Porovnání implementací síťového subsystému v Kubernetes

Lukáš Šiška

Bakalářská práce
2021

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky
Ústav automatizace a řídicí techniky

Akademický rok: 2020/2021

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

(projektu, uměleckého díla, uměleckého výkonu)

Jméno a příjmení:	**Lukáš Šiška**
Osobní číslo:	**A18178**
Studijní program:	**B3902 Inženýrská informatika**
Studijní obor:	**Informační a řídicí technologie**
Forma studia:	**Prezenční**
Téma práce:	**Porovnání implementací síťového subsystému v Kubernetes**
Téma práce anglicky:	**A Comparison of Network Subsystems Implementation in Kubernetes**

Zásady pro vypracování

1. Seznamte se s orchestračním systémem Kubernetes a základními principy tvorby distribuovaných kontejnerových aplikací.
2. Prostudujte problematiku interní síťové komunikace v clusteru Kubernetes.
3. Vhodným způsobem vyberte a popište základní dostupné implementace síťového subsystému pro Kubernetes.
4. Sestavte metodiku testování jednotlivých implementací síťového subsystému.
5. Na základě sestavené metodiky otestujte vybrané implementace a proveďte zhodnocení.

Forma zpracování bakalářské práce: **Tištěná/elektronická**

Seznam doporučené literatury:

1. HUANG, Kaizhe a Pranjal JUMDE. *Learn Kubernetes Security: Securely orchestrate, scale, and manage your microservices in Kubernetes deployments*. Birmingham, UK: Packt Publishing, 2020. ISBN 978-1-83921-650-3.
2. BAIER, Jonathan a Jesse WHITE. *Getting Started with Kubernetes: Extend your containerization strategy by orchestrating and managing large-scale container deployments.* 3rd edition. Birmingham, UK: Packt Publishing, 2018. ISBN 978-1-78899-472-9.
3. BAIER, Jonathan, Gigi SAYFAN a Jesse WHITE. *The Complete Kubernetes Guide: Become an expert in container management with the power of Kubernetes.* Birmingham, UK: Packt Publishing, 2019. ISBN 978-1-83864-734-6.
4. BURNS, Brendan. *Designing Distributed Systems*. O'Reilly Media, 2018. ISBN 978-1-49198-364-5.
5. SAYFAN, GIGI. *Mastering Kubernetes*. Birmingham, UK: Packt Publishing, 2017. ISBN 978-1-78646-100-1.
6. GOASGUEN, Sebastien a Michael HAUSENBLAS. *Kubernetes Cookbook*. O'Reilly Media, 2018. ISBN 978-1-49197-968-6

Vedoucí bakalářské práce: **Ing. Peter Janků, Ph.D.**
Ústav informatiky a umělé inteligence

Datum zadání bakalářské práce: **15. ledna 2021**
Termín odevzdání bakalářské práce: **17. května 2021**

doc. Mgr. Milan Adámek, Ph.D. v.r.
děkan

UNIVERZITA TOMÁŠE BATI VE ZLÍNĚ
Fakulta aplikované informatiky
DĚKAN
L.S.

prof. Ing. Vladimír Vašek, CSc. v.r.
ředitel ústavu

Ve Zlíně dne 15. ledna 2021

Prohlašuji, že

- beru na vědomí, že odevzdáním bakalářské práce souhlasím se zveřejněním své práce podle zákona č. 111/1998 Sb. o vysokých školách a o změně a doplnění dalších zákonů (zákon o vysokých školách), ve znění pozdějších právních předpisů, bez ohledu na výsledek obhajoby;
- beru na vědomí, že bakalářská práce bude uložena v elektronické podobě v univerzitním informačním systému dostupná k prezenčnímu nahlédnutí, že jeden výtisk bakalářské práce bude uložen v příruční knihovně Fakulty aplikované informatiky Univerzity Tomáše Bati ve Zlíně;
- byl/a jsem seznámen/a s tím, že na moji bakalářskou práci se plně vztahuje zákon č. 121/2000 Sb. o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon) ve znění pozdějších právních předpisů, zejm. § 35 odst. 3;
- beru na vědomí, že podle § 60 odst. 1 autorského zákona má UTB ve Zlíně právo na uzavření licenční smlouvy o užití školního díla v rozsahu § 12 odst. 4 autorského zákona;
- beru na vědomí, že podle § 60 odst. 2 a 3 autorského zákona mohu užít své dílo – bakalářskou práci nebo poskytnout licenci k jejímu využití jen připouští-li tak licenční smlouva uzavřená mezi mnou a Univerzitou Tomáše Bati ve Zlíně s tím, že vyrovnání případného přiměřeného příspěvku na úhradu nákladů, které byly Univerzitou Tomáše Bati ve Zlíně na vytvoření díla vynaloženy (až do jejich skutečné výše) bude rovněž předmětem této licenční smlouvy;
- beru na vědomí, že pokud bylo k vypracování bakalářské práce využito softwaru poskytnutého Univerzitou Tomáše Bati ve Zlíně nebo jinými subjekty pouze ke studijním a výzkumným účelům (tedy pouze k nekomerčnímu využití), nelze výsledky bakalářské práce využít ke komerčním účelům;
- beru na vědomí, že pokud je výstupem bakalářské práce jakýkoliv softwarový produkt, považují se za součást práce rovněž i zdrojové kódy, popř. soubory, ze kterých se projekt skládá. Neodevzdání této součásti může být důvodem k neobhájení práce.

Prohlašuji,

- že jsem na bakalářské práci pracoval samostatně a použitou literaturu jsem citoval. V případě publikace výsledků budu uveden jako spoluautor.
- že odevzdaná verze bakalářské práce a verze elektronická nahraná do IS/STAG jsou totožné.

Ve Zlíně, dne

Lukáš Šiška, v. r.
podpis studenta

ABSTRAKT

Bakalářská práce se bude zabývat možnostmi implementace síťového subsystému v Kubernetes. V rámci teoretické části bude nastíněna problematika síťové komunikace v rámci jednotlivých kontejnerů provozovaných v clusteru Kubernetes. Dále bude uveden přehled vybraných implementací síťového subsystému a diskutovány jejich základní vlastnosti. V praktické části bude sestavena metodika testování jednotlivých subsystémů a bude následovat porovnání vybraných. V rámci porovnání se student zaměří jednak na propustnost jednotlivých implementací, tak na složitost jejich nasazení a možnost konfigurovatelnosti.

Klíčová slova: Kubernetes, Docker, síťová komunikace, kontejnerová aplikace, virtualizace (počítače)

ABSTRACT

The bachelor thesis will deal with the possibilities of implementing the network subsystem in Kubernetes. The theoretical part will outline the problematics of network communication within individual containers operated in the Kubernetes cluster. There will be an overview of selected network subsystem implementations and their basic properties will be discussed. In the practical part, the methodology of testing individual subsystems will be constructed and a comparasion of selected ones will follow. In the comparison, the student will focus on the throughput of individual implementations, as well as on the complexity of their deployment and the possibility of configurability.

Keywords: Kubernetes, Docker, network communication, container application, virtualization (computers)

Tímto bych rád bych poděkoval Ing. Peterovi Janků, Ph.D. za cenné rady, věcné připomínky a vstřícnost při konzultacích a vypracování této bakalářské práce.

„SIC PARVIS MAGNA – Veliké věci přicházejí z těch malých“.

Sir Francis Drake

Prohlašuji, že odevzdaná verze bakalářské práce a verze elektronická nahraná do IS/STAG jsou totožné.

OBSAH

ÚVOD **9**
I TEORETICKÁ ČÁST **10**
1 VIRTUALIZACE **11**
1.1 DRUHY VIRTUALIZACE 11
1.1.1 Plná virtualizace 11
1.1.2 Paravirtualizace 12
1.1.3 Kontejnerová virtualizace 12
1.1.4 Emulace 13
2 DOCKER **14**
2.1 DOCKER ENGINE 14
2.1.1 Docker daemon 15
2.1.2 Docker API 15
2.1.3 Docker klient 15
2.2 DOCKER REGISTR 16
2.3 DOCKER OBJEKTY 17
2.3.1 Image 17
2.3.2 Container 17
2.3.3 Volume 18
2.3.4 Network 18
3 KUBERNETES **19**
3.1 NODE 19
3.1.1 Master 20
3.1.2 Worker 20
3.2 POD. 20
3.3 SERVICE 21
3.4 REPLICATION CONTROLLER 22
3.5 NETWORK POLICY 22
4 SÍŤOVÁ KOMUNIKACE **24**
4.1 KONTEJNERY V PODU 24
4.2 POD TO POD 24
4.3 POD TO SERVICE 24
5 IMPLEMENTACE SÍŤOVÉHO SUBSYSTÉMU **26**
5.1 CONTAINER NETWORK INTERFACE 26
5.1.1 ADD 27
5.1.2 DEL 27
5.1.3 CHECK 27
5.1.4 VERSION 27
5.2 ANTREA 28
5.2.1 Instalace 28
5.2.2 Konfigurace 29
5.3 CALICO 29
5.3.1 Instalace 30

5.3.2 Konfigurace 30
5.4 CILIUM 31
5.4.1 Instalace 32
5.4.2 Konfigurace 32
5.5 FLANNEL 33
5.5.1 Instalace 33
5.5.2 Konfigurace 33
5.6 WEAVE NET 33
5.6.1 Instalace 33
5.6.2 Konfigurace 34
II PRAKTICKÁ ČÁST 35
6 PŘÍPRAVA WORKSPACE 36
6.1 GOOGLE COMPUTE ENGINE 36
6.1.1 Vytvoření virtuálního stroje 36
6.1.2 Konfigurace virtuálního stroje 37
6.1.3 Vytvoření clusteru 38
7 METODIKA TESTOVÁNÍ 40
7.1 PROPUSTNOST 40
7.1.1 Iperf 40
7.1.2 Kubernetes Bench Suite 42
7.2 VYUŽITÍ PROCESORU 42
7.2.1 Sar 42
7.2.2 Kubernetes Bench Suite 43
7.3 VYUŽITÍ OPERAČNÍ PAMĚTI 43
7.3.1 Free 44
7.3.2 Kubernetes Bench Suite 44
7.4 LATENCE 44
7.4.1 Qperf 45
7.4.2 Kubernetes Qperf 45
8 TESTOVÁNÍ 47
8.1 PROPUSTNOST 47
8.1.1 TCP protokol 47
8.1.2 UDP protokol 49
8.2 VYUŽITÍ PROCESORU 50
8.2.1 Nečinnost 50
8.2.2 TCP protokol 51
8.2.3 UDP protokol 52
8.3 VYUŽITÍ OPERAČNÍ PAMĚTI 53
8.3.1 Nečinnost 53
8.3.2 TCP protokol 54
8.3.3 UDP protokol 55
8.4 LATENCE 56
8.5 SHRNUTÍ 57
ZÁVĚR 59

SEZNAM POUŽITÉ LITERATURY **61**
SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK **64**
SEZNAM OBRÁZKŮ **65**
SEZNAM TABULEK **67**
SEZNAM PŘÍLOH **68**

ÚVOD

Tématem bakalářské práce je porovnání implementací síťového subsystému, zajišťující síťovou komunikaci mezi kontejnery, používané v rámci orchestračního systému Kubernetes. Cílem této bakalářské práce je získání povědomí o síťové komunikaci v clusteru, především o vybraných implementací, které implementují Container Network Interface v Kubernetes.

Pokud využíváme služeb pro správu Kubernetes clusteru (např. Google GKE, Amazon EKS), implementace síťového subsystému je již vybrána a není potřeba podnikat další kroky v tomto směru. V opačném případě, kdy si cluster spravujeme sami, je třeba implementaci vybrat, nasadit a nakonfigurovat.

Teoretická část má za úkol seznámit obecně s orchestračním systémem Kubernetes a jeho základními komponentami, využívající Docker jako container runtime na principu kontejnerové virtualizace. Součástí je též představení síťové komunikace v Kubernetes clusteru, včetně výběru a popisu později testovaných a porovnávaných implementací síťového subsystému.

Pro možnost otestování a porovnání implementací síťového subsystému v Kubernetes je v praktické části vytvořeno vývojové prostředí potřebné k simulaci Kubernetes clusteru. Toto prostředí je tvořeno dvěma virtuálními stroji, které slouží k potřebné simulaci clusteru. Po nakonfigurování clusteru jsou vybrané implementace síťového subsystému jednotlivě zaintegrovány a nakonfigurovány.

V další kapitole praktické části je sestavena metodika pro testování síťového subsystému, z něhož jsou získána data potřebná k následnému porovnání. Na závěr jsou vybrané implementace porovnány v několika oblastech, jako je například propustnost, využití prostředků, složitost nasazení či konfigurovatelnost samotné implementace.

I. TEORETICKÁ ČÁST

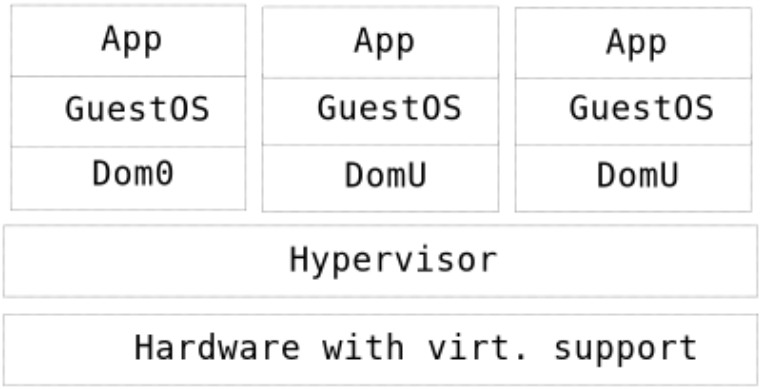
1 VIRTUALIZACE

Virtualizace je technologie umožňující provoz více nezávislých virtuálních strojů na jednom reálném (fyzickém) stroji. V podstatě představuje určitou iluzi, v níž z některých fyzických zdrojů (např. procesor, operační paměť, disk) vytvoříme několik kopií. Jedná se pouze o koncepty, nikoliv reálné objekty. Uživateli může tento princip v konečném důsledku nabídnout kompletní virtuální počítač, složený z právě zmiňovaných virtuálních komponent. Vytváří se tak pocit vlastnictví tohoto počítače, kdy jsou ale reálně fyzické prostředky sdíleny s dalšími uživateli. [9]

1.1 Druhy virtualizace

1.1.1 Plná virtualizace

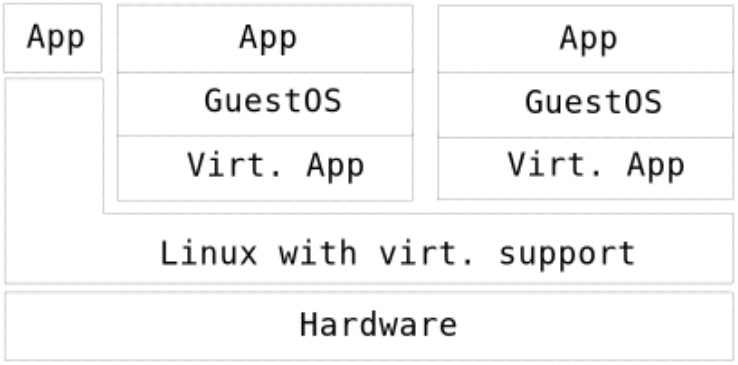
Při plné virtualizaci virtualizujeme všechny součásti počítače, kdy operační systém běžící virtuálně nedokáže rozpoznat, že nemá přístup k fyzickému vybavení. Programy běží na virtuálním hardware, kdežto přístup k fyzickému vybavení je jim pouze zprostředkován. Díky tomu, že virtuální prostředí nemá žádnou vazbu k reálnému vybavení, je možno využívat plné přenositelnosti. Tato vlastnost zajišťuje možnost mít virtuální prostředí provozované nad jakýmkoliv hardwarem bez nutnosti úprav na úrovni virtuálního počítače. Na druhou stranu využití tohoto druhu virtualizace může vést ke snížení výkonu z důvodu nutnosti emulace fyzického vybavení, kdy se většina operací provádí ve vlastním software, nikoliv aby je vykonával přímo hardware. Zjednodušené schéma plné virtualizace lze vidět na obrázku (Obr. 1). [7][8]



Obrázek 1. Schéma principu plné virtualizace [7]

1.1.2 Paravirtualizace

Pokud se některé komponenty virtuálního a fyzického počítače shodují, je možno místo principu plné virtualizace využít tzv. paravirtualizaci, zobrazenou na obrázku (Obr. 2). V takovém případě se provádí pouze částečná abstrakce na úrovni virtuálního počítače, tudíž nabízí prostředí podobné tomu fyzickému. Oproti plné virtualizaci má tu výhodu, že hostovaný systém ví o jeho běhu ve virtuálním prostředí a dokáže tak efektivně komunikovat s hypervisorem, který zprostředkovává přístup těchto počítačů k hardwaru. [7][8]



Obrázek 2. Schéma principu paravirtualizace [7]

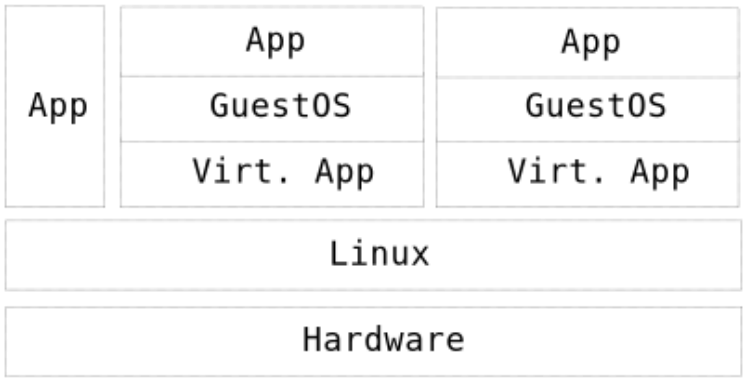
1.1.3 Kontejnerová virtualizace

K virtualizaci dochází až na úrovni operačního systému za pomoci tzv. kontejnerů. V rámci jednoho operačního systému jsou vytvářena oddělená prostředí, sdílející jádro a prostředky hostitelského systému, která jsou izolována od vlivů právě hostitelského systému či jiných procesů (Obr. 3). Příkladem kontejnerové virtualizace je nyní hojně využívaný open source software Docker, jež nabízí jednotné rozhraní pro správu kontejnerů s izolovanými aplikacemi napříč různými prostředí. [7]

Obrázek 3. Schéma principu kontejnerové virtualizace [7]

1.1.4 Emulace

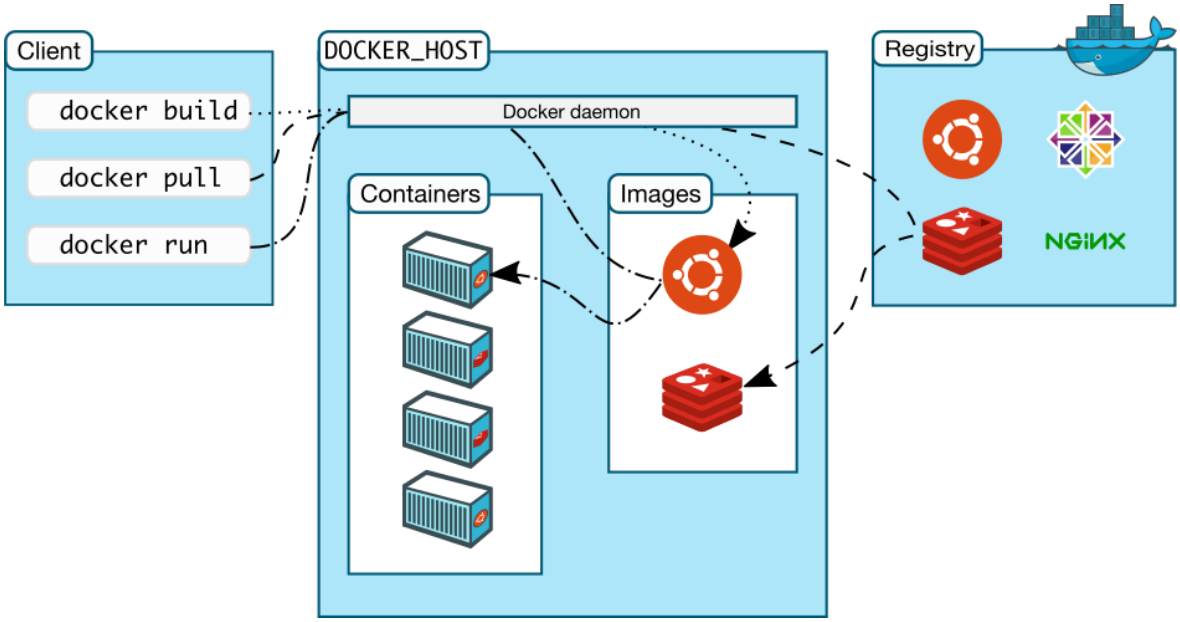
Emulace, na rozdíl od ostatních druhů virtualizace, umožňuje provozovat virtuální prostředí jiné architektury na hostujícím systému. K dosažení této vlastnosti je třeba všechny prováděné operace interpretovat, což má za následek snížení výkonu. Samotná emulace je založena na interpretaci, či statickém nebo dynamickém překladu. V případě interpretace emulátor prochází kód programu instrukci po instrukci a mění stav hostovaného systému. Takové načtení a zpracování každé jednotlivé instrukce je neefektivní. Naopak statický překlad programu nejprve celý program zpracuje a následně provede. Nejefektivnější a nejvíce rozšířenou možnou variantou je využití dynamického překladu, kdy se překlad provádí až za běhu. [7]



Obrázek 4. Schéma principu emulace [7]

2 DOCKER

Open platforma Docker, napsaná v programovacím jazyce Go, poskytuje jednotné rozhraní pro vývoj, běh a dodávání aplikací. Funguje na principu virtualizace na úrovni operačního systému neboli kontejnerových aplikací. Umožňuje tedy izolovat aplikace od ostatních za pomoci kontejnerů, díky čemuž je vývoj i dodávání hotových řešení o mnoho rychlejší. Samotné schéma takové architektury je ukázáno na obrázku (Obr. 5). [11]



Obrázek 5. Schéma architektury Docker platformy [11]

2.1 Docker Engine

Docker Engine je open source technologie pro vytváření kontejnerových aplikací, působící jako klient-server aplikace. Tato aplikace se skládá ze tří hlavních součástí:

- Server jako dlouho běžící proces neboli Docker daemon
- API specifikující rozhraní pro možnost komunikace se serverem
- Klient komunikující se serverem přes API rozhraní

Klient odesílá požadavky přes komunikační rozhraní na server, ten je následně zpracovává a spravuje tak všechny své objekty. Díky tomuto rozhraní nemusí klient běžet na stejném hostitelském systému jako server, ale může se připojit i vzdáleně. [10]

2.1.1 Docker daemon

Docker daemon je schopen zpracovávat přijaté požadavky přes poskytnuté komunikační rozhraní a díky tomu může spravovat objekty jako jsou kontejnery, Docker images, networks či volumes. Vedle správy těchto objektů je poskytnuta možnost komunikace s ostatními Docker daemons. Jakmile server obdrží požadavek, zpracuje jej a provede patřičnou akci. [11]

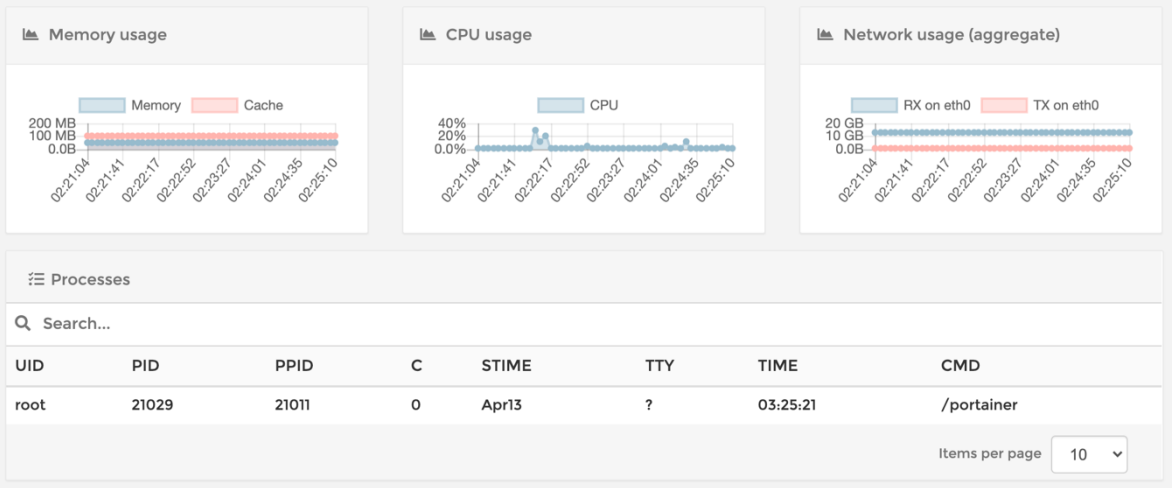
2.1.2 Docker API

Docker poskytuje aplikační rozhraní v podobě RESTful API, díky kterému je klient schopen komunikovat se serverem. Toto rozhraní poskytuje možnost volat konečné body na serveru a tím spravovat zmiňované objekty. [11]

2.1.3 Docker klient

Docker klient je nejrozšířenější cestou, jak uživatel může interagovat s Dockerem. V případě CLI klienta jsou příkazy odesílány přes komunikační rozhraní na server, který je následně zpracuje. Oproti CLI klientům jsou aktuálně dostupní i GUI klienti, jež nabízí stejnou funkcionalitu a v určitých případech je jejich použití efektivnější, nežli použití CLI klienta. Jedním z takových klientů je Portainer, nabízející jednoduchou správu jak samotného Docker, tak i například Kubernetes clusteru či clusteru spravovaného Dockerem v nativním módu Swarm.

Jednou z výhod grafického rozhraní klienta může být například přehledné zobrazení různého druhu statistik, informací či logů o různých součástech systému.

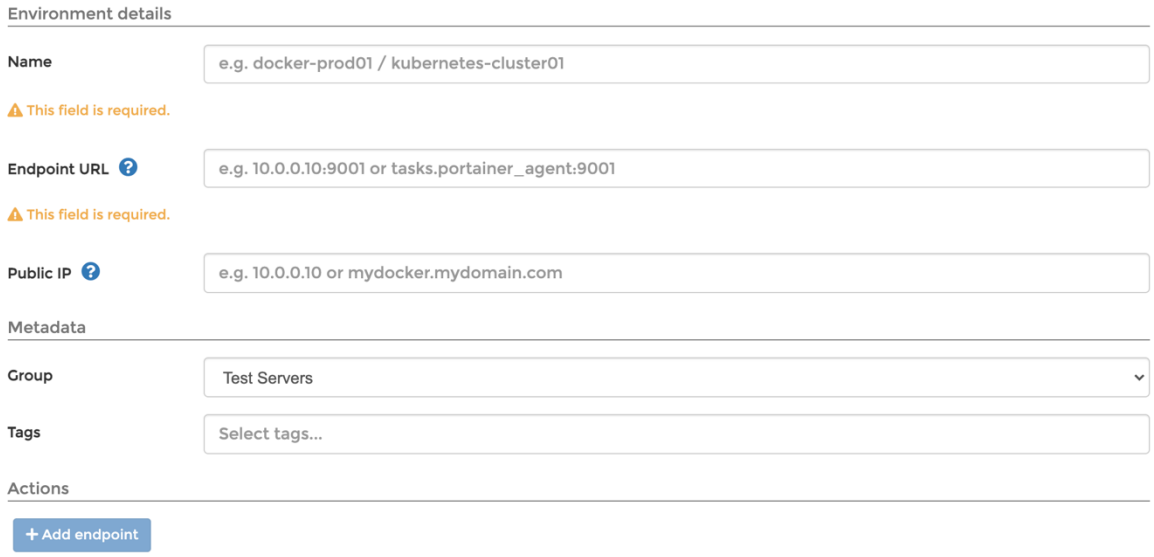


Obrázek 6. Statistiky běžícího kontejneru v Docker

Na obrázku (Obr. 6) lze vidět různé statistiky běžícího kontejneru (v tomto případě samotná aplikace Portainer zabalená v kontejneru), jako je využití operační paměti, procesoru nebo agregované využití sítě.

Naopak v případě CLI klienta je využití určitých funkcionalit mnohdy efektivnější nežli v grafickém rozhraní. Z vývojářského pohledu je pohodlnější se například připojovat do prostředí kontejnerů přes rozhraní konzole nebo čištění částí systému pomocí příkazu „prune“.

Docker klient je též schopen komunikovat s více Docker daemons. Grafické rozhraní Portainer se ve skutečnosti skládá ze dvou částí, serveru a agenta. Server dokáže běžet samostatně bez nutnosti využití agentů. To znamená, že se do Portainer dokáže napojit přímo na Docker API určitého serveru. Díky této možnosti, jak s využitím agentů, tak napojením napřímo, je možnost dosáhnout centralizované správy všech potřebných serverů. Ukázka přidání Portainer agenta je zobrazena na obrázku (Obr. 7).



Obrázek 7. Formulář pro přidání konečného bodu do Portaineru

2.2 Docker registr

Docker registr slouží k ukládání Docker images. Existuje veřejný registr Docker Hub, který obsahuje různé veřejné images jak od prodejců software, open source projektů až po images nahrané komunitou. Tento veřejný registr také nabízí služby pro osobní úložiště images, kdy lze provozovat vedle veřejných osobních úložišť i úložiště soukromé (záleží na vybraném plánu v rámci Docker Hub). [11]

Platforma Docker též nabízí možnost provozovat vlastní soukromý registr. V základu je Docker nakonfigurován tak, že images vyhledává ve veřejném registru. Po úpravě konfigurace a přidání vlastního registru vyhledává images i v něm.

2.3 Docker objekty

2.3.1 Image

Jedním z nejdůležitějších prvků v platformě Docker pro provoz kontejnerových aplikací je image. Je to předpis dostupný pouze pro čtení, skládající se ze sady instrukcí pro vytvoření kontejneru. V případě potřeby je možné si vytvořit vlastní image, který splňuje dané požadavky. Po sestavení image je dostupný pouze lokálně a je třeba jej nahrát do Docker registru pro umožnění přístupu i jiným subjektům. [11]

K vytvoření vlastního image slouží soubor **Dockerfile** obsahující jednoduchou syntaxi pro definování jednotlivých kroků sestavení image. Nově vytvářený image může zároveň také vycházet z již existujícího image, kdy se image může přizpůsobit zvláštním požadavkům. Pokud je upraven **Dockerfile** existujícího image, jsou znovu sestaveny pouze změněné kroky. [11]

Ukázka vytvoření jednoduchého **Dockerfile** je vyobrazena na obrázku (Obr. 8).

```
FROM php:7.4-apache

LABEL maintainer="Lukáš Šiška"

# Install Xdebug 3.0
RUN pecl install xdebug && \
   echo 'zend_extension=/usr/local/lib/php/extensions/no-debug-non-zts-
20190902/xdebug.so' >> /usr/local/etc/php/php.ini && \
   echo 'xdebug.mode=debug' >> /usr/local/etc/php/php.ini && \
   echo 'xdebug.client_host=10.0.2.2' >> /usr/local/etc/php/php.ini && \
   echo 'xdebug.client_port=9000' >> /usr/local/etc/php/php.ini && \
   echo 'xdebug.start_with_request=yes' >> /usr/local/etc/php/php.ini
```

Obrázek 8. Ukázka souboru Dockerfile

2.3.2 Container

Spustitelná instance obrazu se nazývá container (kontejner). S kontejnerem lze díky Docker CLI či API provádět mnoho operací, například jej lze vytvořit, spustit, zastavit, odstranit, případně ho přiřadit do některé z Docker network či vytvořit nový image z aktuálního stavu, kterého kontejner aktuálně nabývá. [11]

Samotné spuštění aplikace v kontejneru je velice jednoduché. Stačí mít k dispozici Docker image z některého repository (veřejné nebo soukromé) a nástroj ke správě Docker engine.

```
docker run -it --rm -v $HOME/.ssh:/root/.ssh -e NAME=Lukáš debian:10 /bin/bash
```

Obrázek 9. Příkaz pro spuštění kontejnerové aplikace

Příkaz na obrázku (Obr. 9) spustí jednoduchý kontejner podle image pro Debian 10. Součástí spuštění je též zrcadlení SSH klíčů z domovského adresáře aktuálně přihlášeného uživatele do uvedeného adresáře v kontejneru. Díky parametru „**-it**“ a „**/bin/bash**“ bude běžet v interaktivním režimu v terminálu a bude v něm spuštěn bash, reprezentující příkazový řádek. Do kontejneru je též předána hodnota tzv. environment proměnné s názvem „**NAME**“ a hodnotou „**Lukas**“, která je v něm ihned po spuštění přístupná. Pomocí těchto proměnných lze řešit nějaká základní konfigurace. Po opuštění příkazové řádky bude kontejner automaticky smazán díky předanému parametru „**--rm**“.

2.3.3 Volume

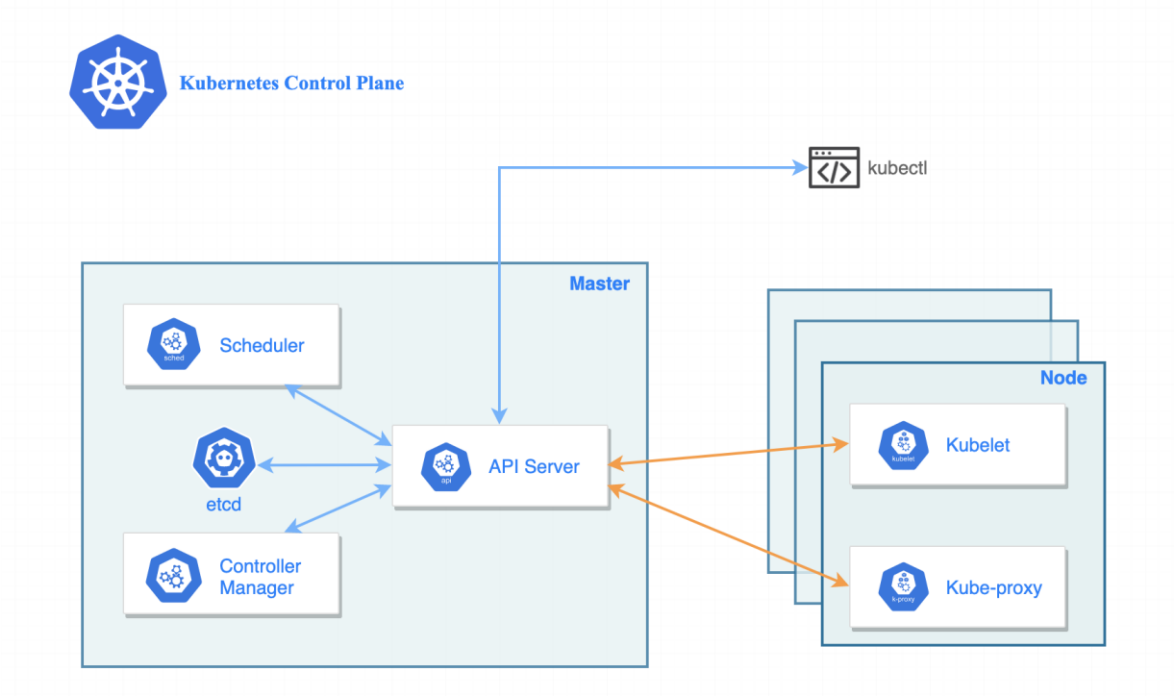
Docker volumes jsou určeny pro uchovávání vytvořených a používaných dat kontejnerů. Nabízejí nezávislost na operačním systému, jelikož jsou kompletně spravovány Dockerem. Nejsou vázány pouze na jeden určitý kontejner, ale mohou být bezpečně sdíleny i mezi více kontejnery. Jako i ostatní objekty je možné volumes spravovat přes aplikační rozhraní, které Docker nabízí. [12]

2.3.4 Network

Samotný Docker Engine nabízí možnosti vytvoření vlastních sítí, do kterých může být následně přiřazen některý z kontejnerů. Určitý kontejner vidí ostatní pouze v případě, že jsou napojeny do některé ze sítí, ve kterých se daný kontejner nachází. Každému z připojených kontejnerů je přiřazena IP adresa z dostupného rozsahu, podle konfigurace sítě.

3 KUBERNETES

Kubernetes je open-source projekt, primárně využívaný pro kontejnerovou orchestraci. To znamená zajistit, aby všechny kontejnery, provádějící různé úlohy, byly správně naplánovány a spuštěny dle požadované konfigurace v podobě YAML definic. Kromě toho musí Kubernetes také sledovat všechny běžící kontejnery a reagovat na změny či problémy v rámci clusteru, jako jsou například změny konfigurace a nahrazení mrtvých nebo nereagujících kontejnerů. [6]



Obrázek 10. Diagram základní struktury Kubernetes custeru [34]

Obrázek (Obr. 10) představuje diagram základní struktury v clusteru, včetně důležitých komponent. Kubernetes cluster je v tomto případě řízen konzolovou aplikací *kubectl*, která komunikuje s *API serverem* a vykonává tak zadané příkazy.

Konzolová aplikace není jediným způsobem, jak řídit Kubernetes cluster. Existuje mnoho dalších aplikací, dokonce i s grafickým uživatelský rozhraním. Příkladem může být aplikace *Portainer*, jež nabízí nově i správu právě Kubernetes clusteru.

3.1 Node

Důležitou částí clusteru jsou master a worker nody, které mohou být tvořeny jak fyzickým, tak virtuálním strojem.

3.1.1 Master

Master node je hlavním řídícím prvkem celého clusteru. Skládá se z několika komponent, jako je *API server*, poskytující možnost řízení clusteru, *scheduleru*, potřebného k plánování úkolů a *controller manageru*. Je zodpovědný za globální plánování spouštění podů a zpracovávání událostí. Může také zastávat roli worker node a provozovat některé z plánovaných podů. [6]

Veškerá data v rámci clusteru jsou ukládána v *etcd* úložišti, což je velmi konzistentní a dostupné tzv. klíč-hodnota úložiště. [34]

3.1.2 Worker

Worker node představuje jednoho z hostitelů v rámci clusteru, kdy jeho úkolem je provoz podů, které mu master node přidělí. V minulosti byl worker node označován také jako minion. Každý worker node obsahuje několik základních komponent, zajišťující základní funkcionalitu. [3]

Důležitou komponentou je *kubelet*, zajišťující komunikaci s API serverem na master nodu. Díky této komponentě lze vytvářet nové objekty naplánované přes *scheduler* či aktualizovat svůj stav. Nedílnou součástí je také container runtime, jež zajišťuje běh a správu všech kontejnerů. [3]

3.2 Pod

Jednotkou práce v Kubernetes je pod. Každý se může skládat z jednoho či více kontejnerů, sdílející totožnou IP adresu a určený rozsah portů. Tyto kontejnery v rámci podu komunikují nejčastěji přes lokální síť, může být ale využito i jiného druhu komunikace. [3]

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx-pod
spec:
  containers:
    - name: nginx
      image: nginx
      ports:
        - name: nginx
          containerPort: 80
          protocol: TCP
```

Obrázek 11. Definice podu v Kubernetes

Na obrázku (Obr. 11) je vyobrazena definice podu, kdy daný pod bude obsahovat pouze jeden kontejner s aplikací *nginx* a portem 80, přes který bude ke kontejneru přistupováno.

3.3 Service

Objekt service v Kubernetes poskytuje určitou míru abstrakce, definující množinu podů a zásad, na základě kterých k nim bude přistupováno. Pody jsou často vytvářeny a mazány z důvodu nutnosti dodržet nadefinovaný stav clusteru. Každý vytvořený pod získává svou vlastní IP adresu, díky které je v rámci clusteru identifikován. Pokud jsou ale pody dynamicky vytvářeny či mazány, nelze se na tyto IP adresy spolehnout. Pro tento moment je ideální využití service, která definuje vstupní bod do některého z podů s požadovanou aplikací a zároveň může řešit problematiku rozložení zátěže mezi patřičné pody. [31]

Service je dělena na několik typů:

- **ClusterIP** – základní typ, service je dostupná pouze v rámci clusteru na vnitřní IP adrese
- **NodePort** – na každém nodu otevírá statický port, kdy je možnost k service přistoupit přes IP adresu nodu a právě otevřený statický port
- **LoadBalancer** – je využito externí funkcionality rozložení zátěže, pokud to poskytovatel umožňuje
- **ExternalName** – service je přiřazen DNS název [31]

```yaml
apiVersion: v1
kind: Service
metadata:
  name: web-service
spec:
  selector:
    app: backend
  ports:
    - protocol: TCP
      port: 80
      targetPort: 8080
```

Obrázek 12. Definice service v Kubernetes

Obrázek (Obr. 12) ukazuje definice jednoduché service, jež směruje všechen tok dat do některého z existujících podů, identifikované díky selektoru *backend* a portu 8080. Jelikož v definici není nastaven typ service, je použit typ *ClusterIP* a komunikace tak probíhá pouze v rámci clusteru.

3.4 Replication controller

Replication controller (RC) definuje počet běžících podů ve stejný čas. Podle definice vytvoření RC je udržován požadovaný stav dané části v clusteru. Pokud existuje ve stejný čas více podů než bylo definováno, jsou přebývající odstraněny. To stejné platí i v případě, kdy je podů méně. Následně pak dochází k vytvoření chybějícího počtu, aby se dosáhlo požadovaného stavu. [3]

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: nginx
spec:
  replicas: 3
  selector:
    app: nginx
  template:
    metadata:
      name: nginx
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx
          ports:
            - containerPort: 80
```

Obrázek 13. Definice replication controller v Kubernetes

Definice uvedená na obrázku (Obr. 13) vytváří RC, kdy požadovaný počet podů je 3. Po dobu běhu RC je tak hlídán stav (počet) těchto podů. V případě změny stavu jsou provedeny patřičné akce (vytvoření nebo smazání podů), aby bylo opět dosaženo požadovaného stavu.

3.5 Network Policy

Network Policy neboli zásady sítě, kontrolují probíhající komunikaci a na základě definovaných pravidel tuto komunikaci řídí.

Existují dva typy zásad, které mohou být definovány současně:

- **Ingress** – řídí příchozí komunikaci do objektů dle definice
- **Egress** – řídí odchozí komunikaci z objektů dle definice

```yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: network-policy
spec:
  podSelector:
    matchLabels:
      app: backend
  policyTypes:
    - Ingress
  ingress:
    - from:
        - podSelector:
            matchLabels:
              role: frontend
```

Obrázek 14. Definice network policy v Kubernetes

Příkladem může být definice na obrázku (Obr. 14), jež využívá pouze zásadu sítě typu *ingress* a řídí tak příchozí komunikaci do podů, které jsou součástí role *frontend*. Pokud nějaký pod, který nenáleží do této role, bude chtít přistoupit k některému podu s rolí *backend*, bude mu tato akce zamítnuta.

4 SÍŤOVÁ KOMUNIKACE

Síťový model v Kubernetes je založen na adresovém prostoru o určitém rozsahu. Každý pod v clusteru má svou vlastní IP adresu a vidí v rámci sítě i ostatní pody. Naopak kontejnery v podech sdílí IP adresu konkrétního podu a mezi sebou komunikují přes lokální síť. Existuje několik způsobů, jakými je komunikace realizována. [5]

4.1 Kontejnery v podu

Pod je vždy spouštěn na některém z nodů, tudíž mohou všechny kontejnery komunikovat jak přes lokální síť, souborový systém nebo jiným způsobem v rámci tohoto konkrétního podu. Pokud například jeden pod obsahuje kontejner s otevřeným portem 1111 a druhý pod kontejner také s otevřeným portem 1111, nedochází ke konfliktu na stejném nodu. Pouze nesmí mít kontejnery totožný port v jednom podu. [5]

4.2 Pod to Pod

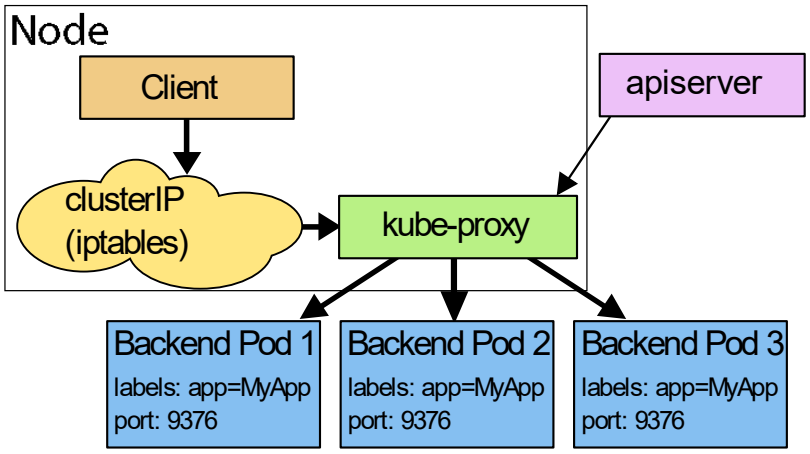
Pody v Kubernetes mají přiřazenou IP adresu, viditelnou v celé síti, tudíž mohou mezi sebou komunikovat bet nutnosti překladu adresu, tunelů a jiných dalších vrstev. Interní IP adresa podu je totožná s jeho externí adresou. Nevýhodou přímé komunikace mezi pody je fakt, že nemají pokaždé stejnou IP adresu. Nelze se tak na ni spolehnout a je potřebné využít jiných způsobů identifikace, například využití service. [5]

Při této přímé komunikaci též nedochází k využívání benefitu rozložení zátěže mezi patřičné pody, pokud taková možnost existuje.

4.3 Pod to Service

Pod se může v případě tohoto typu komunikace spolehnout na identifikaci service a neovlivňuje ji změna stavu podu za ním. O přesměrování toku dat se automaticky stará *Kube proxy* komponenta, obsažená na každém nodu. Poskytuje tak možnost plného využití funkcionality service, především rozložení zátěže mezi více podů. [5]

Obrázek (Obr. 15) zjednodušeně zobrazuje jeden ze způsobů komunikace klienta (v tomto případě podu) se service. Zde využitý typ service, ClusterIP, disponuje IP adresou v rámci clusteru, na kterou může klient přistupovat. Po přistoupení na danou IP adresu je pomocí *Kube proxy* dotaz nasměrován dle vytížení do jednoho ze tří dostupných podů.



Obrázek 15. Diagram komunikace klienta (podu) se service [31]

5 IMPLEMENTACE SÍŤOVÉHO SUBSYSTÉMU

Implementací síťového subsystému je dostupných mnoho, ale každá z nich disponuje rozdílnou složitostí samotné instalace či konfigurace. Některé mohou mimo jiné také poskytovat nastavení síťové politiky a šifrování komunikace. Důležitým faktorem při výběru vhodné implementace je výsledná propustnost sítě, společně s využitím poskytnutých prostředků jako procesor a operační paměť.

K porovnání je vybráno pět implementací na základě autorova uvážení. První vybranou implementací je Flannel, který se řadí mezi nejpoužívanější implementace v Kubernetes. Pětici doplňuje Calico, jež je vybrán z důvodu jeho nativní podpory v Google Kubernetes Engine a je tedy na této platformě hojně využíván. Tyto dvě implementace jsou následně doplněny od tři méně známé (Antrea, Cilium a Weave Net).

5.1 Container Network Interface

Container Network Interface (CNI) poskytuje specifikaci a knihovny pro možnost implementace rozšíření do aplikací, sloužící ke správě síťového rozhraní v Linuxových kontejnerech. V případě Kubernetes je tedy nutné, aby všechny dostupné implementace splňovali tuto specifikaci. CNI specifikace pouze definuje připojování kontejnerů do sítě a správu prostředků v případech, kdy je kontejner vytvořen nebo odstraněn. [1]

Specifikace definuje čtyři základní operace, kterými je implementace řízena. Tyto operace předávají určité parametry implementaci, dle kterých je spouštěná operace vykonána. Seznam parametrů je popsán v tabulce (Tab. 1). Konfigurace je následně předávána pomocí standardního vstupu. [24]

Tabulka 1. Předávané parametry při ovládání síťového subsystému [24]

Název parametru	**Popis parametru**
CNI_COMMAND	Indikuje požadovanou operaci (ADD, DEL, CHECK, VERSION)
CNI_CONTAINERID	Unikátní identifikátor kontejneru
CNI_NETNS	Cesta k network namespace

CNI_IFNAME	Název síťového rozhraní, které bude vytvořeno uvnitř kontejneru (pokud nelze vytvořit s daným názvem, dochází k chybě)
CNI_ARGS	Dodatečné argumenty definované konkrétní implementací
CNI_PATH	Seznam cest ke spustitelným souborům konkrétní implementace

5.1.1 ADD

Operace ADD slouží k přidání kontejneru do sítě nebo také k aplikování změn v rámci předaného network namespace. Na základě parametru s názvem síťového rozhraní je toto rozhraní vytvořeno uvnitř kontejneru v požadovaném network namespace. V druhém případě může být pouze upravena konfigurace již vytvořeného rozhraní. Důležitým parametrem je tedy i samotný unikátní identifikátor kontejneru. [24]

5.1.2 DEL

K odstranění kontejneru ze sítě je využívána operace DEL. Ta podle předaných parametrů s identifikátorem kontejneru a názvem síťového rozhraní toto rozhraní odstraní a kontejner tak již nemůže komunikovat v rámci této sítě. [24]

5.1.3 CHECK

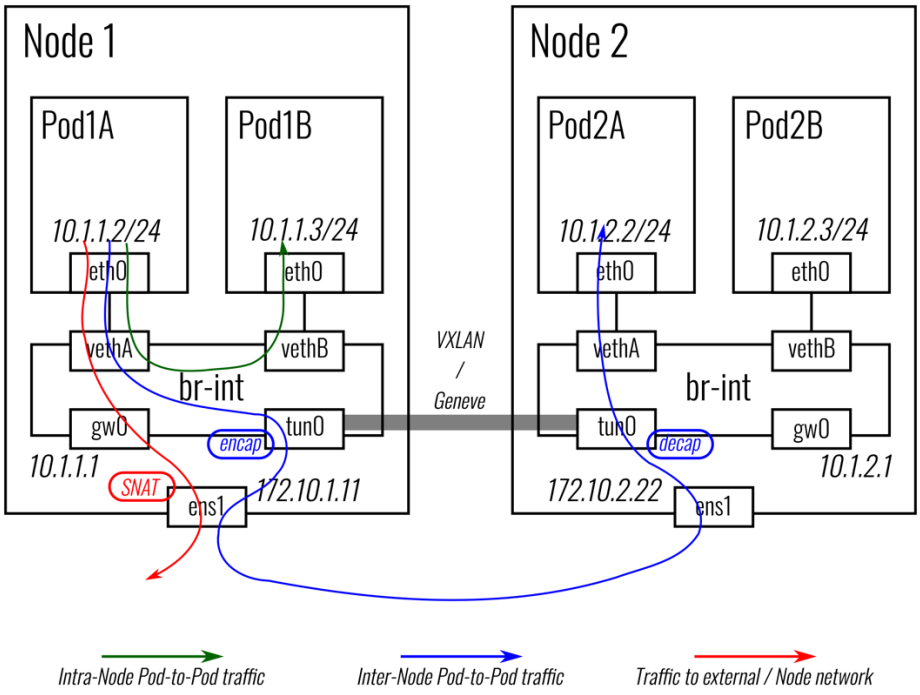
Cílem operace CHECK je kontrola stavu existujícího kontejneru, kdy hlavní kontrolovanou oblastí je samotná síť, podle definovaných pravidel implementace či používaného container runtime. Vedle parametru s názvem operace jsou dále předány totožné parametry jako tomu bylo při vykonání ADD operace v době vytvoření kontejneru. Včetně parametrů je na standardním vstupu předán objekt s daty, který poskytuje container runtime. [24]

5.1.4 VERSION

Úlohou VERSION operace je získání informace o podporovaných verzích CNI. Co se týče parametrů, je předán pouze název operace a na standardním vstupu je uvedena aktuální verze CNI, která je následně vrácena společně se seznamem podporovaných verzí. [24]

5.2 Antrea

Antrea je jednou z možných implementací síťového subsystému v Kubernetes, poskytující správu sítě a bezpečnosti komunikace v clusteru na síťové či transportní vrstvě. K poskytování síťové komunikace mezi pody a bezpečnostních funkcí využívá Open vSwitch (OVS), jež působí jako výkonný virtuální switch a zajišťuje tak efektivně nejen zásady sítě. [25]



Obrázek 16. Komunikace mezi pody s implementací Antrea [26]

Na každém nodu je vytvořen OVS bridge, označen jako „**br-int**“ na obrázku (Obr. 16), ve kterém je pro každý pod vytvořeno virtuální síťové zařízení (veth), propojující bridge s konkrétním podem. Každému podu je přiřazena IP adresa v rámci nodu z dané podsítě. [26]

Komunikace mezi dvěma lokálními pody probíhá napřímo přes zmiňovaný OVS bridge. V případě komunikace podů na rozdílných nodech jsou packety nejprve nasměrovány do tunelu (tun0) umístěného v bridge, zapouzdřeny a odeslány přes tunel do druhého nodu. Po přijetí je získána původní forma packetů a následně předána přes tunel (tun0) v bridge do cílového podu. [26]

5.2.1 Instalace

Složitost instalace závisí na kladených požadavcích na síťový subsystém. V případě využití základní konfigurace dodávané v YAML definici, bez nutnosti dalších úprav, lze implementaci jednoduše nainstalovat pomocí příkazu *kubectl apply -f antrea.yml*, kterému je předána

cesta k definici. Ihned po spuštění příkazu jsou vytvořeny komponenty vyžadované touto implementací.

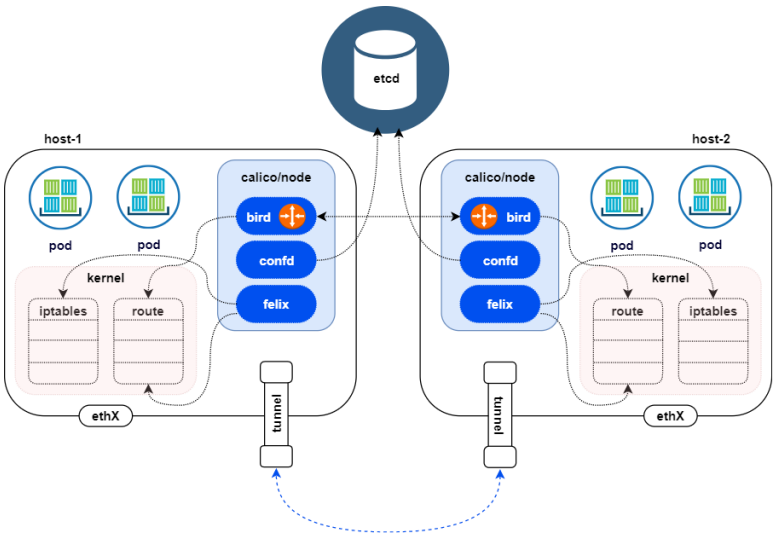
5.2.2 Konfigurace

Antrea nabízí několik možností konfigurace. Počínaje samotnou instalací lze zvolit využitý IP Address Management (IPAM), jež spravuje přiřazování i používání IP adres a také velikost maximální přenosové jednotky (MTU).

Poskytuje také možnost definice síťových zásad jak při příchozí, tak i odchozí komunikaci. Vedle toho implementace poskytuje šifrování komunikace díky bezpečnostnímu rozšíření IPsec.

5.3 Calico

Calico je komunitní projekt poskytující implementaci síťového subsystému v Kubernetes. Vytváří síť na síťové vrstvě, tudíž umožňuje poskytnout plnohodnotnou IP adresu každému podu. Pro uložení nastavení využívá etcd, představující úložiště s páry klíč-hodnota. [27]



Obrázek 17. Diagram Kubernetes clusteru s využitím Calico CNI rozšíření [27]

Calico se skládá ze tří komponent:

- **Node agent** – primární komponenta, která zajišťuje korektní komunikaci v Kubernetes clusteru
- **CNI** – rozšíření, poskytující přiřazování a používání IP adres uvnitř clusteru
- **Kube-controller** – spravuje zásady sítě a udržuje je aktuální vzhledem k uložené konfiguraci

Obrázek (Obr. 17) zobrazuje zjednodušený diagram Kubernetes clusteru s komponentami, poskytované Calico implementací. Každému vytvořenému podu je přiřazena IP adresa v rámci nodu. Hlavní částí komunikace jsou iptables a route, které jsou součástí jádra operačního systému a jsou nastavovány komponentou Calico node. Na základě těchto nastavených pravidel je řízena komunikace mezi všemi pody napříč clusterem.

Důležitou součástí je možné využití tzv. overlay network, což je síť postavená nad jinou sítí. Overlay network v Kubernetes může být využita k řízení komunikace mezi pody na rozdílných nodech, jelikož síť, nad kterou je postavena, nedisponuje IP adresami podů a nemá informace o tom, které pody běží na jakém nodu. Výhodou tak je redukce závislostí na síti v nižší vrstvě. [28]

Jedním z velkých benefitů je vlastní správa zásad sítě, jež rozšiřuje základní chování poskytované v Kubernetes. Poskytuje například možnost aplikovat tyto zásady na kterýkoliv objekt v rámci Kubernetes (pod, kontejner, …). [27]

5.3.1 Instalace

Předpokladem úspěšné instalace je povolení určitých portů ve firewall. Nutnost povolené některých z portů se liší v závislosti na konfiguraci. Prvním otevřeným portem na všech nodech je 179 pro TCP protokol. V případě, kdy je použito VXLAN jako overlay network, je nutné povolit port 4789 pro UDP protokol na všech nodech. [32]

Instalace Calico implementace probíhá pomocí YAML definice, aplikované konzolovým příkazem *kubectl apply -f calico.yml*. Důležitým parametrem při prvotní instalaci je zvolení rozsahu podsítě přes parametr *CALICO_IPV4POOL_CIDR* v definici komponenty Calico node.

5.3.2 Konfigurace

Implementace lze v mnoha směrech konfigurovat. Jedním z konfigurovatelných parametrů je například hodnota MTU, která je v základu nastavena nulu, což znamená automatické rozpoznání. V případě potřeby nastavení této hodnoty je nutno změnit parametr *veth_mtu* v konfiguraci ConfigMap na požadovanou hodnotu.

Pro nastavení overlay network se využívá dvou parametrů, *CALICO_IPV4POOL_IPIP* a *CALICO_IPV4POOL_VXLAN*, kde každý z uvedených přijímá hodnoty *Always* pro povolení či *Never* pro zakázání.

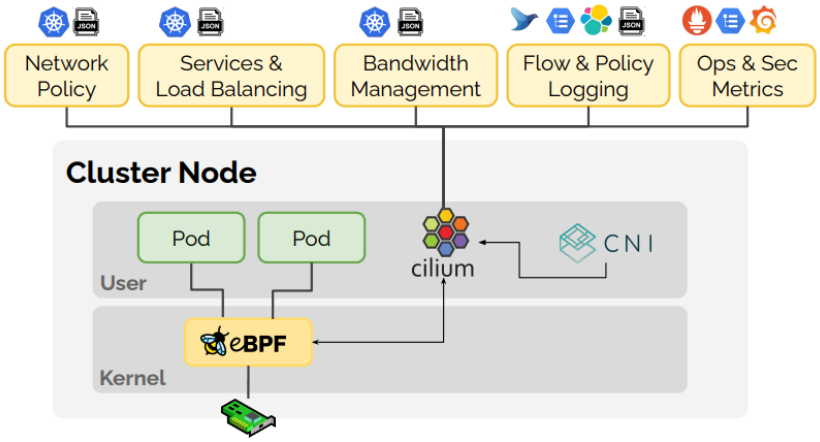
Nastavení zásad sítě může být rozmanité. Obrázek (Obr. 18) zobrazuje ukázku zásady, kdy na pod *service1* může přistoupit pouze pod *service2* přes port 80.

```
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
  name: allow-service
spec:
  podSelector:
    matchLabels:
      service: service1
  ingress:
    - from:
        - podSelector:
            matchLabels:
              service: service2
      ports:
        - protocol: TCP
          port: 80
```

Obrázek 18. Ukázka nastavení zásad sítě v Calico CNI

5.4 Cilium

Tato implementace zajišťuje komunikaci a její bezpečnost v Kubernetes clusteru mezi všemi pody. Mimo klasickou komunikaci na síťové a transportní vrstvě, poskytuje také bezpečné využívání moderních protokolů jako HTTP nebo Kafka v aplikační vrstvě. Cilium je založené na nové technologii eBPF, který podporuje dynamickou úpravu jádra vložením určité formy bajtového kódu. Tímto způsobem lze modifikovat různé části jádra, v tomto případě primárně část spravující síť. [30]



Obrázek 19. Diagram funkcionality poskytované Cilium implementací [30]

Obrázek (Obr. 19) ukazuje, že veškerá funkcionalita společná se sítí je řízena přes hlavní modul Cilium, který dále pomocí technologie eBPF upravuje síťovou část jádra operačního systému.

5.4.1 Instalace

Cilium lze nainstalovat různými způsoby. Jedním ze způsobů je klasické aplikování YAML definice s komponentami, které Cilium tvoří. Další možností je využití správce balíků v Kubernetes, Helm, jak je uvedeno na obrázku (Obr. 20). Při použití těchto příkazů je nutné, aby byl cluster inicializován s adresou podsítě *10.0.0.0/8*.

```
helm repo add cilium https://helm.cilium.io/
helm install cilium cilium/cilium --namespace=kube-system
```

Obrázek 20. Příkazy k instalaci Cilium implementace pomocí Helm

5.4.2 Konfigurace

Mimo konfiguraci zásad v síťové vrstvě, dle které je komunikace upravována, je k dispozici také možnost definovat zásady na aplikační vrstvě pro moderní protokoly jako je například HTTP. Na obrázku (Obr. 21) je ukázána definice takové zásady, kdy mohou ostatní služby přistupovat ke koncovému bodu, označeného jako *service* na portu 80, pouze přes metodu GET a cestu *path* s nastavenou hlavičkou *X-My-Header* na *true*.

```
apiVersion: "cilium.io/v2"
kind: CiliumNetworkPolicy
metadata:
  name: "layer7-rule"
spec:
  endpointSelector:
    matchLabels:
      app: service
  ingress:
    - toPorts:
        - ports:
            - port: '80'
              protocol: TCP
          rules:
            http:
              - method: GET
                path: "/path$"
                headers:
                  - 'X-My-Header: true'
```

Obrázek 21. Ukázka definice zásady sítě v Cilium na aplikační vrstvě

5.5 Flannel

Jednou z nejpoužívanějších implementací v Kubernetes je Flannel. Principem je vytvoření overlay network nad využívanou sítí, kdy každému podu je v této síti přiřazena IP adresa z dostupného rozsahu. Všechny pody v tomto případě komunikují přes přiřazené IP adresy v rámci overlay network. [29]

Flannel na rozdíl od mnoha dalších implementací se zaměřuje pouze na komunikaci v síti, nikoliv na zásady sítě. Vedle Flannelu se pro zajištění zásad sítě může využít například funkcionalita z Calica. Tato kombinace Flannel + Calico nese mezi ostatními implementacemi název Canal. [29]

5.5.1 Instalace

Flannel je možno nainstalovat, obdobně jako každou implementaci, pomocí YAML definice a příkazu *kubectl apply -f flannel.yml*. Před instalací je nutné upravit adresu podsítě v konfiguračním souboru sítě, též vytvářeného přes YAML definice.

5.5.2 Konfigurace

Flannel nabízí možnost výběru své backendové implementace, zajišťující fungování v rámci overlay network. Dostupnými možnostmi jsou například nejpoužívanější VXLAN, host-gw používané ke zlepšování výkonu nebo UDP pro testovací účely. [29]

5.6 Weave Net

Weave Net je implementací síťového subsystému, zajišťující nejen komunikaci mezi jednotlivými komponentami v rámci clusteru. Jako většina implementací také nabízí definování zásad sítě. Ve specifických případech může být využita pouze jako nástroj spravující zásady sítě vedle jiné implementace, která zajišťuje pouze komunikaci (například Flannel).

5.6.1 Instalace

Před samotnou instalací je nutno ve firewall povolit port 6783 pro TCP a porty 6783, 6784 pro UDP protokol. Pokud by některý z portů nebyl povolen, nemohly by některé komponenty implementace mezi sebou komunikovat. [33]

Pro zajištění základní funkcionality nabízené implementací Weave Net není potřeba měnit YAML definici, spouštěnou příkazem *kubectl apply -f weave-net.yml*.

5.6.2 Konfigurace

Mimo možnosti definice zásad sítě, poskytuje Weave Net také šifrovanou komunikace přes technologii IPsec.

II. PRAKTICKÁ ČÁST

6 PŘÍPRAVA WORKSPACE

V teoretické části jsou popsány základy Kubernetes a jeho součástí, potřebné k vytvoření clusteru a jeho základnímu nakonfigurování. Následně v této oblasti praktické části je vytvořen Kubernetes cluster vždy společně s testovanou implementací síťového subsystému, počínaje přípravou samotného workspace.

Pro možnost otestování většiny případů síťové komunikace je nutné, aby vytvořený cluster obsahoval alespoň dvě instance virtuálních strojů. První instance představuje master node, na němž běží hlavní části clusteru jako je API server, etcd úložiště apod. Zbylé instance tvoří skupinu worker nodes.

K testovacím účelům cluster sestává z jednoho worker node a jednoho master node, který zároveň zastává i funkci jako worker node.

6.1 Google Compute Engine

K provozu Kubernetes clusteru a otestování implementací síťového subsystému v prostředí využívané i pro produkční účely je zvolen GCE (Google Compute Engine). Tato služba, provozovaná společností Google, poskytuje uživateli možnost vytvořit a provozovat vlastní virtuální stroje v infrastruktuře Googlu.

Hlavním důvodem výběru této služby je jednoduchá a rychlá možnost vytvořit si vlastní virtuální stroj, případně si z již nastaveného vytvořit obraz disku a lehce vytvořit další v totožném stavu.

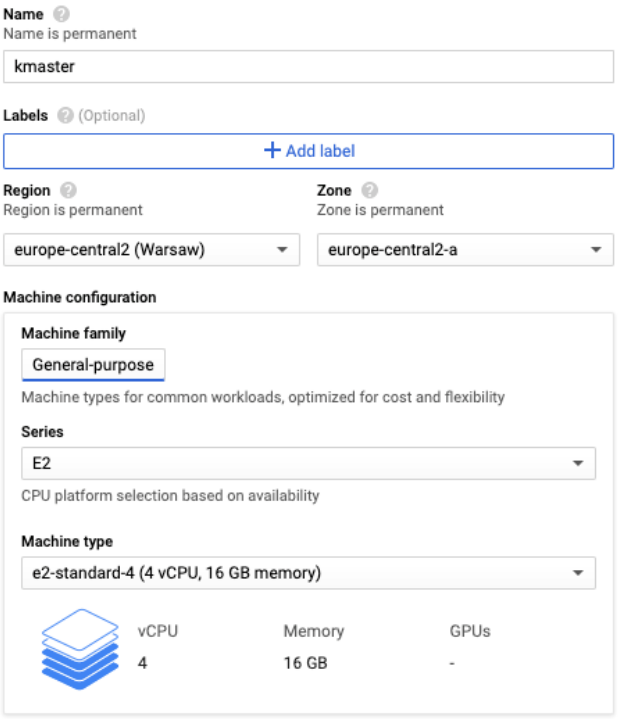
Další výhodou využití GCE je fakturování pouze spuštěných virtuálních strojů po hodinách, tedy ideální možnost využití právě pro různé testovací účely.

6.1.1 Vytvoření virtuálního stroje

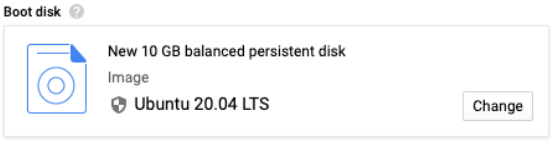
Virtuální stroj bude mít následující konfiguraci:

- procesor Intel(R) Xeon(R) CPU @ 2.20GHz s poskytnutými 4 virtuálními jádry
- operační paměť 16 GB
- velikost pevného disku 10 GB
- operační systém Ubuntu 20.04 LTS
- fyzické umístění ve Varšavě, ležící ve střední Evropě

Pro vytvoření jedné instance virtuálního stroje je třeba vyplnit jednoduchý formulář, jako je vyobrazeno na obrázcích (Obr. 22 a Obr. 23), ve kterém je možnost si nový stroj nakonfigurovat dle vlastních požadavků. Jakmile je stroj vytvořen, lze se do něj ihned připojit přes SSH (zabezpečený komunikační protokol) ve webovém prohlížeči či jinými způsoby, které GCE nabízí.



Obrázek 22. Konfigurace virtuálního stroje na GCE



Obrázek 23. Výběr operačního systému virtuálního stroje na GCE

6.1.2 Konfigurace virtuálního stroje

Z počátku nově vytvořený virtuální stroj obsahuje pouze zvolený operační systém v základním stavu bez nástrojů a závislostí potřebných k vytvoření Kubernetes clusteru. Předpokladem funkčního clusteru je nainstalovaný vybraný container runtime, v tomto případě Docker CE. Alternativou pro Docker container runtime je například containerd. [13]

```
apt-get install apt-transport-https ca-certificates curl software-properties-common
-y
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -
add-apt-repository "deb [arch=amd64] https://download.docker.com/linux/ubuntu
$(lsb_release -cs) stable"
apt-get update -y
apt-get install docker-ce -y
```

Obrázek 24. Instalace Docker container runtime

Další nedílnou součástí přípravy virtuálního stroje před vytvořením samotného clusteru je instalace agenta Kubelet a nástrojů Kubeadm a Kubectl, případně i správce balíků pro Kubernetes, HELM.

```
cat <<EOF | sudo tee /etc/apt/sources.list.d/kubernetes.list
deb https://apt.kubernetes.io/ kubernetes-xenial main
EOF
ls -ltr /etc/apt/sources.list.d/kubernetes.list
apt-get update -y
apt-get install -y kubelet kubeadm kubectl
```

Obrázek 25. Instalace Kubelet, Kubeadm a Kubectl

Důležitou součástí konfigurace je nastavení správného směrování internetové komunikace, kdy v opačném případě dochází často k vynechání iptables. Pro zajištění korektního chodu agenta Kubelet je výrazně doporučováno vypnutí swapování. [14]

```
cat >>/etc/sysctl.d/kubernetes.conf<<EOF
net.bridge.bridge-nf-call-ip6tables=1
net.bridge.bridge-nf-call-iptables=1
net.ipv4.ip_forward=1
EOF
sysctl --system >/dev/null 2>&1

sed -i '/swap/d' /etc/fstab
swapoff -a
```

Obrázek 26. Nastavení síťové komunikace a vypnutí swapování

Na obrázcích (Obr. 24, Obr. 25 a Obr. 26) je zobrazena posloupnost příkazů, potřebná k dosažení chtěné konfigurace virtuálního stroje, včetně nástrojů pro správu clusteru.

6.1.3 Vytvoření clusteru

K vytvoření Kubernetes clusteru slouží příkaz dostupný díky nástroji Kubeadm. V základu tohoto příkazu se uvádí IP adresa serveru, na kterém běží API server a CIDR sítě, jež specifikuje dostupný rozsah IP adres, přidělovaných v rámci sítě mezi jednotlivými pody.

```
sudo kubeadm init --apiserver-advertise-address=10.128.0.9 --pod-network-
cidr=10.244.0.0/16
```

Obrázek 27. Inicializace Kubernetes clusteru

Použití tohoto příkazu je uvedeno na obrázku (Obr. 27). Po dokončení inicializace clusteru na master node je vygenerován příkaz, kdy po jeho spuštění na některém z worker nodes je provedeno připojení do nově vzniklého clusteru.

7 METODIKA TESTOVÁNÍ

V této kapitole praktické části je sestavena metodika pro testování určitých oblastí, nejen síťové komunikace v rámci Kubernetes clusteru. Každá z implementací síťového subsystému implementuje CNI rozdílným způsobem, tudíž se mění i míra náročnosti právě na tyto testované oblasti.

Testování je provedeno nad několika scénáři komunikace mezi jednotlivými objekty clusteru. Prvním scénářem je přímá komunikace mezi pody, jimž je po vytvoření přiřazena IP adresa právě některou z implementací síťového subsystému. Druhým scénářem je komunikace podu s některou ze service, která definuje určité vstupní body do podu pro který je vytvořena. Oba tyto scénáře jsou aplikovány jak na stejném nodu, tak na dvou rozdílných pro TCP i UDP protokol.

Mimo klasické parametry síťové komunikace jako propustnost a latence je v rámci testů měřeno i vytížení procesoru a operační paměti samotných serverů, které v testu působí.

7.1 Propustnost

Prvním testovaným parametrem síťové komunikace je propustnost, jež značí kolik dat může být přeneseno ze zdrojového bodu do bodu koncového během určité doby. Propustnost tedy udává, kolik paketů úspěšně dorazí do koncového bodu.

Jedním z faktorů, který ovlivňuje hodnotu propustnosti, je latence v rámci sítě. Čím větší je latence, tím pomalejší je komunikace v síti. Větší latenci může způsobovat například souběžný přístup více uživatelů do sítě, v horším případě i souběžné stahování dat. [15]

Pro měření propustnosti je využito balíku Kubernetes Bench Suite, dostupného na adrese [22], sestaveného přímo pro testování Kubernetes clusteru s využitím nástroje „iperf“.

7.1.1 Iperf

Iperf je nástroj, umožňující měření přenosu v rámci sítě a tím i získání informací o provedeném přenosu. Funguje ve dvou režimech, serverovém nebo klientském. Na jednom stroji je spuštěn v serverovém režimu s určitými parametry a na druhém právě v klientském, který komunikuje se serverem podle zadané IP adresy. [16]

```
root@kmaster ~# iperf -s
------------------------------------------------------------
Server listening on TCP port 5001
TCP window size:  128 KByte (default)
------------------------------------------------------------
[  4] local 172.16.11.100 port 5001 connected with 172.16.11.101 port 45202
[ ID] Interval       Transfer     Bandwidth
[  4]  0.0-10.0 sec  3.06 GBytes  2.63 Gbits/sec
[  4] local 172.16.11.100 port 5001 connected with 172.16.11.101 port 45268
[  4]  0.0-10.0 sec  3.08 GBytes  2.64 Gbits/sec
[  4] local 172.16.11.100 port 5001 connected with 172.16.11.101 port 45278
[  4]  0.0-10.0 sec  3.12 GBytes  2.68 Gbits/sec
[  4] local 172.16.11.100 port 5001 connected with 172.16.11.101 port 45516
[  4]  0.0-10.0 sec  3.18 GBytes  2.73 Gbits/sec
```

Obrázek 28. Spuštěný nástroj „iperf“ v serverovém režimu

Na obrázku (Obr. 28) je vidět výpis ze spuštěného nástroje v serverovém režimu, jež naslouchá na portu 5001 a pro komunikaci využívá TCP protokol. V tomto režimu může nástroj přijmout další parametry, kterými lze prováděné měření ovlivnit. Těmito parametry lze komunikaci přepnout na využití UDP protokolu, specifikovat port na kterém server naslouchá či nastavit naslouchání pouze na konkrétní adresu nebo síťové rozhraní. [16]

```
root@kworker1 ~# iperf -c 172.16.11.100
------------------------------------------------------------
Client connecting to 172.16.11.100, TCP port 5001
TCP window size:  527 KByte (default)
------------------------------------------------------------
[  3] local 172.16.11.101 port 45516 connected with 172.16.11.100 port 5001
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0-10.0 sec  3.18 GBytes  2.73 Gbits/sec
```

Obrázek 29. Spuštěný nástroj „iperf“ v klientském režimu

Naopak obrázek (Obr. 29) ukazuje základní využití nástroje v klientském režimu, kdy je na vstupu předána pouze adresa serveru a po spuštění započne komunikace se serverem pomocí TCP protokolu. Z výstupu nástroje lze zjistit, že během 10 sekund běhu testu bylo přeneseno 3.18 gigabajtů s šířkou pásma 2.73 gigabitů za sekundu.

Obdobně jako u serverového režimu může nástroj přijmout další parametry, opět ovlivňující výsledky měření. Lze například nastavit maximální rychlost přenosu, velikost přenášeného segmentu, využití UDP protokolu a dalších několik parametrů, uvedených v samotné dokumentaci nástroje. [17]

7.1.2 Kubernetes Bench Suite

Hlavní část balíku tvoří čistý bash script, který podle předaných parametrů spustí testovací sadu nad clusterem. Hlavními parametry jsou názvy nodů, kdy se vždy vybere jeden hlavní, představující server a druhý, sloužící jako klient. Na hlavním node je podle definice, se kterou Kubernetes pracuje, spuštěn nástroj iperf v serverovém režimu jako pod a zároveň k němu i service, zpřístupňující TCP a UDP port pro tento pod. Naopak na druhém nodu je opět podle definice vždy spuštěn nástroj iperf v klientském režimu jako pod s parametry podle právě prováděného testu.

Pro měření propustnosti jsou k dispozici čtyři testovací scénáře jako Pod-to-Pod nebo Pod-to-Service komunikace, v obou případech s využitím TCP i UDP protokolu.

Samotný bash script nabízí několik možností exportu naměřených výsledků, například v textovém formátu, JSON nebo dokonce i jako jednoduché grafy.

```
./knb -t 60 -d 60 -o json -f ~/result/same-node/result.json --client-node kworker --server-node kmaster
```

Obrázek 30. Spuštění testovací sady Kubernetes Bench Suite

Na obrázku (Obr. 30) lze vidět spuštění kompletní sady testů, které balík nabízí s výstupem ve formátu JSON, kdy trvání každého testu je nastaveno na 1 minutu.

7.2 Využití procesoru

Jedním z ukazatelů vytížení stroje při síťové komunikaci je využití procesoru. V případě omezených možností, co se týče dostupných systémových prostředků jako je procesor či operační paměť, může znalost využití těchto prostředků napomoct při výběru správné implementace síťového subsystému.

K měření je využito opět služeb v balíku Kubernetes Bench Suite, jenž má testovací sady připravené přímo pro Kubernetes cluster.

7.2.1 Sar

Nástroj „sar“ umožňuje monitorovat výkon různých subsystémů (procesor, operační paměť, …) v operačním systému Linux v reálném čase. Data může sbírat, ukládat a zjišťovat tak slabá místa v případě zjištění problémů s výkonem. [19]

```
root@kmaster ~# sar -P 0,1,2,3 1 2
Linux 5.4.0-56-generic (kmaster)        04/28/2021      _x86_64_        (4 CPU)

06:04:43 AM     CPU     %user     %nice   %system   %iowait    %steal     %idle
06:04:44 AM       0      5.15      0.00      6.19      0.00      0.00     88.66
06:04:44 AM       1      4.30      0.00      5.38      0.00      0.00     90.32
06:04:44 AM       2      4.21      0.00      6.32      0.00      0.00     89.47
06:04:44 AM       3      4.12      0.00      6.19      0.00      0.00     89.69

06:04:44 AM     CPU     %user     %nice   %system   %iowait    %steal     %idle
06:04:45 AM       0      3.03      0.00      6.06      0.00      0.00     90.91
06:04:45 AM       1      6.25      0.00      1.04      0.00      0.00     92.71
06:04:45 AM       2      2.06      0.00      4.12      0.00      0.00     93.81
06:04:45 AM       3      2.08      0.00      4.17      0.00      0.00     93.75

Average:        CPU     %user     %nice   %system   %iowait    %steal     %idle
Average:          0      4.08      0.00      6.12      0.00      0.00     89.80
Average:          1      5.29      0.00      3.17      0.00      0.00     91.53
Average:          2      3.12      0.00      5.21      0.00      0.00     91.67
Average:          3      3.11      0.00      5.18      0.00      0.00     91.71
```

Obrázek 31. Informace o využití procesoru pomocí příkazu „sar“

Z obrázku (Obr. 31) lze vyčíst několik monitorovaných informací o procesoru ze dne 28.4.2021. V první tabulce jsou uvedeny informace o využití jader procesoru, kdy nejdůležitější je poslední sloupec „%idle“, na kterém lze sledovat vytížení jednotlivého jádra. V tomto případě jsou všechny jádra z větší části nečinné. Poslední tabulka vyjadřuje průměrné hodnoty všech měření.

7.2.2 Kubernetes Bench Suite

Princip monitorování využití procesoru je takový, že je jak na server, tak klienta nasazen pod dle definice, obsahující kontejner s jednoduchým bash scriptem. Tento script běží v nekonečném cyklu a neustále generuje informace o využití do souboru. Jakmile je test pro všechny scénáře dokončen, jsou informace ze souboru vyextrahovány, zpracovány na výstup a vytvořený pod odstraněn.

Monitorování probíhá zvlášť pro komunikaci s využitím TCP a UDP protokolu.

Spuštění této monitorovací služby je provedeno již v rámci spuštění kompletní testovací sady, kterou balík nabízí, jako je uvedeno na obrázku (Obr. 30).

7.3 Využití operační paměti

Pro lepší představu o náročnosti použité implementace síťového subsystému je v rámci testů měřeno i využití operační paměti. Každá z implementací funguje na rozdílném principu, tudíž některé mohou operační paměť využívat méně a některé zase více.

Obdobně jako u propustnosti či využití procesoru je využito balíku Kubernetes Bench Suite, jenž nabízí monitorovací službu připravenou k nasazení do clusteru.

7.3.1 Free

Free je nástroj, umožňující zobrazení informací o volné či využité operační paměti v systému. Vedle těchto dvou základních ukazatelů poskytuje informace o využité paměti v rámci dočasného file systému na operačních systémech obdobných Unixu nebo paměti, využívané bufferem kernelu operačního systému. [18]

```
root@kmaster ~# free --mega
               total        used        free      shared  buff/cache   available
Mem:           16790         275       15840           0         675       16220
Swap:              0           0           0
```

Obrázek 32. Informace o operační paměti pomocí příkazu „free“

Obrázek (Obr. 32) ukazuje využití operační paměti při nečinnosti master nodu, kde využito je pouze necelých 1000 MB z dostupných 16 GB.

7.3.2 Kubernetes Bench Suite

Princip monitorování využití operační paměti je totožný jako u monitorování využití procesoru. Po nasazení podu na oba nody je spuštěn nekonečný proces se zápisem do souboru a při ukončení testu jsou pody odstraněny a výsledek vypsán na výstup.

Monitorování probíhá zvlášť pro komunikaci s využitím TCP a UDP protokolu.

Spuštění této monitorovací služby je provedeno již v rámci spuštění kompletní testovací sady, kterou balík nabízí, jako je uvedeno na obrázku (Obr. 30).

7.4 Latence

Posledním testovaným parametrem při síťové komunikaci je latence. Latence v síti značí dobu potřebnou k přenesení packetu ze zdroje do cílového bodu. Nejčastěji je měřena v milisekundách, ale může být využito i jiných jednotek. Platí, čím menší je latence, tím rychlejší může být komunikace v rámci sítě. Ve většině případů s narůstající velikostí sítě roste i velikost latence. [20]

Pro měření latence je využito balíku Kubernetes Qperf, dostupného na adrese [23], sestaveného přímo pro testování Kubernetes clusteru s využitím nástroje „qperf“.

7.4.1 Qperf

Nástroj „qperf“ poskytuje možnost měření šířky pásma a latence mezi dvěma body v rámci sítě. Obdobně jako nástroj „iperf“ poskytuje dva režimy, serverový a klientský. Na stroji, představující jeden z bodů, je nástroj spuštěn v serverovém režimu, čekající na příchozí požadavky od klienta. Na druhém stroji, opět představující jeden z bodů, je naopak spuštěn v klientském režimu, odkazující se na adresu serveru s uvedením požadovaného testu.

```
root@kworker1 ~# qperf kmaster tcp_lat
tcp_lat:
    latency  =  125 us
```

Obrázek 33. Spuštěný nástroj „qperf“ v klientském režimu

Obrázek (Obr. 33) ukazuje použití nástroje jako klienta, odesílající požadavek na server pro otestování latence komunikace mezi nimi. Výstupem je latence měřená v mikrosekundách, v tomto konkrétním případě 125 mikrosekund.

7.4.2 Kubernetes Qperf

Balík poskytuje Docker image, ve kterém je obsažen právě nástroj „qperf“, společně s definicemi, díky které jej využívají. Po spuštění těchto definic jsou v clusteru vytvořeny dva pody, představující server a klienta. Pro pod, hostující server, je též vytvořena service, se kterou může klient také komunikovat.

Monitorování latence probíhá pro čtyři scénáře. Komunikace mezi pody na stejném nebo dvou rozdílných nodech a komunikace mezi podem a service, opět na stejném či dvou rozdílných nodech.

Jelikož balík poskytuje pouze dvě pevné definice, je třeba při testování měnit v definici klienta jak adresu serveru, tak název nodu, na kterém bude klient spuštěn. Změna těchto parametrů je ukázána na obrázku (Obr. 34), kde **nodeSelector** značí název nodu a **value** naopak adresu serveru. Komentovaná část kódu obsahuje přímou adresu podu se serverem a hodnota nad ní naopak textovou adresu service, tvořící přístupový bod do onoho podu.

```
containers:
- name: qperf-client
  image: danielcoman/docker-qperf:latest
  env:
  - name: SERVICE_TYPE
    value: "client"
  - name: SERVER_ADDR
    value: "server.qperf.svc.cluster.local"
    #value: "10.0.1.72"
  - name: SERVER_PORT
    value: "443"
  - name: QPERF_INTERVAL
    value: "10-60"
nodeSelector:
  kubernetes.io/hostname: kworker
```

Obrázek 34. Definice pro kontejner v podu s nástrojem „qperf“

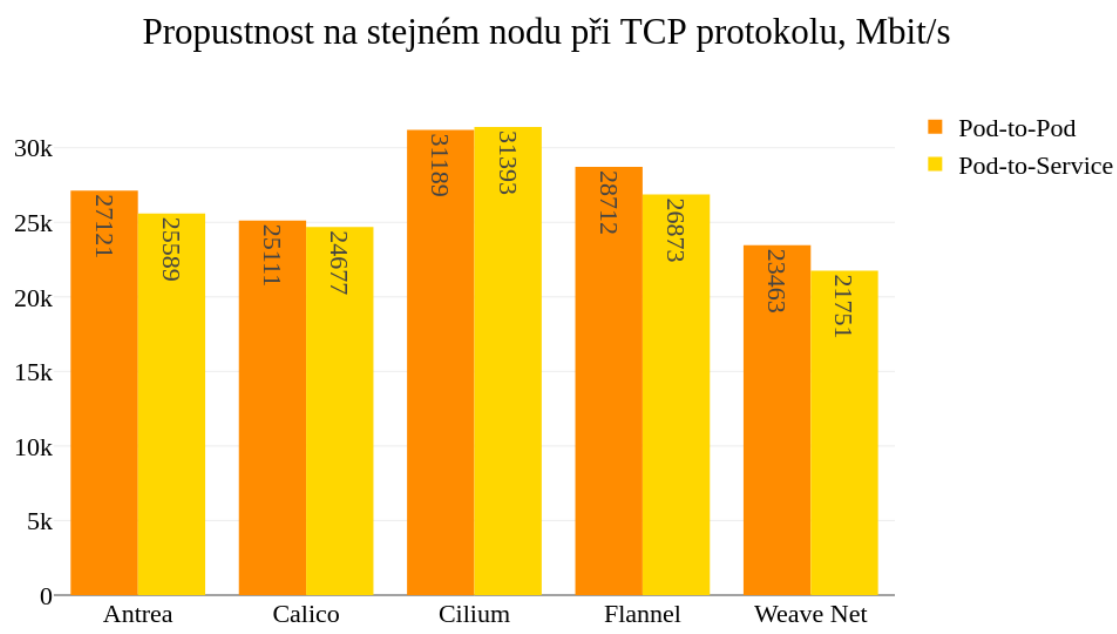
8 TESTOVÁNÍ

V této kapitole je aplikována metodika testování na připraveném prostředí s Kubernetes clusterem, včetně implementace síťového subsystému. Jednotlivé testované oblasti jsou rozděleny do podkapitol, obsahující naměřené hodnoty v podobě grafů a samotné zhodnocení. Poslední podkapitolu tvoří celkové shrnutí provedených měření ve všech oblastech.

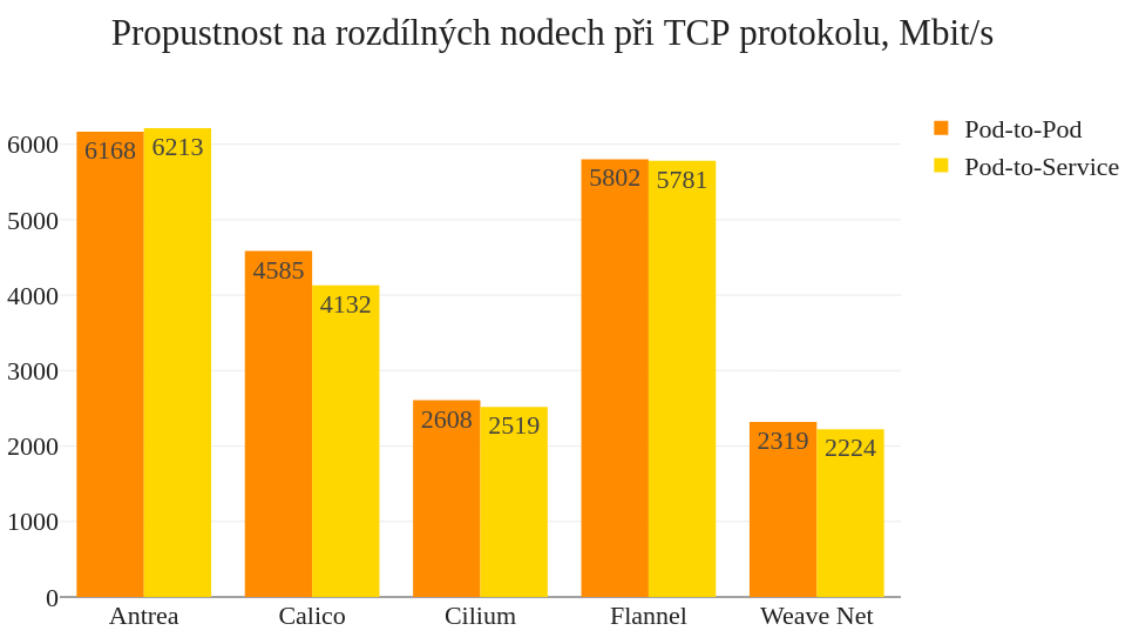
8.1 Propustnost

Všechny hodnoty, získané při měření propustnosti, jsou uváděny v megabitech za sekundu (Mbit/s). Grafy vzniklé z naměřených hodnot představují porovnání velikostí propustnosti mezi jednotlivými implementacemi, kdy první (oranžový) sloupec ukazuje velikost při Pod-to-Pod komunikaci a druhý (zlatý) sloupec naopak při komunikaci Pod-to-Service.

8.1.1 TCP protokol



Obrázek 35. Graf propustnosti při TCP protokolu na stejném nodu



Obrázek 36. Graf propustnosti při TCP protokolu na rozdílných nodech

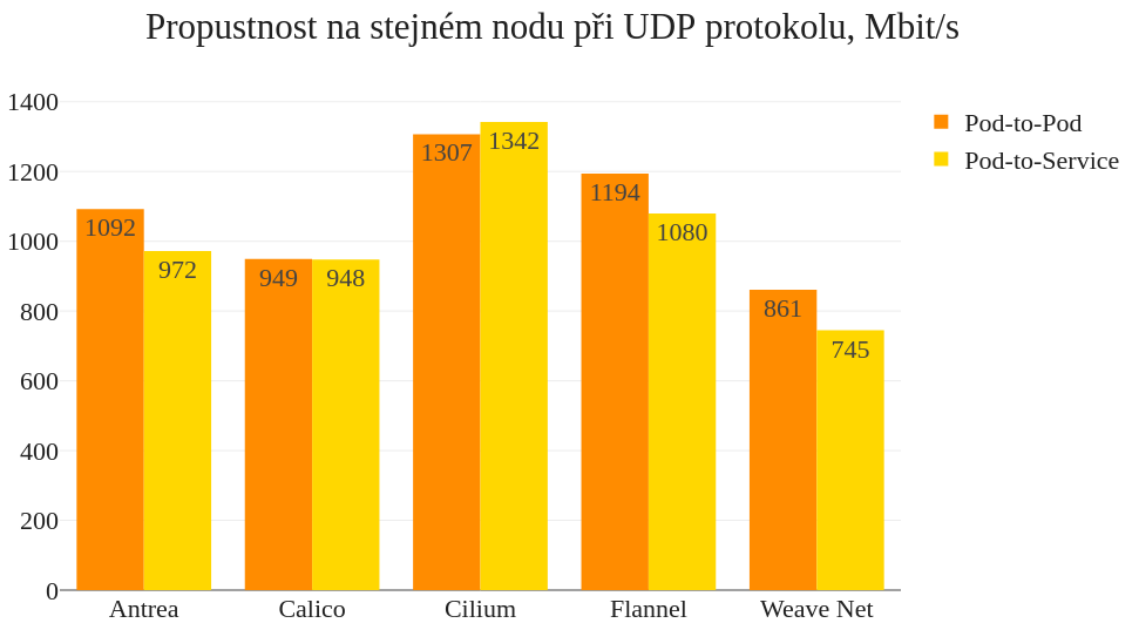
Obrázky (Obr. 35 a Obr. 36) zobrazují graf naměřených hodnot propustnosti při síťové komunikaci přes TCP protokol. Na první pohled je zřejmý markantní rozdíl mezi komunikací v rámci jednoho stejného nodu (Obr. 35) a dvou rozdílných (Obr. 36). Toto zjištění bylo také očekáváno, jelikož v případě dvou rozdílných nodů je velikost propustnosti závislá na jejich síťovém propojení v infrastruktuře.

Dále je zřejmé, že mezi přímou komunikací jednoho podu s podem druhým a service není žádný velký rozdíl, co se týče propustnosti pro každou z implementací. V reálném světě je převážně využívaná komunikace podu se service a provedené měření tak ukazuje, že tento druh komunikace neztrácí na rychlosti oproti té přímé mezi jednotlivými pody.

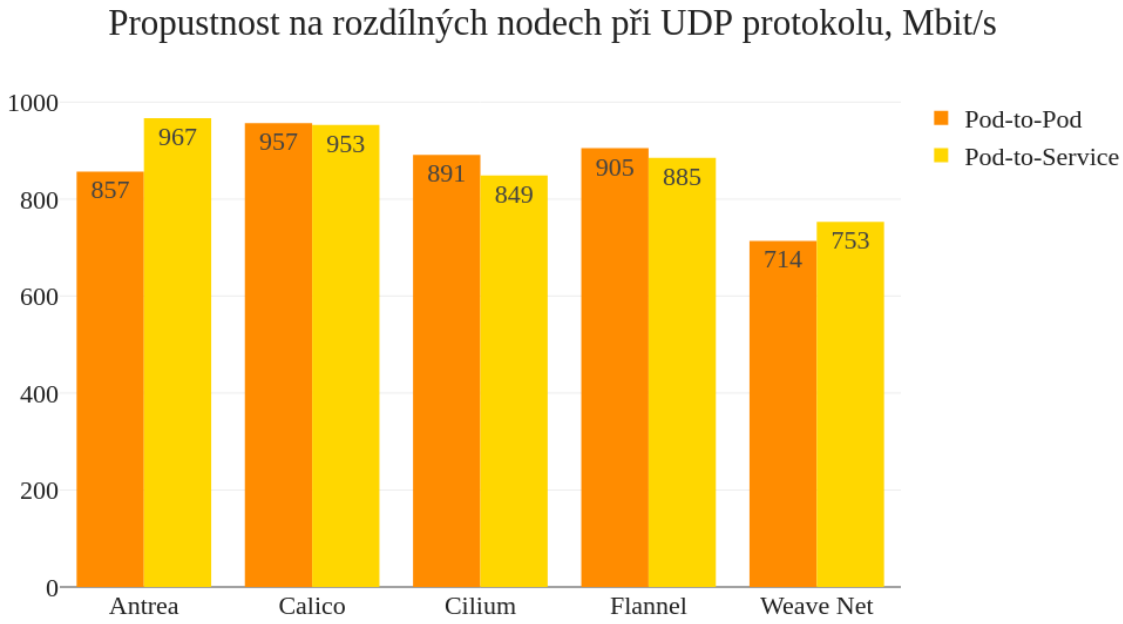
Z grafu je též znát rozdíl mezi velikostí propustnosti implementace Cilium na stejném nodu a tou na dvou rozdílných, kdy v prvním případě svou velikostí překonává ostatní implementace a v případě druhém naopak disponuje velmi malou propustností oproti ostatním.

Dvojice implementací, Antrea a Flannel, mají jednoznačně v těchto měřeních nejlepší vyvážené výsledky, co se týče propustnosti přes TCP protokol jak na stejném nodu, tak při nodech rozdílných.

8.1.2 UDP protokol



Obrázek 37. Graf propustnosti při UDP protokolu na stejném nodu



Obrázek 38. Graf propustnosti při UDP protokolu na rozdílných nodech

Při jakékoliv komunikaci přes UDP protokol na stejném nodu (Obr. 37) nebo rozdílných (Obr. 38) je znát obrovský rozdíl oproti komunikaci přes TCP protokol. V případě stejného nodu jsou hodnoty malé, což se dalo předpokládat z povahy UDP protokolu. Důvodem malé

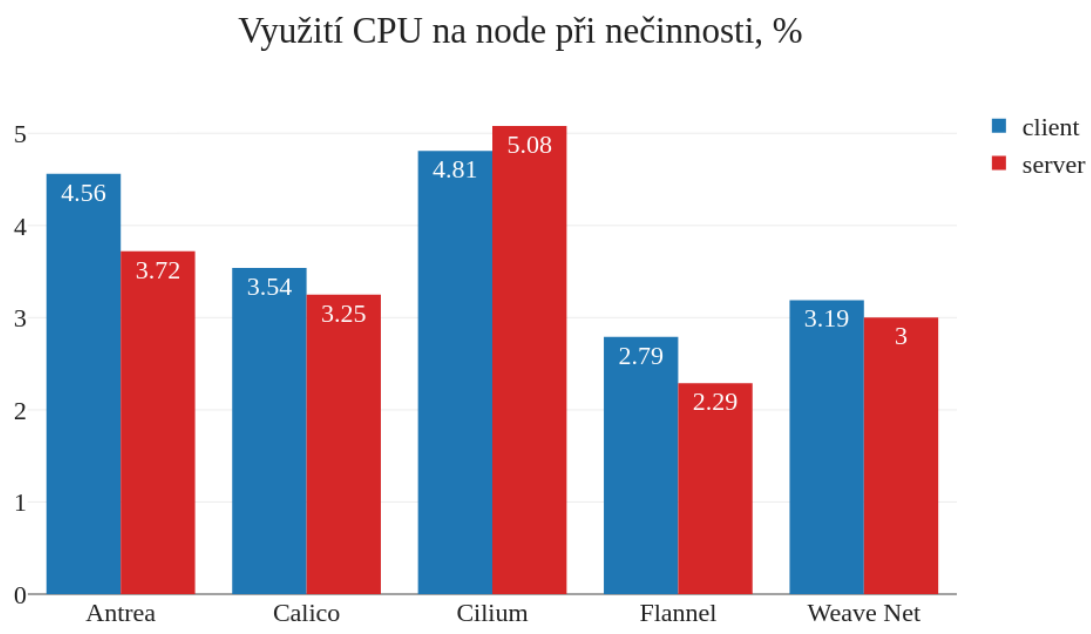
propustnosti může být ta skutečnost, že protokol negarantuje jak už samotné doručení packetů do cíle, tak doručení v pořadí. Proto, aby doručil požadované množství dat v rámci testu, potřebuje více času, čímž je zapříčiněna právě tato malá naměřená propustnost.

Stejně jako u TCP protokolu lze pozorovat nejlíp vyvážené výsledky pro implementaci Flannel.

8.2 Využití procesoru

Získané hodnoty při měření využití procesoru jsou uváděny v procentech využití (%). Grafy vzniklé z těchto hodnot porovnávají využití procesoru mezi jednotlivými implementacemi na použitých nodech jak při samotné komunikaci, tak i při nečinnosti. První (modrý) a druhý (červený) sloupec značí využití procesoru dvou nodů (klienta a serveru) u Pod-to-Pod komunikace, kdežto druhý (fialový) a čtvrtý (hnědý) u Pod-to-Service komunikace.

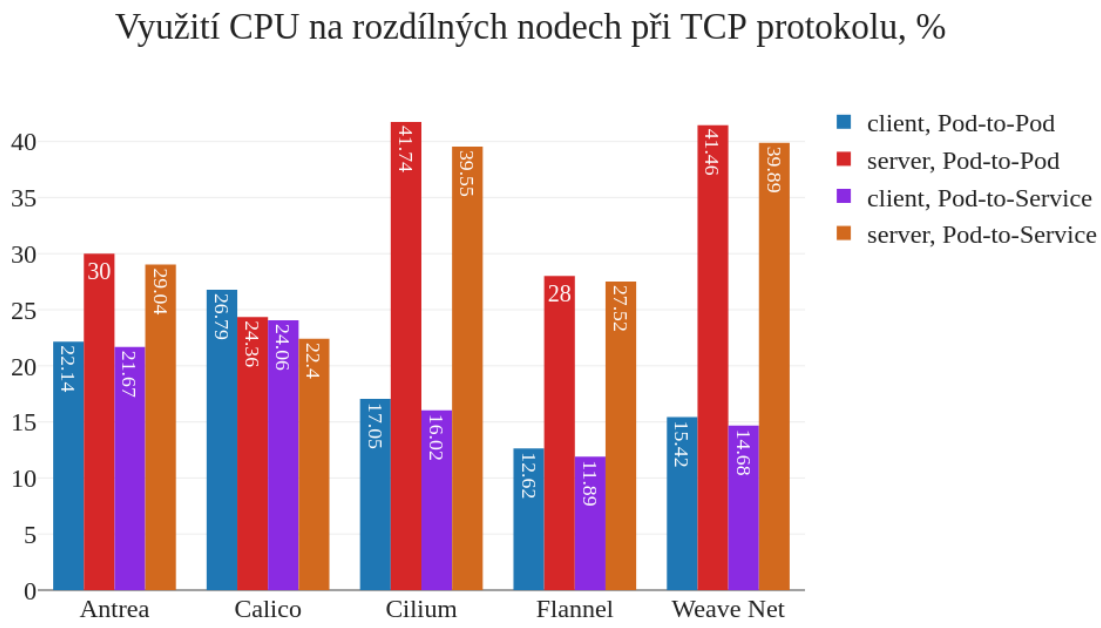
8.2.1 Nečinnost



Obrázek 39. Graf využití CPU při nečinnosti na rozdílných nodech

V případě nečinnosti obou nodů (Obr. 39) jsou hodnoty využití procesoru na velmi podobné rovině. Lze pozorovat pouze malé odchylky, které mohou být způsobeny zapojením master nodu do testovacích scénářů. Jelikož oproti worker nodu obsahuje spuštěné systémové nástroje.

8.2.2 TCP protokol



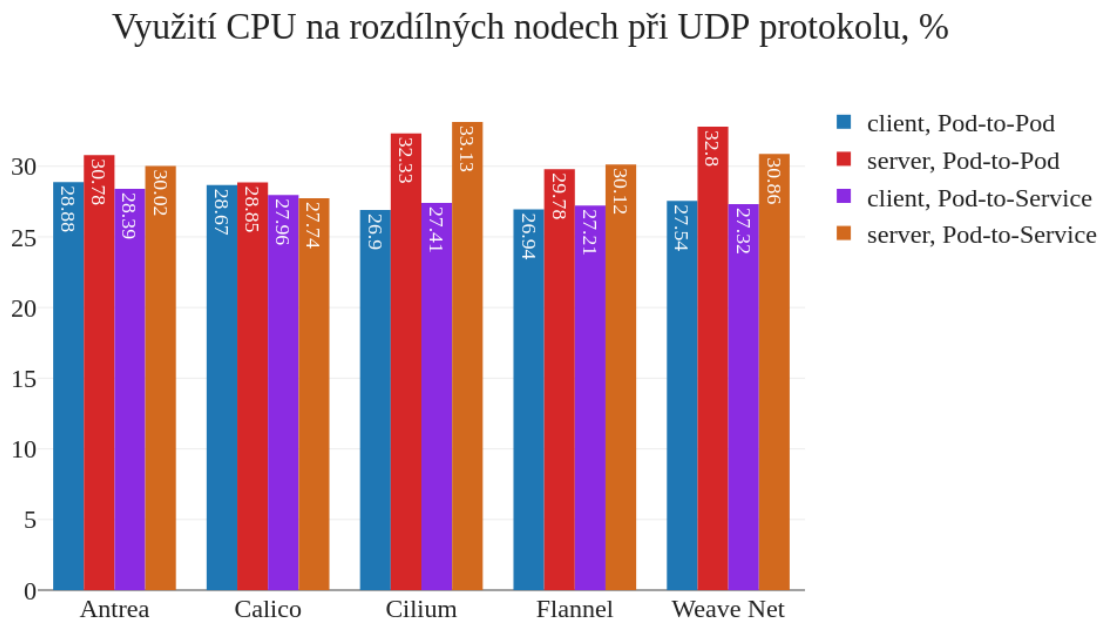
Obrázek 40. Graf využití CPU na rozdílných nodech při TCP protokolu

Jako tomu je u propustnosti, odchylka mezi Pod-to-Pod a Pod-to-Service komunikací mezi rozdílnými nody (Obr. 40) je zanedbatelná. Zajímavým zjištěním je velikost využití v případě implementace Weave Net v porovnání s její naměřenou velikostí propustnosti. I přes to, že mezi vybranými implementacemi nabývá jedny z největších využití procesoru v serverové části, dosahuje nejmenších hodnot, co se týče propustnosti.

Flannel a Antrea, prokazující jedny z nejlepších a nejvyváženějších výsledků v testech propustnosti, dosahují v průměru i nejmenšího využití procesoru.

Je potřeba také vyzdvihnout implementaci Calico, jež dokáže rovnoměrně rozložit zátěž na procesor mezi oba nody.

8.2.3 UDP protokol



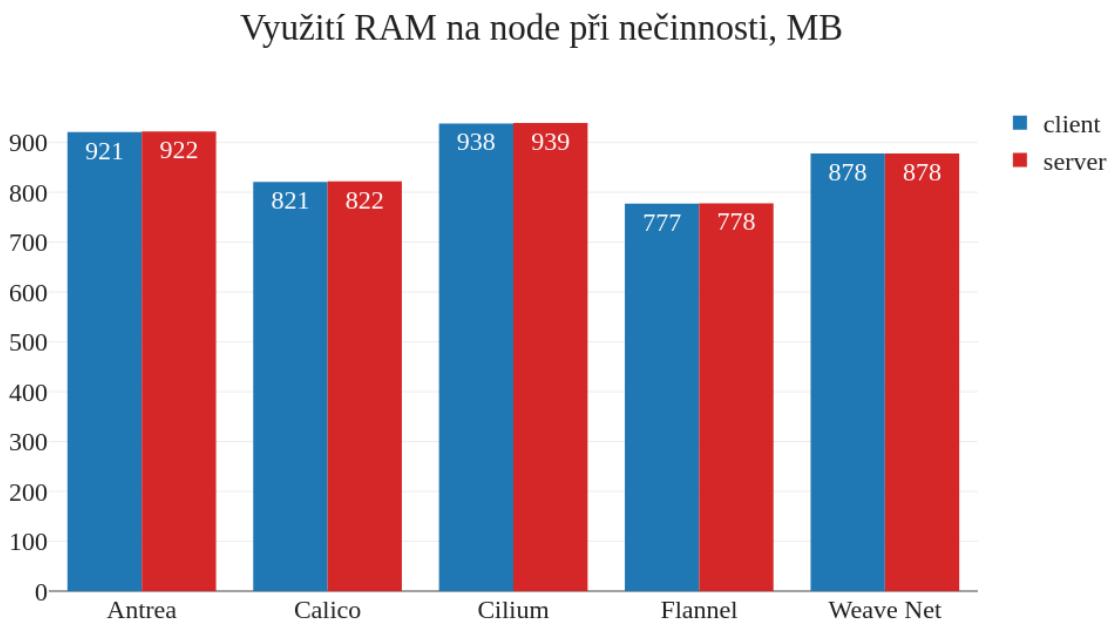
Obrázek 41. Graf využití CPU na rozdílných nodech při UDP protokolu

Oproti využití procesoru při komunikaci přes TCP protokol se hodnoty klienta v případě UDP protokolu téměř zdvojnásobily, jak lze vidět na obrázku (Obr. 41). Na procesor jsou kladeny větší nároky při zpracovávání přijatých dat, které nemusí být kompletní a v pořadí. Tato skutečnost tedy může být jedním z důvodů právě zdvojnásobení využití procesoru na straně klienta.

8.3 Využití operační paměti

Hodnoty přenesené do grafů jsou uváděny v megabajtech (MB). Uvedené grafy porovnávají využití operační paměti mezi jednotlivými implementacemi při komunikaci i nečinnosti. První (modrý) a druhý (červený) sloupec značí využití operační paměti dvou nodů (klienta a serveru) u Pod-to-Pod komunikace, zatímco druhý (fialový) a čtvrtý (hnědý) u Pod-to-Service komunikace.

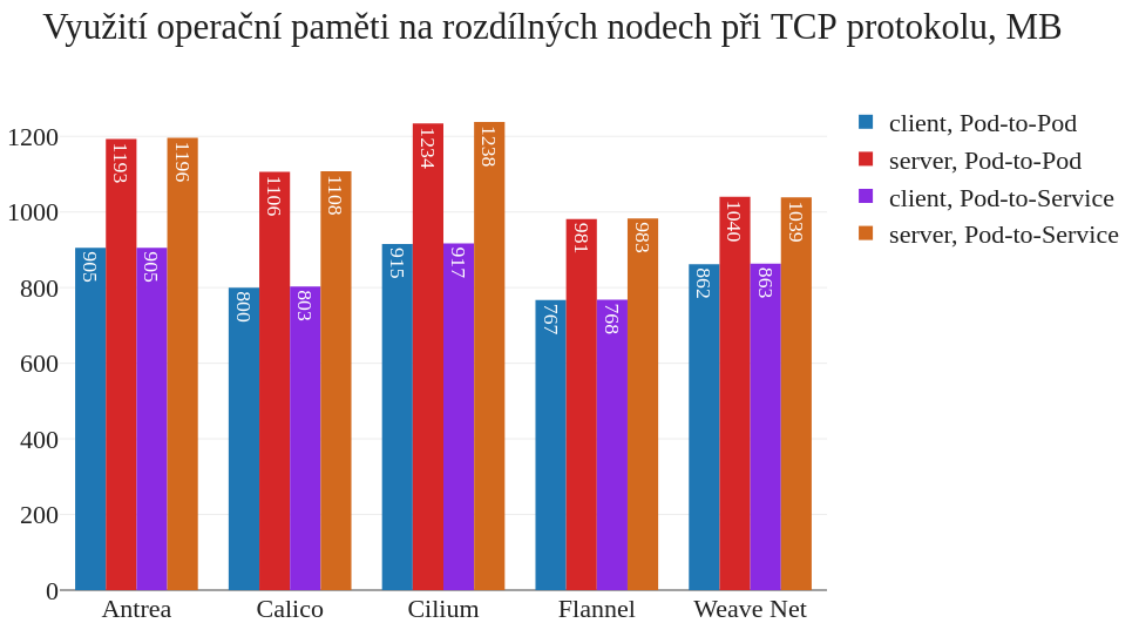
8.3.1 Nečinnost



Obrázek 42. Graf využití RAM při nečinnosti na rozdílných nodech

U využití operační paměti v případě nečinnosti (Obr. 42) není pozorována větší odchylka mezi jednotlivými implementacemi. Vhodné je opět vyzdvihnout Flannel, který stejně jako u využití procesoru, prokazuje jedny z nejmenších hodnot, co se týče využívání prostředků.

8.3.2 TCP protokol

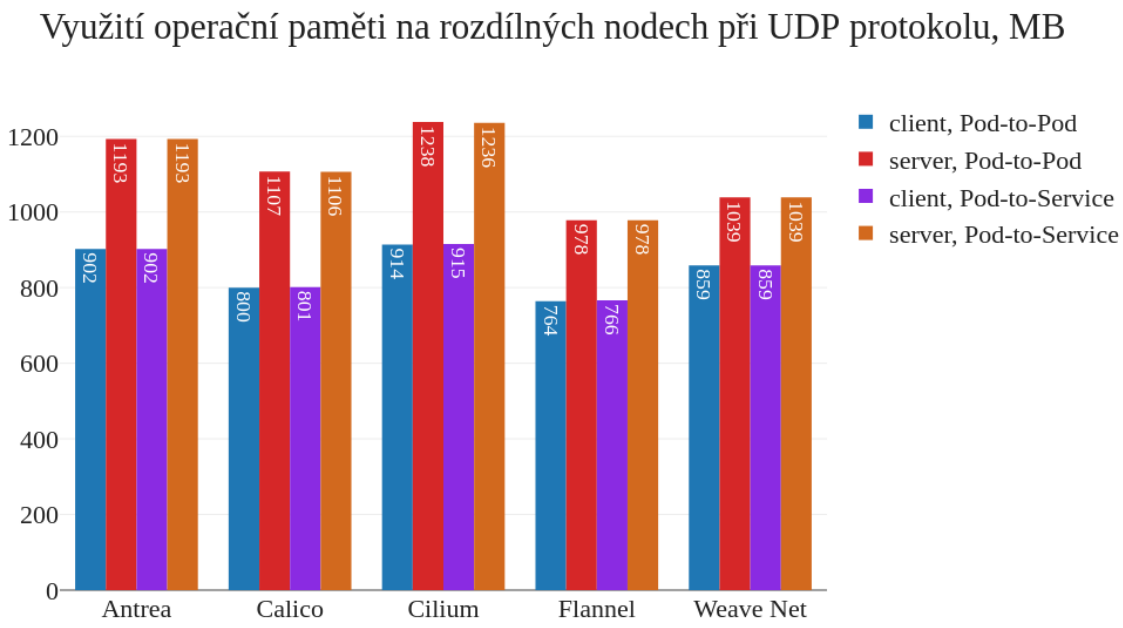


Obrázek 43. Graf využití RAM na rozdílných nodech při TCP protokolu

Odchylky mezi využitím operační paměti na klientovi a serveru, vyobrazené v grafu na obrázku (Obr. 43) mohou být z části způsobené využíváním master nodu stejně jako worker node.

Flannel, stejně jako u měření při nečinnosti, dosahuje opět nejmenších hodnot v rámci využívání prostředků. K jeho hodnotám se lehce blíží i implementace Weave Net, která naopak u využití procesoru nabývala téměř dvojnásobných hodnot oproti Flannelu.

8.3.3 UDP protokol

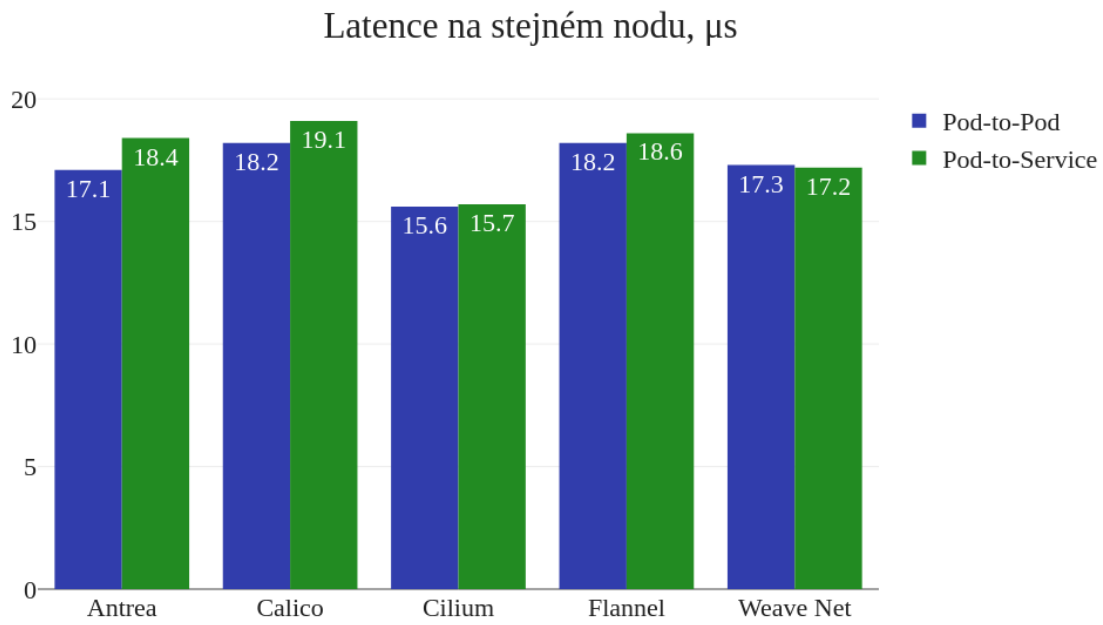


Obrázek 44. Graf využití RAM na rozdílných nodech při UDP protokolu

Z grafů na obrázcích (Obr. 43 a Obr. 44) je zřejmá nezávislost na využitém protokolu. Měření jak pro TCP, tak UDP protokol nabývají téměř totožných hodnot ve všech scénářích.

8.4 Latence

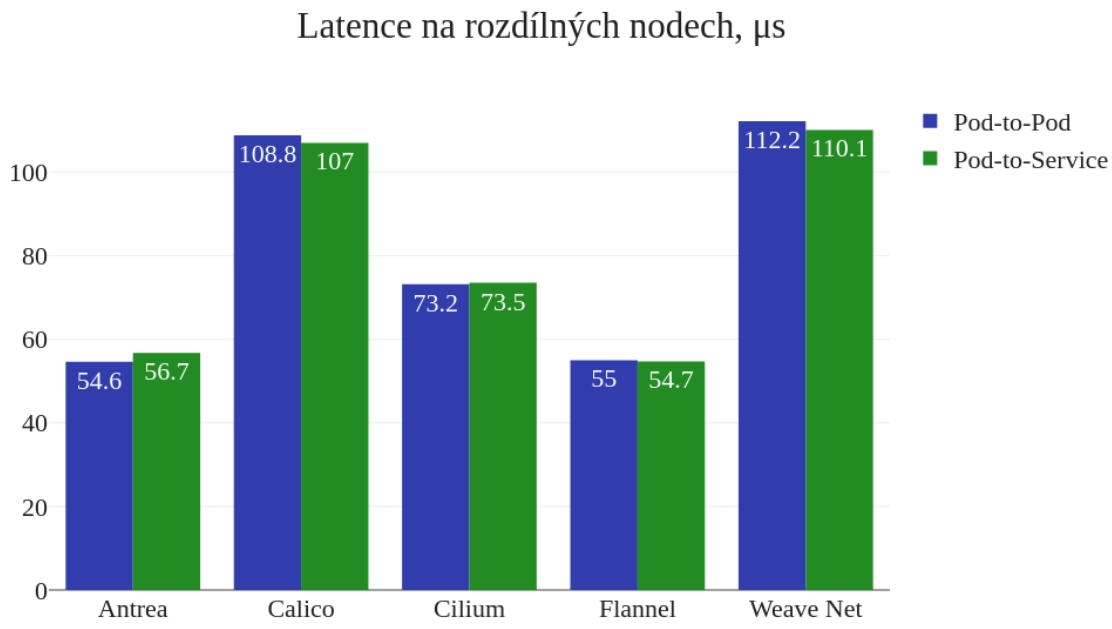
Poslední testovanou oblastí je síťová latence, měřená v mikrosekundách. V grafech jsou opět porovnávány naměřené hodnoty pro jednotlivé implementace, kde první (modrý) sloupec představuje hodnoty při Pod-to-Pod komunikaci a druhý (zelený) sloupec hodnoty Pod-to-Service komunikace.



Obrázek 45. Graf latence při síťové komunikaci na stejném nodu

Co se týče latence na stejném nodu (Obr. 45), jsou odchylky hodnot pro Pod-to-Pod a Pod-to-Service komunikaci opět téměř zanedbatelné.

Nejmenší latenci z vybraných implementací má Cilium, což odpovídá nejvyšším hodnotám propustnosti přes oba protokoly mezi všemi implementacemi (Obr. 35 a Obr. 37) a potvrzuje tvrzení, že vliv na velikost propustnosti má také z určité části síťová latence. [20]



Obrázek 46. Graf latence při síťové komunikaci na rozdílných nodech

Latence v případě rozdílných nodů (Obr. 46) je mnohonásobně větší nežli na stejném nodu. Lze si také povšimnout nízké latence u implementací Antrea a Flannel, kdy, stejně jako tomu je v předchozím případě u Cilia, nabývají jedny z největších propustností na dvou rozdílných nodech (Obr. 36 a Obr. 38).

Naopak vysoká latence u implementace Weave Net potvrzuje jednu z nejmenších propustností v rámci rozdílných nodů (Obr. 36 a Obr. 38).

8.5 Shrnutí

Složitost instalace jednotlivých implementací je u všech vybraných velmi podobná. V případě Calico a Weave Net implementací je nutný zásah do firewall a povolení potřebných portů. Tento fakt může hrát roli v případě, kdyby některý z portů kolidoval s existujícím portem například u již dříve vytvořeného clusteru. Calico nabízí možnost výběru implementace overlay network, což je potřeba zvážit před samotnou instalací. Některé řešení, poskytující virtuální stroje a tím pádem i síť na nižší úrovni, nepovolují využití některých z implementací overlay network.

U možností konfigurace jsou na tom všechny implementace opět podobně, kdy jediný Flannel neposkytuje správu zásad sítě. Často se tedy využívá kombinace dvou implementací, kdy

komunikaci zajišťuje Flannel a zásady sítě spravuje implementace jiná. Vedle zásad sítě také většina implementací umožňuje komunikaci šifrovat. Jedinou výjimkou je Flannel.

Při propustnosti dosahují nejlepších hodnot implementace Antrea a Flannel. V případě Calica by hodnoty mohly být teoreticky o něco lepší, pokud by se správně nastavila hodnota MTU. U Flannelu se takový výsledek očekávat dal, jelikož se primárně zaměřuje na samotnou komunikaci v rámci clusteru.

Flannel dosahuje poměrně nízkých hodnot oproti ostatním implementacím, co se týče využívání prostředků virtuálních strojů (procesor a operační paměť). Na druhou stranu Calico dokáže vytíženost rozložit rovnoměrně mezi oba testované nody.

U porovnání síťové latence se na vrchol vyzdvihuje implementace Cilium, jež má největší propustnost v rámci jednoho stejného nodu, tudíž disponuje i malou latencí. U latence na dvou rozdílných nodech nejlíp vychází opět Antrea a Flannel, kdy na druhou stranu Weave Net má naměřenu nejmenší propustnost a největší latenci mezi dvěma nody.

ZÁVĚR

Hlavním cílem bakalářské práce bylo porovnání implementací síťového subsystému v rámci Kubernetes clusteru.

Celá práce se skládá ze dvou hlavních částí. V první, teoretické části, byla popsána virtualizace jako taková, včetně jejích typů, což tvoří základ k pochopení kontejnerových aplikací, i pro platformu Docker a Kubernetes. Byly popsány základy a komponenty platformy Docker, která je v rámci této bakalářské práce využívána jako container runtime pro Kubernetes. Nedílnou součástí této části bylo také seznámení se samotným orchestračním nástrojem Kubernetes, včetně jeho základních komponent. Samostatná kapitola byla věnována síťové komunikaci mezi jednotlivými komponentami v clusteru. Závěrem teoretické části byly vybrány a popsány implementace síťového subsystému, které následně byly podle sestavené metodiky testovány a zhodnoceny v praktické části.

V rámci druhé, praktické části, byl prvně připraven workspace pro možnost simulace Kubernetes clusteru a spuštění samotných testů nad vytvořeným clusterem. Dále byla sestavena metodika testování určitých oblastí, jako je propustnost, latence a využití procesoru či operační paměti. Dle sestavené metodiky byly všechny vybrané implementace síťového subsystému otestovány a získané výsledky náležitě zpracovány. Posledním krokem praktické části bylo zhodnocení zpracovaných dat pro každou testovanou oblast, včetně konečného shrnutí, které obsahuje také zohlednění náročnosti instalace a konfigurace jednotlivých implementací.

Výsledná měření vycházejí z jednoho statického testovacího modelu, kdy bylo využito externí služby pro provoz virtuálních strojů s určitými omezeními a pevně dané konfiguraci Kubernetes clusteru. V případě změny testovacího modelu se mohou naměření hodnoty lišit, což může ovlivnit výběr vhodné implementace. K dosažení větší objektivity by bylo vhodné mít pro testování cluster s více než jedním worker nodem a oprostit master node od pracovní zátěže, případně zkoumat chování při zatížené síti.

Většina implementací nabízí automatické rozpoznávání hodnoty MTU, kde bohužel některé tuto hodnotu špatně vyhodnocují a je vhodné ji nastavit ručně dle konfigurace samotné sítě. Využitá externí služba nepovolovala vyšší hodnotu MTU nežli jimi nastavených 1500 bajtů. Pro zajímavost by bylo vhodné porovnat testované oblasti při této standardní hodnotě a například s využitím tzv. jumbo packetů (cca 9000 bajtů).

Z naměřených dat vyplývá, že pro malé clustery bez nutnosti definování zásad sítě se jeví jako nejvhodnější Flannel, který ve většině testovaných praktických oblastí, jako je propustnost nebo vytížení prostředků, vyčníval nad ostatními. Calico pak nabízí vyrovnaný výkon napříč clusterem s možností definování zásad, šifrovanou komunikací či jiné další konfigurace. Cilium se naopak může hodit, pokud je cluster složen pouze z jednoho nodu nebo v případě využití funkcionality správy zásad na aplikační vrstvě pro moderní protokoly jako je například HTTP nebo Kafka.

Implementace Antrea vykazovala ve většině testech podobné hodnoty jako Flannel, co se týče propustnosti a síťové latence. Naopak dosahovala většího zatížení prostředků, což může být také způsobeno větší robustností samotné implementace, na rozdíl od Flannelu.

Nejhůře ze všech testovaných implementací vychází Weave Net, který nejenže vykazoval malé hodnoty propustnosti a latence, ale zároveň i vyšší vytížení testovaných prostředků virtuálního stroje.

SEZNAM POUŽITÉ LITERATURY

[1] HUANG, Kaizhe a Pranjal JUMDE. Learn Kubernetes Security: Securely orchestrate, scale, and manage your microservices in Kubernetes deployments. Birmingham, UK: Packt Publishing, 2020. ISBN 978-1-83921-650-3.

[2] BAIER, Jonathan a Jesse WHITE. Getting Started with Kubernetes: Extend your containerization strategy by orchestrating and managing large-scale container deployments. 3rd edition. Birmingham, UK: Packt Publishing, 2018. ISBN 978-1-78899-472-9.

[3] BAIER, Jonathan, Gigi SAYFAN a Jesse WHITE. The Complete Kubernetes Guide: Become an expert in container management with the power of Kubernetes. Birmingham, UK: Packt Publishing, 2019. ISBN 978-1-83864-734-6.

[4] BURNS, Brendan. Designing Distributed Systems. O'Reilly Media, 2018. ISBN 978-1-49198-364-5.

[5] SAYFAN, GIGI. Mastering Kubernetes. Birmingham, UK: Packt Publishing, 2017. ISBN 978-1-78646-100-1.

[6] GOASGUEN, Sebastien a Michael HAUSENBLAS. Kubernetes Cookbook. O'Reilly Media, 2018. ISBN 978-1-49197-968-6.

[7] Virtualizace [online]. Brno: Fakulta informatiky Masarykovy univerzity, 2016 [cit. 2021-03-25]. Dostupné z: https://www.fi.muni.cz/~kas/pv090/referaty/2016-podzim/virt.html

[8] MATYSKA, Luděk. Techniky virtualizace počítačů (2). Zpravodaj ÚVT MU [online]. 2007, XVII(3), 9-12 [cit. 2021-03-25]. ISSN 1212-0901. Dostupné z: http://webserver.ics.muni.cz/bulletin/cisla_tisk/82.pdf

[9] MATYSKA, Luděk. Virtualizace výpočetního prostředí. Zpravodaj ÚVT MU [online]. 2006, XVII(2), 9-11 [cit. 2021-03-25]. ISSN 1212-0901. Dostupné z: http://webserver.ics.muni.cz/bulletin/cisla_tisk/81.pdf

[10] Docker Engine overview [online]. Docker Docs [cit. 2021-03-26]. Dostupné z: https://docs.docker.com/engine

[11] Docker overview [online]. Docker Docs [cit. 2021-03-26]. Dostupné z: https://docs.docker.com/get-started/overview

[12] Use volumes [online]. Docker Docs [cit. 2021-03-26]. Dostupné z: https://docs.docker.com/storage/volumes

[13] Container runtimes [online]. Docker Docs [cit. 2021-4-24]. Dostupné z: https://kubernetes.io/docs/setup/production-environment/container-runtimes/

[14] Kubernetes: Prerequisites for Setup Kubenetes Cluster| Part 2 [online]. Sarasa Gunawardhana, 2019 [cit. 2021-4-24]. Dostupné z: https://faun.pub/kubernetes-prerequisites-for-setup-kubenetes-cluster-part-2-699b3f93d6cc

[15] What Is Throughput in Networking? Bandwidth Explained [online]. Staff Contributor, 2019 [cit. 2021-4-24]. Dostupné z: https://www.dnsstuff.com/network-throughput-bandwidth#what-is-throughput-in-networking

[16] Iperf: měření rychlosti spojení [online]. Adam Štrauch, 2012 [cit. 2021-4-24]. Dostupné z: https://www.root.cz/clanky/iperf-mereni-rychlosti-spojeni/

[17] iPerf – iPerf3 and iPerf2 user documentation [online]. iPerf [cit. 2021-4-24]. Dostupné z: https://iperf.fr/iperf-doc.php

[18] Free(1) – Linux manual page [online]. 2018 [cit. 2021-4-26]. Dostupné z: https://man7.org/linux/man-pages/man1/free.1.html

[19] 10 Useful Sar (Sysstat) Examples for UNIX / Linux Performance Monitoring [online]. Ramesh Natarajan, 2011 [cit. 2021-4-26]. Dostupné z: https://www.thegeekstuff.com/2011/03/sar-examples/

[20] Understanding What Is Network Latency and How You Can Resolve Network Latency Issues [online]. Tek-Tools, 2020 [cit. 2021-4-26]. Dostupné z: https://www.tek-tools.com/network/resolve-network-latency-issues

[21] Qperf(1) - Linux man page [online]. Johann George [cit. 2021-4-26]. Dostupné z: https://linux.die.net/man/1/qperf

[22] GitHub - InfraBuilder / k8s-bench-suite [online]. Alexis Ducastel [cit. 2021-4-29]. Dostupné z: https://github.com/InfraBuilder/k8s-bench-suite

[23] GitHub - danielcoman / kubernetes-qperf [online]. Daniel Coman [cit. 2021-4-29]. Dostupné z: https://github.com/danielcoman/kubernetes-qperf

[24] Container Network Interface (CNI) Specification [online]. [cit. 2021-5-1]. Dostupné z: https://github.com/containernetworking/cni/blob/master/SPEC.md

[25] Antrea Docs - Overview [online]. Antrea Authors [cit. 2021-5-1]. Dostupné z: https://antrea.io/docs/v1.0.1/

[26] Antrea Docs - Antrea Architecture [online]. Antrea Authors [cit. 2021-5-1]. Dostupné z: https://antrea.io/docs/v1.0.1/design/architecture/

[27] Calico [online]. IBM Corporation, 2017 [cit. 2021-5-1]. Dostupné z: https://www.ibm.com/docs/en/cloud-private/3.2.0

[28] Determine best networking option [online]. Calico [cit. 2021-5-1]. Dostupné z: https://docs.projectcalico.org/networking/determine-best-networking

[29] Flannel [online]. Flannel Maintainer Community [cit. 2021-5-1]. Dostupné z: https://github.com/flannel-io/flannel

[30] Cilium [online]. [cit. 2021-5-1]. Dostupné z: https://github.com/cilium/cilium

[31] Service [online]. Kubernetes [cit. 2021-5-1]. Dostupné z: https://kubernetes.io/docs/concepts/services-networking/service/

[32] System requirements [online]. Calico [cit. 2021-5-1]. Dostupné z: https://docs.projectcalico.org/getting-started/kubernetes/requirements

[33] Integrating Kubernetes via the Addon [online]. Weave Works [cit. 2021-5-1]. Dostupné z: https://www.weave.works/docs/net/latest/kubernetes/kube-addon/

[34] Monitoring Kubernetes Control Plane using KubeSphere [online]. KubeSphere, 2020 [cit. 2021-5-12]. Dostupné z: https://itnext.io/monitoring-kubernetes-control-plane-using-kubesphere-7885461f40b1

SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK

CNI	Container Network Interface
API	Application Programming Interface
CLI	Command Line Interface
GUI	Graphical User Interface
GCE	Google Cloud Engine
SSH	Secure Shell
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
RAM	Random Access Memory
CPU	Central Processing Unit
JSON	JavaScript Object Notation
IP	Internet Protocol
OVS	Open vSwitch
IPAM	IP Address Management
MTU	Maximum Transmission Unit
HTTP	Hypertext Transfer Protocol
DNS	Domain Name System

SEZNAM OBRÁZKŮ

Obrázek 1. Schéma principu plné virtualizace [7] 11
Obrázek 2. Schéma principu paravirtualizace [7] 12
Obrázek 3. Schéma principu kontejnerové virtualizace [7] 12
Obrázek 4. Schéma principu emulace [7] 13
Obrázek 5. Schéma architektury Docker platformy [11] 14
Obrázek 6. Statistiky běžícího kontejneru v Docker 15
Obrázek 7. Formulář pro přidání konečného bodu do Portaineru 16
Obrázek 8. Ukázka souboru Dockerfile 17
Obrázek 9. Příkaz pro spuštění kontejnerové aplikace 18
Obrázek 10. Diagram základní struktury Kubernetes custeru [34] 19
Obrázek 11. Definice podu v Kubernetes 20
Obrázek 12. Definice service v Kubernetes 21
Obrázek 13. Definice replication controller v Kubernetes 22
Obrázek 14. Definice network policy v Kubernetes 23
Obrázek 15. Diagram komunikace klienta (podu) se service [31] 25
Obrázek 16. Komunikace mezi pody s implementací Antrea [26] 28
Obrázek 17. Diagram Kubernetes clusteru s využitím Calico CNI rozšíření [27] 29
Obrázek 18. Ukázka nastavení zásad sítě v Calico CNI 31
Obrázek 19. Diagram funkcionality poskytované Cilium implementací [30] 31
Obrázek 20. Příkazy k instalaci Cilium implementace pomocí Helm 32
Obrázek 21. Ukázka definice zásady sítě v Cilium na aplikační vrstvě 32
Obrázek 22. Konfigurace virtuálního stroje na GCE 37
Obrázek 23. Výběr operačního systému virtuálního stroje na GCE 37
Obrázek 24. Instalace Docker container runtime 38
Obrázek 25. Instalace Kubelet, Kubeadm a Kubectl 38
Obrázek 26. Nastavení síťové komunikace a vypnutí swapování 38
Obrázek 27. Inicializace Kubernetes clusteru 38
Obrázek 28. Spuštěný nástroj „iperf“ v serverovém režimu 41
Obrázek 29. Spuštěný nástroj „iperf“ v klientském režimu 41
Obrázek 30. Spuštění testovací sady Kubernetes Bench Suite 42
Obrázek 31. Informace o využití procesoru pomocí příkazu „sar“ 43
Obrázek 32. Informace o operační paměti pomocí příkazu „free“ 44

Obrázek 33. Spuštěný nástroj „qperf“ v klientském režimu.....45
Obrázek 34. Definice pro kontejner v podu s nástrojem „qperf“ 46
Obrázek 35. Graf propustnosti při TCP protokolu na stejném nodu 47
Obrázek 36. Graf propustnosti při TCP protokolu na rozdílných nodech 48
Obrázek 37. Graf propustnosti při UDP protokolu na stejném nodu.....49
Obrázek 38. Graf propustnosti při UDP protokolu na rozdílných nodech 49
Obrázek 39. Graf využití CPU při nečinnosti na rozdílných nodech 50
Obrázek 40. Graf využití CPU na rozdílných nodech při TCP protokolu.....51
Obrázek 41. Graf využití CPU na rozdílných nodech při UDP protokolu 52
Obrázek 42. Graf využití RAM při nečinnosti na rozdílných nodech 53
Obrázek 43. Graf využití RAM na rozdílných nodech při TCP protokolu.....54
Obrázek 44. Graf využití RAM na rozdílných nodech při UDP protokolu.....55
Obrázek 45. Graf latence při síťové komunikaci na stejném nodu.....56
Obrázek 46. Graf latence při síťové komunikaci na rozdílných nodech 57

SEZNAM TABULEK

Tabulka 1. Předávané parametry při ovládání síťového subsystému [24].....26

SEZNAM PŘÍLOH

PŘÍLOHA P I: CD disk s bakalářskou prací a soubory k reprodukci měření