální repozitář Clondike
  - `devel` — adresář se soubory pro vývoj (měření a testy)
  - `doc` — dokumentace generovaná pomocí nástroje doxygen
  - `etc` — konfigurační a inicializační skripty
  - `kernel_simulator` — adresář se soubory simulátoru jádra Clondike
  - `patches` — záplaty pro různé verze linuxových jader
  - `root` — adresář uživatele *root* obsahující konfiguraci a server *npfs*
  - `scripts` — pomocné ovládací skripty
  - `sources` — zdrojové soubory jádra pro různé verze
  - `userspace` — zdrojové soubory uživatelského prostoru
    - `director-api` — soubory pro komunikaci přes Netlink
    - `ruby-director-api` — rozhraní mezi jazyky C a Ruby
    - `simple-ruby-director` — skripty uživatelského prostoru v Ruby
  - `Dockerfile` — soubor s intrukcemi pro vytvoření Docker obrazu
  - `INSTALL` — návod na instalaci
  - `README.md` — popis repozitáře
- `diplomova_prace` — text práce
  - `DP_Friedl_Jan_2016.tex` — zdrojová forma práce ve formátu LaTeX
  - `DP_Friedl_Jan_2016.pdf` — text práce ve formátu PDF

Příloha C

Doplňující grafy rozložení zátěže při simulované kompilaci

Obrázek C.1: Měření distribuce zátěže mezi jednotlivé uzly, 6 uzlů, 1 – 6 úloh na uzel