VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INFORMAČNÍCH SYSTÉMŮ
DEPARTMENT OF INFORMATION SYSTEMS

VYUŽITÍ LINUXOVÝCH KONTEJNERŮ V HW/SW CO-DESIGN

UTILIZATION OF LINUX CONTAINERS FOR HW/SW CO-DESIGN

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE **JIŘÍ JUŘICA**
AUTHOR

VEDOUCÍ PRÁCE **prof. Ing. TOMÁŠ HRUŠKA, CSc.**
SUPERVISOR

BRNO 2020

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav informačních systémů (UIFS) Akademický rok 2019/2020

Zadání bakalářské práce

Student: **Juřica Jiří**
Program: Informační technologie
Název: **Využití linuxových kontejnerů v HW/SW co-design**
Utilization of Linux Containers for HW/SW Co-Design
Kategorie: Analýza a testování softwaru

Zadání:

1. Prostudujte problematiku virtualizace aplikací a strojů. Seznamte se se současnými metodami kontejnerizace a virtualizace.
2. Analyzujte požadavky pro řízení paralelních úloh. Navrhněte řešení pro nahrávání a spouštění paralelních úloh. Ověřte možnosti případné integrace s dalšími systémy například Jenkins či Opennebula.
3. Implementujte systém pro spouštění paralelních úloh. Systém bude umožňovat automatickou přípravu prostředí na vybraných strojích, řízení přidělování zdrojů a rozložení zátěže.
4. Proveďte experimenty a ověřte výkonnost zvoleného řešení.

Literatura:

- Mouat, A.: *Using Docker: Developing and Deploying Software with Containers*. O'REilly Media, 2015, ISBN: 9781491915929.
- Brogi, A., Neri, D., Rinaldi, L. a Soldani, J.: Orchestrating incomplete TOSCA applications with Docker. *Science of Computer Programming*. Elsevier B.V, 2018, **166**, 194-213. DOI: 10.1016/j.scico.2018.07.005. ISSN: 0167-6423.
- *Enterprise Container Platform | Docker* [online]. Docker, 2019 [cit. 2019-10-13]. Dostupné z: https://www.docker.com/

Pro udělení zápočtu za první semestr je požadováno:

- Body 1 a 2.

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/

Vedoucí práce: **Hruška Tomáš, prof. Ing., CSc.**
Konzultant: Dolíhal Luděk, Ing., CODASIP
Vedoucí ústavu: Kolář Dušan, doc. Dr. Ing.
Datum zadání: 1. listopadu 2019
Datum odevzdání: 14. května 2020
Datum schválení: 24. října 2019

Abstrakt

Tato práce pojednává o možnostech nahrazení klasických virtuálních strojů linuxovými kontejnery. Cílem je vytvořit funkční systém ke spouštění paralelních úloh, které mají předchystané prostředí a slouží k testování a buildu softwaru.

Pro kontejnerizaci byla zvolena platforma Docker, která je doplněna o automatickou konfiguraci počítačů, na kterých systém běží. Dále je tento systém rozšířen o propojení s nástrojem pro průběžnou integraci Jenkins.

Vytvořené řešení pomohlo ověřit, že používání kontejnerů je efektivnější než starší technologie virtualizace. Doba provádění úloh tak byla zkrácena v průměru o 24 %. Přínosem této práce je srovnání dvou virtualizačních přístupů a uvedení do technologie Docker.

Abstract

In this bachelor thesis a way of replacement standard virtual machines by Linux containers is described. The goal is to create a fully working system for performing parallel jobs with prepared environments for software build and test purposes.

For containerization the Docker platform was chosen, which is used with automation tool for computers preparation. The system is also connected with continuous integration tool called Jenkins.

This solution proves that usage of the Linux containers is more efficient than older virtualization technologies. The due time of each job was shortened about 24 % in average. The main benefits of this thesis is the comparison of two ways of virtualization and the introduction to the Docker technology.

Klíčová slova

Virtualizace, kontejner, Docker, automatizace, orchestrace, Ansible, Jenkins, DevOps, Zabbix

Keywords

Virtualization, container, Docker, automation, orchestration, Ansible, Jenkins, DevOps, Zabbix

Citace

JUŘICA, Jiří. *Využití linuxových kontejnerů v HW/SW co-design.* Brno, 2020. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce prof. Ing. Tomáš Hruška, CSc.

Využití linuxových kontejnerů v HW/SW co-design

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana prof. Ing. Tomáše Hrušky, CSc. Další informace mi poskytli pan Ing. Luděk Dolíhal, Ph.D. a pan Ing. Jiří Hynek, Ph.D. Uvedl jsem všechny literární prameny, publikace a další zdroje, ze kterých jsem čerpal.

.....
Jiří Juřica
4. června 2020

Poděkování

Tímto bych rád poděkoval panu profesorovi Tomáši Hruškovi za vedení mé práce, panu doktorovi Luďku Dolíhalovi za odborné rady, podporu a pomoc a panu doktorovi Jiřímu Hynkovi za praktické rady ohledně obsahu technické zprávy a rozvržení času.

Děkuji také společnosti Codasip s. r. o. za poskytnutou možnost pracovat na jejich hardwaru a své rodině, přátelům a kolegům za podporu v průběhu celého mého studia.

Obsah

1 Úvod **2**

2 Virtualizace **3**
2.1 Stručná historie . . . 3
2.2 Způsoby virtualizace . . . 4
2.3 Typy hypervizorů . . . 8
2.4 Přínosy používání virtualizace . . . 9
2.5 Virtualizační prostředí . . . 11

3 Docker **13**
3.1 Docker Engine . . . 14
3.2 Dockerfile . . . 18
3.3 Docker registry . . . 19
3.4 Docker Compose . . . 20
3.5 Docker swarm . . . 21

4 Ansible **22**
4.1 Inventory . . . 23
4.2 Playbooks . . . 24
4.3 Roles . . . 26

5 Analýza a návrh řešení pro společnost Codasip **28**
5.1 Návrh nového řešení . . . 29

6 Implementace **31**
6.1 Příprava počítačů a virtuálních strojů . . . 31
6.2 Tvorba rolí v nástroji Ansible . . . 32
6.3 Tvorba imageů . . . 36
6.4 Jenkins . . . 37

7 Testování řešení **40**
7.1 Test uRISC IP compiler regression s jednou větví . . . 41
7.2 Test uRISC IP compiler regression se třemi větvemi . . . 42
7.3 Test uRISC uvm applications se simulátory třetích stran . . . 44

8 Závěr **45**

Literatura **46**

Kapitola 1

Úvod

Tato práce se zabývá metodami virtualizace počítačů. Přestože se jedná o oblast informatiky, která vznikala ještě před dobou osobních počítačů, její přínos je i v dnešní době značný. Má široké využití, snižuje dobu nasazení služeb, náklady na provoz serverů, zvyšuje zabezpečení a vytváří izolovaná prostředí. S postupným vývojem přichází i nové metody, které virtualizaci zefektivňují, z nichž je zde nejvíce probírána virtualizace na úrovni operačních systémů. Druhou doménou, kterou se autor zabývá, je moderní přístup ke správě podnikové infrastruktury, u které je dbáno o snahu vykonat co nejvíce úloh automaticky. Takový přístup usnadňuje práci administrátorům, kteří nemusí strávit tolik času u pravidelně se opakujících úkonů.

Bakalářská práce byla vytvořena pro účely společnosti Codasip, která virtualizaci používá ve velkém množství při procesu vývoje softwaru. Porovnává efektivitu současného řešení s modernějším přístupem a obsahuje jeho kompletní implementaci pro účely srovnání a případného pozdějšího nasazení.

Cílem je nastudování a uvedení do problematiky virtualizace, stanovení jejích typů a hrubé zasazení do historického kontextu. Dále jde o důkladné seznámení se softwarem Docker, který je v rámci práce pro virtualizaci na úrovni operačního systému použit a s nástrojem Ansible. Ten je využit k automatizované přípravě počítačů, instalaci softwaru, spouštění virtualizace a inicializaci nainstalovaných aplikací. Následuje návrh systému určeného k vykonávání úloh pro vývoj softwaru s novým typem virtualizace, porovnání s dosavadním přístupem a implementace nového systému. Nakonec je řešení otestováno a zhodnoceno.

Výše popsané části jsou rozděleny do několika kapitol, z nichž první hned po úvodu se věnuje virtualizaci (kapitola 2). Je zde mimo představení oblasti virtualizace a rozčlenění typů popsána také funkce hypervizoru a k tomu je přidáno srovnání vybraných druhů virtualizačního softwaru. V kapitole 3 je představen software Docker. Po přečtení by měl čtenář získat jak teoretické informace o tom, jakým způsobem program pracuje, tak praktický návod jak jej použít i pro ne zcela triviální případy. Následuje kapitola 4, ve které je v podobné míře rozebrán nástroj Ansible. Kapitoly 5 a 6 popisují podrobně, jaké prostředky se ve společnosti Codasip používají nyní, návrh nového řešení a kompletní popsání jeho implementace. Jsou zde rozebrány veškeré služby, které k vykonání práce byly potřeba, včetně částí sloužících k měření. V části 7 je uvedeno, jakým způsobem bylo prováděno měření a jsou zde představeny veškeré testy pro ověření výsledků práce. V závěru (kapitola 8) je celá práce zhodnocena, je zde vyjmenováno, co všechno se podařilo splnit a také části, kterými by se tato práce dala v budoucnu rozšířit.

Kapitola 2

Virtualizace

Slovo virtuální znamená v běžném světě nějaký objekt, který není skutečný a v počítačové vědě je tento význam velmi podobný. Myslíme tím přidání abstraktní vrstvy nad reálným hardwarem, proměněnou do logických objektů. I tato specifikace je ale poměrně široká a mohli bychom si tak představit prakticky cokoliv od problematiky virtuálních disků, po používání VLAN[1] v počítačových sítích a bylo by to správně. V této práci budou dále tyto pojmy zmíněny spíše okrajově a hlavním předmětem se stane virtualizace celých počítačů a jejich operačních systémů. Virtualizaci tedy rozumíme jako snaze o spouštění více virtuálních strojů[2] na jednom počítači. Počítač, na kterém běží virtualizační prostředí, nazýváme hostitel a virtuální stroje označujeme jako hosty. Výhody této technologie i jednotlivé metody virtualizace jsou rozepsány ve zbytku této kapitoly.

2.1 Stručná historie

Prvotní zmínky ohledně virtualizace pochází již z 60. let 20. století. Tehdejší počítače využívaly dávkového zpracování, kdy byl najednou vykonáván pouze jeden program, který nebyl schopen interakce s uživatelem v reálném čase. Tento přístup byl mnohdy neefektivní, protože nedokázal využít celý výkon počítače a k více uživatelským systémům schopných multitaskingu bylo ještě daleko. Společnost IBM tento problém začala řešit experimenty s virtualizací se softwarem CP-67. [3] V raných 70. letech pak IBM přišlo s jejich první komerční verzí. Jednalo se o podporu virtualizačního prostředí s názvem VM/370 pro mainframy řady System/370. Tento produkt společnost nabízí dosud, pouze označení se změnilo na z/VM podle aktuální nabízené řady mainframů. [5]

V 80. letech pak přichází boom osobních počítačů se systémem MS-DOS od společnosti Microsoft, který se stal brzy na trhu dominantní. Ve stejné době dochází k masovému rozšíření počítačů do bussiness prostředí, společnosti si přivykají na jejich používání ve větší míře, udržují zde např. účetnictví nebo informace o zaměstnancích. Pořizují především mainframy nebo jejich menší verze. U těchto počítačů ale nedominuje žádný operační systém, ten je na počítači od výrobce. To zvyšovalo nároky na IT specialisty, kteří museli umět obsluhovat různá prostředí, která spolu nebyla vzájemně kompatibilní, a přenášet tak data mezi nimi bylo velmi náročné. To dalo za vznik požadavkům na standardy pro výměnu informací a prvním myšlenkám pro vznik operačního systému, který by dokázal fungovat

[1] Virtual Local Area Network – technologie umožňující provoz více vzájemně oddělených lokálních sítí na jedné fyzické.

[2] Virtuální stroj – počítač, který fyzicky neexistuje a je vytvořen pomocí virtualizace. V anglické literatuře se velmi často používá označení Virtual Machine nebo zkratka VM.

na různých zařízeních. Netrvalo dlouho a v Bellových laboratořích vznikl operační systém UNIX. [12]

Společnosti měly často dva systémy na obsluhu. MS-DOS a druhý podle výrobce mianframu, což se začalo ukazovat jako ne zrovna efektivní kvůli školení personálu. Osobní počítače získávaly čím dál více paměti a výkonu, a mohly tak začít spouštět náročnější aplikace, které bylo původně možné provozovat jen na mainframech. Postupně byly skutečně upravovány, aby mohly běžet na PC. To se ale neobešlo bez problémů, protože osobní počítače byly mnohem méně stabilní a tehdejší systém od Microsoftu byl původně jednoprogramový. [16] Firmy to ovšem neodrazovalo, protože pořizovací náklady byly mnohem menší. Veškerá úskalí u osobních počítačů byla nakonec překonána, což vedlo k počátkům vývoje virtualizace na architektuře x86, tak jak ji známe z dnešní doby.

V roce 1988 se pak stala prvním průkopníkem společnost Connectix s jejich produktem Virtual PC, který poprvé umožnil spustit systém z počítače Macintosh na Windows. Společnost Microsoft firmu Connectix v roce 2003 odkoupila. V roce 2008 pak přišla s jejich vlastním virtualizačním prostředím Hyper-V. Od roku 1998 na trhu také působí společnost VMware, která vyvinula produkt VMware server. Z linuxové virtualizace je pak vhodné zmínit KVM[3], na jehož vývoji se podílely od roku 2006 společnosti Qumrane a Red Hat, která Qumrane později odkoupila. [5]

2.2 Způsoby virtualizace

K provádění virtualizace existuje několik přístupů, které se liší především jejich rozsahem a mírou použití abstrakce. To pak souvisí s tím, že mezi způsoby jsou značné výkonnostní rozdíly. Zpravidla však nelze prohlásit, že by některá varianta byla lepší než jiná. Každá má své výhody a spousta virtualizačních produktů nenabízí pouze jednu metodu. V některé literatuře je možné se dočíst o více variantách nebo o jejich mírně jiném rozdělení. Zde je výčet těch nejdůležitějších pro vytvoření si správné představy, jak je virtualizace dosaženo a jak ji autoři odborných publikací rozdělují nejčastěji.

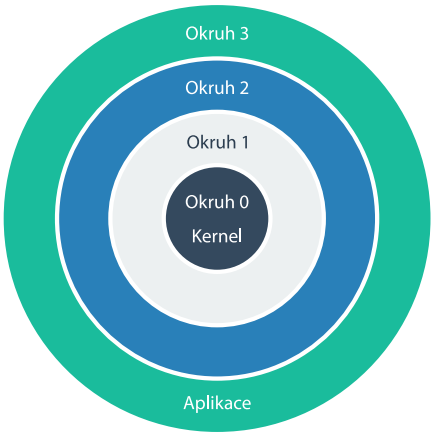
Emulace

Emulace je metoda, která umožňuje simulovat zcela odlišný hardware od toho, na kterém skutečně běží. Můžeme tak například na běžném osobním počítači s architekturou x86-64 programovat a testovat aplikace pro Android určené pro procesory ARM. Jde o způsob, kdy spouštíme binární data zkompilované pro jinou instrukční sadu, než je ta, kterou podporuje hostitelský procesor. Toho je docíleno převodem instrukcí emulovaného programu do instrukcí nebo posloupnosti instrukcí (zde velmi záleží na míře odlišnosti instrukčních sad obou procesorů) pro skutečný procesor. Převodu instrukcí je dosaženo různými technikami s odlišnými výpočetními nároky, obecně lze ale říci, že emulace je nejdražší virtualizační metodou. Mezi techniky emulace patří interpretace a binární překlad. Interpretace načte zdrojovou instrukci, provede její analýzu, vykoná ji a pokračuje v načítání dalších instrukcí. Binární překlad načítá instrukce po blocích a následně je ukládá do mezipaměti, což vede k vyšší náročnosti na začátku emulace. Později to ale celý proces urychluje, pokud se má vykonávat blok, který je již uložen. [15]

Než se dostaneme k dalším virtualizačním metodám, je třeba uvést věc nazývanou jako protection rings, jenž nemá žádný ustálený ekvivalent v českém jazyce. Jde o okruhy před-

[3] Kernel-based Virtual Machine

stavují míru privilegií předávanou hardwarem operačnímu systému v hierarchickém uskupení. [6] Je to způsob, jak chránit data a bránit chybám při přístupu k prostředkům počítače, ať už způsobenými špatně napsanou aplikací nebo záměrným zásahem do prostředků, které nejsou procesu přiděleny. Protection rings tedy významně přispívají k zabezpečení počítačových systémů a mají vazbu na režimy procesorů. Různé architektury mohou mít tuto ochranu aplikovanou odlišně. [16]



Obrázek 2.1: Protection rings pro architekturu procesorů x86.

Tak jak je znázorněno na obrázku 2.1, okruhy jsou sestaveny od největšího oprávnění k tomu nejmenšímu. Okruh úrovně 0 disponuje nejvyššími právy a pracuje v něm jádro operačního systému. To má pak přímý přístup k procesoru, pamětem a vstupně/výstupním zařízením. Další dva okruhy většina operačních systémů nevyužívá (to platí pro oba velmi používané systémy Windows i Linux), ale je možné je uplatnit při virtualizaci. Poslední uživatelský okruh s číslem 3 umožňuje provádění běžných programů. Pokud takový program potřebuje přístup k prostředkům z privilegovanějšího okruhu, musí využít systémového volání a nechat kód zpracovat jádrem operačního systému.

Hypervizor[4] potřebuje také přístup k hardwaru počítače. K tomu se však může dostat jen v okruhu 0 za předpokladu, že procesor hostitele není vybaven hardwarovou podporou virtualizace. Musí být tedy umístěn vedle jádra hostitelského operačního systému. Operační systémy na virtuálních strojích také počítají s přístupem do okruhu 0, což ve skutečnosti není možné, protože v této vrstvě nemůže běžet více systémových jader současně. OS[5] virtuálního stroje je nutno přesunout do jednoho z okruhů s menšími privilegii, o čemž ale takový systém nebude vědět a bude třeba vyřešit provádění privilegovaných instrukcí nebo je třeba virtualizovaný operační systém upravit takovým způsobem, aby dokázal fungovat v privilegovaném okruhu a zároveň neovlivňoval jiné systémy. [6]

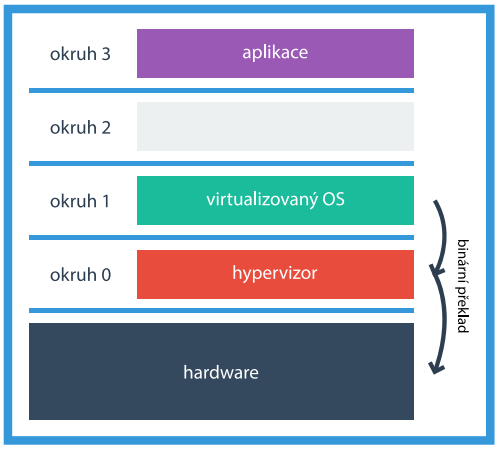
Plná virtualizace

U plné virtualizace je řešen problém virtuálního stroje běžícího v okruhu 1 tím, že instrukce vyžadující vyšší oprávnění jsou emulovány za použití binárního překladu, jak je patrné z obrázku 2.2. Do operačního systému na virtuálním stroji tak není nutné nijak zasahovat.

[4] Hypervizor – je software, který spravuje přístup virtuálních strojů k hardwaru hostitele a řídí jejich běh. Někdy se označuje také jako Virtual Machine Monitor, zkráceně VMM.

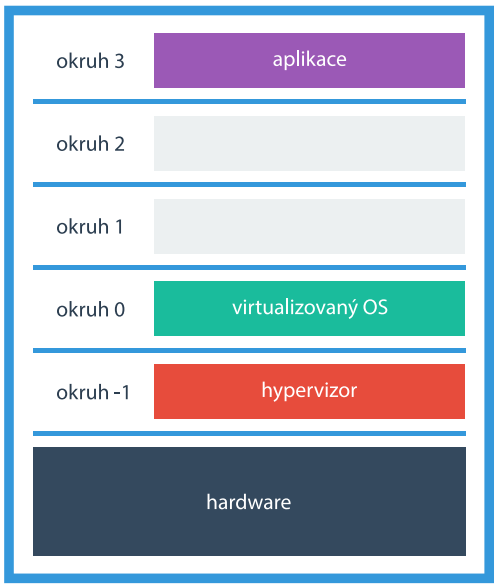
[5] Operační systém

Jde o efektivnější přístup, protože emulace není používána pořád, zároveň se ale jedná stále o výpočetně náročný proces v porovnání s paravirtualizací.



Obrázek 2.2: Schéma plné virtualizace.

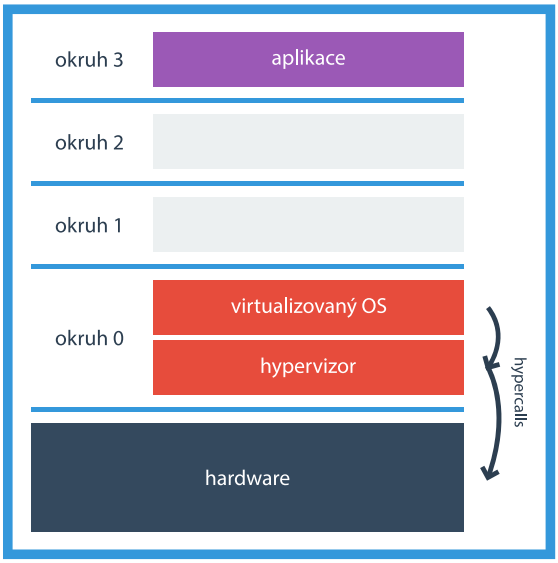
Tuto metodu je však možné výrazně zefektivnit a zrychlit, provozujeme-li ji na procesorech s hardwarovou podporou virtualizace. Poté, co si společnosti Intel a AMD začali uvědomovat důležitost jejího využití, rozšířily instrukční sadu architektury x86 o speciální instrukce pro efektivnější plnou virtualizaci. Každá společnost má pro tuto technologii jiné pojmenování a nejsou vzájemně kompatibilní. Můžeme se tak setkat s pojmy Intel VT-x a AMD-V, které se začaly objevovat u procesorů okolo roku 2006. [18] Hardwarová podpora umožňuje běžet hypervizoru v novém privilegovaném okruhu, někdy označovaném jako -1 a umožňuje tak běh virtuálních operačních systémů v okruhu 0, což je znázorněno na obrázku 2.3. Ty získají přímý přístup k hardwaru a přesto bude mít hypervizor vyšší privilegia, který se tak už nemusí zabývat náročným binárním překladem.



Obrázek 2.3: Schéma plné virtualizace s hardwarovou podporou.

Paravirtualizace

V případě paravirtualizace je třeba upravit jádro virtualizovaného operačního systému tak, aby místo privilegovaných instrukcí komunikovalo s hypervizorem. Hypervizor k tomu poskytuje API[6] a virtualizovaný operační systém se s nim dorozumívá pomocí tzv. hypercalls[7]. V tomto případě host ví o tom, že je virtualizovaný, to mu ale umožňuje běžet v okruhu 0 (obr. 2.4). Mezi výhody této metody patří její rychlost v porovnání s plnou virtualizací bez hardwarové podpory, kde se privilegované instrukce překládají. Zároveň je ale nezbytné mít pro ni softwarovou podporu ze strany virtualizovaného OS, což její používání omezuje. [6]



Obrázek 2.4: Schéma paravirtualizace.

Virtualizace na úrovni operačního systému

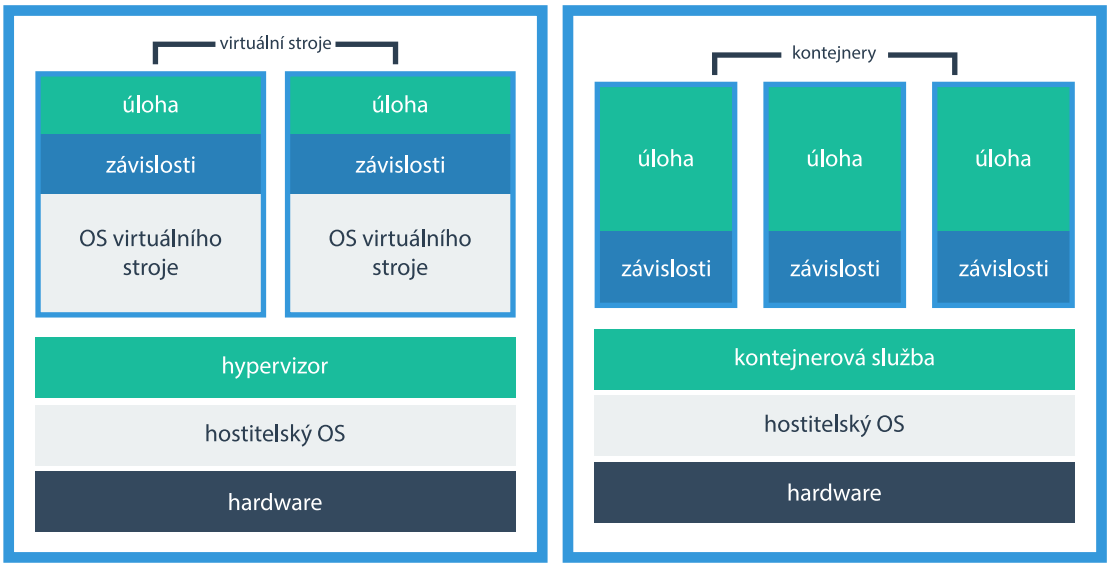
Jde o metodu nejvíce vzdálenou od hardwaru. Pro provoz nepotřebuje žádný hypervizor, který by spravoval přístup ke zdrojům počítače, zároveň ale neumožňuje spouštění více plnohodnotných operačních systémů mimo ten hostitelský. Proto se zde používají i jiné termíny. Místo virtuálních strojů zde mluvíme o kontejnerech. Kontejner je izolované prostředí od zbytku počítače, využívá ale jádro hostitelského operačního systému, a protože v okruhu 0 běží právě jeden kernel, nedochází k žádným problémům, které by bylo nutné řešit. Nad kontejnery pak bývá služba, která zřizuje jejich spouštění, zajišťuje k nim přístup a podobně. Narozdíl od virtuálních strojů, u kterých se doba spouštění blíží spíše spouštění běžného počítače, kontejnery jsou spuštěny nebo vypnuty v řádech vteřin. A protože využívají jádro jednoho operačního systému a veškeré jeho zdroje, jsou efektivnější. Z jejich návrhu však plyne také jedno podstatné omezení. Není možné spouštet kontejner s libovolným OS, ale pouze s tím, jehož jádro má i hostitel. Konkrétně tedy na Linuxu můžeme spouštět kontejnery s různymi distribucemi, ale nikdy ne kontejnery s Windows a opačně. Rozdíl mezi virtuálním strojem a kontejnerem je znázorněn na obrázku 2.5.

[6] Application programming interface – jde o externí rozhraní programu, které umožňuje jinému softwaru spouštět některé funkce a jiné prostředky programu, které API poskytuje.

[7] Hypercalls – jedná se o komunikaci s backendem hypervizoru pokaždé, kdy by měl virtualizovaný operační systém provést privilegované instrukce.

U kontejnerů existuje více možností, jak takové izolované prostředí vytvářet. Některý software nabízí řešení, které je více podobné virtuálním strojům. Pak se jedná o další instanci operačního systému a pokud bychom měli malé povědomí o jejich rozdílech, nebylo by snadné je jako uživatel rozlišit. Docker (kapitola 3) ovšem přišel s jiným přístupem. Tento přístup je založený na tom, že na jeden kontejner spouštíme jen jeden hlavní proces. Tedy nějaký softwarový démon (třeba web server, databázový server apod.), který běží nepřetržitě a ve chvíli, kdyby skončil, skončí i kontejner samotný. [1] Na takovém kontejneru je pak málokdy vytvořených více uživatelů a mimo potřeby propojení s další službou není ani běžné, aby tam běžel SSH[8] nebo telnet server.

Tento přístup se velmi hodí pro vývoj software založeného na architektuře mikroslužeb. Narozdíl od vývoje klasického robustního programu je software využívající tuto architekturu rozdělen do nezávislých komponent, které spolu vzájemně komunikují po síti. Pokud by to bylo potřeba, není pak problém takové aplikace škálovat a dimenzovat je pro vyšší použití, což by se u klasické aplikace řešilo složitěji.



Obrázek 2.5: Rozdíl mezi virtuálními stroji (obrázek nalevo) a kontejnery.

2.3 Typy hypervizorů

Jak již bylo zmíněno v textu výše, hypervizor je software, který řídí přístup virtuálních strojů k výpočetním prostředkům počítače, zodpovídá za jejich správu, spouštění a za vykonávání jejich privilegovaných požadavků, které nemohou provést sami. Co ale dosud nebylo uvedeno je, že hypervizory rozdělujeme do dvou kategorií podle toho, zda je na hostitelském počítači nainstalovaný operační systém. Nutno jen zmínit, že se jedná o vžitou praxi, která není podložená žádným standardem a v některých případech toto rozlišování zcela nestačí. [6]

[8] Secure shell – komunikační protokol a také aplikace pro vzdálené konzolové připojení, které je šifrováno na transportní vrstvě.

Typ 1

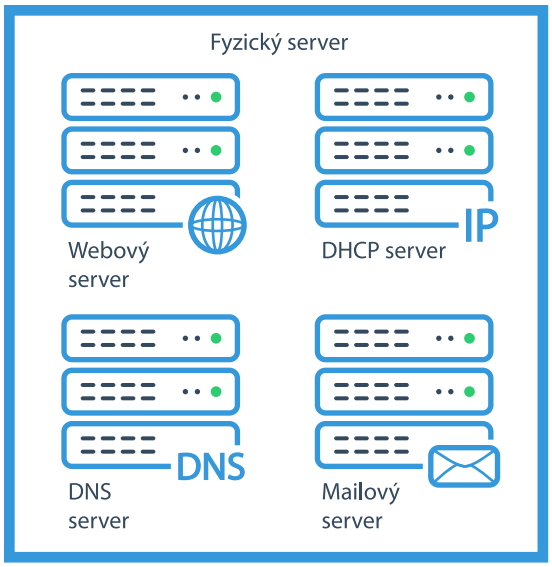
Někdy je označován jako nativní, je hypervizor, který se instaluje na počítač, aniž by na něm byl operační systém. Hypervizor je tak přímo zodpovědný za správu veškerého hardware, který počítač nabízí. Je efektivnější, protože zde není operační systém a aplikace, které by na něm běžely a velice jednoduše se nainstaluje a konfiguruje. Zároveň zde vyvstává problém. Pokud bychom potřebovali třeba nějaké speciální ovladače, hypervizor typu 1 je pravděpodobně nebude umět používat.

Typ 2

Je klasický hypervizor, se kterým se setkáme nejčastěji alespoň na uživatelské úrovni. Instaluje se v operačním systému a nemá přímý přístup k hardwaru, který za něj obstarává hostitelský OS.

2.4 Přínosy používání virtualizace

Virtualizace má velké množství výhod a dnes je již obtížně představitelné, že by se nějaká společnost, která vlastní více serverů nebo datacentrum, dokázala bez této technologie obejít. Jedná se o spolehlivé řešení, které snižuje náklady na provoz potřebných aplikací, zároveň také zvyšuje bezpečnost a do jisté míry zjednodušuje a urychluje správu prostředí aplikací. [6]



Obrázek 2.6: Server s virtualizovanými hosty, na kterých běží izolované aplikace.

Hlavním přínosem je bezesporu lepší využití výpočetních prostředků serverů, jak je patrné z obrázku 2.6. Mnoho počítačů, na kterých běží pouze jedna aplikace, nevyužívá celý jejich výkon. Pro představu velká část serverů v serverovnách nepřekročí využití CPU nad 10 % při běhu jedné aplikace. [17] Za zbylý výkon a energii tedy utrácely společnosti zbytečně. Virtualizace toto řeší a na jednom hostiteli můžeme vytvořit tolik virtuálních strojů nebo kontejnerů, abychom spotřebu výpočetních prostředků dostatečně využili. Dokud však nebyla, neexistovala jiná možnost, než mít pro každý program jiný počítač. To vedlo k vel-

kým nákladům na pořízení těchto zařízení, kterých bylo třeba mnohem více. Poté je tu celá řada nákladů, která souvisí s jejich provozem. Počítače spotřebovávají více elektrické energie, zvyšují se požadavky na chladící systémy v serverovnách, bez kterých by se servery uvnitř přehřály, je nutné mít vyšší kapacitu UPS[9] na překlenutí doby před nastartováním záložních generátorů a výkonnější generátory elektřiny. Také je potřeba pořídit více síťových zařízení nebo s více porty, větší množství kabeláže a více rackových skříní, což zabírá i více místa.

Mezi další výhody patří izolace služeb. Pokud provozujeme jednu aplikaci na virtuální stroj/kontejner, nemusíme se starat, jestli více programů nemá vzájemně se vylučující požadavky na systémové knihovny a další závislosti. Také zvyšujeme bezpečnost, protože tímto přístupem dokážeme předejít problémům, kdy závažná chyba jedné aplikace ovlivní chod druhé. Dále každé virtuální entitě můžeme nastavit jiný firewall, připojit jej do odlišných sítí nebo zcela zamezit přístupu k internetu a nastavit jiné přihlašovací údaje ke správcovským účtům. Pakliže se stane nějaký bezpečnostní incident a některý z hostů bude napaden, škody, které může útočník způsobit, budou daleko menší.

Virtualizací dokážeme usnadnit a zrychlit nasazení nových serverů. Uvažme situaci, kdy bude společnost provozovat jednu aplikaci na počítač bez virtualizace a jejich systémovému administrátorovi přijde nový server, který bude chtít hned začít používat. Server dopraví do serverovny, zapojí síť a další nezbytné náležitosti a začne instalovat operační systém. Po nainstalování doinstaluje specifické ovladače, které se v rámci instalace OS nepřipravily, nakonfiguruje síť a začne chystat prostředí podle požadavků společnosti. Dodá několik programů, které mu pomáhají se správou, přidá uživatelské účty dalších administrátorů a teprve v této chvíli je server připravený na to, aby se na něj mohla nainstalovat požadovaná služba. Šikovný správce si s takovým úkolem poradí v řádu nižších jednotek hodin. Kdyby ovšem využil některou z virtualizačních metod, může použít pro nasazení image[10], který už má připravený s veškerými náležitostmi a dobu přípravy by si tak výrazně zkrátil.

Pokud by nám některý z hostů havaroval a došlo by k poškození jeho disku nebo operačního systému, můžeme jej vrátit zpět do fungujícího stavu pomocí snapshotů[11] nebo nahrát image disku ze zálohy. Způsobené škody tak můžeme řešit mnohem rychleji. Je ale vhodné uvést, že snapshoty nejsou jen výsadou virtuálních strojů a s jejich využitím se můžeme setkat i u běžných počítačů např. při používání LVM[12] nebo souborového systému Btrfs[13].

Společnost, která vlastní více serverů, může také využívat možnosti dynamicky rozkládat zátěž mezi nimi. Kdyby se v určité chvíli stalo, že některý virtuální stroj/kontejner začne spotřebovávat více výpočetních prostředků a u hostitele začne hrozit dosažení maximálního vytížení, je možné hosta automaticky přesunout na počítač, který je vytížený méně. Přesnou definici přesunů je možné určovat podle sady pravidel zadaných administrátory. [6]

Dalším velkým přínosem je možnost vytvořit prostředí pro testování a experimentování. Při nasazování nového softwaru se vyplatí si jeho instalaci dopředu nanečisto vyzkoušet a zjitit případné problémy s chybějícími závislostmi na různých linuxových distribucích. Můžeme bez problémů zkoušet provádět i nejrůznější úkony, u kterých si nejsme jistí, jestli

[9] Uninterruptible Power Supply/Source – zdroj nepřerušovaného napájení využívající energii z baterií v případě výpadku proudu.

[10] Image – v tomto případě mluvíme buď o image disku, což je obraz počítačového disku, který je uložený v souboru nebo o image kontejneru, což jsou binární data obsahující veškeré náležitosti k tomu, abychom mohli kontejner spustit.

[11] Snapshot – stav obsahu disku uložený ke konkrétnímu datu.

[12] Logical Volume Management – je správa logických disků. Oproti běžnému vytváření oddílů nabízí tento přístup více možností, jak s disky nakládat.

[13] B-tree file system – jde o pokročilý linuxový souborový systém.

neohrozí stabilitu systému nebo jen testovat nejnovější verze operačních systémů, které dosud nejsou stanoveny jako vhodné pro produkční nasazení. Ať se nakonec stane cokoliv, je velmi jednoduché takové prostředí vrátit do původního stavu a pokračovat s experimenty. Dělat to ale přímo na počítači by bylo v mnoha ohledech daleko méně příjemné.

2.5 Virtualizační prostředí

Protože v této kapitole byla uvedena celá řada způsobů, jak je možné virtualizaci provádět, v následující části bude zmíněno několik nejznámějších produktů, které virtualizaci umožňují. Vybrány jsou tak, aby byly pokryty různé typy virtualizace, oba typy hypervizorů a rozdílné operační systémy, na kterých software běží.

QEMU

Podporovaný OS	Typ virtualizace	Hypervizor
Windows, Linux a MacOS	Emulace a plná virtualizace	Typ 2

QEMU je multiplatformní software pro emulování hardwaru. Dokáže simulovat celou škálu procesorů, také síťové karty, usb připojení, grafické karty a další. Nabízí dva hlavní režimy, kdy v jednom umožňuje pouze běh binárního souboru určený pro konkrétní architekturu a ve druhém je dostupný i jako hypervizor typu 2. Pro plnou virtualizaci ale většinou nebývá jeho hypervizor použit, protože neumožňuje využití hardwarové akcelerace v případě, že na hostitelském i virtuálním stroji je tentýž procesor. [2]

KVM

Podporovaný OS	Typ virtualizace	Hypervizor
Linux	Plná virtualizace	Nelze určit

Kernel-based virtual machine (zkráceně KVM) je modul jádra Linuxu, který kernelu přidává možnost chovat se zároveň jako hypervizor. Z tohoto důvodu je poměrně náročné zařadit tento software mezi představené typy hypervizorů. Běží jako součást jádra, a proto má přímý přístup k hardware, zároveň ale umožňuje neomezený běh operačního systému a v různých odborných diskuzích se vedou spory, kam jej zařadit. KVM se používá v kombinaci s QEMU, které nabízí široké množství emulovaného hardwaru a knihovnou libvirt, která se používá u více virtualizačních technologií na systémech Linux a poskytuje sjednocené rozhraní pro jejich správu.

Oracle VM VirtualBox

Podporovaný OS	Typ virtualizace	Hypervizor
Windows, Linux a MacOS	Plná virtualizace a paravirtualizace	Typ 2

Jde zřejmě o jeden z nejznámnějších virtualizačních programů, se kterým se v tomto odvětví mnoho lidí setkává jako první. VirtualBox je software umožňující vytvářet virtuální stroje za pomocí hardwarově akcelerované plné virtualizace a dokáže i paravirtualizaci s množstvím emulovaných komponent, který vyvíjí společnost Oracle.

VMware vSphere

Podporovaný OS	Typ virtualizace	Hypervizor
-	Plná virtualizace	Typ 1

VMware vSphere je jedním z mnoha produktů společnosti VMware a právě tento je zde zmíněn, protože se jedná o hypervizor typu 1, který se instaluje přímo na server. Jde o profesionální virtualizační software umožňující například migraci virtuálního stroje z hostitele na hostitele, vše se odehrává v příjemném grafickém rozhraní vSphere klienta, který beží buď jako webová služba nebo je k dispozici jako zvlášť instalovaný program.

Hyper-V Server

Podporovaný OS	Typ virtualizace	Hypervizor
-	Plná virtualizace	Nelze určit

Hyper-V Server je produkt od společnosti Microsoft, který je postavený na jádru systému Windows Server 2008, ale s omezenou funkcionalitou klasického Windows serveru a je k dispozici zdarma. Jedná se o konkurenční produkt k produktům společnosti VMware, Hyper-V je také možno užívat jako rozšíření na operačních systémech Windows Server a uživatelských Windows edice Pro. Zde ovšem opět jako u KVM nelze jednoznačně určit typ hypervizoru.

LXC

Podporovaný OS	Typ virtualizace	Hypervizor
Linux	Virtualizace na úrovni OS	-

LXC je kontejnerová služba, která funguje na systému Linux. Vývoj je podporován společností Canonical, která vyvíjí linuxovou distribuci Ubuntu.

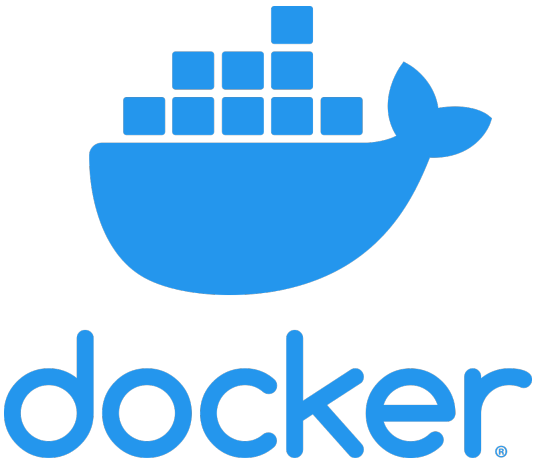
Docker

Podporovaný OS	Typ virtualizace	Hypervizor
Windows, Linux a MacOS	Virtualizace na úrovni OS	-

Docker je velmi oblíbenou kontejnerovou službou. Skládá se z různých částí, na Linuxu vychází z LXC a je kompatibilní i s Windows 10. Více je probrán v kapitole 3.

Kapitola 3

Docker



Obrázek 3.1: Oficiální logo Dockeru. [7]

Docker je v dnešní době velmi populární software, který umožňuje virtualizaci na úrovni operačního systému. Jeho oblíbenost je zapříčiněna kompatibilitou mezi různými operačními systémy (většina ostatních kontejnerových aplikací běží pouze na Linuxu), rozsáhlou komunitou uživatelů a propracovaným manuálem. K dispozici je k používání ve dvou variantách a to jako Docker Community Edition (zkráceně Docker CE) a Docker Enterprise Edition (Docker EE). Obě verze vycházejí v základu ze stejných zdrojových kódů, rozdílů je zde ale hned několik. Docker CE je open-source software a tudíž je k dispozici zdarma. Je možné jej nainstalovat na Windows 10, Mac OS a na linuxové distribuce CentOS, Debian, Fedoru, Raspbian, Ubuntu a jejich deriváty s různými architekturami procesorů. Docker EE je zaměřen na profesionální podnikové nasazení, které obsahuje několik dalších nástrojů pro snadnější používání, větší zabezpečení a prioritní neomezenou podporu. V rámci této verze přibývá podpora dalších operačních systémů, jako jsou RHEL, Oracle Linux a Windows servery. Dále v této práci se zaměříme jen na edici CE. [8]

O správu kontejnerů se stará program Docker Engine, který je zodpovědný za veškeré operace s kontejnery a jejich prostředím. Jak už bylo zmíněno ale dříve, Docker mimo to nabízí ještě další služby, bez kterých by spouštění kontejnerů bylo funkční, zároveň ale určitým způsobem vylepšují jeho používání. Většina z nich, Docker Engine a tvorba imageů bude rozebrána na následujících řádcích.

3.1 Docker Engine

Docker Engine je nejdůležitější částí Dockeru. Jde o aplikaci typu klient-server, serverová část běží jako démon na hostiteli a zpřístupňuje REST API[1], což je důležité a později to bude ještě několikrát zmíněno, protože mimo běžného klienta tak Docker můžeme ovládat přes síť a propojovat s aplikacemi, které Docker podporují. Klient, který je po instalaci Dockeru na hostiteli dostupný, využívá ke komunikaci linuxový socket. Pokud je potřeba zpřístupnit API v síti, je nutné upravit konfiguraci, protože je to ve výchozí podobě zakázáno. Pokud je API zpřístupněna pomocí TCP socketu, je třeba se chovat velmi obezřetně, protože v API standardně není zapnuta žádná forma autentizace, a tak může kdokoliv ve stejné podsíti Docker Engine na hostiteli ovládat.

Správa kontejnerů

Engine spravuje kontejnery, které mohou být ve stavu vytvořený, běžící, zastavený, skončený a mrtvý. Rozdíl mezi skončeným a mrtvým je v tom, že v prvním případě kontejner skončí úspěšně a v druhém kvůli libovolné chybě. Seznam a stavy jednotlivých kontejnerů mohou být zobrazeny po zadání následujícího příkazu. Z něj zjistíme základní informace o kontejnerech běžících na hostiteli jako je id kontejneru, datum vytvoření, jméno, jméno image, ze kterého je kontejner spuštěn a mapování portů, jsou-li nějaké otevřeny.

```
docker ps -a
```

Spouštění kontejnerů

Každý kontejner má definovaný image[2], ze kterého je spuštěn. Image u Dockeru je šablonou pro spouštění určitého typu kontejnerů, kontejner je pak jeho instancí. To znamená, že je možné spouštět libovolný počet kontejnerů ze stejného image. K imageům se dá nejjednodušeji dostat pomocí služby Docker Hub[3], která je jejich největším veřejným úložištěm. Je zde možné nalézt šablony pro libovolné využití, zastoupení zde má veškerý známý software, nalezneme tady třeba jednotlivé linuxové distribuce nebo databázové a webové servery. Protože ale do tohoto úložiště mohou nahrávat data libovolní uživatelé, je nutné být s výběrem opatrný a volit primárně takové image, které jsou označeny jako oficiální. Ty spravují přímo vývojáři Dockeru anebo autoři daného softwaru. Pokud zvolíme neoficiální image, riskujeme tím, že uvnitř může být nainstalovaný nějaký nevyžádaný program. Před dvěma lety se například objevily šablony, na kterých byla aplikace těžící kryptoměnu Monero a které si stáhlo až pět milionů uživatelů. [9] Druhou možností je pak vytvořit si svůj vlastní image, což je výhodnější, protože může být předchystán zcela dle vlastních požadavků.

Spouštění kontejnerů se provádí pomocí příkazu *docker run* doplněný o různé argumenty. Jediný povinný argument je jméno image, ze kterého bude kontejner spuštěn. Pokud šablona není na hostiteli k dispozici, Docker Engine se ji při startu kontejneru pokusí nejprve stáhnout a kontejner spustí až následně. Bez dodatečných přepínačů se kontejner spustí na popředí terminálu bez možnosti interakce, proto má smysl přidat buď přepínač *-d* pro běh kontejneru na pozadí nebo přepínače *-i -t* pro interakci s kontejnerem na popředí.

```
docker run --name web_server -d --rm httpd
```

[1] REST API – aplikační interface zpřístupňující některé metody a funkce pro ovládání jinými programy přes síť pomocí protokolu HTTP/HTTPS.

[2] Image – toto cizí slovo bude dále v textu skloňováno podle mužského neživotného rodu.

[3] Docker Hub – https://hub.docker.com/

Příkaz z příkladu by spustil kontejner s imagem webového serveru Apache na pozadí. K tomu je přidáno jeho jméno (Docker Engine pojmenovává kontejnery po významných vědcích, pokud jméno není uvedeno) a přepínač *--rm*, který zajistí odstranění kontejneru po tom, co bude zastaven. K zastavení a mazání se používají příkazy *docker stop* a *docker rm*. První po zadání jména kontejneru nebo jeho identifikátoru kontejner uvede do stavu zastavený. Takový kontejner se dá spustit znovu. Druhý příkaz, se stejnými možnostmi pro argument, zastavený kontejner smaže.

Specifická nastavení pro konkrétní image

Spousta šablon kontejnerů je vytvořena tak, že je možné upravovat jejich chování podle potřeby uživatele. Toho je možné dosáhnout dvěma způsoby. Prvním z nich je předávání proměnných prostředí[4], které se definují pomocí přepínače *-e*. Na Docker Hubu je u imageů v dokumentaci vždy uvedeno, jaké proměnné prostředí se používají a k čemu slouží. Ty se pak využijí při běhu kontejneru k nastavení požadovaných vlastností. Druhou možností je předání argumentů kontejneru. Každý image pro Docker má uvedenou cestu k programu, který se spustí po startu kontejneru. Kontejner se pak ukončí tehdy, když skončí vykonávání uvedeného programu. Ten může zpracovávat argumenty, které je možné úvést zcela na závěr příkazu *docker run* po uvedení jména image.

```
docker run --name db_server -d -e MYSQL_DATABASE=moje_db -e MYSQL_ROOT_PASSWORD=123456 \
mariadb
```

V příkladu je příkaz ke spuštění kontejneru s databází MariaDB s definovaným jménem databáze, která je vytvořena po jeho startu a heslem pro hlavního uživatele.

Sítě v Dockeru

Dosud uvedené příklady spouštění kontejnerů ale moc užitečné nejsou, protože v nich zatím nebyla uvedena žádná část, která by řešila zpřístupnění serverové aplikace. Pokud to není explicitně uvedeno, neprovede se žádné mapování portů mezi kontejnerem a hostitelem a služba je pak dostupná pouze v rámci běžícího kontejneru a oddělené sítě, kterou spravuje Docker Engine. Otevření je možné docílit pomocí přepínače *-p* nebo *--publish*, který má hned několik možností, jak požadované porty specifikovat. Jednou z možných variant, jak zadat hodnotu přepínače, je uvést jen číslo portu nebo rozsah více portů, které používají aplikace uvnitř kontejneru. Docker Engine pak sám vybere volné porty na hostiteli a provede mapování. Vybrané porty je pak možné vypsat třeba výše uvedeným příkazem *docker ps*. Do přepínače dále můžeme uvést ale i IP adresu, přesné číslo/čísla portů, které se mají využít na hostiteli a typ protokolu, který je použit na transportní vrstvě.

```
docker run --name jenkins_server -p 8080:8080 --rm -d jenkins
```

V tomto případě spouštíme kontejner s nástrojem pro automatizaci úloh zvaný Jenkins, který funguje jako webová služba. Prvním číslem uvádíme port, který má být zpřístupněn na hostiteli a druhý je ten, kterým se určuje port z kontejneru. Protože v přepínači není uvedena IP adresa hostitele, zvolí se adresa 0.0.0.0, která by měla zajistit, že port bude dostupný na všech síťových rozhraních hostitele včetně localhostu.

Mapování portů v Dockeru funguje v závislosti na použitém typu sítě, kterých je k dispozici pro kontejnery více. Pokud síť v příkazu *docker run* nijak neupřesníme, Docker Engine

[4]Proměnná prostředí – je proměnná, která je přístupná v běhovém prostředí operačního systému.

připojí kontejner do výchozí sítě, jejíž typ je v terminologii Dockeru označován jako *bridge*. Sítě typu *bridge* jsou nezávislé na sítích, do kterých je připojen hostitel. Mají definovaný rozsah IP adres, spuštěný DHCP server[5] a poskytují vlastní DNS službu[6]. Mapování portů pak probíhá vytvářením pravidel v iptables[7], které spravuje Docker Engine po vytvoření kontejneru. V tabulce 3.1 je vypsán kompletní výčet všech typů sítí, které Docker podporuje.

Bridge	Výchozí typ sítě, která je izolovaná od sítí hostitele.
Overlay	Síť, která nabízí stejné možnosti, jako typ *bridge*. Rozdíl je v tom, že může být sdílena mezi různými hostiteli, pokud jsou spojeni do clusteru pomocí nástroje Docker swarm. Kontejnery pak mají společnou síť, ve které mohou komunikovat, přestože jsou fyzicky na různých počítačích.
Host	Tento typ sítě odstraňuje izolaci mezi hostitelem a běžícími kontejnery, které dostanou přístup ke stejné síti, do které je hostitel připojen. Pro tento typ sítě nemá přepínač *-p/--public* význam, protože nedochází k žádnému mapování portů.
Macvlan	Speciální typ sítě, který umožňuje kontejnerům komunikovat na linkové vrstvě modelu ISO/OSI. Používá se například u nástrojů k monitorování sítě, které potřebují přímý přístup k síťovému provozu.

Tabulka 3.1: Tabulka typů sítí v Dockeru.

Sítě různých typů mohou být vytvářeny v libovolném množství, pro jejich správu je možné využít příkaz *docker network*.

```
# Vypíše seznam definovaných sítí na hostiteli.
docker network ls

# Vytvoří síť my_network s uvedenou adresou a maskou. Přepínač -d určuje typ sítě a
# jeho použití není povinné, standardně by se použil právě typ bridge.
docker network create -d bridge --subnet 192.168.1.0/24 my_network

# Zobrazí veškeré informace k síti my_network. Výstup je ve formátu JSON.
docker network inspect my_network
```

Spouštění kontejneru se specifikovanou sítí se pak provádí pomocí přepínače *--network* v příkazu *docker run*. Následující příkaz by spustil kontejner CentOS s IP adresou z dříve vytvořené sítě „my_network“.

```
docker run --name centos --network my_network -it --rm centos
```

Kontejnery se mezi sebou dají propojit, klasickým příkladem je případ, kdy je v jednom kontejneru web a ve druhém databáze. Propojení se vytváří pomocí přepínače *--link*, kde se určuje jméno kontejneru a jeho alias. Kontejnery k sobě samozřejmě přístup mají (za předpokladu, že jsou ve stejné síti) a dokáží spolu komunikovat pomocí IP adres, přepínač ale způsobí úpravu souboru „/etc/hosts“[8], kde doplní IP adresu ke jménu i aliasu kontejneru.

[5] DHCP – je název síťového protokolu a také počítačový program typu klient-server pro automatickou konfiguraci IP adres a dalších náležitostí, které připojeným zařízením umožní síťovou komunikaci.

[6] DNS – je síťový protokol a rovněž počítačový program typu klient-server, který překládá doménová jména na IP adresy a opačně.

[7] iptables – jde o softwarový firewall, který je použit v mnoha linuxových distribucích.

[8] /etc/hosts – tento soubor slouží v linuxových operačních systémech k přiřazení jména k IP adrese. Funguje tak jako ten nejjednodušší DNS server, který je dostupný pouze v rámci operačního systému.

Mimo to také zajistí vytvoření dalších systémových proměnných, ve kterých je například informace o IP adrese propojeného kontejneru nebo portech, které využívá.

```
docker run --name db_server -d --rm -e MYSQL_ROOT_PASSWORD=123456 mariadb
docker run --name web_server -p 8080:80 --link db_server:mariadb -d --rm httpd
```

V příkladu výše se spustí nejprve databázový kontejner s jednou proměnnou prostředí, která je pro tento konkrétní image povinná. Následně se spouští webový server, který je propojen s databází. Pokud bychom teď někde v kontejneru konfigurovali adresu databáze, můžeme využít jmen „db_server“ nebo „mariadb“.

Připojování souborů z hostitele

Poslední důležitou věcí u spouštění kontejnerů je možnost zpřístupnit uvnitř umístění, které je na hostiteli nebo i na nějakém externím serveru. Všechny kontejnery, které ukládají libovolná důležitá data, by byly k ničemu, pokud by nebylo možné mít je někde uložené perzistentně. Pokud je totiž kontejner jednou odstraněn, nezůstane po něm vůbec nic. Pro příkaz *docker run* se pro tyto účely používá přepínač *-v* nebo *--volume*. Podobně jako přepínač pro zpřístupnění portů má více možností zápisu, v celé podobě se zadává umístění na hostiteli, umístění uvnitř kontejneru a práva přístupu.

```
mkdir /tmp/my_dir
docker run --name debian -v /tmp/my_dir:/data:ro -it --rm debian
```

Příkaz výše by v kontejneru Debian vytvořil umístění „/data“ a připojil cestu z hostitele „/tmp/my_dir“, která by byla pouze pro čtení. Práce jde ale usnadnit a pokud nejsou žádné požadavky na přesné umístění v rámci hostitele, můžeme použít příkaz *docker volume*. V základu se dá použít pro vytvoření lokálního umístění, pomocí různých ovladačů dokáže ale i zpřístupnit vzdálená úložiště připojená například pomocí SSH nebo NFS[9].

```
# Vypíše seznam definovaných svazků.
docker volume ls

# Vytvoří svazek my_volume. Přepínač -d určuje typ svazku a jeho použití není povinné,
# standardně by se použil právě typ local.
docker volume create -d local my_volume

# Zobrazí veškeré informace ke svazku my_volume. Výstup je ve formátu JSON.
docker volume inspect my_volume
```

Vytvořené svazky jsou standardně umístěny v adresáři „/var/lib/docker/volumes/“ a v případě potřeby je možné se dovnitř z hostitele dostat. Rozhodně ale není vhodné složky v tomto umístění mazat, Docker Engine by se svazkem stále počítal a docházelo by k chybám. Pro tyto potřeby je vhodné použít příkaz *docker volume rm*. Pokud bychom chtěli kontejner s vytvořeným svazkem spustit, hodnota argumentu *-v* se nebude příliš lišit.

```
docker run --name debian -v my_volume:/data -it --rm debian
```

[9] Network File System – jedná se o protokol pro připojování souborových systémů přes síť.

3.2 Dockerfile

Jak již bylo uvedeno, druhou možností, jak získat image pro spouštění kontejneru, je si vytvořit vlastní. Vlastní image se dá připravit pomocí speciálního souboru, který se nazývá Dockerfile. Ten obsahuje instrukce pro jeho vytváření a má vlastní strukturu a syntaxi. Dockerfile musí být ve složce, která je vyhrazená přímo pro tvorbu konkrétního image. V manuálu Dockeru je toto umístění označováno jako *build context*. Často se v něm vyskytují i další soubory, které budou kopírovany do připravovaného image. Mimo vyhrazenou složku nelze kvůli bezpečnosti soubory do image zkopírovat. Je to z důvodů, kdyby byl použit nějaký stažený Dockerfile a byl spuštěn build image, nestane se, že by se do něj dostal libovolný citlivý soubor z hostitele.

První věc, která je třeba uvést v každém takovém souboru, je zdrojový image. Každý image, který je vytvářen, má svého rodiče. Díky tomu je práce velmi jednoduchá. Pokud by byla například potřeba vytvořit image s webovým serverem Apache, není nutné se zaměřovat na nic jiného, než na instalaci tohoto softwaru. Stačí si jen vybrat image na Docker Hub a použít jej jako rodičovský. Příklad souboru Dockerfile pro jednoduchý image s webovým serverem by mohl vypadat následovně.

```
FROM debian

EXPOSE 80

RUN apt-get update
RUN apt-get install -y apache2

ENTRYPOINT ["apachectl", "-D", "FOREGROUND"]
```

Výpis 3.1: Ukázka jednoduchého souboru Dockerfile pro tvorbu image s webovým serverem.

V souboru je instrukce *FROM*, která specifikuje zdrojový image. V tomto případě se vychází z distribuce Debian. Image jednotlivých distribucí jsou velmi odlehčené a obsahují jen zcela základní software. Protože se předpokládá, že v Docker kontejnerech poběží pouze jediná služba, image distribucí neobsahují ani žádný init proces, který by spravoval ostatní běžící procesy (přesto však existují image, které jsou ještě minimalističtější jako je Scratch nebo BusyBox). Dále je zde instrukce *EXPOSE*, která informuje o tom, že image obsahuje službu, která využívá port 80, ale nemá žádný přímý vliv na mapování portů. Pak jsou zde dvě instrukce pro spuštění terminálového příkazu, první pro aktualizaci balíčkovací služby a druhá pro instalaci web serveru. Soubor končí instrukcí *ENTRYPOINT*, do které se uvádí příkaz, který se vykonává po dobu, co je kontejner spuštěn. Přehled dalších důležitých instrukcí je k dispozici v tabulce 3.2.

Image se vytváří pomocí příkazu *docker build*. Jako argument se udává cesta k adresáři *build context* a přepínač *-t* nebo *--tag* pro pojmenování image. Jména imageů mají předepsaný vzor, který se skládá z namespace, jména repozitáře a verze. Namespace je označení pro kategorii imageů, může se jednat například o kolekci uživatele nebo o soubor imageů, které se používají v konkrétním oddělení společnosti. Repozitářem je myšleno jméno konkrétního image a od namespace se odděluje pomocí lomítka. Verze žádnou přesnou konvenci nemá a je na autorovi image, jak ji zvolí. Od repozitáře se odděluje dvojtečkou a pokud se neuvede, použije se výchozí „latest“.

```
docker build -t user/web_server:0.1 -t user/web_server .
```

FROM	Definuje zdrojový image.
LABEL	Přidává doplňující informace k imagi, které jsou k dispozici při výpisu podrobností u vytvořeného kontejneru.
EXPOSE	Používá se pro označení portů, které používá služba uvnitř kontejneru. Instrukce má především informační hodnotu a neovlivňuje nijak nastavení mapování portů.
ENV	Udává se jako *KLÍČ=HODNOTA*. Definuje proměnné prostředí uvnitř kontejneru.
VOLUME	Připojuje svazek do kontejneru. Z bezpečnostních důvodů ale není možné zadat cestu na hostiteli. Tato možnost je pouze při spouštění kontejneru pomocí *docker run*.
COPY	Zkopíruje soubor z hostitele do kontejneru. Je zde i možnost upravit souboru přístupová práva.
RUN	Provede příkaz v shellu.
WORKDIR	Výchozí cesta po startu kontejneru.
USER	Přepnutí do prostředí jiného uživatele. Další instrukce se pak vykonávají pod nim.
ENTRYPOINT	Program, který je spouštěn při startu kontejneru.

Tabulka 3.2: Tabulka typů sítí v Dockeru.

Příklad příkazu *docker build* výše by vytvořil image se dvěma stejnými jmény a rozdílnou verzí. Pokud by byl vytvořen pouze s označením „user/web_server:0.1“, nebylo by možné spouštět jeho kontejnery jen pomocí *docker run user/web_server*, ale musela by být vždy zadána i jeho verze. To není praktické především pro uživatele, kteří nemají přehled o verzích image a spoléhají se na to, že poslední verze je ta, kterou potřebují. Jde o běžnou praxi, kdy je možné docílit, že poslední vytvořený image bude vždy odpovídat i verzi „latest“, což je verze, která je výchozí. Příkaz očekává, že zdrojový Dockerfile je umístěn v aktuálním adresáři, což je uvedeno pomocí tečky na konci.

K pochopení, co se přesně děje, když se z image pomocí příkazu *docker run* spouští kontejner i k tomu, co se děje v průběhu buildu image se hodí zmínit, co je to Union File System. UFS umožňuje překrývání více souborových systémů, z nichž je ve stejné cestě dostupný jen ten, který byl připojen jako poslední. Docker podporuje různé implementace UFS a je možné je v nastavení změnit. Image Dockeru využívají UFS a jsou tvořeny z jednotlivých vrstev souborových systémů, které jsou pouze pro čtení. Každá instrukce vytváří novou vrstvu image (to platí i pro instrukce stejného typu) a pokud je z image spouštěn kontejner, jako poslední se připojí read-write vrstva. [11] Z těchto důvodů je při vytváření souborů Dockerfile častá snaha o úsporu a použití menšího množství instrukcí. Nejlépe se dá šetřit u instrukce *RUN*, kde je nejvhodnější příkazy za sebou řetězit, než pro každý použít instrukci novou.

3.3 Docker registry

Docker registry je obecný název pro úložiště imageů pro Docker. V jedné z předchozích částí již bylo zmíněno veřejné úložiště Docker Hub, které je jedním z příkladů konkrétní implementace registry. Z mnoha různých důvodů může ale vzniknout potřeba nemít své vytvořené image na veřejném úložišti a mít vlastní, které by bylo dostupné například jen

z podnikové sítě. K dispozici je řešení přímo od vývojářů Dockeru v podobě předchystaného image s názvem Docker Registry. Toto řešení je však v základu velmi jednoduché, protože neobsahuje žádné uživatelské rozhraní pro správu imageů. Jedná se pouze o backend[10], který umožňuje nahrávání a stahování imageů pomocí REST API, se kterým umí komunikovat Docker klient. Také zde není implementována žádná forma autentizace uživatelů. Existuje ale několik projektů, které funkci tohoto základního image rozšiřují.

Pro správnou funkci Docker Registry je potřeba mít pro tuto službu nastavené doménové jméno a SSL/TLS certifikát[11]. Pokud budeme chtít do registry něco nahrát a bude podporovat autentizaci uživatelů, je nutné se do něj přihlásit pomocí příkazu *docker login*.

```
docker login registry.example.com
```

Po spuštění příkazu se objeví výzva k zadání uživatelského jména a hesla a nakonec se zobrazí zpráva o potvrzení přihlášení. Odhlášení se provede pomocí příkazu *docker logout*. Pokud by nebyla zadána adresa registry, Docker standardně doplní adresu pro Docker Hub.

Pro nahrávání image je třeba do jeho jména zahrnout i doménové jméno registry, tímto způsobem se pak rozlišuje, kam se má nahrávat. K příkladu bude využit image web serveru z předchozí části, kterému bude přiřazeno další jméno a následně bude nahrán do registry.

```
# Vytvoření nového jména pro image obsahující adresu vlastního Docker registry.
docker tag user/web_server registry.example.com/user/web_server

# Nahrání image do zvoleného Docker registry.
docker push registry.example.com/user/web_server
```

3.4 Docker Compose

Docker Compose je program, který usnadňuje spouštění kontejnerů. Často pro kontejnery, které spouštíme, bývá příkaz *docker run* poměrně obsáhlý, protože obsahuje veškeré nastavení portů, systémových proměnných, svazků, sítě a dalších. Pokud je potřeba jich pak ještě spustit několik a provázat mezi sebou, příkaz *docker run* přestává být pohodlný. Docker Compose tuto práci dělá sám a vyčítá konfiguraci ze souboru s formátem YAML[12]. Výše v části 3.1 byla zobrazena pasáž s propojením kontejneru s databází a webovým serverem, zde je, jak by to vypadalo převedeno do formátu YAML určeného pro Docker Compose.

```
version: "2"
services:
    db_server:
        image: mariadb
        environment:
            MYSQL_ROOT_PASSWORD: "123456"
    web_server:
        image: httpd
        ports:
            - 8080:8080
        links:
            - db_server
```

Výpis 3.2: Ukázka obsahu souboru „docker-compose.yml“.

[10] Backend – je serverová vrstva aplikace, která operuje s daty.
[11] SSL/TLS – jsou protokoly umožňující šifrovanou komunikaci na transportní vrstvě.
[12] YAML – formát pro serializaci dat. Mezi jeho hlavní výhody patří, že je velmi dobře pro člověka čitelný.

Pro spuštění programu Docker Compose je nutné být ve stejném adresáři, jako je soubor s konfigurací. Ten musí být vždy nazván jako „docker-compose.yml“. K dispozici jsou pak možnosti spuštění, vypnutí a prohlížení logů kontejnerů.

```
# Spustí kontejnery z konfigurace. Přepínač -d zajistí, že se na popředí nebudou
# vypisovat logy kontejnerů.
docker-compose up -d

# Zastaví kontejnery.
docker-compose down

# Zobrazuje logy kontejnerů. Přepínač -f zajistí, že se budou vypisovat i budoucí logy.
# Jde o stejnou funkci jako zajišťuje přepínač -F u programu tail.
docker-compose logs -f
```

3.5 Docker swarm

Do této chvíle byly veškeré operace s kontejnery zmíněné v této kapitole jen v rámci jednoho počítače. Existují ale i nástroje pro spouštění a škálování kontejnerů napříč více počítači, které jsou propojeny do clusteru[13]. Jednou z možností je Docker swarm, který je součástí hlavnho programu Docker Engine. Jednotlivé výpočetní uzly ve swarmu jsou označeny jako manager nebo worker. Manager je oprávněn plánovat služby, měnit stavy jednotlivých uzlů a rovněž má veškerá práva jako uzly typu worker. Worker má velmi omezené možnosti. Jde o výpočetní uzel, který nemůže rozhodovat o plánování a jen je na něm spouštěn běh kontejnerů. Obou typů může být libovolné množství.

Ve swarmu se nespouští jednotlivé kontejnery, jako se to děje při použití příkazu *docker run*. Místo toho se definují služby. Službám se nastavují podobné parametry, odlišuje je ale možnost škálování. Při vytváření služby nebo za jejího běhu udáváme požadovaný počet kontejnerů stejného typu, které se mají spustit. Kontejnery jsou pak rozplánovány mezi jednotlivé uzly swarmu. Výchozí plánování probíhá velice jednoduše. Swarm se snaží dosáhnout podobného počtu kontejnerů od stejné služby na všech uzlech, pokud ale máme definovaných služeb více, na celkový počet spuštěných kontejnerů na jednom uzlu se nebere ohled. Plánování může být ale vylepšeno dvěma způsoby. První je s ohledem na využití výpočetních prostředků. Při vytváření služby můžeme uvést jaká je potřebná paměť a počet procesorových jader pro jeden kontejner. V této chvíli plánovač již nespouští kontejnery různých služeb náhodně mezi uzly, ale s ohledem na jejich celkový a využitý výkon. [4] Druhý způsob je přidávání restrikcí, kde se kontejnery mohou spouštět. Lze omezovat spouštění například podle hostname, operačního systému nebo i parametrů definovaných uživatelem pro každý uzel. Swarm se dá také použít pro vysokou dostupnost (v literatuře se běžně vyskytuje pod zkratkou HA z anglického High Availability). Cluster reaguje na výpadky jednotlivých uzlů a služby aktivně přeplánovává.

Ve výchozím nastavení swarm poskytuje také load balancer[14] označovaný jako *routing mesh*. Ten umí zajistit rozložení požadavků mezi více kontejnery jedné služby napříč různými uzly clusteru. Jeho použití je velmi praktické a dokáže i přesměrovat provoz z uzlu, kde kontejner ve skutečnosti neběží. Pokud by swarm například obsahoval tři uzly a byla na něm spuštěna služba web serveru se dvěma instancemi kontejnerů, web server bude dostupný z IP adresy každého uzlu, i když bude existovat jeden, kde kontejner neběží.

[13] Cluster – je skupina počítačů, které spolu spolupracují.
[14] Load balancer – software, který rozkládá síťový provoz mezi víc počítačů se stejnou službou.

Kapitola 4

Ansible

Obrázek 4.1: Oficiální logo nástroje Ansible. [13]

Ansible je jedním z nástrojů pro automatizaci úloh v IT prostředí. Jde o open-source program napsaný v jazyce Python, jehož vývoj podporuje společnost Red Hat. Má široké možnosti využití, používá se pro konfiguraci systémů a instalaci softwaru. Mezi jeho hlavní výhody patří, že je multiplatformní a pro jeho provoz není třeba instalovat žádného agenta[1] na cílových počítačích. Toho je dosaženo tak, že se Ansible z počítače, kde je spuštěn, připojí na počítač, který má být konfigurován, pomocí SSH, kde pod přiděleným uživatelem přímo provádí změny.

Stejně jako u Dockeru zde existují dvě verze. Ansible community, která je vyvíjena komunitou přispěvatelů a Ansible Engine, která je placená a spravována společností Red Hat pro podnikové nasazení s technickou podporou. Mezi další výhody placené verze patří podpůrné nástroje jako je Ansible Tower, který tomuto softwaru přidává webové uživatelské rozhraní pro přehlednější ovládání nebo nástroje pro analytické úkony. I zde bude nadále probírána jen verze Ansible community. Ansible v obou verzích může být instalován na velké množství linuxových distribucí jako je CentOS, RHEL, Debian a Ubuntu, ale i další a na Mac OS. Na Windows se nainstalovat nedá, ale dokáže jej spravovat.

Pokyny pro Ansible se definují v souborech ve formátu YAML označované jako playbooks. Playbook obsahuje soubor úloh, které se mají provést a skupiny hostů, na kterých

[1] Agent – je software, který běží na pozadí počítače a čeká na povely. Může poskytovat různé běhové informace nebo i ovlivňovat prostředí, kde běží.

se mají úlohy vykonat. Úloha (v Ansible dokumentaci se označuje jako task) obsahuje zvolený modul a jeho argumenty. Modul na hostu vykonává akci, pro kterou je určen. Jde o obálku nad klasickými příkazy operačních systémů, jejichž výhoda spočívá v tom, že řeší nekompatibilitu softwaru napříč systémy (jedinou výjimku tvoří moduly pro Windows, které jsou v některých případech k dispozici zvlášť). Ansible nabízí velmi bohatou knihovnu modulů[2], které jsou určeny od běžných činností, jako je třeba kopírování a změna práv souborů, po velmi specificky zaměřené moduly na konkrétní služby, mezi které patří například build imageů pro Docker. Díky jeho komplexnosti má Ansible velmi blízko k programovacím jazykům. K dispozici je větvení a rozhodování podle celé řady skutečností, proměnné a parametrizace úloh a cykly v podobě iterace nad definovanou datovou strukturou.

4.1 Inventory

Po instalaci Ansiblu na počítač, odkud budou ostatní počítače spravovány, je třeba nakonfigurovat hosty do jeho inventáře. Inventář je seznam hostů rozdělených do skupin. Díky tomu je možné tématicky rozlišit tyto uzly třeba podle využití, což následně pomůže jednodušší definici uvnitř playbooku. Klasické umístění je v souboru „/etc/ansible/hosts“, je možné si ale inventář vytvořit i jinde a pak jej pomocí argumentu předat příkazu pro spuštění playbooku. Soubor „hosts“ je výjimkou a nemusí být ve formátu YAML, po instalaci Ansiblu je vyplněn příklady ve formátu INI[3]. Soubor v INI formátu obsahuje vždy jméno skupiny v hranatých závorkách a pod nim na dalších řádcích definice uzlů s jejich případným nastavením.

```
[local]
localhost ansible_connection=local

[web_servers]
web1
web2

[kvm_servers]
192.168.0.100
```

Výpis 4.1: Ukázka obsahu souboru „/etc/ansible/hosts“ ve formátu INI.

Jak je z příkladu patrné, uzel se dá do skupiny začlenit pomocí jeho doménového jména anebo IP adresy. Mimo to můžeme k hostům přidat další informace, jako je to u první skupiny. Ve skupině „local“ je počítač „localhost“, což znamená, že se jedná o počítač, na kterém je Ansible nainstalován. Pomocí klíče *ansible_connection* je zde nastaveno, že se při konfiguraci k němu Ansible nemá připojovat přes SSH, ale má příkazy provádět rovnou. Mezi další nastavení je možné předat například jméno uživatele, pod kterým se Ansible připojí k SSH, SSH port, cestu k interpretru Pythonu a podobně. [10]

Pro jednoho administrátora nebo jejich malou skupinu je toto použití zcela dostatečné. Je vhodné ale uvést, že pro masivní nasazení existuje i řešení označované jako *dynamic inventory*, které umožňuje načítat seznamy uzlů z externích zdrojů. Ansible pak může volat skript pro načtení hostů z databáze namísto načítání statického souboru. [14] Tato funkce nebude dále v práci rozebírána.

[2] Moduly Ansiblu – https://docs.ansible.com/ansible/latest/modules/list_of_all_modules.html

[3] INI – jde o formát dat pro konfigurační soubory. Tyto soubory jsou nejčastěji použity v operačních systémech Windows.

4.2 Playbooks

Možností, jak v Ansible definovat playbook, je spousta a jejich kód může obsahovat rozličnou strukturu. V začátcích je nejjednodušší cestou vložit veškeré náležitosti do jednoho souboru, u komplexnějších úkonů se pak kód rozděluje, což bude nastíněno v části 4.3 o rolích v Ansible. Každý playbook obsahuje část, kde se uvádí skupina uzlů, na které bude vykonáván. K tomu je použit klíč *hosts*. Druhou podstatnou věcí, je klíč *tasks*, kde se definují úlohy. Část *tasks* je z pohledu struktury dat jedno pole, které obsahuje slovník[4] pro každou konkrétní úlohu. Úloha může být pojmenovaná pomocí klíče *name* a dále musí obsahovat jméno modulu, který bude použit. Do něj jsou pak předávány argumenty, pokud jsou třeba.

```
- hosts: kvm_servers
  tasks:
    - name: Print test message
      debug:
        msg: TEST
```

Výpis 4.2: Ukázka souboru simple_playbook.yml.

Playbook z příkladu by se pokusil připojit k uzlu ze skupiny „kvm_servers“ a vypsal by tam zprávu „TEST“ pomocí modulu *debug*. Toto bude ale fungovat, jen pokud bude vyřešena autentizace na vzdáleném počítači při připojování skrze SSH. Ansible standardně používá uživatele root, nejjednodušší možností je tedy povolit k němu na uzlu připojení a přidat mu veřejný klíč z uživatele na serveru, ve kterém Ansible spouštíme. Tím je zajištěno, že přístup k uživateli root nebude vyžadovat žádné heslo, ovšem skutečně jen z uživatele, jehož veřejný klíč byl rootovi na vzdáleném uzlu přidán. Tato cesta není pro zkoušení špatná, bezpečnější by ale bylo nemít možnost se na žádném počítači k privilegovanému uživateli vůbec přihlásit a ani k takovému, u jehož jména se dá předpokládat, že na počítači bude. Root tyto zásady porušuje obě, a tak tento způsob zvyšuje bezpečnostní riziko. V rámci implementace (kapitola 6) je uvedeno, jak Ansible používat bezpečněji.

```
# Kopírování veřejného klíče aktuálního uživatele do adresáře uživatele root na
# vzdáleném počítači.
ssh-copy-id root@192.168.0.100

# Spouštění příkladu.
ansible-playbook simple_playbook.yml
```

Proměnné a fakta

Proměnné se uvádějí jako klíč – hodnota a mohou obsahovat řetězce a čísla nebo i složitější struktury jako jsou pole a slovníky. Jak již bylo zmíněno, Ansible je velmi volný a umožňuje definovat proměnné na více místech. Proměnné se definují pod klíčem *vars*, který může obsahovat přímo playbook na úrovni klíčů *hosts* a *tasks* nebo i v rámci konkrétní úlohy.

V příkladu níže je uvedena jednoduchá úloha pro instalaci balíčku pomocí modulu *apt*. Playbook obsahuje proměnnou „package“, ve které je jméno balíčku. Tato proměnná je předána do argumentu v modulu. Aby Ansible v argumentu rozpoznal, že se jedná o proměnnou, je třeba ji obalit do dvou složených závorek. YAML běžně bere zadaný text jako řetězec (mimo případy, kdy je například hodnotou jen číslo a nic jiného), pokud ale text obsahuje speciální znaky, jako je právě složená závorka, musí být řetězec obalen uvozovkami.

[4] Slovník – v tomto případě mluvíme o asociativní poli v terminologii Pythonu.

```
- hosts: kvm_servers
  vars:
      package: htop
  tasks:
    - name: Install software
      apt:
        name: "{{ package }}"
```

Výpis 4.3: Příklad s využitím proměnné.

Není-li uvedeno jinak, v první fázi, kdy se Ansible připojuje k hostům, se z uzlů stahují informace, které mohou být využity k podmíněnému vykonávání úloh. V terminologii Ansiblu se o těchto datech mluví jako o faktech a k dispozici jsou například informace o názvu a verzi operačního systému, počtu procesorových jader, architektuře procesoru nebo počtu síťových rozhraní. Fakta jsou zpřístupněny ve speciální proměnné „ansible_facts“. Co všechno se dá o počítači zjistit, je možné vyzkoušet pomocí následujícího příkazu, který vypíše fakta pro počítač, na kterém je příkaz spuštěn.

```
ansible localhost -m setup
```

Časté využití je rozhodování na základě operačního systému. Pokud bude ve skupině hostů jeden počítač, na kterém je CentOS a druhý, kde je Debian, bude problém s instalací softwaru, protože obě distribuce Linuxu mají jiný balíčkovací systém a jinak pojmenované balíky. Rozhodování na základě jména operačního systému může vypadat následovně.

```
- hosts: web_servers
  tasks:
    - name: Install Apache on Debian
      apt:
        name: apache2
      when: ansible_facts['os_family']|lower == "debian"
    - name: Install Apache on CentOS
      yum:
        name: httpd
      when: ansible_facts['os_family']|lower == "redhat"
```

Výpis 4.4: Ukázka s instalací softwaru podle linuxové distribuce.

V příkladu se používá modul *apt* pro instalaci balíčku v distribuci Debian a *yum* pro CentOS (pro porovnání se používá řetězec „redhat“, který je nadmnožinou pro CentOS, dále je zde zastoupena i Fedora a další). Požadovaný software se definuje v argumentu *name*. Podmíněné spuštění řeší část *when*, která rozhoduje o provedení úlohy pouze v případě, že je vyhověno podmínce. Zde se přistupuje k faktu „os_family“ jehož volání odpovídá syntaxi Pythonu pro přístup k hodnotě ze slovníku. Druhou možností je přistoupit k faktu jako k atributu objektu pomocí *ansible_facts.os_family*. Za lomítkem je aplikován filtr, který zvolený řetězec převede na malá písmena.

Cykly

Úlohy v nástroji Ansible je možné vykonávat tolikrát, kolik je hodnot, pro které má být určitý modul použit. Toto chování se dá demonstrovat třeba pro případ, že je potřeba přesunout několik souborů s různými jmény z Ansible serveru do odlišných míst na hostech. Ansible narozdíl od klasických programovacích jazyků nabízí mnohem větší výběr cyklů. Všechny ale vycházejí z klasického typu označovaného jako *foreach*, což je speciální příklad

cyklu *for*, který iteruje nad položkami nějaké struktury a každé kolo umístí jeden prvek do stanovené proměnné. Důvodem, proč má Ansible cyklů více, je, že je každý určen pro iteraci nad konkrétní strukturou dat. Existuje cyklus pro procházení prvků z pole, slovníku, ale i komplikovanější s kartézským součinem dvou polí a podobně.

```
- hosts: web_servers
  tasks:
    - name: Copy files
      copy:
        src: "{{ item.src }}"
        dest: "{{ item.dest }}"
      with_items:
        - src: sshd_config,
          dest: /etc/ssh/sshd_config
        - src: site.conf
          dest: /etc/apache2/sites-available/site.conf
```

Výpis 4.5: Ukázka s iterací nad polem slovníků.

V příkladu je v úloze použit modul *copy*, který kopíruje soubory z počítače, kde Ansible běží, na hosty ze zvolené skupiny. Mezi základní argumenty patří *src*, který označuje umístění k souboru na serveru a *dest*, do kterého se předává umístění pro hosta. Iterace je definována v části *with_items*, do které je přiřazeno pole (položky pole jsou ve formátu YAML definovány odrážkou). Položka pole se v každé iteraci nahraje do proměnné *item*. Protože v definovaném poli je položkou slovník, je možné přistoupit k prvkům slovníku.

4.3 Roles

Role v Ansiblu jsou označení pro kategorie úloh. Koncept vychází z představy, že playbook by měl obsahovat jen informace o tom, co kde provést. Jakým způsobem se to ale provede se přenechá na rolích. Playbook se tak rozšíří o klíč s označením *roles*, kerým se přidávají jednotlivé kategorie úloh. Definice rolí s sebou přináší rozčlenění kusů kódu do souborů v rámci několika adresářů a může vypadat jako ve stromu níže.

```
playbook.yml
roles
└── web_server
    ├── files
    │   └── site.conf
    ├── tasks
    │   ├── centos.yml
    │   ├── copy.yml
    │   ├── debian.yml
    │   └── main.yml
    └── vars
        └── main.yml
```

Výpis 4.6: Příklad adresářové struktury pro soubory role v Ansiblu.

Pokud v souboru „playbook.yml“ bude uvedena role „web_server“, Ansible ví, že jednotlivé části role má hledat ve složce „roles/web_server“. Tato role by mohla obsahovat například úkoly k nainstalování webového serveru ve více systémech a přichystat prostředí pro konkrétní web. Uvnitř jsou adresáře *files*, *tasks* a *vars*, kde Ansible hledá data. Jejich jména je tedy nezbytné dodržet. Ve *vars* se definují proměnné, které chceme při provádění

role použít, do *files* přidáváme soubory, které v rámci role chceme kopírovat na spravované uzly a v *tasks* definujeme úkoly, které se mají vykonat. Existuje ještě více adresářů se speciálním významem, jejich použití ale není povinné a v rámci demonstrace rolí nebudou zmíněny. Ve složkách *vars* a *tasks* je vždy hledán soubor „main.yml“. U úloh je rozumné použít ještě rozsáhlejší dělení a v souboru „main.yml“ definovat jen další soubory, odkud se mají úlohy importovat. Součástí může být také třeba podmínka importu podle druhu operačního systému. Níže jsou ukázány příklady souborů, kde došlo ke změnám v porovnání s tím, co bylo dosud uvedeno nebo ty, které byly zmíněny poprvé.

```
- hosts: web_servers
  roles:
    - web_server
```

Výpis 4.7: Ukázka souboru playbook.yml.

```
apache_sites_path: /etc/apache2/sites-available/
packages_centos:
    - httpd
    - bind-utils
packages_debian:
    - apache2
    - dnsutils
```

Výpis 4.8: Ukázka souboru vars/main.yml.

```
- import_tasks: debian.yml
  when: ansible_facts['os_family']|lower == "debian"
- import_tasks: centos.yml
  when: ansible_facts['os_family']|lower == "redhat"
- import_tasks: copy.ym
```

Výpis 4.9: Ukázka souboru tasks/main.yml.

Kapitola 5

Analýza a návrh řešení pro společnost Codasip

Společnost Codasip se zabývá vývojem dvou typů produktů, které spolu souvisí. Prvním z nich je sada nástrojů pro návrh tzv. aplikačně specifických procesorů (zkráceně ASIP), které lze ovládat přes konzoli nebo IDE[1] zvané Codasip Studio a druhým jsou pak přímo návrhy jader procesorů postavených na architektuře RISC-V. Pro vývoj se používá metod průběžné integrace[2] na základě kterých probíhá pravidelný build[3] produktů a jejich testování. To se v současné době odehrává na 76 virtuálních strojích běžících na platformě KVM/QEMU umístěných na 9 serverech.

Úkony na virtuálních strojích jsou spouštěny centrálně pomocí nástroje Jenkins, který je určen pro automatizaci procesu vývoje softwaru. Je napsán v jazyce Java a ovládá se skrze webové grafické rozhraní. Jde o architekturu typu master-slave, ve webovém rozhraní se definují jednotlivé výpočetní uzly (v tomto případě se používají virtuální stroje, nástroj umí ale pracovat i s kontejnery a různými výpočetními cloudy), které je možno sdružovat do pojmenovaných kategorií. Pro komunikaci s uzly společnost využívá propojení pomocí SSH, existují ale i další varianty.

Jednotlivé úlohy pak standardně stahují kód ze soukromých repozitářů, definují se pro ně systémové proměnné, uzly na kterých mají fungovat a kroky, které se musejí provést, aby se úloha dala považovat za splněnou. Úlohy mohou být velmi komplexní a je možné, aby na sebe navazovaly. V případě buildu nástrojů se například spouští úloha paralelně na více druzích operačních systémů a připravený software, který je výsledkem buildu, je použit v dalších úlohách pro testování.

S tím jak se společnost a její produkty vyvíjí, přestává být postupně použití virtuálních strojů dostačující. Přestože se pro virtualizaci používá KVM s hardwarovou podporou plné virtualizace, je zde výpočetní výkon, který je spotřebováván jen pro práci hypervizorů a také celých operačních systémů na virtuálních strojích, které si sami spravují své procesy a rozdělení paměti. Ty mají navíc alokovanou paměť stále, i když ji virtuální stroj celou nepoužívá. Rovněž jsou zde vyšší požadavky na úložiště imageů disků, které v současnosti zabírají dohromady okolo 4,5 TB dat. Ty jsou přitom ve velké míře duplicitní, protože se pro stejný operační systém používá jen jedna varianta, která ale musí být zkopírována pro

[1] IDE – je vývojové prostředí pro programátory s grafickým rozhraním.

[2] Metody průběžné integrace – z anglického Continuous Integration. Jde o soubor metod pro vývoj softwaru, které jsou součástí metodik extrémního programování. Jedná se o pravidelné testování funkčnosti programu po jeho sestavení.

[3] Build – sestavení programu z aktuálních zdrojových kódů.

každý virtuální stroj zvlášť. Image jsou ukládány na síťové úložiště NAS, odkud jsou přímo připojovány pomocí protokolu NFS. To také klade vyšší požadavky na síťovou infrastrukturu a její propustnost. V případě výpadku v rámci firemní LAN sítě anebo úložiště dojde k okamžitému přerušení funkčnosti virtuálních strojů, což vede k selhání všech v té době spuštěných úloh, jejichž délka běhu může být i několik hodin.

V tomto počtu virtuálních strojů začíná být složitá také jejich správa. Na serverech se vyskytují virtuální stroje s operačními systémy CentOS 6, CentOS 7, Debian 8, Debian 9 a Windows 10 a jejich počty jsou přibližně vyvážené. Jejich prostředí je vytvářeno tak, že se spustí virtuální stroj na počítači zaměstnance, který má přípravu prostředí virtuálních strojů na starosti, k němu se připojí image disku a začne se s konfigurací. Po úpravě je třeba image disku rozdistribuovat, což je nepříjemný manuální proces. Image je nutné nakopírovat na síťové úložiště, zduplikovat jej a následně vypnout všechny virtuální stroje napříč různými servery, kterých se úprava týká a vyměnit jim disky. Po zapnutí je třeba zkontrolovat, jestli se ke všem těmto uzlům dokázal Jenkins připojit. K tomu je ve společnosti navíc potřeba komunikace dvou týmů, protože zaměstnanci řídící běh úloh na serveru Jenkins nemají přímý přístup k serverům, na kterých virtuální stroje běží, což může způsobovat prodlevy.

Z těchto důvodů přichází myšlenka nahradit původní virtuální stroje kontejnery. Očekávaným zlepšením je pak efektivnější využití výpočetních prostředků, úspora místa (image pro kontejner zabírá místo v řádech jednotek GB, jeden image pro virtuální stroj desítky GB) a velmi jednoduchá možnost úpravy prostředí, kdy jsou kontejnery startovány pouze na vyžádání a jen s nejnovějším imagem. Spolu s těmito vlastnostmi přicházejí také nové možnosti skladby úloh, kdy pro jejich vykonávání není limitní celkový počet virtuálních serverů, který je z krátkodobého pohledu neměnný. Kontejnerů je možné spustit současně takové množství, které umožňuje výpočetní síla používaných serverů. Proto by bylo možné jednotlivé úlohy rozdělit a spouštět je současně, pokud to daný proces umožňuje. Tím by se dalo dobu vykonávání úloh zkrátit.

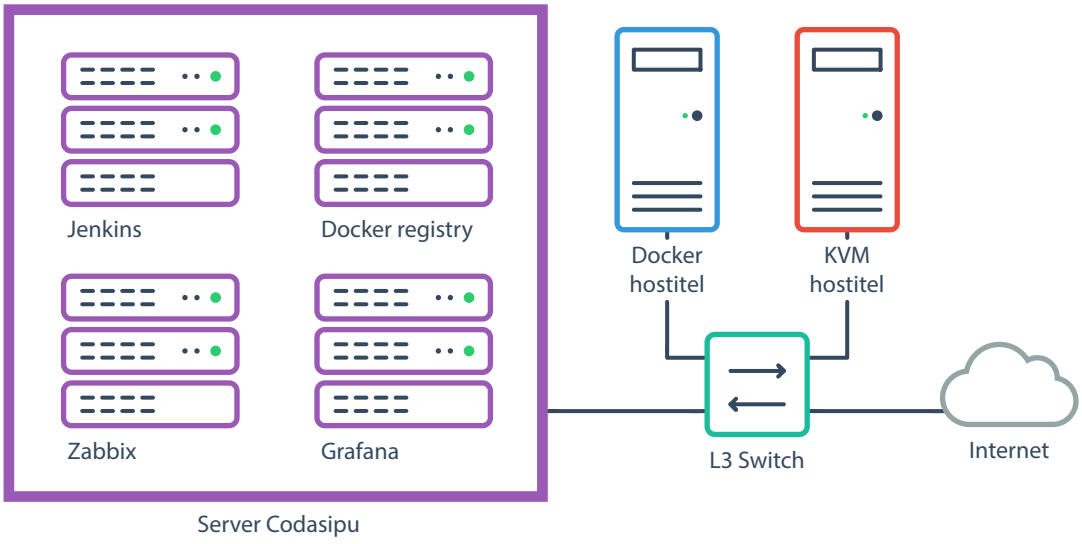
5.1 Návrh nového řešení

Pro realizování zadání jsem se rozhodl nahradit virtualizační software KVM a použít Docker a to především z důvodu, že se dá používat i na systémech Windows. To je důležité, protože bez podpory operačních systémů od společnosti Microsoft by nebylo možné provádět build produktů pro všechny podporované platformy. K tomu, aby byl koncept funkční, jsem do práce zahrnul také přípravu vlastního Docker registry s vylepšením o autentizaci uživatelů a webové grafické rozhraní. Díky tomu budou moci pracovníci společnosti efektivně spravovat jednotlivé image a umožní jim to rozdělovat úrovně oprávnění a přístupů mezi uživatelskými účty. Propojení se serverem Jenkins jsem zajistil pomocí pluginu pro Docker, který sám dokáže spouštět na vyžádání kontejnery mezi hostiteli a stahovat pro ně aktuální image. V průběhu příprav jsem zvažoval také možnost využít propojení této služby s hostiteli, kteří by byli spojeni pomocí nástoje Docker swarm. Ukázalo se ale, že přestože by požití nástroje swarm mohlo být vhodnější kvůli lepšímu plánování podle využitých zdrojů, Docker Swarm plugin pro Jenkins nedokáže delegovat přihlašovací údaje k soukromému Docker registry, a nebylo by tak zajištěno použití aktuálních imageů. Zajímal jsem se také o použití softwaru OpenNebula, ten jsem se ale rozhodl nevyužít, protože správu kontejnerů dokáže plně zařídit server Jenkins. Rovněž se jedná o nástroj, který je určen primárně pro správu virtuálních strojů a Docker zde lze používat částečně pomocí neaktuálního rozšíření. Jeho použití mi proto nepřišlo být užitečné.

Dále jsem stanovil způsob, jak porovnávat a měřit spotřebovaný výkon na strojích, kde probíhají úlohy v nástroji Jenkins. V Codasipu se k dohledu serverů používá monitorovací nástroj Zabbix, a tak jsem se rozhodl toho využít. Na testovací hostitele jsem navrhl instalovat agenty pro tento software a v pravidelných intervalech měřit využití paměti a procesoru. Přestože Zabbix dokáže naměřená data vizualizovat, nastavování grafů není v tomto nástroji moc příjemné. Rozhodl jsem se proto použít ještě vizualizační nástroj Grafana, který umí využít data ze serveru Zabbix.

Protože práce obnáší spoustu úloh pro konfiguraci počítačů a virtuálních strojů, rozhodl jsem se nachystat tuto část v Ansible tak, aby bylo možné mít k dispozici zcela stejné prostředí, jako jsem měl k dispozici já, bez velké námahy. To mi pomohlo v průběhu příprav, kdy bylo jednoduché kdykoliv změny na libovolném počítači zahodit a rychle jej připravit znovu. Role Ansiblu budou užitečné i v případě, že se některá z mých částí práce využije ve společnosti více. Nebude těžké je modifikovat podle individuálních požadavků a nasazení bude rychlé.

Pro mou bakalářskou práci jsem dostal k dispozici dva počítače se stejnými parametry pro testování a porovnávání staré a nové metody virtualizace a možnost spouštět virtuální stroje na jednom ze serverů. Z toho vychází následující úvaha o tom, jak by celý systém měl jako celek fungovat. Jeden počítač jsem určil jako Docker hostitele a druhý bude referenční KVM hypervizor. Úlohy se pak budou vykonávat na obou zároveň, aby bylo možné porovnat jejich dobu provádění a využití výpočetních prostředků. Na serveru jsem se rozhodl použít 4 virtuální stroje z nichž jeden bude pro Jenkins, další pro Docker registry a zbylé dva pro Zabbix a nástroj Grafana na monitorování. Pro provoz nástroje Jenkins jsem využil jeden z předchystaných imageů společnosti, které obsahovaly konfiguraci potřebných pluginů, které Codasip používá. Pro sledování využitých výpočetních prostředků jsem zvolil firemní server Zabbix, protože použít vlastní instanci tohoto softwaru se jevilo jako nadbytečné. Zbylé služby byly zcela v mé režii. Schéma propojení je možné vidět na obrázku 5.1.



Obrázek 5.1: Schéma návrhu propojení jednotlivých prvků systému. V poli serveru se nachází virtuální stroje.

Kapitola 6

Implementace

V této kapitole bude podrobněji rozebráno, jakým způsobem byla praktická část bakalářské práce realizována. Kapitola obsahuje pasáž o přípravě virtuálních strojů a počítačů a o tom, co všechno se podařilo vyřešit automatizovaně pomocí Ansible. Později bude zmíněna tvorba Docker imageů pro firemní nasazení, propojení s nástrojem Jenkins, spouštění úloh a jejich měření.

6.1 Příprava počítačů a virtuálních strojů

První věcí, kterou bylo třeba udělat, bylo nainstalovat operační systém na dva přidělené počítače pro spouštění virtuálních strojů a kontejnerů. Protože zde neexistovalo žádné omezení, rozhodl jsem se podle osobních preferencí a zvolil aktuální Debian verze 10 (s označením Buster). Počítačům byla zapnuta podpora hardwarové virtualizace v UEFI[1]. Dále byl připraven obecný image se stejným operačním systémem pro použití ve virtuálních strojích pro Docker registry a nástroj Grafana, zkopírován image pro Jenkins a vytvořeny virtuální stroje. Parametry fyzických a virtuálních strojů jsou uvedeny v tabulce 6.1.

Využití	Docker	KVM	Jenkins	Docker registry	Grafana
Hostname	jj-host-1	jj-host-2	jj-jenkins	jj-docker-registry	jj-grafana
Typ stroje	fyzický	fyzický	virtuální	virtuální	virtuální
Procesor	AMD Ryzen 7	AMD Ryzen 7	AMD Epyc 7281	AMD Epyc 7281	AMD Epyc 7281
Počet jader	16	16	4	4	4
Velikost paměti [GB]	32	32	4	4	4

Tabulka 6.1: Přehled parametrů strojů použitých v práci.

Každý z připravovaných strojů byl nastaven jen základně. V systému je vytvořen uživatel „jurica“, který je členem skupiny „sudo“ a který může po zadání hesla vykonávat privile-

[1]Unified Extensible Firmware Interface – nástupce firmwaru BIOS, jenž je zodpovědný za základní inicializaci počítače při startu a předání řízení operačnímu systému.

gované operace pomocí stejnojmenného příkazu. Přihlašování pomocí SSH k uživateli root je vypnuto a jinak je umožněno pouze uživatelům, kteří se prokáží na základě povoleného veřejného klíče. K tomu je nastavena síť, hostname a nainstalováno několik nejzákladnějších balíků, jako je *htop* a *vim*. Hostnamy strojů jsou zaregistrovány v interním autoritativním DNS[2] serveru společnosti, aby mohlo být využito šifrovaného připojení k webovým službám. Ty v sobě obsahují předponu „jj“, aby byla dostatečně odlišena doménová jména, která slouží pro účely této bakalářské práce.

Při implementaci byl kladen důraz na konfiguraci strojů a služeb tak, aby jejich použití bylo co nejvíce bezpečné a možné nasadit do běžného používání. Umožnění přihlašování uživateli root přes SSH je velké bezpečnostní riziko kvůli privilegiím a také protože je to známý uživatel, který by mohl být vhodný k hádání hesla. Zároveň je upřednostňováno vykonávat privilegované operace pomocí příkazu sudo, protože je pak možné zjistit, jaká operace byla kterým uživatelem vykonána. Z práce je ale vypuštěna konfigurace firewallů, protože se odvíjí od sítí a jejich počtu, kde je počítač připojen a nebylo by možné ji univerzálně použít.

6.2 Tvorba rolí v nástroji Ansible

V Ansible bylo vytvořeno dohromady 5 souborů playbook, které jsou definovány pro Docker hostitele, KVM hostitele, Docker registry, server Grafana a pro libovolný počítač, kde se provede build nachystaných Docker imageů. Každý playbook využívá jednu nebo více rolí, které byly rozděleny podobným způsobem. Byla vytvořena role pro instalaci Dockeru, KVM, nástroje Grafana, Docker Compose, Zabbix agenta, pro konfiguraci Docker registry a build imageů. Přestože byl jako výchozí systém zvolen Debian 10, role byly psány a částečně testovány i pro CentOS 8 a starší verze obou těchto distribucí.

Docker Engine

Role obsahuje úlohy pro stažení a instalaci programu Docker Engine a jeho závislostí z oficiálních repozitářů Dockeru. V době psaní této práce bylo obtížné instalovat Docker Engine na CentOS 8, protože mu chyběla správná verze balíku „containerd.io“ a systém ji nedokázal zajistit. Problém je vyřešen samostatnou instalací balíčku, což má za následek jen to, že v případě, kdy bude Docker Engine odinstalováván, tento balík se automaticky neodstraní a je třeba to udělat zvlášť.

V této a i v následujících rolích jsou použity další adresáře, které nebyly v kapitole 4 uvedeny. Jednou z nich je složka *defaults*, která rovněž umožňuje definici proměnných. Rozdíl mezi *defaults* a *vars* je v jejich prioritě, kdy proměnné z adresáře *defaults* mohou být přepsány definicí proměnné kdekoliv jinde. Proměnné z této složky jsou v práci použity jako výchozí argumenty, které upravují chování vykonávání rolí například tím, že se nějaká část aktivuje až jejich přepsáním. Tímto způsobem je docíleno možnosti volat roli pro Docker Engine s variantou pro otevření REST API[3] pro vzdálené řízení, která je ve výchozím stavu vypnutá. Tuto možnost potřebuje mít přístupnou Docker hostitel, aby se k němu mohl připojit server Jenkins.

Pokud se role vykoná s požadavkem na zpřístupnění API, provede se několik dalších úkonů. Na začátku kapitoly 3 o Dockeru bylo zmíněno, že pokud je API otevřena, má k ní ve výchozím stavu přístup kdokoliv, kdo je ve stejné síti, jako je Docker hostitel. Zneužití

[2] Autoritatvní server DNS – DNS server, který spravuje pouze záznamy pro konkrétní doménu.
[3] Dále v textu bude pro zkrácení zmiňována jen jako API.

by mělo velmi nebezpečné následky, protože je možné pomocí API spustit privilegovaný kontejner, který může mít připojen kompletní filesystém hostitele. Výsledek by byl rovný tomu, jako mít přihlašovací údaje přímo k uživateli root. Komunikaci je možné ale celou zabezpečit pomocí použití oboustranného TLS. Tento způsob se od běžného šifrované komunikace, jak je známa například u HTTPS, liší tím, že i klient musí mít vygenerovanou sadu certifikátů, které jsou podepsány stejnou autoritou, jako certifikáty pro server. V tomto případě se hodí vygenerovat si tzv. self-signed certifikáty, které se vytvářejí včetně certifikátů zaštiťující autority. Tyto certifikáty jsou přeneseny v rámci úloh v roli a otevření API je nastaveno přepsáním služby v SystemD[4] úpravou spouštění Docker démonu.

Docker Compose

Docker Compose byl uveden v části 3.4 a je použit pro spuštění kolekce kontejnerů s Docker registry. Jeho instalace by byla za normálních okolností velmi jednodnoduchá, protože se stahuje pouze binární soubor, který se umístí na příslušné místo v operačním systému a po přidělení práv ke spuštění je připraven k použití. Problém zde nastal v souvislosti s používání nástroje Ansible, který má modul na ovládání tohoto programu, ten ovšem nebyl schopen si zajistit správné moduly do Pythonu. Role tedy k instalaci Docker Compose zajišťuje stažení správných modulů pro Python 2 i Python 3.

Součástí adresářů je složka *meta*, ve které se specifikují závislé role, které musí být provedeny předtím, než je spuštěna tato. Zde je uvedena role Docker Engine, bez které by používání tohoto programu nemělo smysl. Díky tomuto mechanismu pak není třeba uvádět veškeré závislé role přímo v souboru playbook a dá se tak vyhnout problémům, pokud by se na nějaké předchozí kroky zapomnělo nebo byly definovány ve špatném pořadí.

Docker registry

Pro řešení ukládání imageů jsem zvolil software s názvem Portus[5], který byl vyvíjen společností SUSE. Portus nabízí rozšíření standardního kontejneru Docker Registry o chybějící část řízení přístupu a grafické rozhraní. Pomocí aplikace Portus je možné řídit přístup ke jmenným prostorům, které mohou být vázány na firemní týmy, ty jsou rozšířením. K dispozici jsou tři úrovně přístupu rozdělené na viewer, publisher a owner. Viewer může image pouze stahovat, publisher je může také nahrávat a owner je k tomu dokáže i odstranit. Portus rovněž nabízí možnost zpřístupnit image z vybraného jemnného prostoru pro nepřihlášené uživatele. Mezi další výhody patří podpora LDAP[6] a REST API, které je možné využít k inicializaci prostředí po spuštění. Portus byl v průběhu práce na implementaci na zkoušku nasazen i přímo v Codasipu, kde jej využívá jeden z týmů. Ukázalo se ovšem, že tento software není zcela odladěn a vyskytují se u něj občasné chyby. Také se zdá, že se na vývoji od minulého roku (2019) přestalo pracovat, proto jej pro běžné nasazení není možné doporučit.

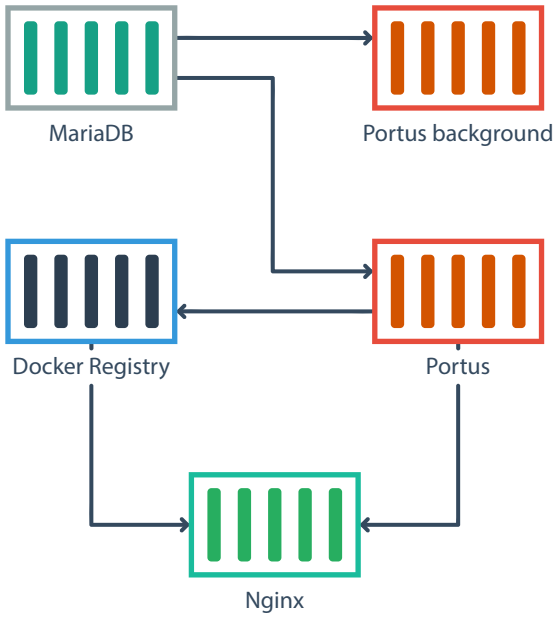
Portus je systémem složeným z 5 kontejnerů, ve kterém je zahrnut oficiální image Docker Registry, databázový server MariaDB, dvakrát image Portus (jednou v roli webové aplikace a podruhé jako nástroj pro udržení integrity dat mezi databází a registry) a webový server Nginx. Nginx zaujímá pouze úlohu proxy serveru, který na základě tvaru URL přepošle

[4]SystemD – je init proces Linuxu, který nahradil starší System V. Jednou z jeho funkcí je správa softwarových démonů.

[5]Portus – http://port.us.org/

[6]LDAP – protokol a adresářový server sloužící k evidenci dat jako jsou přihlašovací údaje. Časté je jeho podnikové nasazení. Microsoft má svou vlastní implementaci, která se nazývá Active Directory.

dotaz webové aplikaci Portus anebo Docker Registry. Na obrázku 6.1 je k dispozici schéma, ve kterém je znázorněno, jak mezi sebou kontejnery komunikují. Porty jsou mapovány jen mezi hostitelem a proxy serverem, ostatní kontejnery fungují jen v rámci *bridge* sítě.

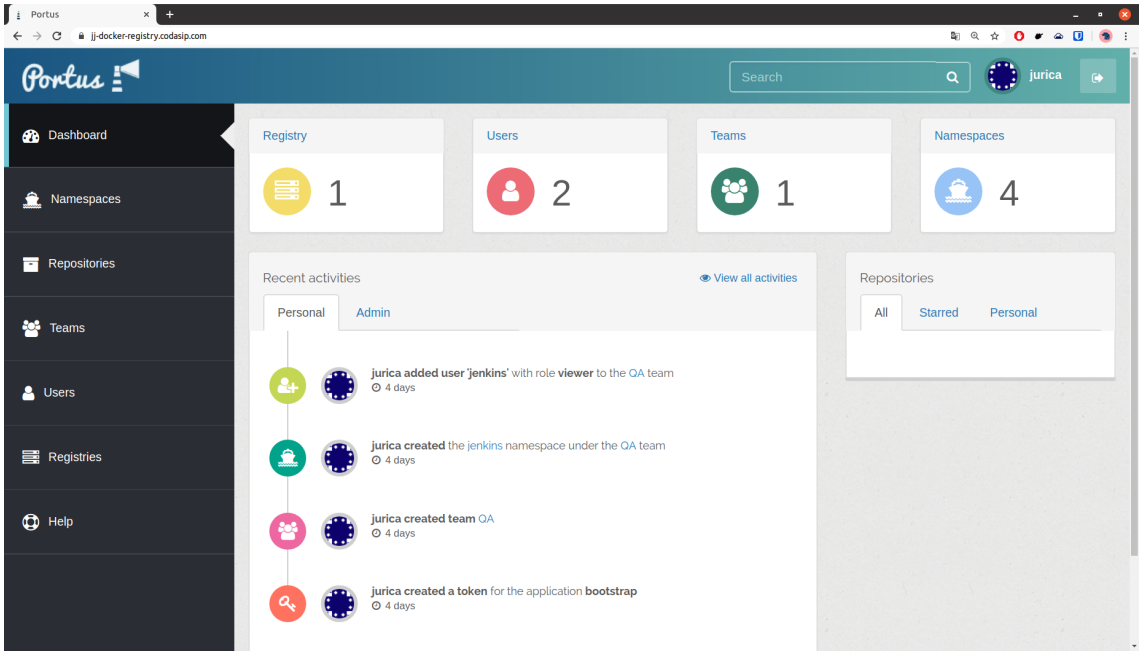


Obrázek 6.1: Schéma komunikace a provázání kontejnerů pro úložiště imageů. Šipky znázorňují, kde je provedeno linkování.

V rámci role jsou překopírovány potřebné soubory na stroj, kde se má registry spustit. Pro správný běh úložiště a funkci HTTPS se na mnoha místech používá FQDN (celé doménové jméno), které je vygenerováno pomocí zjištěného hostname stroje a doplněno o řetězec „.codasip.com“. Soubor s proměnnými obsahuje další nastavení, které je možné upravit na míru, jako je jméno použité databáze, heslo k uživateli databáze, cesty k TLS certifikátům a podobně. Na základě těchto dat je pak generován soubor „docker-compose.yml“, který se spolu s dalšími parametrizovatelnými konfiguračními soubory nalézá v adresáři *templates*. Role je závislá na předchozí roli pro Docker Compose.

Poté co je systém kontejnerů poprvé spuštěn, zahájí se inicializace Portusu. Pomocí Ansible modulu pro vytváření HTTP/HTTPS požadavků se vytvoří uživatelé „jurica“ jako administrátor a „jenkins“ jako bot. Poté se již jen vytvoří tým QA a namespace „jenkins“, do kterého má uživatel se stejným jménem přístup pouze pro čtení. Tohoto uživatele bude využívat server Jenkins pro stahování aktuálních imageů na Docker hostiteli. Aby bylo možné REST API použít, kontejner Portus s webovou aplikací musí být na poprvé spuštěn s povolenou registrací uživatelů, což je nastavováno pomocí proměnné prostředí pro kontejner v „docker-compose.yml“. Toto nastavení umožní vytvořit prvního uživatele, aniž by bylo třeba mít v požadavku autentizační token (způsob ověření uživatele při komunikaci s API). Úložiště má být ale soukromé, proto je po inicializaci služby systém vypnut, proměnná prostředí v souboru „docker-compose.yml“ upravena a kontejnery opětovně spuštěny. Spuštěná webová aplikace Portus je vidět na obrázku 6.2.

V této roli se poprvé vyskytují i proměnné obsahující hesla a další hodnoty, které by měly být utajeny. Toho je docíleno pomocí programu Ansible Vault, což je samostatný program, který umožňuje zašifrovat jednotlivé proměnné nebo celý soubor pomocí hlavního hesla. Jakmile se v nějaké roli použije šifrovaná proměnná, je nutné spouštět playbook, ve



Obrázek 6.2: Úvodní stránka aplikace Portus po přihlášení administrátorským účtem.

kterém je obsažena, jen se zadaným hlavním heslem. Tímto způsobem se také řeší ukládání hesel pro uživatele, kteří mohou spouštět příkaz *sudo*.

Grafana a KVM

Instalace serveru Grafana a balíků pro KVM v sobě nezahrnuje nic náročného. Role obsahují úlohy se specifickými balíky pro Debian i CentoOS. Protože Grafana sloužila v práci jen jako nástroj pro vykreslování grafů v rámci porovnávání virtualizačních metod a její využití není nutné, služba není inicializována uživatelem. Je zde pouze nahrán dodatečný plugin, který umožňuje v Grafaně použít jako zdroj dat Zabbix.

Zabbix agent

Pro měření spotřebovaného výpočetního výkonu na Docker a KVM hostitelích je potřeba nainstalovat Zabbix agenta, který pravidelně přeposílá pozorovaná data na server. Z oficiálních repozitářů společnosti Zabbix jsou stáhnuty potřebné balíky pro verzi 4.0 LTS a provedena základní konfigurace. Ta obsahuje vyplnění názvu monitorovaného počítače a adresu serveru. Druhou částí bylo rovněž využití REST API, aby byl ihned po instalaci agenta počítač zaregistrován na serveru. Registrace počítače je prováděna inteligentně a při každém běhu role se nejprve kontroluje, jestli daný počítač již není vytvořen. Pokud není, požadavkem jsou zaslány nezbytné informace o stroji, je zvolena skupina hostů, kde má být zařazen a výber šablon pro monitorování.

Build imageů

Po tom, co je zprovozněno úložiště imageů, je možno image vytvořit a nahrát. O tom, jak jsou image tvořeny, je následující část (6.3). Zde se nachází popis role, která předpokládá, že již existují připravené Dockerfily. Role má nastavenou výchozí adresu registry

a namespace, které se používají k tvorbě názvu imageů, konkrétní jméno image je použito podle složky, obsahující Dockerfile. Cesta k adresářům se soubory Dockerfile je definována další proměnnou. Dále je použito globálních proměnných, pro získání přihlašovacích údajů k administrátorovi v registry, aby bylo možné image nahrát.

Prvně se provede přihlášení do registry a následně se jeden po druhém prochází adresáře se soubory Dockerfile. Před buildem je definována proměnná, která obsahuje aktuální číslo verze image. To je vyčítáno ze souboru Dockerfile, který obsahuje instrukci *LABEL*, v níž je verze uvedena. Modul pro práci s Docker image dokáže v jednom kroku provést build i nahrání na úložiště. Neumí to ovšem udělat s více názvy současně, a tak je prvně proveden build a nahrání s přesnou verzí v názvu. Následně je pro image přidáno druhé jméno s verzí „latest“ a rovněž zaregistrováno v úložišti (Docker Registry dokáže rozeznat na základě ID image, že jej již obsahuje, proto se podruhé image skutečně nenahrává).

6.3 Tvorba imageů

V rámci práce bylo připraveno celkem 10 imageů pro distribuce Debian 9, Debian 10, CentOS 6, CentOS 7 a CentOS 8. Pro každý image existují dvě varianty, které se liší tím, jestli je uvnitř nainstalován a spuštěn SSH server nebo ne. Varianty jsou vytvořeny na základě dvou možností, jak se server Jenkins může připojovat ke spuštěným kontejnerům. Pokud se používá SSH, image musí obsahovat SSH server a Javu, aby bylo možné jej ovládat. Propojování přes SSH probíhá bez zadávání hesla na základě veřejného klíče. To funguje tak, že byla vygenerována dvojice RSA klíčů pro uživatele „jenkins“, které se pro komunikaci nahrají v grafickém rozhraní serveru Jenkins a veřejný klíč z dvojice je překopírován do všech imageů s podporou SSH pro stejnojmenného uživatele, který je v imagích vytvořen. Pokud by bylo třeba z nějakého důvodu využít jiných klíčů, je možné kontejneru předat nový veřejný klíč, aniž by musel proběhnout nový build image. Toho je docíleno pomocí proměnné prostředí „JENKINS_SERVER_SSH_PUBKEY“. Ta je při startu kontejneru kontrolována a v případě, že není prázdná, defaultní veřejný klíč je přepsán. Ve variantě bez SSH je pro připojení potřebný pouze balík Javy a o ostatní se stará příslušný plugin v nátroji Jenkins. Varianty bez SSH byly vytvořeny primárně pro Docker Swarm plugin, protože ale nakonec nebyl použit, jejich testování proběhlo pouze v začátcích a následně byly jen upravovány podle poznatků z testování SSH varianty. To může znamenat, že je v některém zanesena chyba znemožňující připojení ke kontejneru s tímto imagem.

Soubory Dockerfile obsahují instrukci *LABEL*, ve které jsou zaneseny základní informace o image, které jsou dostupné po spuštění kontejneru pomocí příkazu *docker inspect*. Tyto informace se zapisují ve formě *KLÍČ=HODNOTA*, kdy klíč je tvořen z obráceného pořadí domény instituce, která image spravuje a konkrétního typu informace. Hodnotou je pak libovolný řetězec, který musí být ovšem z obou stran obalen uvozovkami v případě, že obsahuje bílé znaky. Příkladem může být tento klíč s hodnotou *com.codasip.maintainer="Jiri Jurica"*. Dále jsou v imagích zmíněny v komentáři základní informace pro práci s nimi, jako jsou příkazy pro spouštění a build. Následuje instrukce pro zkopírování skriptu, který je spuštěn po startu kontejneru. Jedná se o jednoduchý skript napsaný v programu Bash[7], který je ovlivňován pomocí předaného argumentu. Pokud je kontejner spuštěn bez argumentů pro tento skript, v SSH variantě proběhne kontrola proměnné „JENKINS_SERVER_SSH_PUBKEY“ a následně se spustí SSH server na popředí. V druhé variantě je spuštěn příkaz *tail -f /dev/null*, což je způsob, jak spustit kontejner, který není závislý na běhu žádné konkrétní

[7]Bash – často používaný typ shellu, což je program, který v Linuxu interpretuje příkazovou řádku.

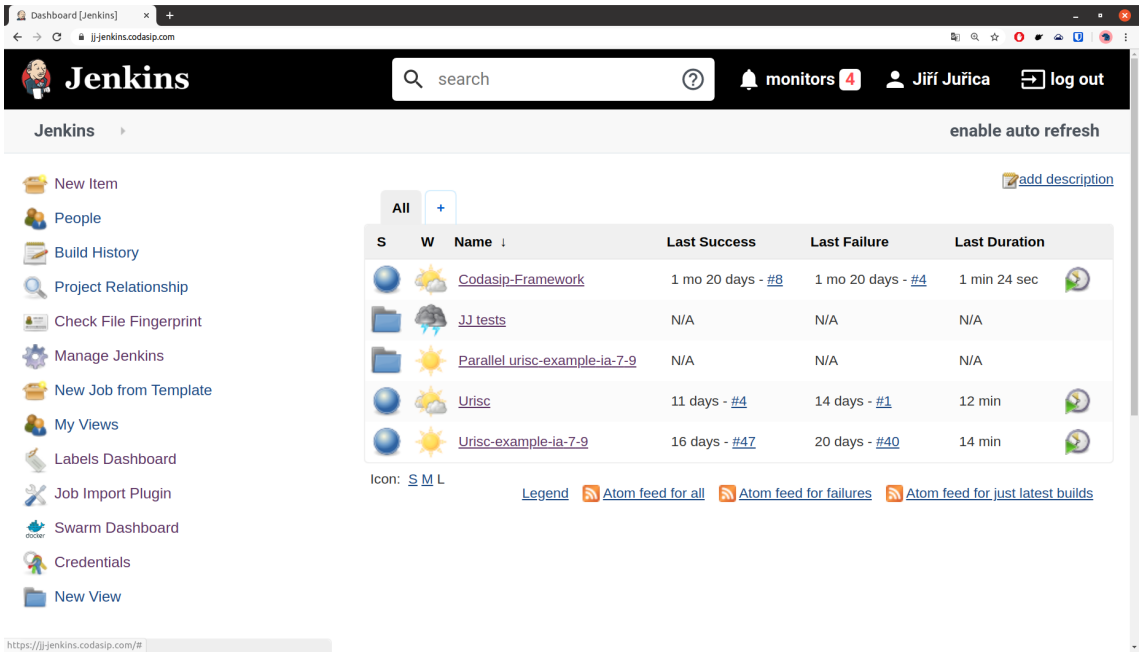
služby, což je pro potřeby propojení se serverem Jenkins nezbytné. Skript reaguje na argument „inspect“, který je vhodné předávat při spouštění kontejneru v interaktivním režimu. Případné záležitosti okolo přípravy prostředí a spouštění SSH serveru jsou vykonány na pozadí a na popředí je spuštěn Bash. Uživateli je tak umožněno kontejner prozkoumat. Dále se při přípravě image vytvoří uživatel „jenkins“, kopírují se SSH klíče (veřejný klíč pro komunikaci se serverem Jenkins a oba klíče pro stahování repozitářů z firemního serveru Gitlab) a volí se „entrypoint“. Nakonec se instalují balíky, které jsou nezbytné pro vykonávání úloh v Codasipu a jsou závislé na distribuci a její verzi.

Připojování NFS

Některé nástroje, které jsou ve společnosti používány, jsou nainstalovány na síťovém úložišti, aby je mohli používat jak zaměstnanci, tak virtuální stroje. Z tohoto důvodu je pro vykonávání některých úloh nutné připojit ke kontejneru NFS úložiště. Tento požadavek nijak neovlivňuje tvorbu imagů a ani by to nebylo vhodné, protože v rámci kontejneru není bez privilegovaného běhu[8] možné připojovat síťové svazky. K připojení NFS se používá propojování adresářů z hostitele do kontejneru, které je specifikováno v příkazu *docker run*, jak bylo uvedeno v části 3.1.

6.4 Jenkins

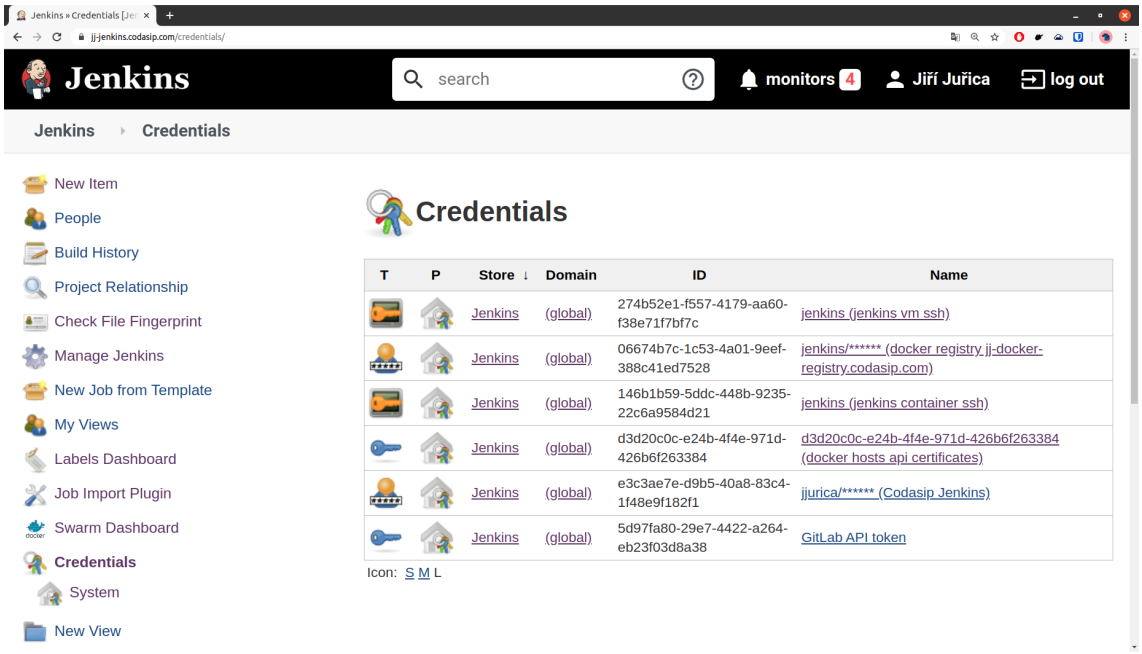
Server Jenkins byl pro použití v mé práci již předkonfigurován a mým cílem bylo v první fázi pouze zprovoznit provádění úloh na Docker a KVM hostitelích. Druhou části pak bylo importovat zde několik úloh, které se ve společnosti skutečně používají, upravit je pro potřeby této bakalářské práce a spustit testování. Na obrázku 6.3 je úvodní strana Jenkinse.



Obrázek 6.3: Úvodní stránka nástroje Jenkins.

[8] Privilegovaný režim kontejneru – spouští se přidáním přepínače *--privileged* do příkazu *docker run*. Ve výchozím stavu má kontejner zamezen přímý přístup k zařízením hostitele.

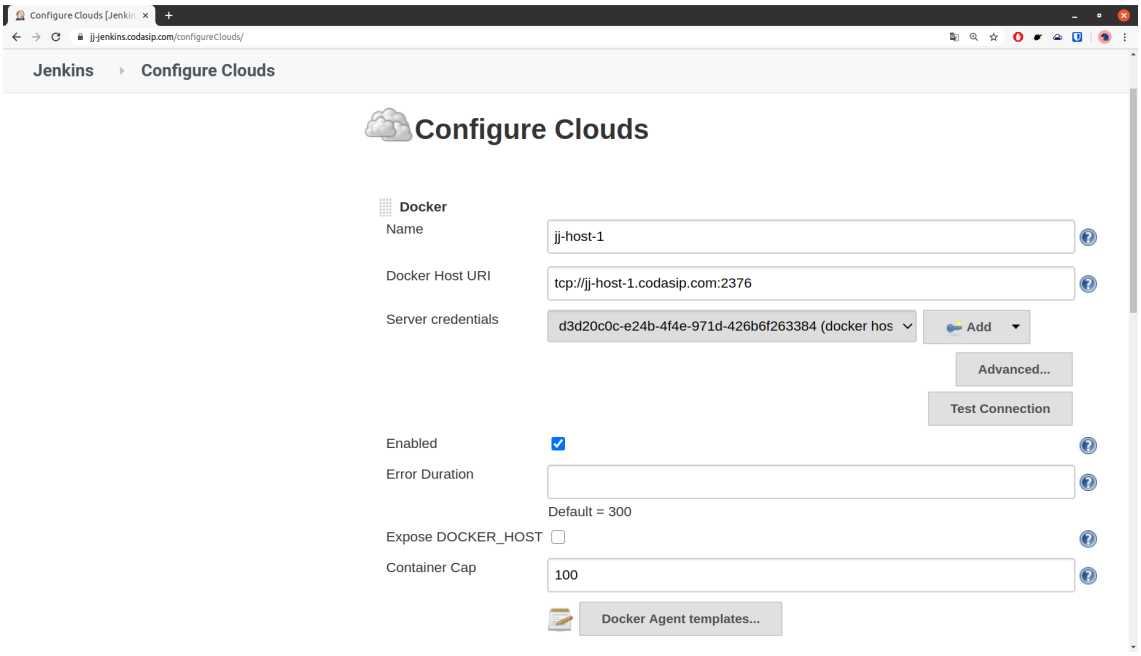
Do serveru byl doinstalován plugin Docker, který umožnil přidávat Docker hostitele. Než byl nakonfigurován, bylo třeba nastavit některé přístupové údaje. Ukládání přístupových údajů je v nástroji Jenkins řešeno pomocí pluginu Credentials, který umožnuje na serveru ukládat různé typy přihlašovacích údajů. Pro tuto práci bylo třeba nastavit SSH klíče pro připojení ke kontejnerům, SSH klíče pro připojení k virtuálnímu stroji na referenčním KVM hostiteli, přidat klientské certifikáty pro zabezpečenou komunikaci s Docker REST API a přihlašovací údaje pro Docker registry. Nastavení je možné vidět na obrázku 6.4.



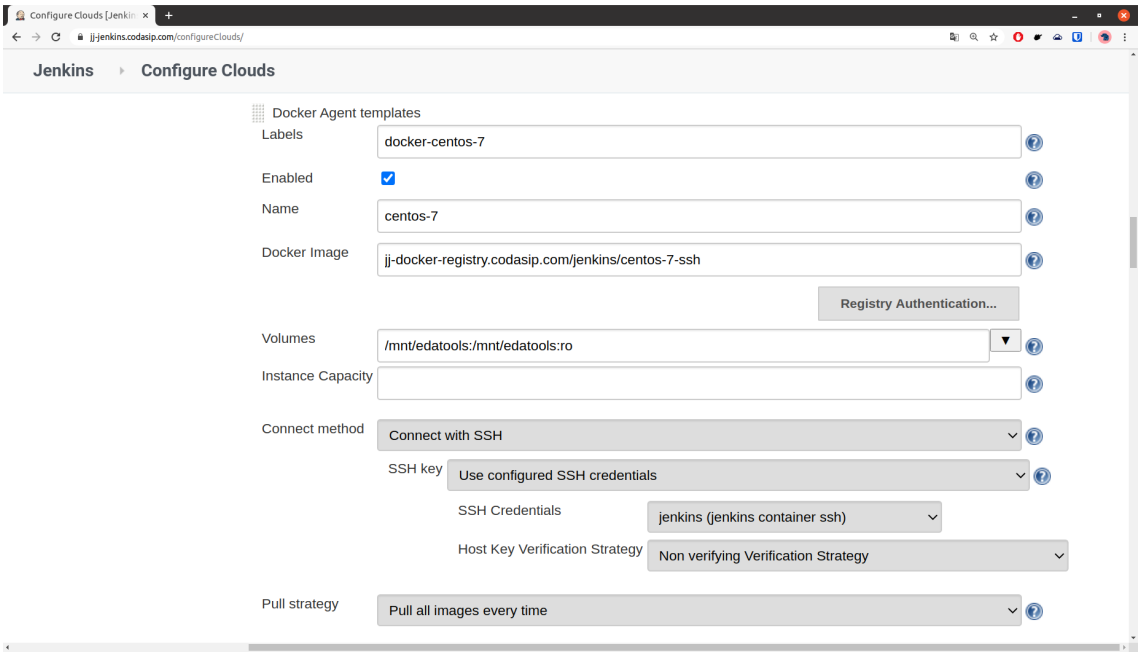
Obrázek 6.4: Přihlašovací údaje uložené na serveru Jenkins.

Pak již bylo možné přidat stroje, na kterých úlohy poběží. Na KVM hostiteli jsem vytvořil jediný virtuální stroj, který měl nastaveny stejné výpočetní prostředky, jako měl sám hostitel. Image disku byl zvolen produkční s jednou z linuxových distribucí a veškerými závislostmi, které jsou ke spouštění úloh potřeba. Tento virtuální stroj byl na serveru Jenkins zaregistrován, nastavila se komunikace pomocí SSH a specifikovaných klíčů a bylo provedeno ověření, že se na něj server dokáže úspěšně připojit. V části nastavení cloudů byl zaregistrován Docker hostitel, což obnáší zadání adresy počítače a portu, na kterém je zpřístupněna Docker API. V cloudu Docker je také možné nastavit maximální počet kontejnerů, který se může na hostiteli spouštět. V rámci jeho nastavení byly pak specifikovány šablony pro jednotlivé SSH image. V této části se volí jméno image, label, přihlašovací údaje k Docker registry, způsob připojení, SSH klíče, připojování NFS a podobně. V šabloně je rovněž možné nastavit maximální počet kontejnerů stejného typu, který může běžet na hostiteli současně a také je zde možnost stanovit maximální hodnotu paměti, kterou smí kontejner použít. S nastavení využití jader procesoru je to složitější, protože na to chybí podpora ve zvoleném pluginu. Lze toho docílit zatím jen vytvořením skriptu na serveru, který upravuje spouštění příkazu *docker run*, když je volán tímto pluginem.[9] Těmito způsoby je možné řídit využití prostředků spuštěných kontejnerů. Na obrázku 6.5 je zachyceno nastavení cloudu Docker a na obrázku 6.6 je nastavení pro konkrétní Docker image.

[9] https://stackoverflow.com/questions/58193256/limit-cpus-with-jenkins-and-docker-plugin



Obrázek 6.5: Nastavení připojení k Docker hostiteli.



Obrázek 6.6: Nastavení šablony pro spouštění kontejneru na základě specifického image. V tomto případě se jedná o CentOS 7. Některá pole, která nebyla vyplněna, byla z obrázku vymazána, aby bylo vidět kompletní nastavení jedné šablony.

Kapitola 7

Testování řešení

Pro testování řešení byly zvoleny dvě úlohy pro testování procesoru ASIP s označením Codasip uRISC, který je v rámci společnosti vyvíjen. V úlohách je spuštěn interní testovací framework napsaný v jazyce Python, ve kterém je volen typ testů, produkt a větve z git repozitáře se zdrojovými kódy, které se mají otestovat. Při testování nebyl virtuální stroj ani kontejnery nijak omezovány a měly k dispozici veškerý výpočetní výkon počítače. Ten byl mimo hypervizor KVM na jednom počítači a Docker Engine na druhém testovacím počítači spotřebováván už jen samotným operačním systémem a Zabbix agenty, jejichž vliv je zanedbatelný. Pro běh testů byl zvolen image s distribucí CentOS 7[1].

Součásti každého provedeného testu je informace o době běhu v kontejneru a ve virtuálním stroji, procentuální porovnání časů a grafy využití procesoru a paměti hostitelů. Experimentoval jsem také s měřením spotřebovaných výpočetních prostředků jen na základě konkrétních procesů, to se ovšem ukázalo jako nevhodné. Takové měření neprobíhá dobře, protože procesy mezi sebou část paměti sdílejí. Zabbix tento jev nedokáže detekovat a udává zkreslené hodnoty, které jsou mnohem vyšší, než hodnoty, které udává jako celkové využití paměti počítače. Druhým zásadním problémem je, že KVM je součástí kernelu, a proto se jako proces měřit nedá.

U procesorů je měřen tzv. „CPU load“, což je metoda, kdy je stanoveno množství požadavků vzhledem ke kapacitě jednoho jádra procesoru, na kterém běží. Hodnota je uváděna v procentech a udává se v průměru za časovou jednotku (v případě tohoto testování jde o průměr za jednu minutu). Hodnota z grafu obsahuje výsledek pro celý procesor, čehož je dosaženo pomocí sumy hodnot pro každé jádro ve stejném čase vyděleno počtem jader procesoru. V grafech je na některých místech možné pozorovat vytížení procesoru, které je vyšší než 100 %. To značí stav, kdy je počítač přetížen a požadavků je v dané chvíli více, než může procesor zpracovat. Využití paměti je měřeno v jednotkách GB, kdy maximum je označováno oranžovou čarou a je stanoveno v hodnotě 32 GB, což odpovídá kapacitě RAM počítačů. Z grafů paměti je také možno odpozorovat, že po skončení úlohy na počítači s KVM využití paměti zpátky neklesne. To je způsobeno přístupem KVM, které paměť pro virtuální stroj postupně alokuje podle potřeby, neumí s ní ale nakládat dynamicky a zpět ji uvolnit, dokud virtuální stroj běží.

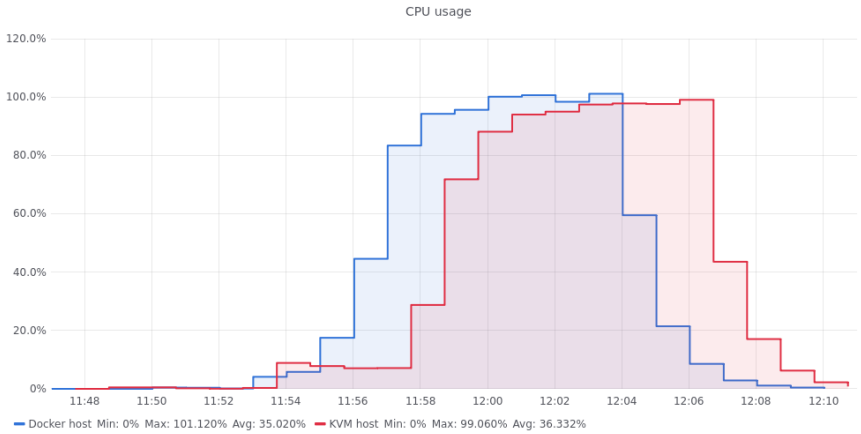
[1] Původně se měl použít CentOS 6, ovšem v průběhu práce bylo zjištěno, že originální image CentOS 6 nemůže být na novém Debianu v Dockeru spuštěn. Tato chyba je evidována v oficiálním repozitáři na serveru Github vývojářů distribuce CentOS a souvisí s verzí kernelu, na které běží novější verze distribuce Debian.

7.1 Test uRISC IP compiler regression s jednou větví

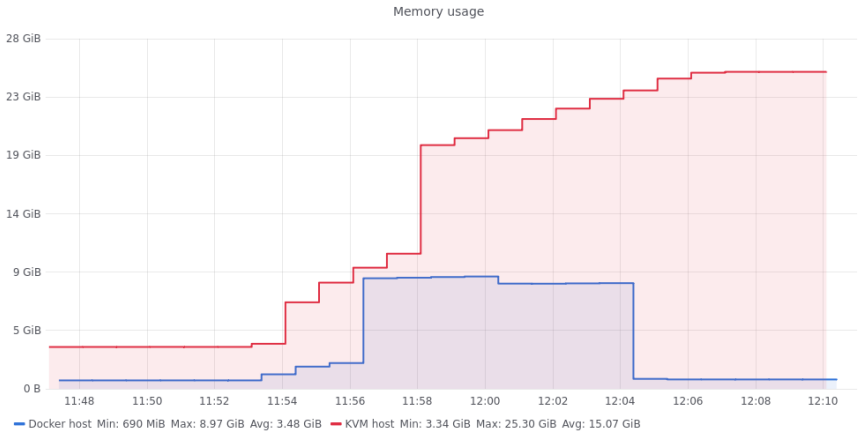
Tento test byl spuštěn dohromady třikrát pro tři různé větve. Změřené hodnoty jsou k dispozici v tabulce 7.1 a grafy využití procesoru a paměti jsou uvedeny pro první měření v obrázcích 7.1 a 7.2. Pro další běhy testu byly grafy vynechány, protože obsahují velmi podobný průběh jako z prvního běhu.

Branch	Doba běhu v Dockeru	Doba běhu v KVM	Zrychlení
test-fpu	11 m	14 m	21 %
test-legal	6 m 55 s	9 m 10 s	25 %
test-jtag	11 m	14 m	21 %

Tabulka 7.1: Výsledky měření prvního testu.



Obrázek 7.1: Graf využití procesoru v průběhu prvního běhu testu.



Obrázek 7.2: Graf využití paměti v průběhu prvního běhu testu.

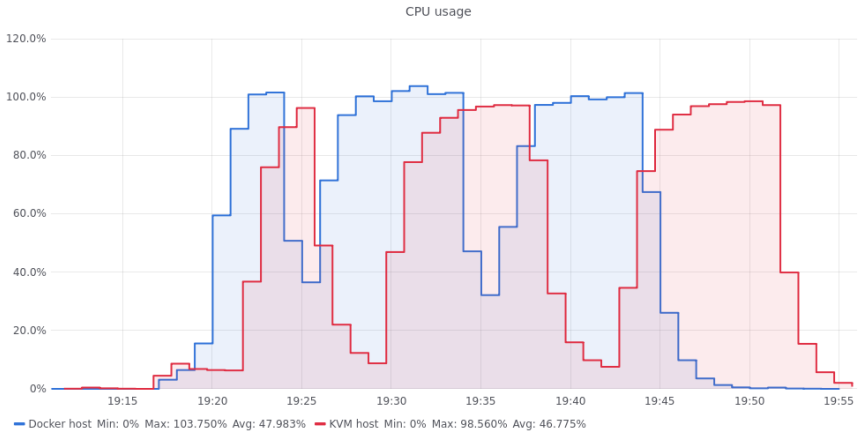
Z grafů je dobře patrná chvíle, kdy je úloha zpracovávána. Modrá křivka Docker hostitele naznačuje, že se úloha v kontejneru vykonává rychleji dřívějším poklesem zpět do klidových hodnot.

7.2 Test uRISC IP compiler regression se třemi větvemi

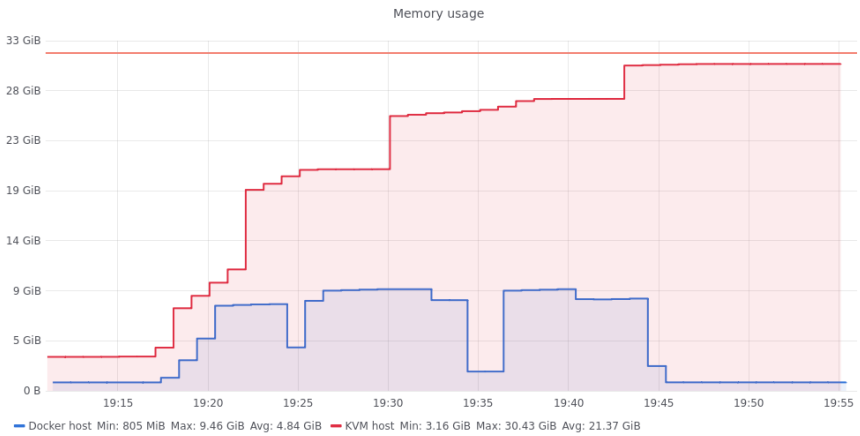
V testu jsou použity stejné větve jako v předchozí části. Nyní se však spouštějí všechny v jedné úloze. Ve virtuální stroji se vždy spustí postupně a u Dockeru je nejprve spuštěn jeden kontejner s postupným vykonáváním a napodruhé jsou spuštěny 3 kontejnery pro každou větev zvlášť. Změřené hodnoty jsou k dispozici v tabulce 7.2. Grafy využití procesoru a paměti jsou pro postupný běh znázorněny v obrázcích 7.3 a 7.4. Pro paralelní vykonávání jsou k dispozici v 7.5 a 7.6. Přestože paralelní test vychází lépe, mohl by být pravděpodobně ještě zrychlen. Úloha totiž vždy na začátku rozbaluje balíky s nástroji, ať už zpracovává jen jednu větev nebo všechny. Pokud by toto bylo již připraveno, výsledky by byly ještě lepší.

Provádění úlohy	**Doba běhu v Dockeru**	**Doba běhu v KVM**	**Zrychlení**
postupně	28 m	35 m	20 %
paralelně	25 m	35 m	29 %

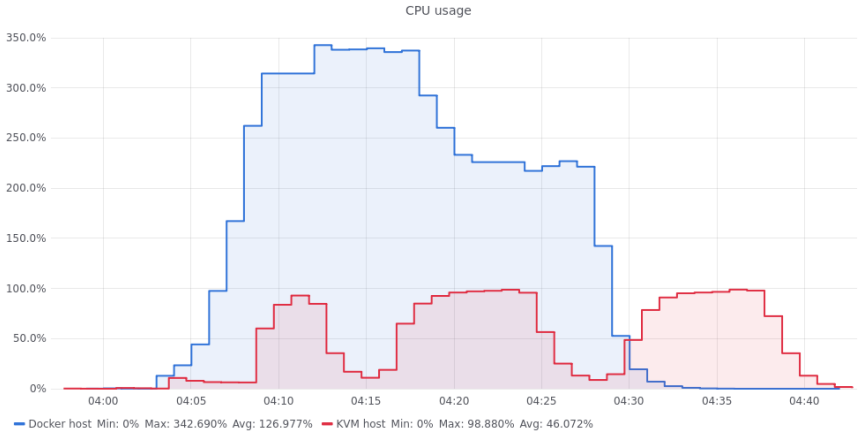
Tabulka 7.2: Výsledky měření druhého testu.



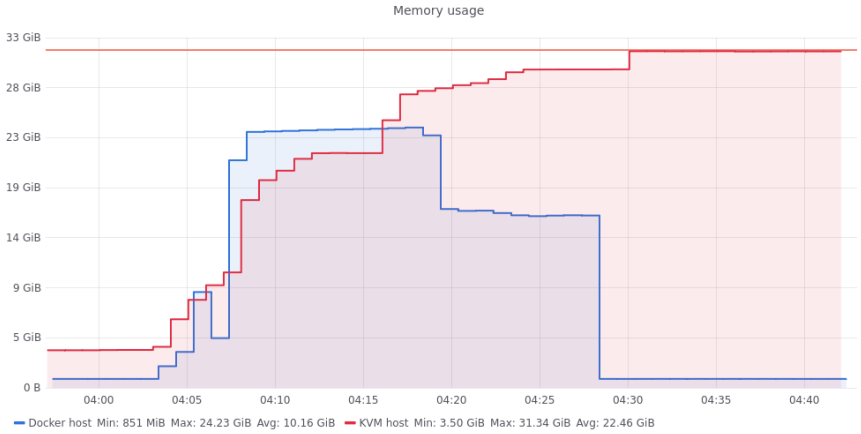
Obrázek 7.3: Graf využití procesoru při postupném vykonávání úlohy.



Obrázek 7.4: Graf využití paměti při postupném vykonávání úlohy.



Obrázek 7.5: Graf využití procesoru při paralelním vykonávání úlohy.



Obrázek 7.6: Graf využití paměti při paralelním vykonávání úlohy.

V případě grafů 7.3 a 7.4, kdy je zpracování úlohy prováděno postupně, je dobře patrné vykonávání jednotlivých větví, které je rozděleno do tří extrémů. Zde je ještě více patrné, že v kontejneru proběhne úloha rychleji (zpracování poslední větve v kontejneru končí v podobnou dobu, jako začíná být zpracována na virtuálním stroji).

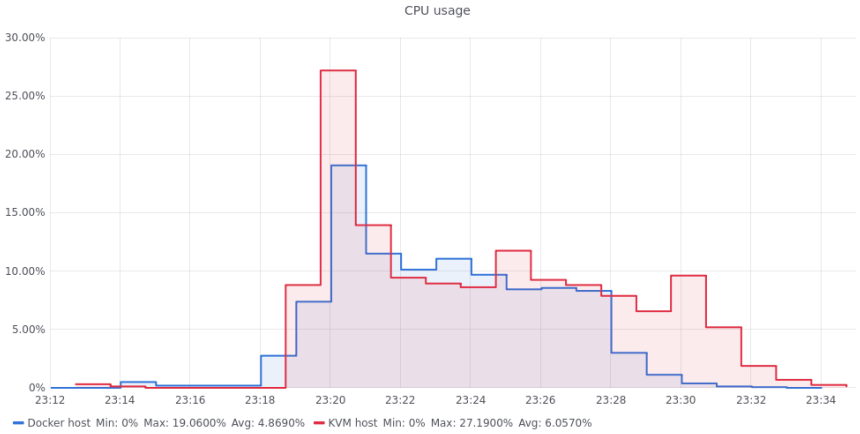
V grafech 7.5 a 7.6 je průběh na virtuálním stroji totožný, protože se jedná o stejnou úlohu. Na Docker hostiteli byly ale spuštěny tři nezávislé kontejnery, které se vykonávají všechny najednou. To vede k dočasnému přetížení počítače, které je patrné z grafu využití CPU, které dosahuje v maximu využití až okolo 300 %.

7.3 Test uRISC uvm applications se simulátory třetích stran

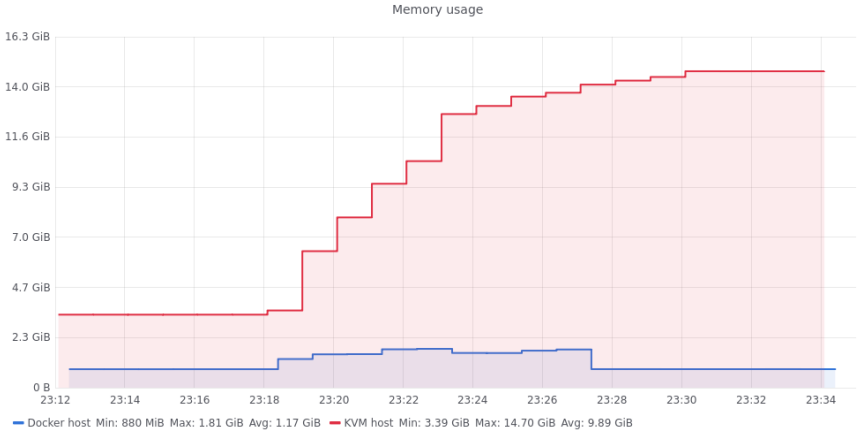
Tento test provádí druhou úlohu nad procesorem Codasip uRISC a využívá při tom simulátory Questa a Riviera. Test je spuštěn pro dvě větve a hodnoty měření jsou k dispozici v tabulce 7.3. Grafy využití procesoru a paměti jsou uvedeny pro první měření v obrázcích 7.7 a 7.8.

Doba běhu v Dockeru	Doba běhu v KVM	Zrychlení
8 m 56 s	12 m	26 %

Tabulka 7.3: Výsledky měření třetího testu.



Obrázek 7.7: Graf využití procesoru pro třetí test.



Obrázek 7.8: Graf využití paměti pro třetí test.

Z grafů je opět patrné, že zpracování v kontejneru je pro tuto úlohu skutečně rychlejší. Zároveň je možné diskutovat o využití paměti hostitele s KVM hypervizorem. Ten spotřebovává paměti výrazně více, přitom KVM alokuje paměť až poté, co je skutečně potřebována. Z tohoto důvodu by bylo možné test rozšířit o pozorování paměti uvnitř virtuálního stroje.

Kapitola 8

Závěr

Cílem práce bylo vytvořit funkční prostředí pro spouštění paralelních úloh k vývoji a testu softwaru na linuxových kontejnerech. To se podařilo splnit včetně drobných vylepšení, jako je uživatelsky přívětivá správa imageů pomocí aplikace Portus nebo plně automatické sestavení konfigurace nástrojem Ansible. Součástí bylo také na těchto úlohách srovnání plné virtualizace a virtualizace na úrovni operačního systému v podání softwaru KVM a Docker. Výsledky jsou pro Docker velmi příznivé, rychlost prováděných úloh byla zvýšena v průměru o 24 %, čímž byla potvrzena myšlenka, že použití kontejnerů bude efektivnější.

Práce obsahuje obecné porovnání virtualizačních technik i přehled zástupců softwaru, který se v dnešní době k virtualizaci používá. Proběhlo podrobné seznámení se s požadavky na zadaný systém, ze kterého vzešel jeho návrh, který počítá s propojením s nástrojem Jenkins pro automatizaci procesu vývoje softwaru. Systém byl implementován tak, aby sám dokázal připravit nezbytné prostředí pro vykonávání úloh. Toho je dosaženo pomocí předchystaných imagů pro kontejnery v Dockeru. Pomocí nástroje Jenkins je také možné řídit využití výpočetních prostředků na Docker hostiteli. Nakonec byly provedeny experimenty, které dopadly pozitivně.

Dalšími kroky, které by mohly práci rozšířit, je použití Dockeru a spouštění kontejnerů na systému Windows, pro co nezbyl čas ani ke krátkému vyzkoušení. Je to část nutná k docílení toho, aby mohly být kontejnery v Codasipu skutečně využity. Také nedošlo k vyzkoušení softwaru Kubernetes, který konkuruje nástroji swarm od Dockeru a umí spouštět i jiné kontejnery. Tím by mohlo být docíleno lepších možností rozplánování napříč více hostiteli se softwarem Docker a také lepší správy výpočetních prostředků. Do budoucna by bylo vhodné k tomu nahradit úložiště imageů Portus jinou aplikací, jejichž vývoj bude stále probíhat. Poslední věcí, která byla v rámci práce i částečně testována, je možnost odkládání kontejnerů, na kterých se úloha nezdaří. Zatím se za každých okolností kontejner zastaví a smaže, což je ale pro zjišťování důvodů, proč se úloha nepodařila, nepříjemné. Tuto funkcionalitu je možné řešit pomocí řídící úlohy, která by spravovala běh kontejneru a spouštěla na něm potřebnou podúlohu. Docker plugin k tomu dokonce nabízí speciální kroky v definici úlohy pro správu Dockeru. Ty však bohužel nefungují správně a není možné je použít. Nahradit by se však daly skriptem, který by byl v rámci řídící úlohy spuštěn a který by připravil kontejnery pomocí komunikace s Docker REST API.

Literatura

[1] Banerjee, T. Understanding the key differences between LXC and Docker. *Flockport* [online]. 2014, [cit. 2020-05-01]. Dostupné z: https://archives.flockport.com/lxc-vs-docker/.

[2] Bellard, F. *QEMU 4.2.94 Documentation*. 2020 [cit. 2020-04-85]. Dostupné z: https://www.qemu.org/docs/master/.

[3] Brodkin, J. With long history of virtualization behind it, IBM looks to the future. *Network World* [online]. 2009, [cit. 2020-04-13]. Dostupné z: https://www.networkworld.com/article/2254433/with-long-history-of-virtualization-behind-it--ibm-looks-to-the-future.html.

[4] Brown, N. Scheduling Services on a Docker Swarm Mode Cluster. *Semaphore* [online]. 2017, [cit. 2020-05-20]. Dostupné z: https://semaphoreci.com/community/tutorials/scheduling-services-on-a-docker-swarm-mode-cluster.

[5] Cerling, T., Buller, J., Enstall, C. a Ruiz, R. *Mastering Microsoft Virtualization* [online]. Sybex, 2009 [cit. 2020-04-04]. ISBN 978-0-470-44958-5. Dostupné z: https://learning.oreilly.com/library/view/mastering-microsoft-virtualization/9780470449585/.

[6] Chirammal, H. D., Mukhedkar, P. a Vettathu, A. *Mastering KVM Virtualization* [online]. Birmingham: Packt, 2016 [cit. 2020-04-04]. ISBN 978-1-78439-905-4. Dostupné z: https://learning.oreilly.com/library/view/mastering-kvm-virtualization/9781784399054/.

[7] DOCKER. *Docker Logo Blue* [online]. [cit. 2020-05-04]. Dostupné z: https://www.docker.com/company/newsroom/media-resources.

[8] DOCKER. *Docker Documentation*. 2020 [cit. 2020-05-14]. Dostupné z: https://docs.docker.com/.

[9] Goodin, D. Backdoored images downloaded 5 million times finally removed from Docker Hub. *Ars Technica* [online]. 2018, [cit. 2020-05-14]. Dostupné z: https://arstechnica.com/information-technology/2018/06/backdoored-images-downloaded-5-million-times-finally-removed-from-docker-hub/.

[10] Keating, J. *Mastering Ansible* [online]. Birmingham: Packt, 2015 [cit. 2020-05-22]. ISBN 978-1-78439-548-3. Dostupné z: https://www.ansible.com/resources/ebooks/mastering-ansible.

[11] Mouat, A. *Using Docker* [online]. Sebastopol: O'Reilly Media, Inc., 2015 [cit. 2020-05-14]. ISBN 978-1-49191-576-9. Dostupné z: https://learning.oreilly.com/library/view/using-docker/9781491915752/.

[12] Portnoy, M. *Virtualization Essentials* [online]. 2. vyd. Sybex, 2016 [cit. 2020-04-08]. ISBN 978-1-11-926772-0. Dostupné z: https://learning.oreilly.com/library/view/virtualization-essentials-2nd/9781119267720/.

[13] RED HAT. *Ansible Mark Mango* [online]. [cit. 2020-05-21]. Dostupné z: https://www.ansible.com/logos.

[14] RED HAT. *Ansible Documentation*. 2020 [cit. 2020-05-21]. Dostupné z: https://docs.ansible.com/ansible/latest/index.html.

[15] Smith, J. a Nair, R. *Virtual Machines: Versatile Platforms for Systems and Processes* [online]. Morgan Kaufmann, 2005 [cit. 2020-04-24]. ISBN 978-1-55-860910-5. Dostupné z: https://learning.oreilly.com/library/view/virtual-machines/9781558609105/.

[16] Vavrečková Šárka. *Operační systémy: Přednášky* [online]. Opava: Slezská univerzita v Opavě, 2017 [cit. 2020-04-13]. Dostupné z: http://vavreckova.zam.slu.cz/obsahy/os/ospredn/ospredn.pdf.

[17] Victor, J., Savit, J., Combs, G., Hayler, S. a Netherton, B. *Oracle Solaris 10 System Virtualization Essentials* [online]. Prentice Hall, 2010 [cit. 2020-04-04]. ISBN 978-0-13-708188-2. Dostupné z: https://learning.oreilly.com/library/view/oracle-solaris-10/9780137084067/.

[18] VMWARE. *Understanding Full Virtualization, Paravirtualization, and Hardware Assist*. 2007 [cit. 2020-04-25]. Dostupné z: https://www.vmware.com/content/dam/digitalmarketing/vmware/en/pdf/techpaper/VMware_paravirtualization.pdf.