FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

Zadání bakalářské práce

Název:	Síťová komunikace aplikací v Kubernetes s externími zařízeními v privátní síti
Student:	Jan Troják
Vedoucí:	Ing. Tomáš Vondra, Ph.D.
Studijní program:	Informatika
Obor / specializace:	Bezpečnost a informační technologie
Katedra:	Katedra počítačových systémů
Platnost zadání:	do konce letního semestru 2023/2024

Pokyny pro vypracování

Kubernetes se stává přední technologií pro správu aplikací ve formě kontejnerů. Kubernetes řeší komunikaci ve vnitřní síti klastru. V případě potřeby komunikace s externím zařízením, které není a nemůže být součástí sítě klastru, nenabízí technologie Kubernetes jednoduché řešení. Cílem práce je rozšířit funkcionalitu Kubernetes tak, aby umožila komunikaci právě se zařízeními mimo vnitřní síť klastru.

Prozkoumejte, jaké možnosti pro síťování Kubernetes nabízí.
Prozkoumejte možnosti adresace a komunikace se zařízeními, které se nachází v privátní síti, mimo síť klastru.
Zaměřte se na komunikaci pomocí TCP, UDP a HTTP protokolů.
Navrhněte a implementujte řešení, které umožní navázat komunikaci mezi kontejnery v Kubernetes a zařízeními mimo interní síť klastru.

Implementaci je možné provést čistě ve virtuálním prostředí. V případě implementace ve virtuálním prostředí dodejte kompletní virtuální prostředí nebo definici prostředí.

Elektronicky schválil/a prof. Ing. Pavel Tvrdík, CSc. dne 4. ledna 2023 v Praze.

Bakalářská práce

SÍŤOVÁ KOMUNIKACE APLIKACÍ V KUBERNETES S EXTERNÍMI ZAŘÍZENÍMI V PRIVÁTNÍ SÍTI

Jan Troják

Fakulta informačních technologií
Katedra počítačových systémů
Vedoucí: Ing. Tomáš Vondra, Ph.D.
10. května 2023

Obsah

Seznam obrázků

Seznam výpisů kódu z příkazové řádky

Chtěl bych poděkovat vedoucímu práce Ing. Tomáši Vondrovi, Ph.D., který velmi pomohl při vzniku této práce ať už věcnými podněty, nebo nasměrováními které vedly k lepším výsledkům. Zároveň bych mu chtěl poděkovat za poskytnutí velké volnosti, díky které jsem se mohl konkrétněji zaměřit na témata mně blízká.
Dále bych chtěl poděkovat Daně Suchomelové, Danielu Hromádkovi a Samuelu Švalichovi za poskytovanou psychickou podporu a prostory pro hacketony, při kterých primárně tato práce vznikla.
V neposlední řadě bych chtěl poděkovat open-source komunitě za vytváření skvělých produktů, které jsou volně dostupné široké veřejnosti.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 2373 odst. 2 zákona č. 89/2012 Sb., občanský zákoník, ve znění pozdějších předpisů, tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou a veškeré jejich dokumentace (dále souhrnně jen „Dílo“), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené.

Abstrakt

Tato práce se zabývá možnostmi síťování v systému Kubernetes. Cílem práce bylo rozšířit Kubernetes o možnost adresace a komunikace cloudu se zařízeními v privátních sítích. Známá řešení poskytují pouze komunikaci pomocí vysokoúrovňových protokolů. Cílem bylo nalézt řešení, které by podporovalo protokoly nižších vrstev ISO/OSI.

Práce představuje možnost rozšíření systému Kubernetes o zmíněnou funkcionality síťové komunikace. Tento způsob umožňuje komunikaci pomocí TCP a UDP protokolů se zařízeními v privátních sítích. Představené řešení nabízí flexibilitu použití a nepředstavuje žádná omezení pro standardní použití systému Kubernetes. Řešení je realizováno pomocí zavedených standardů pro rozšiřování systému.

Výsledky této práce poskytují širší možnosti pro použití systému Kubernetes. Díky tomuto rozšíření je možné lépe využít systém Kubernetes v oblastech testování, smart cities a dalších oblastech pracujících se zařízeními v privátních sítích.

Klíčová slova Kubernetes, CNI, network driver, privátní síťový segment, edge cloud computing, Kubernetes operátor, síťování, TCP, UDP, K8S

Abstract

This thesis explores the networking capabilities of Kubernetes. The aim of the thesis was to extend Kubernetes with the possibility of addressing and communicating with devices in private networks. Known solutions only provide communication using high-level protocols. The goal was to find a solution that would support communication using lower layer ISO/OSI protocols.

This thesis presents the possibility of extending the kubernetes system with the mentioned functionalities of network communication. This method allows communication with devices in private networks using TCP and UDP protocols. The presented solution offers flexibility of use and does not present any limitation restricting standard use of Kubernetes. The solution is implemented using established standards for extending the system.

The results of this work provide wider possibilities for the use of Kubernetes. With this extension, it is possible to make better use of Kubernetes in the areas of testing, smart cities and other areas working with devices in private networks.

Keywords Kubernetes, CNI, network driver, private network segment, edge cloud computing, Kubernetes operator, networking, TCP, UDP, K8S

Seznam zkratek

API	Application Programming Interface
ARP	Address Resolution Protocol
AWS	Amazon Web Services
BGP	Border Gateway Protocol
CNI	Container Network Interface
CRD	Custom Resources Definition
CRI	Container Runtime Interface
CRUD	Create, Read, Update, Delete
ČVUT	České vysoké učení technické v Praze
DNS	Domain Name Server
FIT	Fakulta informačních technologií ČVUT - Praha
GRASP	General Responsibility Assignment Software Patterns
HIL	Hardware In Loop
HTTP	Hypertext Transfer Protocol
HW	Hardware
IP	Internet Protocol
ISO/OSI	Referenční model ISO/OSI
JSON	JavaScript Object Notation
LSM	Linux Security Modules
MQTT	Message Queue Telemetry Transport
NAD	Network Attachment Definition
NAT	Network address translation
NPWG	Network Plumbing Working Group
OCI	Open Container Initiative
OS	Operační systém
QUIC	Quick UDP Internet Connection
REST	Representational State Transfer
RFC	Request For Comments
SDK	Software Development Kit
SIL	Software In Loop
SSL	Secure Sockets Layer
TCP	Transmission Control Protocol
TLS	Transport Layer Security
UDP	User Datagram Protocol
URI	Uniform Resource Identifier
UTS	UNIX Time-Sharing
VXLAN	Virtual Extensible Local Area Network
YAML	YAML Ain't Markup Language

Úvod

Orchestrační nástroj (zkráceně orchestrátor) je nástroj, který slouží pro ulehčení práce s různými informačními systémy. Často se jedná o technologii složenou z malých programů a modulů, které automatizují a ulehčují práci. Ochestrátorů je velká řada z mnoha různých kategorií. Mezi nejznámější orchestrátory kontejnerových aplikací patří Kubernetes. Právě tímto orchestrátorem se tato práce zabývá. [1]

Kubernetes je orchestrační technologie, která poskytuje prostředí pro provoz aplikací na více serverech. Kubernetes tak vytváří a poskytuje jednotné prostředí pro správu aplikací, nazývané cloud. Kubernetes podporuje různé způsoby komunikací mezi aplikacemi a službami v síti cloudu. Toto prostředí je tvořeno více výpočetními uzly. Pro potřeby síťování Kubernetes používá interní privátní virtuální síť, která je sdílena mezi všemi uzly systému. Tuto síť mohou využívat všechny objekty, které jsou součástí daného systému Kubernetes.

Pro propojení vnitřní sítě s okolním světem poskytuje Kubernetes standardní řešení. Tato standardní řešení komunikace, které systém nabízí jsou primárně jednostranné, spoléhají se na komunikaci s veřejnými adresami a nenabízí přímou kontrolu nad tokem dat. Pro spolehlivou oboustrannou komunikaci se zařízeními v privátních sítích, které se nacházejí mimo zmíněnou virtuální síť, není technologie v základu připravena. To je značné omezení v případě, že do systému je potřeba připojit reálný hardwarový prvek, který nelze přímo integrovat do sítě Kubernetes. Takovými prvky jsou například jednoduchá zařízení, která sbírají data, různé periferie, testovaná zařízení apod. Obecně je lze tato zařízení označit jako externí hardwarové prvky.

Tato práce se zaměřuje na to, jak rozšířit možnosti orchestrátoru Kubernetes o možnost adresace a komunikace s hardwarovými zařízeními v privátních sítích. Zkoumaná komunikace s hardwarovými prvky bude probíhat pomocí *TCP*, *UDP* a *HTTP* protokolů.

Navržené řešení by mělo být obecné a nezávislé na nestandardním nastavení Kubernetes. Zároveň by nemělo nijak ovlivňovat jakékoliv funkcionality systému.

0.1 Motivace

Tato práce vnikla pro potřeby HIL[1] testování v prostředí cloudu. Hlavní myšlenkou je nalézt způsob, jak umožnit testování komunikace různých hardwarových prvků a simulací tak, aby bylo možné prvky a simulace jednoduše kombinovat. Pro tyto účely je zapotřebí umožnit komunikaci mezi Kubernetes a zařízeními, které nejsou připojené do interní sítě. Díky integraci HIL testování do prostředí cloudu se zlepší možnosti testování. Zároveň se zjednoduší práce potřebná pro nastavování prostředí.

0.2 Cíle práce

Cílem této práce je nalézt způsob, jak zajistit adresaci a komunikaci s hardwarovými prvky, nacházejícími se mimo interní síť Kubernetes. Zkoumána bude komunikace pomocí protokolů *TCP* , *UDP* a *HTTP*. V případě, že nebude známo žádné řešení, které by splnilo kladené nároky, pak je za cíl považován návrh a implementace řešení pro výše popsaný problém.

0.3 Struktura práce

Práce je strukturována do tří kapitol. První kapitola představuje základní koncepty systému Kubernetes. Hlavní část bude věnována možnostem síťování, které tento systém nabízí. Informace obsažené v teoretické části slouží jako stavební bloky pro zbytek této práce.

Druhá kapitola je věnována samotnému problému adresace zařízeních v přilehlé privátní síti. Zde je problematika představena převážně na teoretické úrovni. V této části jsou diskutovány možná řešení umožňující zkoumanou komunikaci.

Poslední, třetí kapitola popisuje realizaci řešení představeného v kapitole předchozí. Zde je popsáno prostředí použité při implementaci, konkrétní způsob podpory komunikace a implementace rozšíření systému Kubernetes.

0.4 Dohoda se čtenářem

V této práci se budou často vyskytovat názvy objektů ze systému Kubernetes. Tyto názvy budou uvedeny s velkými počátečními písmeny. Toto je zavedená konvence proto, aby se názvy objektů nepletly se slovy běžného jazyka. Tato konvence dává dobrý smysl zejména v anglicky psané literatuře. I přesto, že tato práce je psaná v jazyce českém, bude tato konvence dodržována. Příkladem objektu může být objekt typu Deployment s velkým počátečním „D“.

V případě, že je v práci uveden výpis z příkazové řádky, nebo část zdrojového kódu, bude použit specifický blok. Ukázkový blok je uveden ve výpisu kódu 1.

Pokud se příkazy v ukázkách provádějí v různých prostředích, budou prostředí uvedena v hranatých závorkách. Pokud je prostředí jednotné, bude použit symbol podtržítka. Příkazy vždy začínají symbolem `$`, výstupy konzole jsou uvedeny symbolem `>>>` .

Příkazy z těchto bloků jsou převeditelné na spustitelný script pomocí příkazu. který je uveden v ukázce 2.

[1] HIL (hardware in loop) je technika testování hardwarových zařízení, kde je zařízení testováno v simulovaném prostředí. Simulace prostředí nejčastěji probíhá pomocí matematických modelů, které generují signály pro daná zařízení.

Výpis kódu 1 Ukázkový blok výpisu konzole

```
[1]$ ip -4 --brief address show eth0        # this is environment (PC) 1
>>> eth0         UP       192.168.124.176/24
[2]$ ip -4 --brief address show eth0        # this is environment (PC) 2
>>> eth1         DOWN     192.168.124.177/24
```

Výpis kódu 2 Příkaz převedení výpisu z konzole na spustitelný script

```
[_]$ sed 's␣/^\[[[:digit:]_]\]\$␣//␣;␣/^>>>/␣d' block.sh | tee script.sh
```

0.5 Prostředí a použité verze softwaru

Veškeré příklady jsou prováděny ve virtuálním stroji na systému `CentOS/7`. Virtuální prostředí lze spustit pomocí služby `Vagarnt`. V případě potřeby lze virtuální prostředí nastavit pomocí následujícího příkazu, který je uveden ve výpisu 3.

Výpis kódu 3 Nastavení prostředí pomocí Vagrant

```
[1]$ cat > Vagrantfile <<EOF
Vagrant.configure("2") do |config|
  config.vm.box = "centos/7"
end
EOF
[1]$ vagrant up
[1]$ vagrant ssh
[2]$ hostnamectl
>>>   Static hostname: localhost.localdomain
>>>         Icon name: computer-vm
>>>           Chassis: vm
>>>        Machine ID: d1a6b9d5e7f4af49b5c53c99d86d520b
>>>           Boot ID: 077a2c59fd4545889bb1566fd23d5c58
>>>    Virtualization: kvm
>>>  Operating System: CentOS Linux 7 (Core)
>>>       CPE OS Name: cpe:/o:centos:centos:7
>>>            Kernel: Linux 3.10.0-1127.el7.x86_64
>>>      Architecture: x86-64
```

Pro správné použití je potřeba mít nainstalovaný Vagrant a libovolný podporovaný virtualizační nástroj (hypervisor).

Veškeré informace v této práci se vztahují k verzi Kubernetes `1.26.0`.

Kapitola 1
Teoretická část

V této kapitole jsou vysvětleny a popsány základy kontejnerizace a orchestrátoru Kubernetes. Se znalostí těchto konceptů je následně vysvětlena problematika síťování kontejnerů a síťování v Kubernetes.
Porozumění této kapitoly je kritické pro pochopení navazujících kapitol.

1.1 Kontejnery

Kontejnerizace je způsob virtualizace a izolace prostředí na úrovni operačního systému. Tato práce se zabývá pouze aplikačními kontejnery, kdykoliv v textu je uveden výraz kontejner, řeč je o kontejneru aplikačním. Výraz kontejner reprezentuje běžící instanci kontejnerového obrazu (container image). Dále jsou probírány pouze linuxové kontejnery splňující Open Container Initiative specifikaci.

Pro lepší přiblížení kontejnerů a kontejnerizace obecně, doporučuji článek *Learning Containers From The Bottom Up* [2] od *Ivan Velichko* a přednášku *Kontejnery – principy a Docker* od *Ing. Tomáše Vondry, Ph.D.* [3]

1.1.1 Open Container Initiative

Za velký pokrok v oblasti kontejnerizace z velké části může společnost Docker, Inc, která je autorem stejnojmenné technologie Docker. Docker vznikl jako interní nástroj pro poskytování služeb ve společnosti dotCloud. V roce 2013 se společnost dotCloud přetransformovala na společnost Docker, Inc. [4]

Technologie Docker zažila masivní úspěch. Právě kvůli vzrůstající popularitě kontejnerizace vznikl projekt s názvem Open Container Initiative (OCI).

Dle oficiálních stránek, OCI je projekt, který vznikl za účelem vytvoření a udržovaní otevřených standardů pro formát kontejnerů a běhových prostředí kontejnerů (container runtimes). Na projektu se podílí jak nadšení jednotlivci, tak i velké společnosti jako je například RedHat, IBM, Docker a další. Projekt poskytuje sadu standardů pro kontejnerové technologie. Díky těmto standardům jsou dnes jasně definovaná rozhraní, na které se mohou spoléhat jiné technologie pracující právě s kontejnery. [5]

Open Container Initiative momentálně spravuje tři standardy. Konkrétně se jedná o *Runtime Specification*, *Image Specification* a *Distribution Specification*. [5]

Image Specification (česky specifikace obrazu kontejneru) definuje převážně podobu manifestů pro kontejnery a podobu rejstříků kontejnerů. První část standardu definuje formát manifestu pro obraz kontejneru. Účelem je zajistit, adresovatelnost jednotlivých konfigurací obrazů kontejneru.

Toho je docíleno pomocí hašovaní a generování unikátních identifikátorů. Další část specifikace popisuje rejstřík, pro uchovávání jednotlivých manifestů kontejnerů. Třetí část specifikace popisuje způsob, jakým serializovat filesystém kontejneru a případně změny tohoto filesystému. Poslední část specifikace definuje formát pro popis obrazu kontejneru. Tento formát obsahuje potřebné informace, které následně využívá běhové prostředí kontejnerů. Jedná se převážně o metadata obrazu kontejneru a popis filesystému. [6]

Runtime Specification (česky specifikace běhového prostředí) specifikuje konfiguraci, běhové prostředí a životní cyklus kontejneru. V první části jsou vydefinovány možné stavy kontejnerů a jejich význam, podporované operace s kontejnery (spuštění, pozastavení atd.) a životní cyklus kontejneru. Druhá část specifikace popisuje konfigurační soubor, který je použit při práci s kontejnery. Zbylé části obsahují různá rozšíření a popis běhového prostředí již pro konkrétní platformy. Popisovanými a proto i podporovanými platformami jsou *Linux*, *Solaris*, *Windows*, *virtuální stroje* a *z/OS*[1]. Nejdůležitější platformou pro účely této práce je Linux. V runtime specifikaci pro Linux je určeno, jaké prostředky mají být použity pro korektní běh kontejnerů. Jedná se o *namespaces*, *cgroups*, *capabilities*, *LSM* a *chroot*. Díky těmto nástrojům lze dosáhnout požadované virtualizace na linuxových systémech. [7]

Poslední specifikací OCI je *Distribution Specification*. Jedná se o nejnovější specifikaci v rámci Open Container Initiative. Tato specifikace popisuje API protokol, který slouží pro komunikaci s image container registry.[2] [8]

1.1.2 Síťování kontejnerů

Linuxové kontejnery dle standardu OCI pro izolaci síťování používají síťové jmenné prostory (network namespaces). Síťový jmenný prostor je jedním z osmi jemných prostorů jádra linuxových operačních systémů, které slouží k izolaci globálních prostředků jádra. Díky této izolaci lze procesy oddělit od nepotřebných systémových zdrojů. Síťový jmenný prostor abstrahuje veškeré prostředky spojené se síťováním. Mezi abstrahované prostředky patří například síťová rozhraní, IP adresy, IP tables a další. [9], [10]

V manuálových stránkách o síťovém jmenném prostoru jsou zmíněné následující informace. *„Síťové zařízení může být součástí právě jednoho síťového jmenného prostoru.“ „Pár dvou virtuálních síťových rozhraní (veth) může sloužit pro propojení dvou síťových zařízení v dvou rozdílných jmenných prostorech.“* [10] Tyto informace popisují, jak je možné propojit kontejnery s okolním světem. Právě virtuálních síťových rozhraní využívají i jednotlivé implementace pracující s kontejnery.

Způsobů jak kontejnery propojit s okolí sítí pomocí veth je mnoho. Níže bude popsán jeden z nejčastěji používaných způsobů[3].

Předpokládejme, již běžící docker container bez nastaveného síťování. Toho lze dosáhnout pomocí následujícího příkazu 1.1.

Výpis kódu 1.1 Vytvoření kontejneru bez nasatveného síťování

```
[1]$ docker run --net none travelping/nettools
```

Uvedeným příkaz vytvoří běžící proces `/bin/sh`, který bude oddělený od hostujícího sytému pomocí prázdného síťového jmenného prostředí. V tuto chvíli je vytvořen funkční linuxový kontejner, který není nijak připojen k okolní síti. Aktuálně se proces bude nacházet v novém síťovém prostoru. Tento prostor bude obsahovat pouze síťové rozhraní typu loopback, jak je vidět ve výpisu 1.2.

[1] *z/OS* je operační systém vyvíjeny spojeností IBM

[2] Container registry označuje službu která, implementuje API dle zmíněné specifikace. Container registry poskytuje vzdálené úložiště pro obrazy kontejnerů. Příkladem takové služby je DockerHub.

[3] Ukázka různých způsobů nastavení síťování v prostředí docker je uvedena na stránkách společnosti Docker

Výpis kódu 1.2 Kontejner s prázdným síťovým jmenným prostorem

```
# Create docker container without empty network namespace
[1]$ docker run -it --net none travelping/nettools

# List network interface inside the container
[2]$ ip --br l
>>> lo   UNKNOWN     00:00:00:00:00:00 <LOOPBACK,UP,LOWER_UP>
```

Proto aby bylo možné se z kontejneru připojit do okolní sítě, případně se ze sítě připojit do kontejneru, je potřeba spojení vytvořit. Jedním ze způsobů, jak spojení vytvořit je pomocí síťového ovladače (driveru) *bridge*. Tento způsob se skládá z následujících kroků. (Pořadí kroků odpovídá pořadí provádění v implementaci ovladače.)

1. Vytvoření síťového jmenného prostoru
   Nejprve se vytvoří jmenný prostor pro kontejner. Každý proces, běžící uvnitř kontejneru bude součástí tohoto prostoru.

2. Vytvoření rozhraní typu `bridge`
   Dalším krokem je vytvoření rozhraní typu `bridge`. Linuxový `bridge` je virtuální rozhraní, které slouží primárně k propojení více síťových segmentů. Propojení probíhá na druhé respektive třetí (záleží na použití) vrstvě referenčního modelu ISO/OSI. Toto rozhraní je vytvořeno v kořenovém prostoru hostujícího zařízení. V implementaci Dockeru se toto rozhraní nazývá `docker0`.

3. Vytvoření páru rozhraní typu `veth peer`
   Pro komunikaci mezi jmennými prostory je vytvořen pár virtuálních rozhraní. Tato rozhraní si navzájem přeposílají veškerou komunikaci. Pár slouží, jako prostředek komunikace mezi síťovým prostorem kontejneru a kořenovým jmenným prostorem.

4. Vložení jednoho rozhraní `veth` do síťového jmenného prostoru
   V tomto kroku se vloží jeden z páru rozhraní do vytvořeného jmenného prostoru.

5. Připojení druhého konce `veth` páru do vytvořeného `bridge` rozhraní
   Nyní se druhý z virtuálních páru připojí do rozhraní `bridge`. V tuto chvíli je vytvořeno spojení mezi rozhraním uvnitř kontejneru a vytvořeným `bridge` rozhraním. Pokud by se v systému nacházely další kontejnery, které mají síťování nastavené stejným způsobem, pak by byly tyto kontejnery propojeny mezi sebou právě pomocí zmíněného `bridge` rozhraní.

6. Přidělení IP adres pro `bridge` a `veth` uvnitř jmenného prostoru
   Následně se v kontejneru nakonfigurují síťová rozhraní, což zahrnuje přidělení IP adresy, masky podsítě a dalších parametrů.

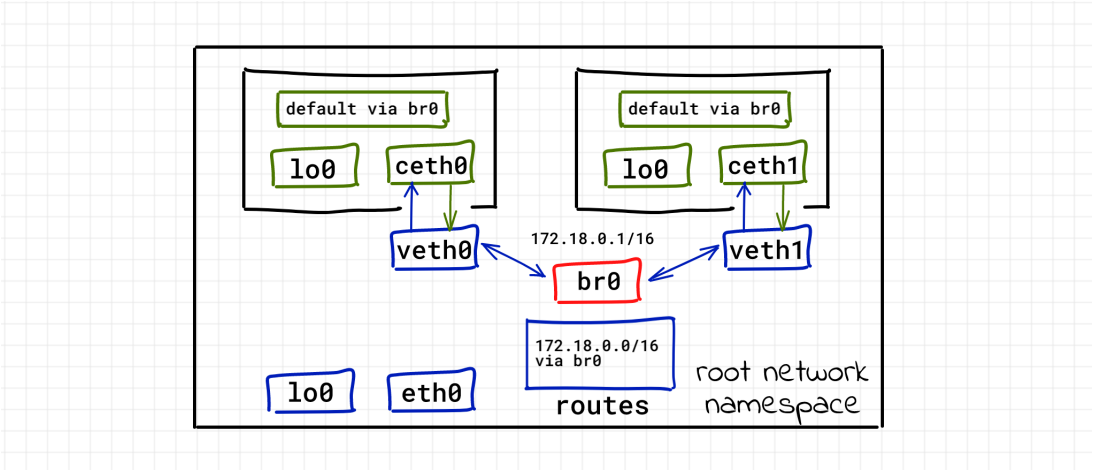
7. Zapnutí potřebných rozhraní

8. Nastavit NAT a IP Masquerade v hostujícím jmenném prostoru
   Proto aby bylo možné se z rozhraní `bridge` propojit na okolní síť, je na hostujícím systému nastavena NAT a IP Masquerade. Toto je prováděno pomocí prostředků kernelu. Konfigurace je prováděna pomocí netfilter za pomocí iptables.

   [11]
   Zmíněné kroky lze provést následujícími příkazy, které jsou uvedené ve výpisu 1.3.
   Výše popsaný postupu odpovídá implementace pro docker pomocí sítového ovladače *bridge*. Pro lepší pochopení výsledného stavu je k dispozici schéma 1.1 ilustrující ukázkovou výslednou konfiguraci.

Výpis kódu 1.3 Ruční nastavení síťování konteneru - docker bridge

```
# Create container with an empty network namespace
[_]$ docker run --rm -it --net none --name example travelping/nettools &

# Reference the namespace that docker has created
[_]$ mkdir -p /var/run/netns
[_]$ touch /var/run/netns/docker
[_]$ mount -o bind
  ↪ /proc/‘docker inspect -f '{{.State.Pid}}' example‘/ns/net
  ↪ /var/run/netns/docker

# Create a veth pair
[_]$ ip link add ceth0 type veth peer name veth0

# Move one interface to the docker network namespace
[_]$ ip link set netns docker dev veth0

# Create a bridge intreface
[_]$ ip link add br0 type bridge

# Connect second interface to the bridge
[_]$ ip link set veth0 master br0

# Add IPs to interaces
[_]$ ip netns exec docker ip addr add 192.168.1.100/24 dev ceth0
[_]$ ip addr add 192.168.1.50/24 dev br0

# Set default gateway for docekr to bridge
[_]$ ip netns exec docker ip route add default via 192.168.1.50

# Set all interface to up state
[_]$ ip netns exec docker ip link set up dev ceth0
[_]$ ip link set up dev br0
[_]$ ip link set up dev veth0
```



Obrázek 1.1 Schéma síťování pro ovladač *bridge* [12]

1.1.3 Container Network Interface

Kontejner má díky OCI jasně definovaný způsob jakým izolovat síťový provoz v kontejneru (pomocí jmenného prostoru). Žádná z OCI specifikací nedefinuje jakým způsobem umožnit kontejnerům komunikaci i mimo jeho jmenný síťový prostor (výše popsaný způsob je jen jeden z mnoha způsobů a implementací). Z důvodu, že konfigurace komunikace a nastavení komunikace mimo prostor kontejneru není součástí specifikace, různé implementace kontejnerů řeší problematiku různými způsoby. Jednou z těchto implementací je právě již zmíněný ovladač *bridge*.

Proto aby jednotlivé implementace měly jednotné rozhraní, byl vytvořen standard Container Network Interface (CNI). Díky tomuto je možné oddělit implementaci běhového prostředí kontejnerů a implementaci síťování. CNI specifikace určuje, že realizace CNI standardu mají být implementovány formou binární spustitelných souborů. Tyto spustitelné soubory se označují jako moduly (pluginy), případně CNI ovladače (drivers).

Ve specifikaci je CNI označován jako *„množina standardu, definující rozhraní pro síťování kontejnerů“*. [13] Specifikace definuje následující:

1. Formát souboru pro definici síťové konfigurace
   CNI definuje formát souboru pro konfiguraci dostupných CNI modulů implementující standard. Tento soubor slouží administrátorům pro nastavení a definování podporovaných modulů. Následně je konfigurační soubor používán aplikacemi spravující kontejnery (runtimes). Dle dodané konfigurace runtimes pracují s jednotlivými CNI moduly.
   Konfigurace je serializována pomocí JSON objektu, který obsahuje aktuální verzi CNI standardu, název a seznam podporovaných modulů. Jednotlivé položky seznamu modulů obsahují reference na dané moduly, standardem definované parametry a parametry specifické pro jednotlivé realizace modulů.

2. Protokol pro komunikaci mezi runtime aplikacemi a CNI moduly
   Implementace CNI specifikace přijímají parametry formou globálních proměnných a standardního vstupu. Každý CNI modul musí podporovat čtyři základní operace (*ADD*, *DEL*, *CHECK*, *VERSION*), které jsou ve standardu specifikovány.

3. Postup pro spouštění modulů
   Pomocí zmíněného způsobu je možné, aby runtimes využívaly služeb CNI modulů a tím mohly spravovat síťování uvnitř kontejnerů. Proto, aby komunikace fungovala definuje CNI řadu pravidel pro CNI moduly a runtimes, které tyto moduly spouští. Mimo pravidla, specifikace

rozděluje i zodpovědnosti pro implementace kontejnerových běhových prostředí a implementace CNI modulů.
Runtime je zodpovědný za vytváření nových síťových prostorů a korektní volání operací modulů. Runtime nesmí volat více paralelních operací nad jedním kontejnerem. Runtime musí volat CNI v případě mazání kontejneru, na kterém byl CNI použit. CNI moduly musí správně pracovat s více kontejnery, v případě potřeby je zodpovědný za správné zacházení se sdílenými prostředky.

4. Postup delegování práce modulů na jiné moduly
   Pro určité potřeby dává smysl umožnit CNI modulům volat jiné moduly. Pro tyto případy je ve specifikaci popsán způsob, který unožuje CNI modulům delegovat práci i na jiné implementace modulů. Tato funkcionalita umožňuje velikou flexibilitu při vykonávání operací. Příkladem využití je modul *multus*, který umožňuje volat více CNI pluginů, bez jakékoliv nutnosti modifikovat samotné runtimes.
   Tato část specifikace bude důležitá v následujících kapitolách. Konkrétně při využívání meta-pluginu Multus.

5. Návratové hodnoty CNI operací
   Poslední část specifikace definuje datové typy, které se souží jako návratové hodnoty při volání CNI modulů. Volané moduly mohou vracet jeden ze tří typů návratových dat (*Success*, *Error*, *Version*). Tato data obsahují bližší informace o výsledku volání.

Díky tomuto standardu je problematika síťování kontejnerů abstrahována na jednotlivé implementace standardu. Specifikace přesunula zodpovědnost konfigurace síťování z běhových prostředí kontejnerů na jednoduché modulární programy. Při využití CNI již běhové prostředí nemusí implementovat logiku konfigurace a mohou se spolehnout na již vytvořené obecné CNI moduly. Toto velmi zjednodušuje vývoj běhových prostředí kontejnerů a umožňuje snáší administraci. [13]

1.2 Kubernetes

Kontejnerizace velmi ovlivnila způsob doručování a nasazování aplikací. Myšlenka kontejnerů a OCI vytvořila jednoduchý prostředek komunikace mezi vývojáři aplikací a administrátory. Kontejnery velmi standardizovalo správu a nasazování aplikací na servery. Pro ulehčení práce s kontejnery dává smysl zajímat se o orchestraci těchto kontejnerů. Mezi dnes nejznámější orchestrátor kontejnerů patří právě Kubernetes. [1]

Kubernetes je orchestrační nástroj, který umožňuje snadnější práci a administraci aplikací, které jsou provozovány formou OCI kontejnerů.

Tento nástroj vznikl pro správu aplikací ve společnosti Google. Z počátku byl vyvíjen jako interní nástroj pro Google. V roce 2014 Google daroval tento systém nadaci *Cloud Native Conputing Foundation*. [14]. Následně se Kubernetes stal velmi populární open-source technologií. Dnes je nadšenci poznačován za operační systém cloudu. [14]

1.2.1 Základy systému Kubernetes

Kubernetes je orchestrační nástroj pro správu a nasazování aplikací v prostředí cloudu. Nástroj slouží ke správě aplikací formou kontejnerů (splňující OCI) na více serverech. Mezi hlavní funkce, které Kubernetes nabízí patří například automatické škálování aplikací, automatické opravy aplikací, zajišťování vysoké dostupnosti aplikaci, rozkládání zátěže mezi aplikace apod. [15]

Pro základní pochopení orchestrátoru je dobré vědět, jakým způsobem orchestrátor pracuje. Díky tomu je možné pochopit možnosti a limity této technologie.

Kubernetes typicky operuje na více serverech, které tvoří takzvaný klastr. Klastr nejčastěji označuje množinu počítačů, které spolu spolupracují. Pro účely této práce klastr označuje množinu

serverů, na kterých je Kubernetes provozován. Na těchto serverech pak Kubernets vytváří prostředí pro samotnou správu a nasazování aplikací. Toto prostředí bude označováno jako cloud.

Kubernetes se skládá z více mikroservisních modulů, které společně tvoří samotnou technologii. Tyto moduly je možné rozdělit do dvou základních kategorií. První kategorií jsou moduly tvořící takzvaný *control plane* (dříve také označovaný jako *master node*). Druhou kategorií jsou moduly, které tvoří *worker node* (česky pracovní uzel).

Množina modulů *control plane* je jádrem Kubernetes. Základním úkolem této množiny je správa pracovních uzlů v cloudu. [16] Tato množina obsahuje pět základních prvků.

- Etcd
  Etcd je distribuovaná NoSQL databáze, která uchovává data ve formě „klíč-hodnota“. Tato databáze je jediným prvkem Kubernetes, který uchovává perzistentní data o stavu cloudu. Případná ztráta dat v této databázi vede k nefunkčnosti celého systému.

- API Server
  API server je komponenta, která vystavuje rozhraní pro komunikaci s cloudem. API Server je označován jako front-end celého systému Kubernetes. Komunikace s vystavovaným API probíhá za pomocí protokolu HTTP a REST architektury.
  API Server je jediná komponenta, která přímo komunikuje s databází Etcd. Všechny zbylé komponenty interagují s databází prostřednictvím API Serveru.

- Plánovač
  Plánovač, jak už název napovídá, slouží k plánovaní úloh v cloudu. Jeho úkolem je reagovat na změny v Etcd databázi – konkrétně na požadavky pro běh kontejnerů. V případě potřeby má za úkol naplánovat (zvolit) vhodný server klastru, na kterém daný kontejner bude spuštěn.

- Správce kontrolerů
  Správce kontroleru spravuje programy, které se označují jako kontrolery. Tyto programy jsou zodpovědné za většinu práce systému Kubernetes. Příkladem zodpovědnosti kontroleru může být správa stavu serverů (v případě *node kontroleru*), nebo správa jednotlivých objektů Kubernetes.
  Více detailněji jsou kontrolery popsány ve třetí kapitole sekci 3.3, která se zabývá rozšiřováním funkcí systému Kubernetes.

- Správce cloudu
  Poslední komponentou je správce cloudu. Jedná se o komponentu, která provádí komunikaci s API poskytovatelů veřejných cloudu, jako jsou například AWS a Azure. Tento modul je zodpovědný za dynamickou správu serverů klastru v případě, že Kubernetes je provozován v prostředí veřejných cloudu.

Zmíněné moduly tvoří jádro systému Kubernetes. [16] Tyto moduly mohou běžet na jednom, nebo více serverech. Tyto moduly nejsou přímo součástí cloudu Kubernetes, pouze prostředí cloud tvoří.

Druhá kategorie modulů pracuje přímo s Pody (objekty abstrahující kontejnery). Kontejnery spravované těmito moduly představují samostatné aplikace, které uvnitř Kubernetes běží. Lze říct, že tvoří pracovní sílu klastru – z tohoto plyne název „pracovních uzly“.

Tato skupina se skládá z následujících modulů:

- Kubelet
  Kubelet je démon, který běží na každém serveru v klastru. Úkolem tohoto démona je komunikovat s kube-api serverem a spravovat kontejnery, které běží na daném serveru.

- Kube-proxy
  Kube-proxy je modul, který běží na každém uzlu klastru a nastavuje síťování pro daný uzel. Kubernetes dodává výchozí implementaci tohoto modulu, zároveň ale umožňuje nastavit systém tak, aby používal jinou implementaci této služby.

- Container runtime
  Poslední komponentou, která je přítomná na každém stroji je container runtime. Container runtime musí být nainstalovaný na každém uzlu a splňovat runtime specifikaci OCI. V současné době Kubernetes doporučuje jeden z následujících runtimes: *containerd*, *CRI-O*, *Docker Engine*, *Mirantis Container Runtime*. [17]

Výše popsané komponenty tvoří celý systém Kubenretes. [16], [15] Již bylo uvedeno, že Kubernetes slouží pro orchestraci kontejnerů v rámci klastru. Následující odstavec představí způsob, kterým jsou kontejnery v rámci Kubernetes nasazovány. Průběh procesu nasazení kontejneru dobře ukazuje jak Kubenretes pracuje jako celek.

Nasazení Podu (pro tento účel lze Pod chápat jako kontejner) začíná tím, že *kube-api* přijme požadavek na jeho vytvoření. V tuto chvíli *kube-api* zaznamená požadavek do *etcd* databáze. V databázi se pak bude nacházet definice Podu, který má být vytvořen. Tímto se změní očekávaný stav klastru, obsahující informaci o novém Podu. Nyní na řadu přichází *plánovač*. Ten získá informace o daném Podu a stavu jednotlivých serverů klastru. Ze skupiny serverů vybere ten, který nejvíce vyhovuje potřebám pro provoz Podu. Informaci o zvoleném serveru, na který má být Pod naplánován *plánovač* (prostřednictvím *kube-api*), zapíše do databáze *etcd*.

V tuto chvíli obsahuje databáze *etcd* definici požadovaného Podu společně s informací, kde má být Pod provozován. Nyní je příslušný *kubelet* vyzván, aby si aktualizoval informace o Podech, které má provozovat na daném serveru. S aktualizovanými informacemi pak vytvoří příslušný Pod a kontejnery. *Kubelet* sám objekty nevytváří, pouze volá příslušnou implementaci běhového prostředí kontejnerů (OCI), který kontejnery tvoří. Veškeré provedené změny změny a akce jsou v průběhu zapsány opět do *etcd* databáze. [15], [14]

Díky výše popsaným mechanismům a modulům Kubernetes, tvoří Kubernetes funkční orchestrátor kontejnerů.

1.2.2 Pod

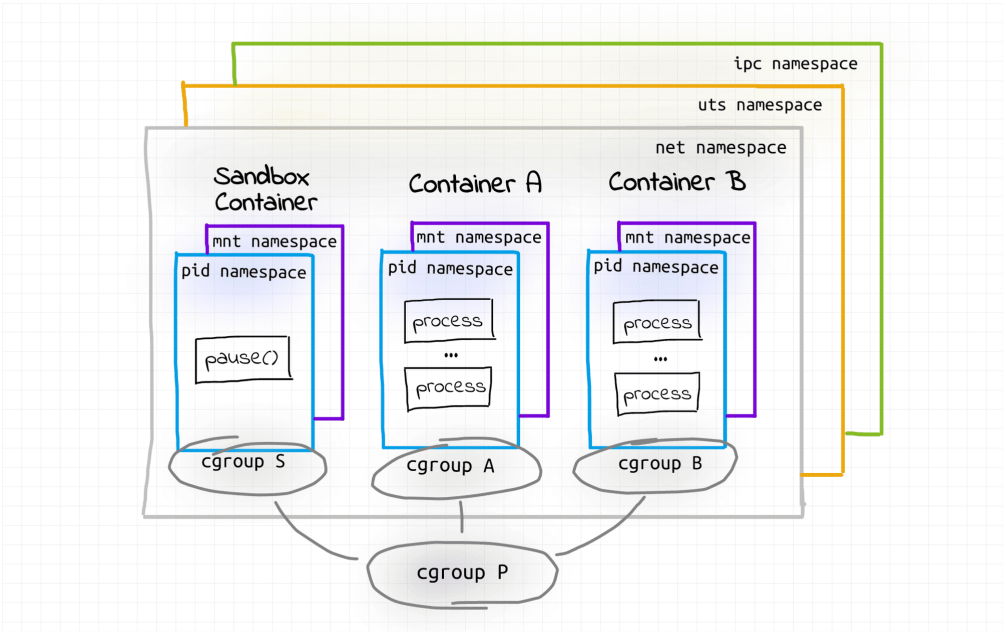
Pod je základním objektem Kubernetes. Jedná se o abstrakci kontejneru, se kterou Kubernetes pracuje. Pod je často chápán jako kontejner, který obsahuje další kontejnery. Pro jednoduché porozumění problematiky je toto vysvětlení dostačující i přesto, že není stoprocentně pravdivé.

Stejně jako kontejnery jsou i Pody implementovány pomocí jmenných prostorů jádra. Pod je prostředí formované jemnými prostory, ve kterém je možné spouštět kontejnery. Tímto způsobem Kubernetes zaobaluje kontejnery. Pod je atomický objekt, který lze v Kubernetes plánovat.

Při vzniku Podu se vytváří celkem tři jmenné prostory, které jsou sdílené se všemi kontejnery uvnitř Podu. Konkrétně se jedná o jmenný prostor mezi-procesové komunikace (IPC namespace), jmenný prostor názvu, domény systému (UTS namespace) a síťový jmenný prostor (net namespace). Všechny zmíněné prostory jsou sdílené napříč kontejnery v Podu. Jednotlivé kontejnery jsou izolované pouze pomocí jmenného prostoru proces ID (pid namespace) a jmenného prostoru přístupových bodů (mnt namespace). Izolace procesů v Podu je výborně ilustrována na schématu 1.2 od Ivan Velichko.

Mezi hlavní vlastnosti Podu patří nestálost (často popisováno anglickým slovem emhemeral) a neměnnost (anglicky immutability). Nestálost v kontextu Podu značí nestálost při zániku Podu (v anglické literatuře se Pod občas označuje za smrtelný – mortable). Při zániku Podu se veškeré informace a data spjatá s daným Podem ztratí. Proto není dobré se jakýmkoliv způsobem spoléhat na data uvnitř Podu. Druhou klíčovou vlastností je zmíněná neměnnost. Tato vlastnost znamená, že již běžící objekt Pod nelze nijak měnit. V případě potřeby změny je nutné vytvořit Pod nový a starý smazat. [14]

Z výše popsaných vlastností lze odvodit i vlastnosti další. Zde jsou příklady těch, které jsou důležité pro účely práce: Každý Pod má vlastní nepředvídatelnou IP adresu, pokud Pod zemře a namísto něj vznikne nový, nelze nová IP adresa předvídat. V případě zániku Podu se ztrácí veškerá



Obrázek 1.2 Schéma jmenných prostorů v Podu [18]

data v něm uložená. Při vzniku Podu nelze předpovědět na jaký server bude Pod naplánován (pokud není explicitně uvedeno).

1.2.3 Deployment

Deployment je další objekt Kubernetes, který slouží převážně pro nasazování takzvaných bezstavových aplikací pomocí Podů. Bezstavové aplikace jsou aplikace, které nemají žádný vnitřní stav a neukládají žádná perzistentní data. Často se jedná o různé jednoduché webové API, webové front-endy apod. Typické pro tyto aplikace je, že jsou nezávislé na předchozích a budoucích požadavcích. Právě pro potřeby provozování těchto aplikací je určený objekt Deployment.

Deployment je objekt, který zaobaluje již popsaný objekt Pod. Tento objekt se skládá celkem ze dvou částí. První je specifikace šablony pro Pod. Tato šablona slouží pro popis Podu, který náleží danému Deploymentu. Druhá část Deploymentu specifikuje, jakým způsobem se má s Pody zacházet. Tento popis obsahuje informaci o počtu Podů, které mají v jeden okamžik běžet, jakým způsobem provádět aktualizace Podů atd.

Deplymenty jsou spravovány vlastním kontrolerem, který se stará o veškeré instance objektu v klastru. Kontroluje, zda vše běží tak jak má a zda jsou dodržena veškerá pravidla nastavená daným objektem. [14]

Deplyment umožňuje kontrolu Podů pro nestavové aplikace.

1.3 Standardní síťování v Kubernetes

Síťování je velmi důležitou částí orchestrátoru Kubernetes. Orchestrátor poskytuje celkem čtyř řešení pro síťování uvnitř clusteru. V následující části se zaměříme primárně na komunikaci, která pobíhá na transportní vrstvě, nebo vyšší – dle modelu ISO/OSI. Předpokládejme, že pro síťovou vrstvu používáme protokol *IP* verze 4.

1.3.1 Kontejner s kontejnerem uvnitř Podu

Díky tomu, že jednotlivé kontejnery v jednom Podu sdílí stejný síťový jmenný prostor, mají všechny kontejnery přístup ke stejným síťovým systémovým prostředkům. Všechny kontejnery v Podu mají sdílené síťové zařízení, stejnou IP adresu a další prostředky. Zároveň platí, že každý Pod obsahuje síťové rozhraní typu *loopback*. Komunikace mezi kontejnery tedy probíhá pomocí *loopback* rozhraní. To v praxi znamená, že kontejnery v Podu posílají data na rozhraní *loopback* a ostatní kontejnery na *loopback* adrese naslouchají. Tím lze docílit komunikace mezi kontejnery v rámci jednoho Podu.

1.3.2 Komunikace Pod s Podem

Při komunikaci Podu s Podem nastává problém, jelikož samotný standard OCI přímo řešení nenabízí. Při komunikaci mezi Pody, je často zapotřebí zprostředkovat komunikaci mezi více než jedním uzlem klastru. Tento typ komunikace musí vyřešit problémy jako přidělování IP adres Podům, sdílení dat mezi více uzly, kontrolu kolizí portu, routování mezi uzly apod.

Řešení zmíněných problémů je netriviální a velmi těžko obecně implementovatelné tak, aby vyhovovalo každému použití Kubernetes. Z tohoto důvodu není řešení této komunikace přímo součástí Kubernetes. Namísto toho se Kubernetes odvolává na zmíněný standard CNI. Proto, aby byla komunikace mezi Pody umožněna, musí být po instalaci a inicializaci systému Kubernetes nainstalován modul (CNI plugin), který požadovanou komunikaci dokáže zprostředkovat.

První požadavek na funkční CNI modul pro potřeby Kubernetes je zajištění nastavení síťových rozhraní v Podu. I přesto, že specifikace je určena převážně pro práci s kontejnery, je možné implemntace tohoto standardu pro účely konfigurace Podu použít. Stačí nastavit rozhraní jednomu z kontejnerů v Podu a díky sdílení jmenného prostoru nastavení se projeví v celém Podu. Tento požadavek je jednoduché splnit, jelikož se jedná o povinné vlastnosti určené ve specifikaci.

Druhým požadavkem, který je kladen systémem Kubernetes na vývojáře CNI modulů, je umožnit přímou komunikaci mezi Pody v celém klastru. Tento požadavek je netriviální problém. Kubenretes nijak více tento požadavek nespecifikuje. Absence této specifikace klade na implementace CNI modulů vysoké nároky. Většina řešení tohoto požadavku se dá rozdělit do čtyř základních skupin (*full mesh of static routes*, *orchestrating the underlay*, *encapsulating in the overlay*, *cloud API*). [19], [20]

V případě, že se všechny uzly nachází na stejné linkové ISO/OSI vrstvě, je možné routování dosáhnout pomocí statické konfigurace routovacích pravidel (*full mesh of static routes*).

Druhým způsobem je využití směrovacích protokolů (například BGP), případně dynamickým nastavováním routovacích pravidel. Tento způsob umožňuje práci s uzly, které se nenacházejí ve stejné linkové vrstvě. Tento způsob se nazývá *orchestrating the underlay* a je implementován například modulem *Calico*.

Velmi často využívaným způsobem je využití VXLAN, díky které lze zaobalit síťování. Tato možnost je označována jako *encapsulating in the overlay*. Ukázkovou implementací je modul *Flannel*.

Posledním způsobem jsou často proprietární řešení, která umí komunikovat přímo s prvky zajištující síťování. Tento způsob je typický při používání služeb nabízených veřejnými cloudovými službami. Tento způsob se označuje jako *cloud API*.

Pro správné fungování je nutné, aby CNI moduly správně přidělovaly IP adresy jednotlivým Podům. Ve většině případech jsou Podům přidělovány adresy z rozsahu, který náleží danému uzlu klastru. Tento rozsah je uložen v proměnné *podCIDR*, která se nachází ve specifikaci každého uzlu. Rozsah je odvozen z proměnné *clusterCIDR*. Tato proměnná uchovává rozsah IP adres, které mají být přidělovány Podům v celém klastru. *PodCIDR* jsou tedy jednotlivé podsítě *clusterCIDR*.

Užití těchto definovaných rozsahů je pouze doporučený postup pro implementaci CNI a jejich využití není nijak vynucováno. [20]

Tím, že Kubernetes deleguje síťování na CNI pluginy, které mají relativně velkou volnost implementace, je možné síťování přizpůsobit přímo na míru konkrétnímu použití systému Kubernetes.

Pro přiblížení zmíněných informací může posloužit následující ukázka 1.4 nastavení klastru. Tento klastr se skládá celkem ze tří serverů pojmenovaných *kmaster*, *kworker1* a *kedge1*. [21]

Výpis kódu 1.4 Konfigurace nastavení síťování v Kubernetes [22]

```
# Get list of nodes and their podCIDR range
[_]$ kubectl get nodes -o json | jq
  ↪ '[.items[]|{node:.metadata.name,podCIDR:.spec.podCIDR}]' | yq -P
>>> - node: kedge1
>>>   podCIDR: 10.244.2.0/24
>>> - node: kmaster
>>>   podCIDR: 10.244.0.0/24
>>> - node: kworker1
>>>   podCIDR: 10.244.1.0/24

# Get global cluster-cidr for cluster
[_]$ kubectl cluster-info dump | grep -m 1 -oE 'cluster-cidr=[^"]+'
>>> cluster-cidr=10.244.0.0/16

# Get list of pods and their internal IP addresses
[_]$ kubectl get pods -o json | jq '[.items[]␣|␣
  ↪ {name:.metadata.name,ip:.status.hostIP,node:.spec.nodeName}]' |
  ↪ yq -P
>>> - name: debug-edge-7b9bff85d4-j9t26
>>>   ip: 172.16.16.111
>>>   node: kedge1
>>> - name: http-server-67b6fc474b-dhzll
>>>   ip: 172.16.16.101
>>>   node: kworker1
>>> - name: kube-controller-manager-kmaster
>>>   ip: 172.16.16.100
>>>   node: kmaster
```

Dokumentace Kubernetes nabízí seznam doporučených CNI modulů. Tento seznam lze nalézt na stránce Kubernetes (kubernetes.io).

1.3.3 Komunikace pomocí Service

Výše popsaný způsob komunikace je zcela funkční, ale značně limitující. Přímá komunikace mezi Pody je závislá na konfiguraci jednotlivých Podů. Konkrétně na jejich IP adresách, které jsou Podům přiděleny. V případě, že jeden z Podů chce komunikovat s Podem jiným musí znát jeho IP adresu a na tu komunikaci adresovat. Bohužel na stálost a neměnnost IP adres u Podů není spoleh. Z tohoto důvodu je potřeba umožnit jednotlivým aplikacím v cloudu dynamicky objevovat IP adresy služeb, které chce daná aplikace adresovat. Této funkcionalitě se v anglickém jazyce říká *service discovry*, do českého jazyka by se dalo přeložit jako *objevování služeb*.

Service discovry je proces dynamického, objevování IP adres, případně routovacích pravidel v síti. Jednou z nejznámějších služeb poskytujících *Service discovry* je DNS. Pro dynamické objevování služeb v zásadě existují dvě řešení. Aplikace (často označované jako klientské) mohou

service discovry provozovat samy. Druhou možností je delegovat problém na jiné služby, jako například DNS, a jiné. Oba tyto způsoby je možné aplikovat v prostředí Kubenretes.

Zmíněné první řešení v Kubernetes by znamenalo, že aplikace běžící v cloudu by komunikovaly přímo s *kube-api*. Takto by bylo možné, aby si aplikace samy zjišťovaly potřebné informace o okolních službách v klastru, které by následně mohly adresovat. Tento způsob má určité výhody, ale několik zásadních nevýhod. Mezi ty nejzásadnější nevýhody patří: související bezpečnostní problémy a zvýšené nároky kladené na vývojáře aplikací. Zároveň tento způsob jde proti GRASP[4] návrhovému vzoru. Také není použitelný pro aplikace, které nejsou přímo připraveny pro běh v prostředí Kubernetes. Kvůli zmíněným nevýhodám není tento způsob řešení *service discovery* obecně preferovaný.

Druhý způsob nabízí možnost delegace problému na jinou službu poskytující *service discovery*. Jelikož problém *service discovry* je již dobře známý a zdokumentovaný, existují pro něj standardní řešení. Kubernetes nabízí řešení, které je velmi podobné takzvanému *reverse proxy*.

Reverse proxy je služba, která funguje jako prostředník mezi klientem a cílovým serverem. Služba abstrahuje spojení ke koncovému serveru. Ve většině případů je *reverse proxy* implementována jako aplikace, která stojí mezi klientskou aplikací a nabízenou službou. Tato aplikace pak následně sama vystavuje body pro připojení a zprostředkovává doručení komunikace koncovým službám. Tím, že klient se nepřipojuje přímo na server, ale na určitého prostředníka, nemusí provádět samotnou *service discovery*. *Service discovery* je prováděna pouze proxy serverem. Tímto lze problém *service discovery* delegovat na jinou službu.

V praxi to znamená, že se klienti přímo nepřipojují na server, ale na *reverse proxy*, která požadavky přepošle na koncovou aplikaci. Proto, aby toto řešení fungovalo, musí být adresa *reverse proxy* známá a neměnná, *reverse proxy* musí být spolehlivá a dostupná v nejvyšší možné míře. Zároveň musí zajistit správné přeposílání požadavků klientů na poskytované služby, což v důsledku znamená, že musí znát koncové aplikace. [15]

Variací *reverse proxy* ve světě Kubernetes je právě objekt *Service*. Oficiální dokumentace Kubernetes uvádí, že *„Service je metoda vystavování síťových aplikací, které běží v jednom nebo více Podech.“* [24]. Stejným způsobem by se dalo popsat výše zmíněné fungování *reverse proxy*. Metoda Service má následující vlastnosti: je adresovatelná v celém cloudu pomocí IP adresy i DNS jména, zprostředkovává komunikaci s Pody, je spolehlivá a perzistentní (její IP adresa je neměnná).

Pro jednoduché pochopení fungování Service je dobré se seznámit s definicí Service, kterou poskytuje *kube-api*. Ukázku této definice lze vidět níže ve výpisu 1.5.

Výpis kódu 1.5 Ukázka definice objektu Service

```
---
kind: Service
apiVersion: v1
metadata:
  name:  service-name
spec:
  selector:
    app: pods-label # label of pod - for identification
  type: ClusterIP   # type of service (LoadBalancer|ClusterIP|NodePort)
  ports:
  - port:  80               # exposed port by servise
    targetPort:  8080       # exposed port on pod
```

[4] Grasp označuje General Responsibility Assignment Software Patterns je návrhový vzor, který prosazuje nízkou provázanost a vysokou soudržnost. [23]

Prvním důležitým parametrem je název, který slouží jako DNS klíč při adresování dané Service. O správný překlad adres se postará interní Kubernetes DNS server. Druhým velmi důležitým atributem je `selector`. Selector slouží k propojení dané Service s Pody, které daná Service abstrahuje (jedná se o Pody, na které bude objekt komunikaci směřovat). V příkladu 1.5 budou adresovány všechny Pody, které obsahují označení `app: pods-label`. Posledním důležitým parametrem je pole portů, které určují jakým způsobem má být případná komunikace na jednotlivé Pody přeposílána. V uvedeném příkladu všechny požadavky, které přijdou na IP adresu objektu *service-name* a port 80, budou přeposlány na příslušné Pody na port 8080. Adresování jednotlivých Podů je automaticky prováděno objektem Service.

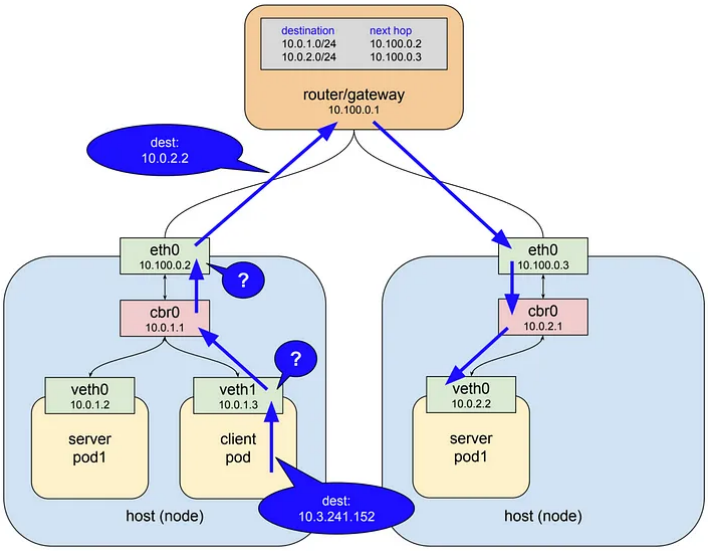
Díky těmto parametrům je možné popsat způsob, jak pomocí Service komunikovat s koncovými aplikacemi běží v koncových Podech. Jediné, co v definici chybí je IP adresa, dané Service. Tento parametr lze v definici vynutit pomocí `clusterIP`. Pokud daná IP adresa není specifikovaná, pak Kubernetes vybere IP adresu sám. Tato IP adresa je po dobu existence Service neměnná a zaznamenaná v interním DNS serveru. [15]

Příklad komunikace mezi Pody pomocí Service může vypadat následujícím způsobem. Mějme klientskou aplikaci běžící v Podu `C` (client), která by chtěla komunikovat se službou běžící v Podu `A` (addressee). Zároveň v Kubernetes existuje objekt Service `S` (service), který přeposílá požadavky na zmíněný Pod `A`. Pod `C` chce vyslat požadavek na službu běžící, která je dostupná pomocí `S`. Proto zašle požadavek přímo na `S`, na daný port, který je předem dohodnutý. To je spolehlivé, jelikož `S` má neměnnou IP adresu, jejíž hodnota je získatelná z interního DNS Kubernetesu.

V tuto chvíli opouští paket Pod s cílovou adresou `S`. Ještě než paket zcela opustí daný uzel klastru, je díky Kubernetes pozměněna cílová adresa tak, aby směřovala přímo na daný Pod `A`, kde běží potřebná služba. Tuto část lze chápat jako službu poskytující *reverse proxy*.

Při cestě paketu opačným směrem, který obsahuje odpověď pro Pod `C`, je adresa opět pozměněna, aby Pod `C` nepoznal, že k nějaké změně vůbec došlo. Za povšimnutí stojí, že klientský Pod `C` nemusí nic o existenci Podu `A` vědět, celá komunikace je abstrahována prostřednictvím Service.

Způsob komunikace pomocí Service je ilustrována na schématu 1.3.



Obrázek 1.3 Průběh komunikaci pomocí Service [25]

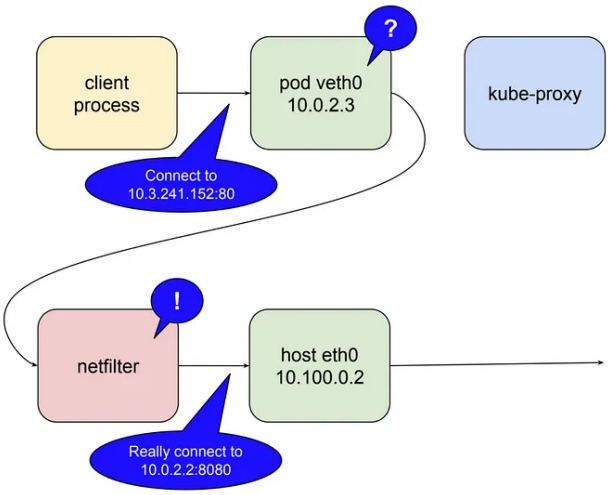
Problematika Service je složitá. Pro detailnější pochopení Service doporučuji nastudovat oficiální dokumentaci Kubenretes a web The Kubernetes Networking Guide. Níže jsou uvedeny

dva fakty, které je dobré pro účel práce znát.

Překlad adres na jednotlivých uzlech klastru zajišťuje komponenta *kube-proxy* často pomocí *netfilter* pravidel, které jsou spravovány pomocí *IPtables*. Viz obrázek 1.4.

Implementace překladu adres se může měnit dle použitého CNI.

Adresovatelnost objektu Service zajišťuje objekt Endpoint, respektive objekt EndpointSlice. Tento objekt je úzce spojen s objektem Service. Rozsah přidělovaných IP adres pro Services je uchován v proměnné `service_cluster_ip_range`, ve výchozí konfiguraci se jedná o rozsah `10.43.0.0/16`. [26]



Obrázek 1.4 Překlad adres pomocí netfilter při použití Service[27]

Existují celkem tři základní módy, ve kterých Service může fungovat. Výše popsaný příklad použití Service je pouze jedním z nich. Těmito módy jsou *ClusterIP*, *NodePort*, *LoadBalancer*. [14]

1.3.3.1 ClusterIP

ClusterIP je nejzákladnější způsob fungování Service. V tomto módu plní všechny výše popsané služby pro překlad adres. Tento mód se používá pro interní komunikaci Podů v cloudu. Aplikace, vystavené objektem Service *ClusterIP*, jsou dostupné pouze v rámci interní sítě cloudu. [24] Výše uvedený příklad odpovídá tomuto použití.

1.3.3.2 NodePort

Node port je variace předešlého *ClusterIP* módu. Tento mód obsahuje překlad adress v rámci cloudu, a zároveň vystavuje službu i mimo interní síť cloudu. Daná služba (v tomto módu) bude dostupná pod definovaným portem na všech IP adresách pracovních uzlů klastru. Za tuto funkcionalitu je také zodpovědná komponenta *kube-proxy*. Při příchodu paketu na adresu serveru je adresa přeložena na adresu Podu, stejně jako v případě komunikace v rámci interní sítě. [24]

1.3.3.3 LoadBalancer

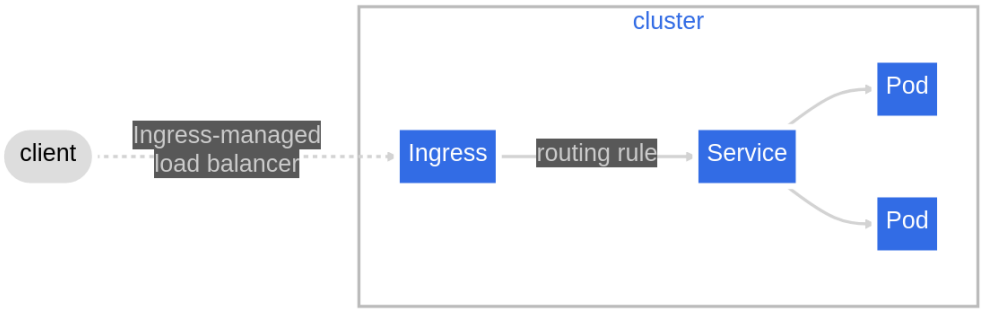
LoadBalancer je nejpoužívanějším módem Service. [14] Tento mód je do jisté míry podobný variantě *NodePort*. Také je variací *ClusterIP* zároveň umožňuje adresovat interní objekty Service z externí sítě. Pro vystavení Service do okolní sítě vytváří unikátní IP adresu, která je adresovatelná z vnějšku sítě. Proto, aby tato funkcionalita fungovala, musí být zajištěna funkcionalita load-balancingu. V případě použití poskytovatelů cloudových služeb, je tato funkcionalita zajištěna

pokytovately. V případě on-premise řešení lze využít softwarových implementaci, jako například *MetalLB*. Pro robustnější řešení lze zajistit hardwarovou podporu load-balancingu. [24]

1.3.4 Ingress

Jedním z velmi často využívaným objektem Kubenretes z kategorie síťování je i Ingress. Ingress je relativně nový objekt v Kubernetes. Do standardního API byl zařazen ke konci roku 2020 [28].

Ingress je v oficiální dokumentaci uveden jako *„API objekt, který spravuje externí přístup k Services uvnitř cloudu, typicky pomocí HTTP. Ingress může poskytovat služby load-balancingu, SSL a routování na základě doménových jmen.“* [29] Jedná se tedy o objekt, který vystavuje přístup k objektům typu Servcice uvnitř cloudu. Ingress si lze představit jako vstupní bránu celého cloudu, která umožňuje dostupnost některých aplikací v interní síti Kubernetes. Velmi dobře je toto znázorněno na schématu 1.5 níže, které znázorňuje klienta přistupujícího pomocí Ingress a Service až k jednotlivým Podum.



Obrázek 1.5 Ukázka funkce objektu Ingress [30]

Mohlo by se zdát, že *Ingress* je duplicitním řešením pro již existující objekt *Service* v režimu *NodePort* nebo *LoadBalancer*. Je pravda, že podobné funkcionality nabízené objektem Ingress lze docílit i za použití Service. Ingress ale navíc doplňuje a rozšiřuje funkce objektu Service, zároveň řeší některé z problémů, které mohou při použití Service nastat. Ingress je často preferovaným způsobem pro vystavování aplikací, namísto Service a to z dvou hlavních důvodů.

Prvním důvodem je kontrola routování. V případě použití Services pro účely vystavení služeb běžících v sítí Kubernetes lze routování konfigurovat pouze na síťové respektive transportní vrstvě dle ISO/OSI modelu. Service v módu nodePort nám umožňují nastavovat routování pouze za pomocí portů a typu transportního protokolu (TCP a UDP). V případě Service typu *LoadBalancer* lze routování kontrolovat navíc pomocí IP adres. Uvedené způsoby kontroly routování mohou být značně omezující pro různé typy služeb. V některých případech dává smysl kontrolovat routování pomocí protokolů výších vrstev. Právě možnost pokročilé kontroly routování přináší objekt Ingress.

Druhým důvodem, proč je Ingress hojně využívaný, je jeho vlastnost umožňující redukovat finanční nároky na vystavování aplikací, při použití poskytovatelů veřejných cloudových služeb, jako jsou Azure, AWS, Google apod. Velmi často se stává, že poskytovatele cloudových služeb si účtují poplatky za každou existující veřejnou IP adresu. Tyto poplatky se pak mohou účtovat za každý objekt Service typu *LoadBalaner*. V případě použití objektu Ingress se zodpovědnost za load-balancing přenese dovnitř samotného klastru. Díky tomu lze znatelně omezit výdaje při použití veřejných poskytovatelů cloudů.

1.3.4.1 Implementace Ingress

Kubernetes přímo neposkytuje implementaci samotného objektu Ingress. Vývojáři systému poskytují pouze API, pro daný objekt. Jednotlivá implementace tohoto objektu se může lišit v závislosti na použitém kontroleru. V případě jednotlivých implementací se bavíme o implementaci samotného Ingress kontroleru. Kontroler je pak modul, který rozumí definovanému API, dle kterého plní potřebné funkce. Jednotlivé implementace kontrolerů se mohou výrazně lišit, dle dané infrastruktury, prostředí a potřeb použití.

Jak již bylo zmíněno, Ingress dlouhou dobu nebyl součástí Kubernetes. To vedlo ke vzniku mnoha řešeních třetích stran, implementujících funkci objektu. [31] Z tohoto důvodu je samotné Ingress API navrženo velmi volným způsobem, aby co nejméně limitovalo již existující řešení.

Příklady kontrolerů implementující funkci Ingress jsou: *AWS Load Balancer Controller*, *GoogleCloud LoadBalancer controller*, *Ingress NGINX Controller*. [29]

1.3.4.2 Routování pomocí aplikační vrstvy - Ingress

Ingress umožňuje nastavovat routování na základě nejvyšší vrstvy abstrakce ISO/OSI modelu, konkrétně pomocí HTTP protokolu.

Pro definici routování Ingres používá list pravidel (rules). Tyto pravidla konfigurují samotný Ingress kontroler, prvky listu definují pravidla pro routování. Příklad pravidel lze vidět ve výpisu kódu 1.6.

Výpis kódu 1.6 Ukázka routovacích prvidel objektu Ingress

```
---
rules:
- host: back-end.example.com
  http:
    paths:
    - path: /v1
      pathType: Prefix
      backend:
        service:
          name: be-v1-service
          port:
            number: 80
```

Prvním způsobem, kterým lze kontrolovat routování je pomocí DNS jména, které je použito pro samotný přístup k Services v cloudu. DNS jméno lze specifikovat dle standardu *RFC 3986* jako část URI označovaná *host*. Momentálně Ingress nepodporuje specifikování portu pomocí oddělovače „:“. Možnost specifikace pomocí portu je diskutována a je možné, že se v dalších verzích systému Kubernetes vyskytne. Jediný podporovaný port v aktuální verzi Kubernetes, je 80 respektive 443 pro HTTPS. Pomocí nastavení routování na základě DNS jména lze dobře oddělovat jednotlivé objekty v cloudu, ke kterým má být přistupováno. Příkladem může být odlišný objekt Service pro back-end a front-end aplikace, kdy obě Services jsou dostupné ze stejné IP adresy, ale pod jiným doménovým jménem *front-end.example.com* respektive *back-end.example.com*.

Druhý způsob routování, které je možné specifikovat pomocí Ingress je za pomocí cest. V kontextu Ingress cesta označuje část URI dle *RFC 3986*, která je označovaná jako „path“. Díky tomuto lze oddělit například verze aplikací. Takové použití by mohlo vypadat tak, že *back-end.example.com/v1* bude odkazovat na verzi v1 aplikace a *back-end.example.com/v2* bude odkazovat na verzi *v2*. Při konfiguraci lze cesty lze specifikovat, zda danému pravidlu mají odpovídat všechny cesty s daným prefixem, nebo pouze cesty, které přesně odpovídají danému vzoru. [32]

Oba výše popsané způsoby routování lze kombinovat.

Toto jsou základní dva způsoby routování, které Ingress nabízí. Ve chvíli, kdy Ingerss přijme požadavek, který uspokojí vydefinované pravidlo, objekt přepošle požadavek dále do cloudu a plní tak funkci *reverse proxy*. Ingress je určen převážně k tomu, aby požadavky přeposílal na objekty Service. Tyto požadavky jsou pak díky Service doručeny na potřebné Pody a aplikace běžící uvnitř systému.

1.3.4.3 Šifrování pomocí Ingress

Doposud popsaná funkce routování je závislá na datech HTTP protokolu. Pokud komunikace není nijak zabezpečena, pak jsou tato data volně dostupná a Ingress s nimi může pracovat. V případě šifrované komunikace mezi klientem a aplikacemi v Kubernetes, nemůže Ingress parametrům v HTTP rozumět. Data obsažená v HTTP protokolu jsou šifrovaná a Ingress na základě těchto dat nemůže routování poskytovat. Proto, aby Ingress podporoval routování i šifrovaných dat, nabízí objekt možnost zprostředkování TLS šifrování pro komunikaci mezi klienty a samotným Ingress. Tímto je šifrování delegováno na objekt Ingress a umožněno routování i šifrovaných dat. Delegace šifrování přináší dvě výhody oproti šifrování až na straně aplikace v cloudu. [14]

První výhodou je, že Ingress má přístup k datům. Jelikož komunikaci sám šifruje, pak může nahlížet i do dat komunikace a díky tomu poskytovat zmíněné routování dle parametrů HTTP protokolu.

Druhá výhoda delegace šifrování na objekt Ingress souvisí s bezpečností systému. Ingress poskytuje jednoduché řešení pro zabezpečení veškeré komunikace, která je vedena přes daný objekt. Díky tomu není zodpovědnost zabezpečení na vývojářích samotných aplikacích, ale na administrátorech daného klastru. Implementace šifrování je obsažena v Ingress kontroleru, což je preferovanější řešení, než implementace logiky přímo v samotných aplikacích. [33]

1.3.5 Egress

Výše popsaný objekt Ingress je určen pouze pro jednosměrnou komunikaci směrem do prostředí cloudu. Při komunikaci směrem z klastru ven se často hovoří o takzvaném egress. Jak již název napovídá, egress označuje opačnou funkcionalitu a myšlenku oproti zmíněnému objektu Ingress. I přesto, že výraz egress je několikrát zmíněn v oficiální dokumentaci Kubernetes, tak samotný Kubernetes funkcionalitu egressu nepodporuje. Komunikace z klastru ven není nijak oficiálně zdokumentovaná.

O komunikaci z Podů klastru do veřejného internetu se primárně starají CNI pluginy. Výchozí nastavení většiny známých CNI pluginů, je provádět překlad adres přímo na výpočetních uzlech klastru, kde daný Pod běží. Tento způsob dává dobrý smysl, jelikož nevyžaduje žádnou konfiguraci ani složitou implementaci. Zároveň, díky plánovači, určitým způsobem balancuje síťovou zátěž mezi jednotlivé uzly klastru. Může nastat chvíle, kdy je třeba vést komunikaci určitým směrem, například přes jeden z dostupných uzlů klastru. Pro tyto účely je možné použít pokročilé implementace CNI modulů, které tuto funkcionalitu nabízejí. [34] Dalším možným řešením je použití nástroje třetích stran pro správu Service, jako je například Consul.

Je možné, že v budoucnu bude tento problém řešen přímo pomocí Kubernetes. Momentálně dokumentace uvádí že neexistuje oficiální řešení implementující funkci egress. [35]

Kapitola 2

Definice problému a možná řešení

Tato kapitola je zaměřena na hledání problému popsaného v úvodní části. Cílem této kapitoly je představit možné způsoby řešení problému a poukázat na výhody respektive nevýhody daného řešení.
Pro porozumění této kapitoly je dobré být seznámen se základním fungování orchestrátoru Kubernetes a principy síťování, které tento orchestrátor nabízí.

2.1 Definice problém

V úvodu a zadání této práce je nastíněný problém komunikace mezi vnitřní sítí klastru a privátními sítěmi, které přiléhají alespoň k jednoho z uzlů klastru.

V rámci této kapitoly bude síťová komunikace označkovat komunikaci probíhající na síťové vrstvě referenčního ISO/OSI modelu – konkrétně pomocí protokolu IP. Pro účely této práce se omezíme na protokoly TCP, UDP a HTTP.

Kubernetes poskytuje řešení pro několik typů síťové komunikace. Nejzákladnějším z nich je komunikace mezi Pody, které jsou součástí jednoho cloudu. Pro tento druh datového přenosu nepřináší Kubernetes žádnou formu omezení. Přenos je umožněn plně a všemi směry.

Komunikace směrem do klastru z okolní sítě také není nijak zvláštně omezena. Pro tuto komunikaci lze využít některé ze standardních objektů, které Kubernetes nabízí (Ingress, Service). Tento druh přenosu vyžaduje konfiguraci, ale je umožněn bez jakéhokoliv omezení.

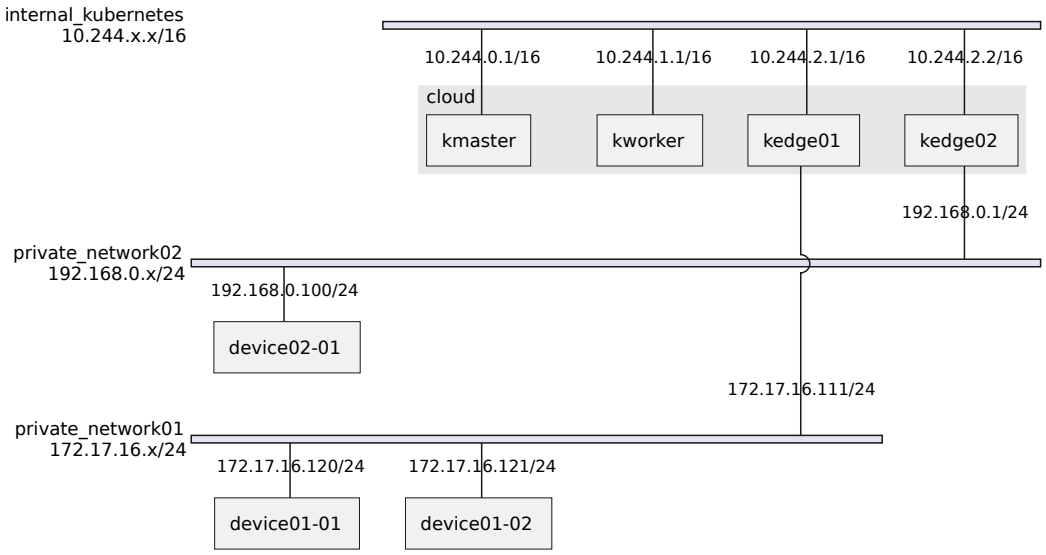
Posledním druhem komunikace je spojení klastru s okolní sítí, ve směru komunikace do okolní sítě. Toto spojení je bezproblémové v případě, že aplikace v klastru adresuje objekty, které jsou veřejně dostupné[1] v okolní síti. Příkladem takového spojení je požadavek na veřejnou DNS službu. V případě, že některá z aplikací běžících v Kubernetes se obrátí například na server `8.8.8.8`, pak komunikace proběhne bez potíží. Požadavek je na jednom z uzlů klastru pozměněn tak, aby byl validním v okolní síti mimo klastr (typicky je pozměněna hlavička IP protokolu - pomocí SNAT) a poté je požadavek vyslán do okolní sítě.

Problém nastává, pokud je vyžadována komunikace s okolní *privátní* sítí. Privátní sítí myslíme síťový segment, který není dostupný ve veřejném internetu a zároveň není dostupný ze všech uzlů klastru. Tato situace je ilustrována na schématu 2.1 níže.

Ve schématu jsou uvedeny celkem tři síťové segmenty. Síť *internal_kubernete* reprezentuje interní síť klastru. Rozsah této podsítě `10.244.x.x/16` je definovaný proměnnou `PodCIDR`. Klastr se skládá z čtveřice výpočetních uzlů *kmaster*, *kworker*, *kedge01*, *kedge02*.

Sítě *private_network01* a *private_network02* označují přilehlé privátní síťové segmenty. Omezme

[1] Veřejně dostupným objektem budeme chápat libovolný server, který je dostupný ve veřejné síti internetu, a je dostupný ze všech výpočetních uzlů klastru



Obrázek 2.1 Ukázka síťového zapojení klastru

se na *private_network01*. Segment *private_network01* má přidělen rozsah IP adres `172.17.16.x/24` a obsahuje dvě koncová zařízení (*device01-01*, *device01-02*). Síťový segment je zcela izolovaný s výjimkou napojení na zařízení *kedge01*.

V uvedeném příkladu je privátní síť a interní síť spojena pouze jedním z uzlů klastru *kedge01*. Tento uzel je do privátní sítě *internal_kubernetes01* a interní Kubernetes sítě *internal_kubernete* připojen pomocí dvou různých síťových karet. V obecném případě může více uzlů být připojeno do více privátních sítí a zároveň jedna privátní síť může být propojena s klastrem pomocí více uzlů. Uzly, které takto propojují privátní síť a vnitřní síť Kubernetes budou označovány jako *edge uzly*.

V případě takového nastavení není přímo komunikace objektů klastru se zařízeními v privátních sítích umožněna (ve směru z klastru k zařízením).

Následující část práce bude zaměřena na hledání způsobu, jakým zmíněnou komunikaci umožnit tak, aby co nejvíce vyhovovala stanoveným požadavkům.

2.1.1 Požadavky na řešení problému

Pro účel práce má smysl stanovit požadavky na hledané řešení. Požadavky jsou následující:

- Obecnost řešení
  Řešení by mělo pokrýt všechny možné, výše zmíněné, situace – mělo by umožňovat propojení s více privátními sítěmi za použití více než jednoho uzlu.
- Podpora TCP, UDP
  Řešení by mělo plně podporovat komunikaci pomocí síťových protokolů TCP a UDP a protokolů z vyšších abstrakčních kategorií.
- Jednoduchost
  Řešení by mělo být jednoduché z pohledu uživatele případně administrátora klastru. Jeho používání by nemělo být nijak zvlášť náročné s porovnáním používání jiných služeb Kubernetes.

To samé platí pro jeho zavedení a případnou integraci do již existujícího Kubernetes klastru. Používání by mělo podporovat zavedené způsoby komunikace se systémem Kubernetes.

- Bezpečnost řešení
  Řešení by mělo být bezpečné a nijak významně by nemělo snižovat bezpečnost systému při použití.

Zmíněné komunikace lze dosáhnout dvěma způsoby, každý ze způsobů má jisté výhody a nevýhody. Jako první zde bude uveden způsob za použití existující technologie *Kube edge*, která by daný problém mohla do jisté míry řešit.

2.2 Řešení pomocí Kube edge

Kube edge je projekt společnosti cloud native computing foundation, který rozšiřuje Kubernetes pro použití na koncových zařízení (edge devices). Tento projekt je vyvíjen primárně pro potřeby *edge cloud computing architektur* systému. [36] Cílem projektu Kube edge je usnadnit provoz Kubenets na zařízeních, která mohou mít omezené prostředky, jejich připojení není stále, stabilní a jsou provozovány na různých lokacích. Kube edge umožňuje tyto zařízení integrovat do existujících Kubernetes klastrů. Díky tomut lze dobře pracovat s různými zařízeními, jako jsou například IoT zařízení, různé senzory, zařízení pro chytrá města atd. [37]

Kube edge je převážně určen k integraci zařízení, na kterých je možné provozovat samotné kontejnery a službu kubelet. Primární účel Kube edge tedy neřeší výše popsaný problém, ale je velmi úzce spjatý s komunikací určitých zařízení a interní sítě Kubernetes. Z tohoto důvodu lze v implementaci Kube edge nalézt alespoň částečné řešení zmíněného problému a to pro komunikaci za pomocí MQTT protokolu, případně protokolu HTTP.

Kube edge obsahuje celkem šest komponent tvořících projekt. Jednou z těchto komponent je EventBus. EventBus je komponenta, která umožňuje komunikaci s externími zařízeními pomocí MQTT protokolu. Pokud by byl problém omezen pouze na komunikaci protokol MQTT, pak by EventBus řešil definovaný způsob komunikace a zároveň by splňoval veškeré stanovené požadavky.

Pro potřeby komunikace pomocí protokolu HTTP obsahuje Kube edge komponentu ServiceBus. Tato komponenta umožňuje komunikovat se službami běžícími na výpočetních uzlech koncových zařízení (jsou součástí klastru). Jedná se o komunikaci se službami, které nejsou přímo součástí systému Kubernetes. V uvedeném příkladu to znamená, že Kube edge by umožnilo komunikovat se službou běžící na uzlech *kedge01* respektive *kedge02*. Tuto komunikaci lze představit jako loopback v rámci jednoho uzlu.

Komponenty ServiceBus lze rozšířit dvěma způsoby. Prvním způsobem je úprava zdrojového kódu Kube edge tak, aby podporoval komunikaci i mimo klastr (výpočetní uzly). To by znamenalo, že *edge uzly* by vytvořil most mezi privátní sítí a interní sítí Kubernetes. Tento způsob by byl funkční, ale velmi nestandardní a těžce udržitelný, proto není vhodným řešením. Druhý způsob je vytvoření HTTP služby, která by běžela na *edge uzlech* a sloužila by jako prostředník komunikace. Tato služba by naslouchala na lokální adrese serveru a HTTP požadavky delegovala na daná koncová zařízení v privátní síti. Takové řešení by bylo funkční, ale velmi náročné na provoz. Zároveň by omezovalo problém pouze na HTTP protokol, což je v rozporu s požadavky na řešení. Pro stanovené požadavky je tento způsob také nedostačující. [38]

V obou zmíněných případech by Kube edge poskytoval funkci prostředníka komunikace mezi sítí Kubernetes a privátní sítí.

Kube edge je velmi zajímavá technologie, která je užitečná při integraci koncových zařízení do Kubernetes, ale pro účel propojení klastru s privátní je sítí nedostatečná. I přesto, že dnes Kube edge nenabízí plné řešení problému, je dost pravděpodobné, že potřebná funkcionalita bude do projektu v budoucnu přidána.

Cindy Xing (vedoucí projektu Kube edge) a Kevin Wang (co-founder a vývojář Kube edge) na konferencích KubeCon uvedli, že plánují rozšířit projekt tak, aby podporoval více protokolů. Zároveň zmínili, že se chtějí dále věnovat rozšíření stávající infrastruktury pro komunikaci a adresaci. [39], [40]

2.3 Řešení pomocí Proxy

Popsané možné řešení za pomocí Kube edge by používalo tzv. prostředníka komunikace. V oblasti síťování se služba prostředníka nazývá Proxy. Proxy je služba, která umožňuje přistupovat k službám skrze tzv. proxy serveru (prostředníka). Často se používá pro přístup mezi více síťovými segmenty. Tato služba plní řadu funkcí. Mimo přeposílání komunikací může poskytovat zabezpečení, filtrovat tok dat, zaznamenávat informace o komunikaci apod. Proxy servery mohou pracovat na různých vrstvách ISO/OSI modelu, nejčastěji operují na vrstvě čtvrté respektive sedmé, kde operují s protokoly TCP, UDP respektive HTTP.

Proxy tímto nabízí velmi zajímavou funkcionalitu, která by mohla řešit zmíněný problém. Pokud, by bylo možné provozovat proxy na jednom z edge uzlů, pak by tato služba mohla sloužit jako most mezi privátní sítí a sítí Kubernetes. Taková služba by přeposílala komunikaci mezi interní sítí Kubernetes a přilehlou privátní sítí. K tomu, aby to bylo možné je zapotřebí, splnit následující požadavky:

- Schopnost proxy serveru operovat na čtvrté a sedmé vrstvě ISO/OSI modelu
- Možnost spravovat jednotlivé proxy servery na edge uzlech
- Proxy musí mít přístup k oběma síťovým segmentům (privátní síti a Kubernetes síti)

První požadavek je jednoduše splnitelný vhodným výběrem proxy serveru. Aplikací, které poskytují službu proxy a operují s protokoly TCP, UDP a HTTP je celá řada. Mezi nejznámější patří: Envoy, NGINX, SoCat, HAProxy atd. Každá ze zmíněných služeb je více či méně vhodná na určitý druh použití. Každá lze použít jako most mezi Kubernetes a privátní sítí.

Druhý požadavek se vztahuje ke správě jednotlivých proxy služeb. Je potřeba zajistit, aby proxy služba běžela na konkrétním edge uzlů, byl garantován její běh a aby bylo možné tuto službu jednoduše spravovat. K tomut lze využít stávájících služeb Kubernetes. Aplikaci proxy lze provozovat formou kontejnerů a naplánovat daný kontejner právě na potřebný *edge uzel*.

Posledním požadavkem je přístup k oběma síťovým segmentům. Provozovat proxy, v rámci systému Kubernetes, přináší řadu výhod, ale jednu zásadní nevýhodu. Aplikace provozované v systému Kubernetes mají přístup pouze do interního síťového segmentu. V každém podu se nachází právě jedno síťové rozhraní, které je napojeno do virtuální sítě Kubernetes. Toto je v rozporu s posledním uvedeným požadavkem. Z tohoto důvodu je potřeba umožnit kontrolu nad konfigurací síťových prostředků pro jednotlivé Pody tak, aby bylo možné kontrolovat dostupné síťové segmenty.

V případě, že by bylo možné korektně provozovat proxy službu jako aplikaci v Kubernetes (všechny výše uvedené požadavky by byly splněny), pak by způsob za použití proxy serveru umožnil hledaný způsob komunikace a řešil by definovaný problém. Tento způsob by splňoval většinu z požadavků pro řešení, které jsou uvedeny v sekci 2.1.1. Oboustrannost komunikace by byla splněna ze samotné podstaty fungování proxy. Pro oboustrannou komunikaci lze využít dvou služeb, kde každá z nich bude poskytovat komunikaci jedním směrem. Některé implementace umožňují oboustrannou komunikaci automaticky. Obecnost řešení je splněna díky vlastnostem Kubernetes. Služba lze jednoduše plánovat a spravovat na více uzlech zároveň. Podpora TCP a UDP je možná v případě, že se podaří správně nastavit síťové prostředky uvnitř Podu, ve kterém služba poběží. Bezpečnost a jednoduchost řešení je pak závislá na dané implementaci.

V tuto chvíli je potřeba zajistit výše zmíněnou konfiguraci síťových prostředků pro Pod, ve kterém služba poběží. Pro pochopení jak tohoto dosáhnout, je dobré si připomenout, jak je nastavené síťování uvnitř Podu.

Ve zbytku této kapitoly budou diskutovány dva způsoby, kterými lze dosáhnout, aby Pod mohl komunikovat s více sítovými segmenty (obsahoval více než jedno síťové rozhraní).

2.3.1 Síťování uvnitř Podu

Každý Pod sdílí síťový jmenný prostor mezi všemy kontejnery daného Podu. Tento síťový prostor je vytvořen běhovým prostředím kontejnerů na daném uzlu klastru. Běhové prostředí je ovládáno komponentou kubelet na daném výpočetním uzlu. Ve chvíli, kdy je jmenný prostor vytvořen, CNI pluginu vytvoří a vloží virtuální síťové rozhraní `eth0` do daného prostoru. Zároveň nastaví komunikaci tak, aby splňovalo požadavky pro komunikaci mezi Pody definované systémem Kubernetes. Tímto způsobem vznikne nový síťový prostor pro Pod, ve kterém se budou nacházet rozhraní `eth0` pro komunikaci s vnitřní sítí klastru a navíc rozhraní `lo` (loopback). Ve výpisu 2.1 níže je vidět popisované nastavení Podu.

Výpis kódu 2.1 Nastavení síťových prostředků v podu

```
# Create sample Pod and attatch to its terminal
[1]$ kubectl run my-pod --image=nicolaka/netshoot:v0.9 -it --rm

# List network interfaces
[2]$ ip --br l
>>> lo           UNKNOWN 00:...:00 <LOOPBACK,UP,LOWER_UP>
>>> eth0@if11   UP      06:...:8a <BROADCAST,MULTICAST,UP,LOWER_UP>

# List network interfaces and IP adresses assiged to them
[2]$ ip --br a
>>> lo           UNKNOWN 127.0.0.1/8
>>> eth0@if11   UP      10.244.0.9/16
```

2.3.2 Root namespace

Pro potřeby použití proxy uvnitř Kubernetes je nutné, aby Pod, ve kterém poběží služba proxy měl přístup k dvěma síťovým segmentům. K interní sítí Kubernetes a k privátní síti, do které bude proxy služba tvořit „bránu“. To je potřeba, aby bylo možné přímo přistupovat k potřebným síťovým prostředkům uvnitř Podu a tak umožnit službě proxy korektně operovat na čtvrté vrstvě ISO/OSI modelu. Z toho důvodu je potřeba, aby Pod obsahoval alespoň dvě síťové rozhraní (pro interní Kubernetes síť a pro privátní síť se zařízeními). Ve výchozím nastavení Podu toto není možné, jelikož je vždy vytvořen nový namespace pouze s jedním rozhraním a tím je Pod uzavřen pouze do interní sítě Kubernetes a izolován o ostatních síťových segmentů.

Pro konfiguraci síťového jemného prostoru poskytuje Kubernetes API pouze jeden parametr `pod.spec.hostNetwork`. Tento parametr je součástí definice Podu. Hodnota tohoto parametru nabývá pouze logické hodnoty „True“ nebo „False“. Výchozí hodnota tohoto parametru je „False“. Tímto parametrem lze specifikovat, zda Kubernetes má pro daný Pod vytvářet nový izolovaný síťový prostor, nebo využít stávající kořenový síťový prostor. V případě, že je tento parametr nastaven na hodnotu „True“, pak všechny procesy uvnitř daného Podu mají přístup k síťovým prvkům v kořenovém jmenném prostoru (stejně jako běžící proces kubelet...). V takovém případě budou pro Pod dostupná všechna síťová rozhraní a navíc rozhraní `eth0`, kterém bude připojeno do virtuální sítě Kubernetes. Ukázku tohoto si lze prohlédnout níže ve výpisu 2.2.

Výpis kódu 2.2 Nastavení síťových prostředků v podu při využití hostNetwork

```
# Crete Pod with ".spec.hostNetwork=true"
[1]$ kubectl apply -f - <<EOF
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
spec:
  hostNetwork: true
  containers:
    - image: nicolaka/netshoot:v0.9
      name: my-pod
      command: ["tail"]
      args: ["/dev/random"]
EOF

# Attach to newly created Pod
[1]$ kubectl exec -it my-pod -- bash

# List network interfaces
[2]$ ip --br l
>>> lo               UNKNOWN 00:...:00 <LOOPBACK,UP,LOWER_UP>
>>> bridge           UP      8e:...:0c <BROADCAST,MULTICAST,UP,LOWER_UP>
>>> veth2553bd1f@if2 UP      0e:...:49 <BROADCAST,MULTICAST,UP,LOWER_UP>
>>> eth0@if10        UP      02:...:02 <BROADCAST,MULTICAST,UP,LOWER_UP>

# List network interfaces and IP adresses assiged to them
[2]$ ip --br a
>>> lo               UNKNOWN 127.0.0.1/8
>>> docker0          DOWN    172.17.0.1/16
>>> bridge           UP      10.244.0.1/16
>>> veth2553bd1f@if2 UP
>>> eth0@if10        UP      192.168.49.2/24
```

Tímto lze jednoduše splnit poslední podmínku pro provozování proxy, jako aplikace v Kubernetes. Nyní stačí správně spustit a nastavit proxy tak, aby propojovala síť privátní se sítí Kubenretes. Tato proxy by běžela v Podu s nastavenou hodnotou `pod.spec.hostNetwork` na „True“. Pod by běžel na potřebném edge uzlu.

I přesto, že je tento způsob zcela funkční, nesplňuje požadavek na bezpečnost, který je specifikován v sekci 2.1.1. Řešení velmi narušuje virtualizaci, kterou nám kontejnery a Kubernetes poskytují. Tím je výrazně oslabena bezpečnost tohoto systému. Procesy běžící v Podu, by měly plný přístup k síťovým prostředkům hostujícího systému. To velmi rozšiřuje možnost útoku na hostující systém respektive na prostředí Kubernetes. Používání `pod.spec.hostNetwork` lze označít za anti-pattern, právě kvůli narušení virtualizace. To potvrzuje i Mark Betz ve svém blogu *Understanding Kubernetes networking*.

> *„In fact I would suggest that they are anti-patterns for 99.99 per cent of use cases, and any implementation that makes use of either should get an automatic design review.“* [41]
> (Ve skutečnosti bych řekl, že jsou to anti-vzory pro 99,99 procent případů použití, a každá implementace, která je používá, by měla být přezkoumána.)
>
> *„I don't think it's a stretch to suggest that you are never, ever going to need to do this.“* [41]
> (Nemyslím si, že by bylo přitažené za vlasy naznačovat, že tento způsob, nikdy nebudete potřebovat.)

Z výše zmíněných důvodů nelze toto označit za vyhovující řešení problému. Při použití by se mohlo stát, že by byl tento přístup zamítnut na základě stanovených bezpečnostních politik návrhářů systémů.

I přesto, že zmíněné řešení nelze označit za dostatečné, tak alespoň poskytuje možnost vyzkoušet, zda řešení pomocí proxy serveru je funkční a zda má smysl se jim dále zabývat. Pro účely implementace byla myšlenka testována a následně prezentována (pro účely HIL testování) právě za pomocí využití `pod.spec.hostNetwork`. Pro test a prezentaci bylo nastavení služby proxy identické s nastavením, které je představeno v následující kapitole sekci 3.2.4.

2.3.3 Kubernetes a CNI

Předchozí sekce ukazuje, jakým způsobem lze v Podu přistupovat k síťovým prostředků z kořenového jmenného prostoru. Zároveň zmiňuje, že způsob je funkční, ale nebezpečný kvůli zmíněnému oslabení virtualizace. Pro uspokojení požadavku na bezpečnost je potřeba umožnit Podu interagovat s více sítovými rozhraními, bez narušení virtualizace. V případě, že by bylo možné vytvářet Pody s více než jedním síťovým rozhraním a zároveň jednoduše konfigurovat síťové prostředky uvnitř izolovaného jmenného prostoru Podu, pak by bylo možné využít stejné řešení, jako při využití `pod.spec.hostNetwork` a zachovat bezpečnost systému. Zbytku této kapitoly se budeme zabývat tím, jak tohoto dosáhnout. Pro účely porozumění řešení je potřeba si připomenout a ukázat, jak Kubernetes využívá CNI.

Již v teoretické části je , že Kubernetes k síťování používá CNI standardu. Container Network Interface specifikuje formát souboru, kterým lze definovat a konfigurovat jak mají být CNI moduly používány. Jedná se o informace ve formě JSON souboru, který je uložen na discích serverů klastru (pracovních uzlu). Tento soubor se nachází v adresáři `/etc/cni/net.d/` na každém severu, který je součástí klastru. Kubernetes vždy pracuje se souborem, který se nachází v příslušném adresáři a je první dle abecedního pořadí. Tento soubor nese informace o verzi CNI a jménu CNI modulu, který má být využíván. CNI moduly jsou ve výchozím nastavení ukládány do `/opt/cni/bin`. V případě, že systém využívá například modulu *Flannel*, pak konfigurace CNI, uložená na jednotlivých serverech, bude konfigurovat daný modul a odkazovat na něj. Spustitelný modul implementující CNI (*Flannel*) bude uložen v adresáři `/opt/cni/bin`.

V dřívějších verzích Kubernetes bylo možné definovat cestu k adresáři s konfigurací pomocí `--cni-conf-dir`, tento parametr byl ve verzi `1.24` odstraněn. [28], [42] Cestu k CNI modulům je možné natavit pomocí proměnné prostředí `CNI_PATH` na daném uzlu.

2.3.4 Kubernetes Network Custom Resource Definition De-facto Standard

Potřeba lépe kontrolovat síťové prostředky uvnitř Podu a přímo možnost vytvářet Pody s více síťovými rozhraními byla diskutovaná na konferenci *KubeCon + CloudNativeCon North America* v roce 2017 v Austinu. [43] Na této konferenci vznikla skupina *NPWG* (Network Plumbing Working Group). NPWG vznikla primárně za účelem umožnit vytváření Podů s více síťovými rozhraními. Cílem skupiny bylo vytvořit specifikaci pro konfiguraci přídavných síťových rozhraní a následně implementovat jednoduché řešení pro účely prezentace jejich výsledků.

Zmíněná specifikace, která byla vytvořena skupinou NPWG se nazývá *Kubernetes Network Custom Resource Definition De-facto Standard*. Tento standard slouží pro definici objektů a pro definici API, umožňující rozšíření standardního chování Kubernetes, o možnost vytváření Podů s více sítovými rozhraními. Specifikace definuje následující:

- Objekt Network Attachment Definition
  Objekt NAD (Network Attachment Definition) je nový objekt, který specifikace přidává do Kubernetes API. Tento objekt je definován v doméně `k8s.cni.cncf.io`. NAD slouží pro definici přídavných síťových rozhraní. Tento objekt obsahuje jediný parametr `NetworkAttachmentDefinition.spec.conf`. V tomto parametru lze ukládat informace o přídavném rozhraní. Pro definici přídavného rozhraní stačí do zmíněného parametru uložit CNI konfigurační soubor, formou JSON.
  NAD umožňuje ukládat nastavení pro přídavná síťová rozhraní. Při vytváření Podu s přídavnými rozhraními jsou NAD objekty referencovány.

- Požadavky pro delegující CNI modul
  Network Custom Resource Definition specifikace definuje i základní požadavky pro implementaci tzv. delegující CNI moduly. Delegující CNI moduly označují moduly, které splňují tuto specifikaci. Pro účely práce nejsou tyto požadavky klíčové.
  Implementace a kompatibilita delegujících modulů je umožněna díky sekci standardu CNI o delegování práce modulů. Tato sekce je zmíněna v první kapitole 1.1.3, odrážce 4.

- Komunikaci pluginu s existujícími objekty Kubenretes
  Poslední důležitou částí, která je definovaná zmíněným standardem, je způsob integrace se stávajícími objekty Kubernetes. Zde je definováno, jakým způsobem definovat přídavné rozhraní v definici Podu. K tomut jsou využívaný anotace v metadatech Podu. Anotace označuje objekt, který je součástí metadat Podu a umožňuje uchovávat libovolné textové hodnoty ve formě páru (klíč, hodnota).
  Specifikace využívá anotací právě pro komunikaci s delegující CNI moduly a pro nastavování přídavných rozhraní. V případě potřeby nastavení přídavného rozhraní, stačí na daném Podu vytvořit anotaci s klíčem `k8s.v1.cni.cnf.io/network` a hodnotou, která odpovídá názvu dříve vytvořeného NAD objektu.
  Specifikace udává i další způsoby, jakým lze definovat přídavné rozhraní přímo v definici Podu. Zároveň udává další pravidla pro komunikaci s delegující CNI moduly. Tyto způsoby a pravidla jsou specifická pro různé případy použití a proto zde nebudou uvedeny. V případě zájmu je možné tyto informace najít v samotné specifikaci.

Díky této specifikaci a delegujícím CNI modulům, které specifikaci implementují, je možné vytvářet Pody s více síťovými rozhraními. Network Custom Resource Definition tak řeší poslední

požadavek pro možnost použití proxy (definovaný v sekci 2.3). TímWto je možné využít proxy pro řešení problému propojení interní sítě Kubernetes a přilehlé privátní sítě.

Navrhované řešení, kombinující použití proxy serveru a *Kubernetes Network Custom Resource Definition (CRD) de-facto standardu*, je efektivní a praktický přístup k umožnění komunikace mezi interní sítí Kubernetes a přilehlou privátní sítí. Navrhované řešení nabízí potřebnou flexibilitu, bezpečnost a škálovatelnost.

Kapitola 3

Návrh řešení a implementace

Tato kapitola popisuje návrh a implementaci řešení problému z předchozí kapitoly. Cílem této kapitoly je navrhnout a vytvořit modul pro systém Kubernetes, který umožní jednoduchým způsobem propojit interní síť Kubernetes a přilehlou privátní síť.

3.1 Prostředí

Celý vývoj, testování a ukázka probíhá ve virtuálním prostředí. Díky tomu je možné řešení vyzkoušet bez potřeby hardwaru. Při definici prostředí byl kladen důraz na jeho snadnou replikovatelnost. Veškeré definice a konfigurace jsou dostupné v přiloženém archivu, který je kopií veřejného repositáře na platformě GitHub. Přiložené médium zároveň poskytuje dokumentaci a návod k rozběhnuti samotného prostředí. Tato dokumentace je uložena v adresáři `doc/` v přiloženém archivu[1].

Vzhledem k tomu, že předmětem práce není instalace, případně konfigurace prostředí a návod na zprovoznění je obsažen v přiložených souborech, bude zde postup nastavení prostředí zmíněn jen stručně.

Následující nastavení prostředí probíhalo na sytému s operačním systémem *debian-11*.

3.1.1 Virtuální stroje

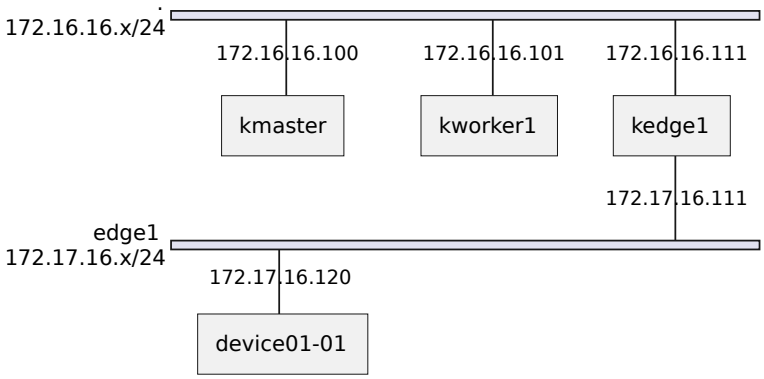
Pro virtualizaci byl využit hypervizor *virtualbox-6.1* od společnosti Oracle. Virtualbox byl ovládán a konfigurován pomocí nástroje *vagrant-2.3.4* od společnosti HashiCorp. Použité technologie a nástroj byly zvoleny pro jednoduchou ovladatelnost a možnost definice prostředí formou zdrojového kódu. Díky tomu je možné deklarativně definovat prostředí, které je snadno nasatvitelné a replikovatelné.

Pro účely vývoje byly vytvořeny celkem čtyři virtuální stroje. Každé z těchto zařízení vychází z definice vagrant boxu `ubuntu/focal64`. Stroje *kmaster*, *kworker1*, *kedge1* jsou servery, na kterých bude provozován systém Kubenretes. Všechny tyto stroje jsou součástí jednoho klastru. Kube-edge je jediný virtuální stroj, který je připojen do dvou síťových segmentů. Tento server plní roli edge uzlu[2]. Čtvrtým virtuální strojem je *device01-01*. Tento stroj reprezentuje zařízení umístěné v privátní síti. Zařízení *device01-01* není součástí sítě Kubernetes. Konfigurace síťování virtuálního prostředí je znázorněna na schématu 3.1 níže.

Kompletní konfigurace virtuálního prostředí je definována v souboru `vagrant/Vagrantfile`. Tento soubor je parametrizovatelný pro potřeby úprav prostředí.

[1] Tato dokumentace prostředí je dostupná i na adrese bt.project.dvojak.cz

[2] Edge uzel označuje stroj, který propojují privátní síť klastru a vnitřní síť Kubernetes.



Obrázek 3.1 Síťové nastavení virtuálního prostředí

3.1.2 Konfigurace serverů

Aby bylo možné provozovat Kubenretes na serverech, je nutné uzly nakonfigurovat a nainstalovat potřebné programy pro provoz systému. Konfigurace a instalace potřebných komponent je definovaná pomocí ansible playbooku[3]. Playbook pro konfiguraci serverů je uložen v souboru `ansible/playbook/infra.yaml`.

Konfigurace a instalace se skládala z následujících kroků, jdoucích po sobě.

1. Nastavení firewall
2. Vypnutí SWAP (odkládací prostor na disku)
3. Zavedení modulů kernelu pro potřeby Kubernetes a Containerd
4. Nastavení parametrů kernelu pro Containerd
5. Instalace Containerd
6. Konfigurace Containerd a nastavení podpory Cgrups
7. Instalace Kubernetes
8. Vytvoření uživatele *kube* pro ovládání Kubernetes
9. Nastavení parametrů kubelet
10. Nastavení parametru kernelu pro přeposílání ARP komunikace

Jedná se o standardní instalaci Kubernetes s Containerd poskytující běhové prostředí kontejnerů dle OCI. Předposlední bod, z výše uvedených, je přidán, aby bylo možné provozovat systém Kubernetes na serverech s více síťovými rozhraními. Poslední krok je součástí konfigurace pro případné testování podpory obousměrnosti komunikace. Tato možnost je krátce zmíněna na konci sekce 3.2.4.

[3]Playbook označuje soubor definující úlohy pro konfiguraci pomocí ansible.

3.1.3 Konfigurace zařízení

Pro možnost testování je nutné vhodně nastavit i virtuálním stroj *device01-01* tak, aby provozoval aplikace komunikující za pomocí TCP, UDP a HTTP protokolu. Pro jednoduchost provozování aplikací je použit Docker. Konfigurace a instalace byla provedena pomocí ansible playbooku `ansible/playbook/device.yaml`.

3.1.4 Instalace a konfigurace Kubernetes

Instalace Kubernetes je prováděna pomocí nástroje *kubeadm*. Kubeadm umožňuje jednoduchou instalaci Kubernetes. Pro konfiguraci instalace systému lze využít parametry, které kubeadm nabízí. Důležitými parametry pro instalaci cloudu jsou `--apiserver-advertise-address` nastavující primární IP adresu API serveru, `--pod-network-cidr` nastavující proměnnou *podCIDR* a `--apiserver-cert-extra-sans` definující parametry pro generování TLS certifikátu.

Součástí instalace Kubernetes je instalace samotného CNI, který bude použit pro síťování uvnitř cloudu. Pro ukázkové prostředí byl použit modul *Flannel*. Jedná se o jednoduchý a často používaný CNI Kubernetes plugin[4].

3.1.5 Výsledné prostředí

Výsledné prostředí poskytuje dobré možnosti pro testování a realizaci samotného návrhu rozšíření. Následující části textu budou vyvíjeny a prezentovány na prostředí odpovídající instalaci popsané výše. Postup rozběhnutí prostředí přesně odpovídá postupu, který je popsán v přiloženém archivu.

3.2 Návrh řešení a implementace

Před implementací řešení je nutné provést návrh tak, aby splňoval již deklarované požadavky. Návrh vychází z probíraného řešení pomocí proxy, které je popsáno v sekci 2.3 předchozí kapitoly. Návrhu bude používat již zmíněný a probraný koncept Podu s více síťovými rozhraními. Konfigurace Podu s více rozhraními bude docíleno za použití implementace standardu *Kubernetes Network Custom Resource Definition De-facto Standard*.

Návrh řešení se bude skládat z dvou hlavních částí. V první části bude diskutován výběr aplikace pro provozování služby proxy. Druhá část bude popisovat konkrétní návrh konfigurace síťování v Podu. Na závěr bude provedena ukázka navrženého řešení.

3.2.1 Vymezení implementace

V předešlé kapitole jsou deklarovány požadavky na hledané řešení problému a následné rozšíření funkcionality Kubernetes. Návrh si klade nároky splnit každý z těchto požadavků. Zároveň se snaží o to, aby nepřinášelo žádná výrazná omezení pro případné použití. Veškeré koncepty a myšlenky, které budou dále popsány, budou klást důraz na snadnou konfigurovatelnost a případnou rozšiřitelnost.

I přesto tuto snahu dává smysl se omezit na určitý způsob použití výsledného rozšíření. Tato omezení jsou použita zejména pro umožnění automatizace práce formou operátoru. Automatizace bude popsána v sekci 3.3.2, zabývající se rozšířením Kubernetes.

Pro každé zmíněné omezení bude nabídnuta alternativa, která dané omezení umožňuje realizovat.

[4] Řešení podporuje i použití jiných CNI pluginy. Funkcionalita byla testována pouze na modulech Flannel a Calico

Podpora protokolů

V požadavcích pro řešení problému je zmíněná podpora TCP, UDP a HTTP protokolu. TCP a UDP protokoly se nachází na stejné abstrakční vrstvě dle ISO/OSI modelu. Protokol HTTP se nachází na sedmé nejvyšší abstrakční vrstvě. Tento protokol se spoléhá na zmíněné protokoly TCP a UDP. Dnes používané verze HTTP (verze v1 a v2) jsou provozovány za pomocí TCP protokolu. Verze v3 protokolu HTTP je stále vyvíjena. Dnešní návrh počítá, že protokol bude využívat QUIC protokol, který je postaven na UDP. [44].

Pro potřeby práce bude navrhované řešení zaměřeno převážně na komunikací pomocí protokolů TCP a UDP. Toto je možné, jelikož nalezení řešení pro tyto dva protokoly implikuje i podporu HTTP. Přímo podpora protokolu HTTP bude diskutována v sekci níže. Jeho podpora může přinést zajímavé možnosti a rozšiřující funkcionality. Podpora HTTP ale výrazně komplikuje konfiguraci a návrh řešení. Pro automatizaci není vhodná.

Jednosměrnost komunikace

Dalším vymezením je podpora převážně jednosměrné komunikace z interní sítě clusteru do sítě privátní. Jednosměrnou komunikaci se myslí pouze směr, kterým je vyslána první zpráva komunikace. Takto zůstane podporována oboustrannost komunikace pomocí TCP i UDP v případě, že komunikace je zahájena z vnitřní sítě klastru.

Směr adresace z privátní sítě do sítě klastru je velmi specifický případ užití a proto nebude přímo součástí implementace. Možnost podpory toho směru bude diskutována níže.

3.2.2 Výběr proxy

Pro propojení interní sítě Kubernetes a přilehlé privátní sítě bude použita služba proxy. Tato služba by měla splňovat následující požadavky: možnost provozování v kontejneru, snadná konfigurovatelnost a možnost provozování více proxy spojení v jeden okamžik.

Aplikací poskytující službu proxy je celá řada. Z tohoto důvodu nedává smysl vyvíjet nové vlastní řešení. Namísto toho lze využít již existujících implementací. Mezi známe implementace, které splňují výše deklarované požadavky patří například: HAProxy, Nginx, Envoy, SoCat. Právě SoCat byl vybrán jako vhodná pro provoz služby proxy.

SoCat je nástroj pro přenos dat vyvinutý společností Red Head. Tento nástroj je navržen tak, aby byl jednoduchý a zároveň vysoce konfigurovatelný. To nabízí velké možnosti použití tohoto nástroje. Jedno z možných použití je i poskytování TCP respektive UDP proxy. [45]

SoCat je velmi známý a minimalistický nástroj. Díky tomu je velmi jednoduchý na použití a vhodný pro provoz v kontejneru. SoCat je zároveň dostupný, jako veřejný obraz kontejneru na platformě Docke Hub. Z tohoto je vhodným řešením pro provoz v rámci systému Kubernetes.

Další podmínkou pro vhodnou volbu proxy je možnost provozování proxy pro více spojení najednou. Pro tuto funkci nabízí SoCat přepínač *fork*. Při použití tohoto přepínače program vytvoří pro každé nové spojení nový proces, který bude dané spojení spravovat. Tímto je zajištěna podpora více spojení najednou.

Společně s jednoduchou konfigurací splňuje SoCat všechny požadavky a je vhodnou volbou pro provozování proxy v prostředí Kubenretes formou kontejneru. Následující část textu již bude předpokládat použití tohoto nástroje.

SoCat přímo neumí pracovat s protokolem HTTP. V případě potřeby práce je možné využít některý z nástrojů, které tuto možnost nabízí. Vhodným kandidátem může být služba Nginx, kterou lze provozovat jako proxy podporující HTTP. Nginx nabízí možnost proxy na čtvrté a sedmé vrstvě ISO/OSI modelu. Možnost řídit proxy pomocí HTTP protokolu pak nabízí velkou řadu možností – oproti řešením podporujíc pouze čtvrtou vrstvu ISO/OSI. Příkladem může být pokročilý load-balancing, podpora šifrování a kontrola routování pomocí HTTP atd. Pro účely práce nebude tento proxy server použit, z důvodu náročné konfigurace.

3.2.3 Nastavení sítě

Důležitou částí návrhu řešení je i nastavení síťování v Podu tak, aby podporoval více síťových rozhraní. Způsob, kterým lze tohoto dosáhnout, je na teoretické úrovni popsán již v předchozí kapitole. Z této kapitoly bude návrh síťování vycházet. Pro podporu Podu více síťových rozhraní bude využit *Kubernetes Network Custom Resource Definition De-facto Standard*.

Tento standard deklaruje API rozhraní pro popis přídavných síťových rozhraní v rámci Kubernetes. Zároveň definuje základní podobu tzv. delegujících pluginů. Jako delegujícími pluginy jsou ve standardu označovány programy a řešení, které implementují zmíněný standard. Kubernetes *Network Custom Resource Definition De-facto Standard* odkazuje na část standardu CNI o delegování práce mezi moduly. Zmíněná část je krátce popsána v sekci 1.1.3 první kapitoly. Právě díky této části je umožněna snadná implementace delegujících pluginů. Jedním z těchto pluginů je Multus CNI (Multus).

Multus je delegující plugin, který je vyvíjen přímo skupinou Network Plumbing Working Group, která spravuje zmíněný standard. Multus není přímo implementace CNI specifikace, Multus projekt je označován jako tzv. meta-plugin. [46]

Multus byl navržen jako zásuvný modul, který rozšiřuje základní Kubernetes CNI o funkci tvorby Podu s více rozhraními. Této možnosti je dosaženo tím, že Multus umožňuje volání více implementací CNI standardu. Při použití to znamená, že Multus volá tzv. „hlavní“ CNI modul (výchozí, pro umožnění komunikace v cloudu) a následně „vedlejší“ modul, který do Podu vkládá a nastavuje přídavné rozhraní. Tímto způsobem je umožněno vytvářet Pody s více síťovými rozhraními. [46]

Způsob konfigurace přídavných síťových rozhraní je popsán ve specifikaci Kubernetes *Network Custom Resource Definition De-facto Standard*. Více informací o standardu je popsáno v minulé kapitole sekci 2.3.4.

3.2.4 Ukázka nastavení proxy

Pomocí služby SoCat a modulu Multus je již možné vytvořit funkční způsob pro komunikaci se zařízeními v privátních sítích. Pro snadnější pochopení, jakým způsobem je adresace v privátních sítích umožněna, bude řešení ukázáno na konkrétním příkladu.

Cílem příkladu bude zprovoznit službu proxy přímo v systému Kubernetes tak, aby poskytla komunikaci mezi interní sítí klastru a jedním zařízením v privátní síti.

Jako první je potřeba vytvořit objekt Network Attachment Definition. Konkrérně se jedná o objekt s názvem *bridge-conf*. Definice tohoto objektu je znázorněna ve výpisu kódu 3.1. Uvedená specifikace objektu *bridge-conf* odpovídá navrženému API v rámci *Kubernetes Network Custom Resource Definition De-facto Standard*. Objekt definuje přídavné síťové rozhraní, které bude vytvořeno pomocí CNI modulu *bridge*[5].

Tento objekt popisuje způsob, jakým bude vytvořeno přídavné síťové rozhraní v Podu obsahující proxy (běžící proces SoCat). Popis je obsažen v atributu objektu `.spec.connection`. Tento atribut obsahuje kompletní popis a konfiguraci pro použití „vedlejšího“ pluginu CNI. Jako vhodným modulem pro ukázku byl zvolen CNI plugin *bridge*. Jedná se o stejný CNI plugin, který byl popsán v první kapitole této práce 1.1.2.

V tuto chvíli je vytvořen objekt popisující způsob vytvoření přídavného síťové rozhraní. Nyní je třeba vytvořit Pod, který bude provozovat potřebou službu proxy. Definice tohoto Podu je znázorněna v ukázce 3.2.

[5] Uvedená konfigurace je pouze ukázková, při používání je důležité objekt nakonfigurovat konkretním potřebám pro konkretní systém. Pro více informací je dobré být plně seznámen s projektem CNI, zejména se specifikací.

Výpis kódu 3.1 Ukázka konfigurace Network Attachment Definition

```
---
apiVersion: k8s.cni.cncf.io/v1
kind: NetworkAttachmentDefinition
metadata:
  name: bridge-conf
spec:
  config: |-
    {
      "cniVersion": "0.3.1",
      "type": "bridge",
      "name": "bridge-conf",
      "bridge": "eb",
      "isGateway": true,
      "ipMasq": true,
      "ipam": {
        "type": "host-local",
        "ranges": [[{ "subnet": "10.10.0.0/24"}]],
        "routes": [{"dst": "0.0.0.0/0"}],
        "dataDir": "/run/kube-edge/ipam-state"
      }
    }
```

Výpis kódu 3.2 Ukázka konfigurace podu s Proxy

```
---
apiVersion: apps/v1
kind: Pod
metadata:
  name: sample-proxy
  annotations:
    k8s.v1.cni.cncf.io/networks: bridge-conf
spec:
  containers:
    - name: proxy-tcp
      image: alpine/socat
      args: ["tcp4-listen:8080,fork", "tcp4-connect:172.17.16.120:8080"]
```

Takto vytvořený Pod bude provozovat proxy pomocí nástroje SoCat. Konfigurace SoCat je určena v objektu `.spec.containers`. Přídavné síťové rozhraní je deklarováno pomocí atributu `.metadata.annotations.'k8s.v1.cni.cncf.io/networks'`, který odkazuje na již vytvořený Network Attachment Definition objekt. Pro kontrolu konfigurace je uveden výpis 3.3 z vytvořeného Podu.

Výpis kódu 3.3 Ukázka nastavení proxy Podu

```
# Exec /bin/sh inside Pod
[1]$ kubectl exec -it sample-proxy -- sh

# Show SoCat process (pid=1)
[2]$ ps -p 1 -o args
>>> COMMAND
>>> socat tcp4-listen:8080,fork tcp4-connect:172.17.16.120:8080

# List network interfaces and IP addresses
[2]$ ip -br a
>>> lo         UNKNOWN 127.0.0.1/8 ::1/128
>>> eth0@if38 UP      10.244.2.17/24
>>> net1@if39 UP      10.10.0.32/24
```

V uvedené ukázce lze vidět, že v Podu běží proces SoCat, který provozuje službu proxy. Zároveň je možné vidět, že Pod obsahuje celkem tři síťová rozhraní. První rozhraní je typu *loopback* toto rozhraní je obsaženo v každém Podu pro potřeby komunikace v rámci samotného Podu. Druhým rozhraním je `eth0@if38`. Toto rozhraní je vytvořeno CNI modulem *Flannel* a slouží pro komunikaci uvnitř cloudu. Posledním rozhraním je `net1@if39`. Toto zařízení je vytvořeno díky meta-pluginu Multus. Konkrétně se jedná o rozhraní vytvořené „vedlejším“ modulem *bridge*, jak je definováno v objektu Network Attachment Definition.

Uvedená konfigurace korektně nastavuje proxy tak, aby spojení, které přijdou na otevřený port 8080 daného Podu, byla přeposlána dále. V uvedeném případě, na zařízení (s IP adresou `172.17.16.120`) v privátní síti, dostupné z pracovního uzlu, kde Pod běží. O správné doručení paketů komunikace se postará již hostující systém Podu. Samotné opuštění interní sítě klastru je umožněno díky přidanému síťovému rozhraní `net1@if39`.

Uvedený příklad vyžívá CNI pluginu *bridge* jako „vedlejší“ plugin pro nastavení síťování. Takto navržené použití podporuje i libovolné jiné implementace CNI standardu. Díky tomuto řešení poskytuje úplnou flexibilitu při konfiguraci sítě, jelikož lze použít libovolný CNI plugin.

Tento příklad demonstruje použití nástroje SoCat a meta-pluginu Multus tak, aby umožnil komunikaci objektů v systému Kubernetes se zařízeními v přilehlé privátní síti. Pod bude plně abstrahovat komunikaci se zařízením. V reálném případu užití se hodí využít navíc objektu Service odkazujíc na daný Pod. Tím lze docílit zpřístupnění standardních funkcí jako je například load-balancing a podpora DNS.

Následující a poslední sekce této práce se bude zabývat integrací uvedeného řešení do prostředí Kubernetes a automatizací konfigurace pro snadnější použití.

Podpora obousměrnosti komunikace

Zmíněná ukázka poskytuje jednosměrné řešení umožnující komunikaci se zařízeními v privátní síti. Adresace směrem do cloudu z privátní sítě není podporována. To odpovídá omezení zmíněnému v sekci o vymezení implementace.

V případě potřeby adresace do vnitřní sítě klastru se nabízí dvě možnosti. První z nich je použití objektu Ingress. Pomocí Ingress lze dosáhnout možnosti adresace do sítě klastru. Takové

řešení je validní, ale přináší nemalé nároky na konfiguraci. Pro podporu více privátních síťových segmentů je použití Ingress neefektivní a pro automatizaci nevhodné.

Druhým způsobem je možnost využití upraveného již navrženého řešení. Pro realizaci je možné využít proxy stejným způsobem, jako je popsáno výše, jen v opačném směru. Proto, aby byla komunikace umožněna, musí být vhodně zvolen „vedlejší“ CNI plugin. Příklady pluginů, které lze využít jsou *host-device* a *macvlan* z množiny referenčních modulů, které jsou vyvíjeny v rámci specifikace CNI. Pro správné použití může být třeba povolit přeposílání ARP paketů na edge uzlech (záleží na zvoleném „vedlejší“ CNI modulu).

V případě implementace podpory obousměrné adresace se nabízí i poskytnout DNS překlad doménových jmen na edge uzlech, pro usnadněni adresování. Provozovat takovou službu pro objekty interní sítě Kubernetes je snadno realizovatelné pomocí CoreDNS, s případně upraveným Kubernetes pluginem.

3.3 Rozšiřování Kubernetes

Cílem této sekce je rozšířit funkce Kubernetes tak, aby se usnadnilo vytváření spojení mezi interní sítí Kubernetes a zařízeními v privátních přilehlých sítích. Zmíněné rozšíření bude automatizovat kroky uvedené výše. Tato sekce převážně popisuje implementaci operátoru *EdgeOperator*. Zdrojové kódy operátoru jsou součástí přiloženého archivu v adresáři `code/EdgeOperator/`.

Kubernetes je velmi dobře navržen proto, aby byl jednoduše rozšiřitelný. Na oficiálních stránkách projektu Kubernetes je uvedeno:

> *„Kubernetes is highly configurable and extensible. As a result, there is rarely a need to fork or submit patches to the Kubernetes project code.“* [47]
> (Kubernetes je vysoce konfigurovatelný a rozšiřitelný. Díky tomu je jen zřídka potřeba kód projektu Kubernetes forkovat nebo zasílat záplaty.)

V případě potřeby, Kubernetes nabízí způsoby, kterým lze systém rozšířit. Dokumentace projektu popisuje různé potřeby pro rozšíření systému a zároveň odkazuje na způsoby, jak toho dosáhnout. Pro tvorbu operátoru je důležitá sekce popisující rozšiřování Kubernetes API a automatizace práce.

3.3.1 Rozšiřování Kubernetes API

Kubernetes objekty jsou prvky, které uchovávají stav systému. Lze si je představit jako datové struktury nesoucí informace. Tyto objekty slouží pro komunikaci s Kubernetes API serverem. API server poskytuje základní CRUD operace k těmto objektům a díky těmto operacím lze konfigurovat, nastavovat a ovládat samotný cloud.

Kubernetes poskytuje základní objekty pro práci s cloudem. Příklady těchto objektů jsou Pod, Deployment, Endpoint, Namespace atd. Tyto objekty jsou navrženy a spravovány autory Kubernetes. Jedním ze standardních objektů je i CustomResourceDefinition. CustomResourceDefinition (zkráceně CRD) je meta-objekt, který umožňuje definovat vlastní nové objekty. CRD pak umožňuje definovat strukturu nových objektů (datových struktur). Příkladem použití CRD je již zmíněný Network Attachment Definition, který byl definovaný v dokumentu Kubernetes Network Custom Resource Definition De-facto Standard. V případě, že je nový CRD objekt vytvořen, je tento objekt přidán do Kubernetes API a tím je možné s ním Kubernetes pracovat. Pro nový datový objekt jsou automaticky vytvořeny základní CRUD operace.

CustomResourceDefinition poskytuje elegantní způsob pro rozšiřování Kubernetes API.

3.3.1.1 Návrh vlastních CRD objektů

Jedním z požadavků pro řešení problému zmíněných v sekci 2.1.1 je jednoduchost při používání. Pro jednoduchost je požadováno, aby řešení podporovalo zavedené způsoby komunikace s cloudem.

Toho lze dosáhnout právě za použití CRD, pro rozšíření standardního Kubernetes API.

Pomocí CRD objektů je možné udržet standardní API pro uživatele. Tento způsob přináší řadu výhod a usnadnění. I přesto, že tato komunikace (s operátorem) se může stát nestandardní, a z počátku složitá, jedná se o zavedený způsob, který je v oblasti administrace populární. V oblasti DevOps lze hovořit o zavedeném standardu.

Následující podkapitoly uvedou návrh CRD objektů pro výsledný operátor. Samotná implementace operátoru uvedena v poslední části této kapitoly 3.3.2.2.

Device

Prvním z objektů rozšiřujících standardní API je *Device*. Objekty *Device* reprezentují zařízení, která se nachází v privátní síti, mimo cloudu. Tato zařízení by měla být dostupná přes jeden či více uzlů v klastru. Tyto uzly souží jako brány k zařízení. Objekty slouží primárně pro ukládání informací o zařízeních. Ve výpisu kódu 3.4 lze vidět ukázkovou definici navrženého objektu.

Výpis kódu 3.4 Ukázka CRD Device

```
---
apiVersion: edge-operator.k8s.dvojak.cz/v1
kind: Device
metadata:
  name: device01-01
spec:
  nodeName: kedge1
  up: true
  ipAddress: 172.17.16.120
  components:
    - name: backend
      up: true
      handlers:
        - name: api server
          protocol: HTTP
          port: 443
          endPoints:
            - https://<path>/api/*
            - https://<path>/version
```

Objekt *Device* uchovává data formou – klíč, hodnota. Data jsou serializována pomocí YAML souborů. Níže jsou vypsány jednotlivé klíče, spolu s vysvětleným významem hodnot v nich uložených.

`.spec.nodeName` — Název pracovního uzlu, který je připojen do privátní sítě a sítě Kubernetes. Tento uzel bude použit pro vytvoření mostu a provozování proxy.

`.spec.up` — Definuje, zda je dané zařízení zapnuté a může být použito.

`.spec.ipAddress` — IP adresa zařízení v privátní síti.

`.spec.components` — Seznam komponent, které na zařízení běží a jsou dostupné po síti. Komponenta označuje běžící aplikaci. Příkladem takové komponenty může být webový server či jiná aplikace.

`.spec.components.name` — Název komponenty pro identifikaci.

`.spec.components.up` — Určuje, zda je daná komponenta zapnuta a může být použita.

`.spec.components.handlers` — Seznam portů, které daná aplikace obsahuje.

`.spec.components.handlers.name` — Název portu. Tento parametr nemá přímé použití, slouží pouze pro dokumentační účely.

`.spec.components.handlers.protocol` — Protokol označuje typ použitého protokolu, povolené hodnoty jsou TCP, UDP a HTTP

`.spec.components.handlers.port` — Číslo daného portu

`.spec.components.handlers.endpoints` — Seznam koncových bodů (endpointů). Tento seznam je nepovinný a slouží pouze pro dokumentační účely.

Objekt *Device* je součástí API skupiny *edge-operator.k8s.dvojak.cz*. Zároveň je navržen jako globální a tudíž nepodléhá žádnému Kubernetes namespace.

Od této chvíle budeme zařízení v privátní síti značit jako *Device*.

Connection

Druhým definovaným CRD objektem je *Connection*. *Connection* reprezentuje již vytvořené spojení se zařízením v privátní síti *Device*. Objekt slouží pro uchovávání informací o spojení a zároveň k ovládání daného rozšíření. Struktura objektu je nastíněna v ukázce kódu 3.5.

Výpis kódu 3.5 Ukázka CRD Connection

```
---
apiVersion: edge-operator.k8s.dvojak.cz/v1
kind: Connection
metadata:
  name: sample-connection
  namespace: default
spec:
  deviceName: device01-01
  networkName: bridge-conf
  componentNames:
    - backend
```

Níže jsou vypsány jednotlivé klíče, spolu s vysvětleným významem hodnot v nich uložených.

`.spec.deviceName` — Název objektu *Device*, ke kterému bude vytvořeno spojení proxy.

`.spec.networkName` — Názez objektu *NetworkAttachmentDefinition*. Tento objekt definuje nastavení sítě v Podu, které bude použito pro Pod s proxy.

`.spec.componentNames` — Seznam komponent objektu *Device*, ke kterému bude vytvořeno spojení proxy.

Vytvořením objektu *Connection* se vytvoří již samotné spojení mezi interní sítí Kubernetes a daným zařízením. Postup vytvoření takového spojení je vysvětlen v sekci 3.3.2.2.

Objekt *Connection* je součástí API skupiny *edge-operator.k8s.dvojak.cz*. Tento objekt je Kubernetes namespace závislý.

3.3.2 Automatizace práce

Objekty slouží jako datové struktury pro ukládaní informací o stavu systému. Tyto objekty jsou zpracovávány pomocí Kubernetes kontrolerů. Kontrolery jsou nekonečné smyčky, které mají za úkol udržovat systém v požadovaném stavu, který je definovaný pomocí objektů uložených v *etcd* databázi. Kontrolery lze chápat jako pracovníky, kteří spravují celý systém Kubernetes. Oficiální dokumentace uvádí příklad na termostatu. Termostat je typicky nastaven na fixní požadovanou teplotu, kterou má za úkol v místnosti udržet. V případě, že teplota klesne Pod požadovanou hodnotu, pak provede potřebné kroky pro zvýšení teploty v místnosti.

Stejným způsobem fungují kontrolery v systému Kubernetes. Neustále porovnávají aktuální stav cloudu Kubernetes se stavem požadovaným (definovaným objekty). V případě že se tyto dva stavy liší, pak se pokusí aktuální stav cloudu co nejvíce přiblížit stavu požadovanému.

Funkce kontroleru je názorně ilustrována na následujícím příkladu zdrojového kódu 3.6. [48]

Výpis kódu 3.6 Implementace kontroleru [48]

```
for {
  desired := getDesiredState()
  current := getCurrentState()
  if current != desired {
    makeChanges(desired, current)
  }
}
```

Stejně jako Kubernetes obsahuje základní skupinu standardních objektů, tak nabízí i standardní množinu kontrolerů. Tyto kontrolery jsou spravovány vývojáři Kubernetes a jsou součástí základní instalace systému. Příkladem těchto kontrolerů jsou Deployment Controller, Endpoint Controller, Namespace controler atd.

Kontrolery společně s objekty tvoří základní stavební bloky Kubernetes, které představují způsob, jakým je Kubernetes navržen. Tento koncept (návrhový vzor) usnadňuje správu a škálování aplikací. Zároveň umožňuje snadnou rozšiřitelnost systému.

V předchozí částí byl zmíněn způsob rozšiřování Kubernetes API pomocí CRD. Právě s CRD úzce souvisí návrhový vzor operátor. Operátor je návrhový vzor definovaný Kubernetes, který (společně s CRD) umožňuje vytvářet moduly podobné kontrolerům.

3.3.2.1 Operátor

Operátor je aplikace, která rozšiřuje funkcionalitu Kubernetes o specifické vlastnosti. Velmi často pracuje s Custom Resource Definitions (CRD). Operátor navazuje na principy kontrolerů v Kubernetes a zavádí možnost automatizace pro správu aplikací v prostředí cloudu. Hlavní myšlenkou operátoru je umožnit automatizace práce administrátorům aplikací. Operátor lze označit jako modul, který přidává velmi specifické funcionality do systému Kubernetes.

Operátor Kubernetes je obvykle složen z kombinace CRD (Custom Resource Definitions) a kontroleru, který spravuje a zpracovává instance těchto objektů. Většina operátorů implementuje nebo navrhuje jedno nebo více rozhraní, které Kubernetes poskytuje. Tato rozhraní zahrnují:

- Custom Resource Definitions (CRD)
  Objekty umožňující ukládání dat o cloudu a komunikaci s operátorem prostřednictvím Kubernetes API.

- Kontroller
  Při implementaci kontroleru je implementace často abstrahována do tří základních funkcí:

  - Reconcile
    Udržuje cloud v požadovaném stavu a je volána při vzniku nebo změně definice pozorovaného CRD.
  - Status Modified
    Reaguje na změnu vnitřního stavu objektu a je volána při změně stavu pozorovaného CRD.
  - Deleted
    Je volána pokaždé, když je pozorovaný CRD objekt smazán.
- Finalizers
  Souvisí s implementací kontrolerů a umožňuje dokončit a uklidit prováděné operace. Finalizer je typicky volán při neočekávané chybě systému nebo po smazání objektu.
- Webhooks
  Umožňuje napojení na inplementaci stávajícího API serveru a ovlivnění zpracování dat při použití API serveru. Kubernetes nabízí dva základní typy:
  - Mutator
    Umožňuje modifikovat požadavek na API server, aplikovat různé politiky nebo opravit nekonzistence dat.
  - Validator
    Umožňuje validovat příchozí požadavky a odmítnout ty, které neodpovídají předem stanoveným pravidlům nebo očekávaným hodnotám.

Pro detailnější pochopení návrhového vzoru doporučuji blog *Exploring Kubernetes Operator Pattern*, oficiální dokumentaci Kubernetes a přednášku *Tutorial: Zero to Operator in 90 Minutes!* od *Solly Ross* [49], [50], [51], [52]

3.3.2.2 Oprátor EdgeOperator

Poslední částí implementace je realizace samotného operátoru, který bude pracovat s výše vydefinovanými CRD a automatizovat tvorbu proxy Podu, jak je ukázáno v sekci 3.2.4. Od této chvíle bude tento operátor označován jako *EdgeOperator*.

Oficiální dokumentace Kubernetes vybízí k použiti některé z frameworků pro implementaci operátorů. Frameworků pro realizaci operátoru je celá řada. Kubernetes na svých stránkách uvádí tyto příklady: *kubebuilder (GO)*, *KubeOps (.NET)*, *kube-rs (Rust)* a *Java Operator SDK*. Pro implementaci operátoru *EdgeOperator* byl použit *KubeOps - The dotnet Kubernetes Operator SDK*. [53]

KubeOps je framework obsahující sadu nástrojů pro implementaci operátoru v prostředí .NET. Jedná se o udržovaný projekt, který je stále vyvíjen. Jeho autor udává, že projekt je silně inspirován projektem *Kubebuilder*. [53] *KubeOps* je implementován v nejnovější verzi frameworku s dlouhodobou podporou (`.NET 7`). Implementace *KubeOps* využívá technologii `ASP.NET`, která slouží pro vývoj webových služeb. Použití zmíněné technologie pro tvorbu webu jako základ pro realizaci operátorů dává smysl. Právě `ASP.NET` je dle průzkumu *Stack Overflow Developer Survey 2022* nejpoužívanější technologií pro vývoj webových backendových aplikací profesionálními vývojáři. [54] Toto velmi zpřístupňuje možnost implementace operátorů.

V následujících sekcích bude krátce vysvětlen mechanizmus fungování operátoru E*dge-operator* a nastíněn způsob, kterým byl operátor implementován pomocí *KubeOps* frameworku.

CRD

Objekty, které *EdgeOperator* implementuje jsou *Device* a *Connection*. Jedná se o objekty, které jsou definovány a popsány v sekci 3.3.1.1. Tyto objekty jsou v operátoru definované pomocí tříd, které korespondují navrženému API. Mimo standardní datové typy a konstrukce, které nabízí

jazyk `C#`, jsou k definici použity i atributy (anotace), které nabízí framework *KubeOps*. Tyto atributy slouží převážně k dodatečnému definování metadat o objektu, jako je dokumentace a podoba specifikace výsledných CRD.

Objekty jsou definovány pomocí tříd `DeviceEntity` a `ConnectionEntity`, které dědí příslušné třídy z *KubeOps*.

Validátor

EdgeOperator implementuje validátor pro každý ze zmíněných objektů. Oba tyto validátory slouží pro kontrolu správnosti definic objektů. Tato kontrola probíhá vždy, když je jeden z objektů vytvořen, nebo modifikován. Kontrolované vlastnosti jsou detailně popsány v dokumentaci projektu.

Ve výchozím nastavení, operátor zamítne požadavek pro vytvoření respektive editaci objektu v případě, že objekt nesplní jednu z kontrolovaných vlastností. V případě potřeby je možné operátor přepnout do nestriktního módu validace. V této situaci, kdy je porušena některá z požadovaných vlastností objektu, operátor pouze informuje o jaká porušení se jedná a požadavek nezamítne. Po provedení požadavku jsou data propsána do interní Kubernetes databáze *etcd*. Tento mód není určen pro reálné nasazení, jelikož může docházet k nedefinovanému chování. To vede k nekonzistenci stavu cloudu. Tyto nekonzistence mohou vyžadovat přímý zásah administrátora do vnitřní logiky fungování operátoru.

Validátory jsou realizovány pomocí tříd `DeviceEntityValidator` a `ConnectionEntityValidator`. Tyto třídy implementují rozhraní dodávaná *KubeOps*. Realizace validátorů je umožněna díky webhooks, která Kubernetes poskytuje.

Implementace (formou pseudokódu) obou validátoru je uvedena ve výpisech kódu 3.7 a 3.8.

Výpis kódu 3.7 Implementace validace Device v Edge-Operator

```
func ValidateDevice(Device device){
    var errors = new([]string);

    errors += validateIpAddress(device);
    errors += validateNodeExists(device);
    errors += validateNodeHasLabels(device);
    errors += validateComponentsHasUniqueNames(device);
    errors += validateProtocolCollision(device);

    if(cloud.operator.strictValidation){
        return Fail(errors);
    }
    return Success(errors);
}
```

Kontroler

Hlavní částí *EdgeOperator* je kontroler objektu *Connection*. Implementace tohoto kontroleru automatizuje vytváření služeb proxy, které umožňují samotnou komunikaci. Jedná se o automatizaci kroků, které jsou uvedeny v sekci 3.2.4 této kapitoly.

Samotný kontroler implementuje metody Reconcile a Delete kontroleru. Tyto metody se spoléhají na korektní definici CRD objektů. Pro správné fungování je vyžadováno, aby databáze *etcd* obsahovala objekty, které jsou validní dle výše popsaného validátoru.

Reconcile metoda implementuje logiku pro vytváření nový spojení a udržování již existujících spojení (*Connection*). Při vzniku nového spojení (vytvoření objektu *Connection*) metoda vytvoří

Výpis kódu 3.8 Implementace validace Conection v Edge-Operator

```
func ValidateConnection(Connection connection){
    var errors = new([]string);

    errors += validateDeviceExists(connection);
    errors += validateDeviceIsUp(connection);
    errors += validateDeviceContainsComponents(connection);
    errors += validateComponentsAreUp(connection);
    errors += validateNadExits(connection);

    if(cloud.operator.strictValidation){
        return Fail(errors);
    }
    return Success(errors);
}
```

objekt Deployment zaobalující objekt Pod, který slouží jako proxy služba do privátní sítě k danému zařízení. Pro každé spojení je vytvořen právě jeden Pod. Každý handler definovaný v objektu *Device* odpovídá jednomu kontejneru běžícímu ve vytvořeném Podu. Mimo Deployment vyváří kontroler i objekt Service, díky které umožňuje adresovat daná zařízení pomocí DNS jména. Registrované DNS jméno zařízení, ve výchozím konfiguraci, odpovídá `dop-<name_of_service>`. Vystavené porty objektem Service odpovídají portům na daném zařízení.

Druhou implementovanou funkcionalitou je Delete. Tato metoda je volána vždy, když dojde ke smazání objektu Service.

Kontroler je konfigurovatelný, díky tomu je možné upravit jeho chování konkrétním potřebám užití. Implementace kontroleru je součástí třídy `ConnectionController`.

Pro snazší pochopení jsou uvedeny ukázkové implementace *EdgeOperator* kontroleru. Uvedené ukázky pseudokódu jsou uvedeny ve výpisu kódů 3.9 a 3.10

Výpis kódu 3.9 Implementace Reconcile v Edge-Operator

```
func Reconcile(ConnectionEntity entity){
    var name = GetProxyName(entity);
    var namespace = entity.Namespace();

    if (cloud.ConnectionExists(name, namespace)){
        var deployment = updateDeployment(name,entity);
        var service = updateService(name,entity);

        cloud.UpdateV1Deployment(deployment, namespace);
        cloud.UpdateV1Service(service, namespace);
    }
    else{
        var deployment = createDeployment(name,entity);
        var service = createService(name,entity);

        cloud.NewV1Service(deployment, namespace);
        cloud.NewV1Deployment(deployment, namespace);
    }
}
```

Výpis kódu 3.10 Implementace Delete v Edge-Operator

```
func Delete(ConnectionEntity entity){
    var name = GetProxyName(entity);

    cloud.DeleteV1Deployment(name, namespace);
    cloud.DeleteV1Service(name, namespace)
}
```

Instalace a ukázka použití operátoru

Zdrojový kód operátoru *EdgeOperator* je dostupný v přiloženém archivu, případně na platformě GitHub v repositáři `dvojak-cz/Bachelor-Thesis`. Archiv obsahuje mimo zdrojového kódu i definice prostředí (popsáno výše), dokumentaci k operátoru a vše ostatní spjaté s touto prací.

Pro jednoduché nasazení a použití operátoru lze využít dokumentaci na webu *bt.project.dvojak.cz*. Instalace operátoru je umožněna pomocí sady konfigurovatelných manifestů, které jsou součástí každého release daného repositáře. Potřebné kontejnery pro testování a instalaci operátoru jsou dostupné v kontejnerovém repositáři GitHub.

Zmíněná dokumentace obsahuje i příklad použití samotného kontroleru. Definice objektů jsou dostupné v repositáři.

Kapitola 4
Závěr

V závěru této bakalářské práce se podíváme na hodnocení dosažených výsledků a přínosů představeného řešení. Hlavním cílem této práce bylo rozšířit funkcionalitu Kubernetes o komunikaci se zařízeními mimo interní síť cloudu. Konkrétně se jednalo o zařízení v privátních síťových segmentech, která jsou propojená s klastrem pomocí jednoho či více pracovních uzlů. Tato práce byla motivována potřebami při testování HIL a integrací testování do oblasti cloudu. Nalezení řešení a následné rozšíření funkcionality Kubernetes umožňuje z prostředí cloudu testovat zařízení, které není možné připojit přímo do Kubernetes.

První část práce byla zaměřena na porozumění stávajících řešení síťování a standardů použitých pro síťování. Zároveň částečně ukázala možnost implementací síťování v systému Kubernetes. Tato část pomohla převážně k pochopeni problematiky a ujištění se, že systém Kubernetes opravdu nenabízí dostatečnou možnost adresace do přilehlých privátních síťových segmentů.

V další části se práce zaměřila na řešení daného problému. V této části byly deklarovány požadavky pro hledané řešení. Díky těmto požadavkům bylo možné hodnotit jednotlivá možná řešení, identifikovat jejich nedostatky a přínosy. Jako první byla diskutována možnost využití stávajícího projektu *kube edge*. Tento projekt nabízí možnost řešení problému, která je omezena na MQTT případně HTTP protokol. I přesto, že tato technologie nesplňuje všechny nastavené požadavky pro řešení, jedná se o velmi ambiciozní projekt. Je určitá pravděpodobnost, že projekt bude zkoumanou funkcionalitu implementovat a proto je velmi vhodné pozorovat vývoj tohoto projektu. Druhá zkoumaná možnost se zabývala možností využití proxy. To bylo určeno jako vhodné řešeni problému. Navržené řešení pomocí technologie proxy bylo do značné míry umožněné díky aktivitám komunity *Network Plumbing Working Group*.

Poslední část této práce byla zaměřena na rozšířen systému Kubernetes o možnost diskutovaného způsobu síťování. Tato kapitola popisuje konkretní řešení a návrh rozšíření. Zároveň zmiňuje prostředí, které bylo pro vývoj a testováni použito, společně s důvody volby řešení.

Výsledek práce poskytuje způsob, pro adresaci zařízeních v přilehlých privátní sítích z interní sítě Kubernetes. Konkretní řešení je navrženo tak, aby umožnilo adresaci převážně směrem z cloudu do privátní sítě. Tento směr dává pro reálně použití největší smysl. Řešení je navrženo tak, aby bylo jednoduše rozšiřitelné a podporovalo zavedené standardy v systému Kubernetes.

Výsledky této práce přinesou nové možnosti pro použití orchestrátoru v oblasti testování. Zároveň mohou pomoci při práci s IoT zařízeními v oblastech smart cities a podobných oblastech, které se dotýkají tématu edge cloud computing.

Tato práce zkoumala přímo komunikaci a adresaci s externími zařízeními. Možná rozšíření, jako podpora šifrování nebo autentizace nejsou součástí této práce. Určitě se jedná zajímavé oblasti, které dává smysl dále zkoumat. Mezi další možná rozšíření patří integrace s dalšími nástroji (dynamická konfigurace firewallu atd.). Pro potřeby HIL testování dává smysl se zabývat

spolehlivostí a efektivitou daného řešení.

Celkově lze konstatovat, že tato bakalářská práce dosáhla svého hlavního úkolu – posílit schopnosti Kubernetes v oblasti komunikace s externími zařízeními, která se nacházejí mimo interní síť cloudu. Díky výsledkům této práce se otevírají nové možnosti pro efektivní řešení problémů a významně se rozšiřuje spektrum aplikací systému Kubernetes.

Bibliografie

1. GOLDBERG, Joe. *Workflow orchestration: An introduction* [online]. DevOps Blog, 2019. [cit. 2022-12-14]. Dostupné z: `https://www.bmc.com/blogs/workflow-orchestration/`.
2. VELICHKO, Ivan. *Learning Containers From The Bottom Up* [online]. Learning Containers, Kubernetes, a Backend Development, 2021. [cit. 2023-03-13]. Dostupné z: `https://iximiuz.com/en/posts/container-learning-path/`.
3. VONDRA, Tomáš. *Kontejnery - principy a Docker*. NI-VCC. Fakulta informačních technologií České vysoké učení technické v Praze: Katedra počítačových systémů, 2022.
4. POULTON, Nigel. *Docker deep dive : zero to Docker in a single book*. Nigel Poulton, 2020.
5. FOUNDATION, The Linux. *About the open container initiative - open container initiative* [online]. opencontainers.org, [b.r.]. [cit. 2023-01-18]. Dostupné z: `https://opencontainers.org/about/overview/`.
6. INITIATIVE, Open Container. *Image Format Specification* [online]. Open Container Initiative, GitHub, 2022 [cit. 2023-01-03]. Dostupné z: `https://github.com/opencontainers/image-spec/blob/main/spec.md`.
7. INITIATIVE, Open Container. *Open Container Initiative Runtime Specification* [online]. Open Container Initiative, GitHub, 2022 [cit. 2023-01-03]. Dostupné z: `https://github.com/opencontainers/runtime-spec/blob/main/style.md`.
8. INITIATIVE, Open Container. *Open Container Initiative Distribution Specification* [online]. Open Container Initiative, GitHub, 2022 [cit. 2023-01-03]. Dostupné z: `https://github.com/opencontainers/distribution-spec/blob/main/spec.md`.
9. PROJECT, The Linux man-pages. *namespaces(7) - Linux manual page* [online]. man7.org, 2022. [cit. 2023-01-20]. Dostupné z: `https://man7.org/linux/man-pages/man7/namespaces.7.html`.
10. PROJECT, The Linux man-pages. *network_namespaces(7) - Linux manual page* [online]. man7.org, 2022. [cit. 2023-01-20]. Dostupné z: `https://man7.org/linux/man-pages/man7/network%5C_namespaces.7.html`.
11. VELICHKO, Ivan. *Container Networking Is Simple!* [online]. Learning Containers, Kubernetes, a Backend Development with Ivan Velichko, 2020. [cit. 2023-02-16]. Dostupné z: `https://iximiuz.com/en/posts/container-networking-is-simple/`.
12. VELICHKO, Ivan. *Connecting multiple network namespaces with a bridge* [online]. How Container Networking Works: Practical Explanation, 2020. [cit. 2023-04-18]. Dostupné z: `https://iximiuz.com/container-networking-is-simple/router-4000-opt.png`.

13. CONTAINERNETWORKING. *Container Network Interface (CNI) Specification* [online]. GitHub, 2022. [cit. 2022-12-29]. Dostupné z: https://github.com/containernetworking/cni/blob/main/SPEC.md.
14. POULTON, Nigel. *The Kubernetes Book*. 2023 Edition. Nigel Poulton, 2022.
15. KEBBANI, Nassim; TYLENDA, Piotr; MCKENDRICK, Russ. *The Kubernetes bible : the definitive guide to deploying and managing Kubernetes across major cloud platforms*. Packt Publishing, 2022.
16. KUBERNETES, The Authors. *Kubernetes Components* [online]. Kubernetes, 2022. [cit. 2023-03-20]. Dostupné z: https://kubernetes.io/docs/concepts/overview/components/.
17. KUBERNETES, The Authors. *Container Runtimes* [online]. Kubernetes, 2023. [cit. 2023-03-11]. Dostupné z: https://kubernetes.io/docs/setup/production-environment/container-runtimes/.
18. VELICHKO, Ivan. *Kubernetes pod.* [online]. Containers vs. Pods - Taking a Deeper Look, 2021. [cit. 2023-04-18]. Dostupné z: https://iximiuz.com/containers-vs-pods/pod-2000-opt.png.
19. KASHIN, Michael. *CNI* [online]. The Kubernetes Networking Guide, 2022. [cit. 2023-03-26]. Dostupné z: https://www.tkng.io/cni/.
20. FOUNDATION, Cloud Native Computing. *Kubernetes and the CNI: Where We Are and What's Next - Casey Callendrello, CoreOS* [online]. www.youtube.com, 2019. [cit. 2023-03-26]. Dostupné z: https://youtu.be/Vn6KYkNevBQ?t=1990.
21. KUBERNETES, The Authors. *kubectl Cheat Sheet* [online]. Kubernetes, 2023. [cit. 2023-04-07]. Dostupné z: https://kubernetes.io/docs/reference/kubectl/cheatsheet/.
22. KUBERNETES, The Authors. *Cluster Networking* [online]. Kubernetes, 2022. [cit. 2023-03-05]. Dostupné z: https://kubernetes.io/docs/concepts/cluster-administration/networking/.
23. MLEJNEK, Jiří. *Architektonické vzory*. BI-SI1. České vysoké učení technické v Praze Fakulta informačních technologií, 2018.
24. KUBERNETES, The Authors. *Service* [online]. Kubernetes, 2023. [cit. 2023-03-15]. Dostupné z: https://kubernetes.io/docs/concepts/services-networking/service/.
25. BETZ, Mark. *Service routing* [online]. Understanding kubernetes networking: services, 2017. [cit. 2023-04-18]. Dostupné z: https://miro.medium.com/v2/resize:fit:720/format:webp/1*UetnYP8uE05GAqQD0tbtBQ.png.
26. RANCHER, SUSE. *Rancher Kubernetes Engine* [online]. rke.docs.rancher.com, 2023. [cit. 2023-03-27]. Dostupné z: https://rke.docs.rancher.com/config-options/services.
27. BETZ, Mark. *Netfilter - Service* [online]. Understanding kubernetes networking: services, 2017. [cit. 2023-04-18]. Dostupné z: https://miro.medium.com/v2/resize:fit:720/format:webp/1*4XyIJ5Tdvs8f6zhLwdVt3Q.png.
28. K8S-CI-ROBOT. *Merge pull request #97075 from adtac/apfe2e-5 · kubernetes/kubernetes@af46c47* [online]. GitHub, 2020. [cit. 2023-04-19]. Dostupné z: https://github.com/kubernetes/kubernetes/commit/af46c47ce925f4c4ad5cc8d1fca46c7b77d13b38.
29. KUBERNETES, The Authors. *Ingress* [online]. Kubernetes, 2023. [cit. 2023-03-16]. Dostupné z: https://kubernetes.io/docs/concepts/services-networking/ingress/.
30. KUBERNETES, The Authors. *ingress-diagram* [online]. Kubernetes, 2023. [cit. 2023-04-18]. Dostupné z: https://d33wubrfki0l68.cloudfront.net/91ace4ec5dd0260386e71960638243cf902f8206/c3c52/docs/images/ingress.svg.
31. KASHIN, Michael. *Gateway API* [online]. The Kubernetes Networking Guide, 2021. [cit. 2023-03-26]. Dostupné z: https://www.tkng.io/ingress/gateway.

32. KUBERNETES, The Authors. *Ingress* [online]. Kubernetes, 2022. [cit. 2023-03-27]. Dostupné z: https://kubernetes.io/docs/reference/kubernetes-api/service-resources/ingress-v1/.
33. KOKEŠ, Josef. *Bezpečné použití kryptografie*. BI-BEK. České vysoké učení technické v Praze Fakulta informačních technologií: Katedra informační bezpečnosti, 2023.
34. YAMAMOTO, Hirotaka. *Introducing Coil v2, a Kubernetes network plugin to build Egress gateway* [online]. Kintone Engineering Blog, 2020. [cit. 2023-03-27]. Dostupné z: https://blog.kintone.io/entry/coilv2.
35. KUBERNETES, The Authors. *Network Policies* [online]. Kubernetes, 2022. [cit. 2023-03-07]. Dostupné z: https://kubernetes.io/docs/concepts/services-networking/network-policies.
36. BIGELOW, Stephen. *What is edge computing? Everything you need to know* [online]. Tech Accelerator, 2021. [cit. 2023-04-15]. Dostupné z: https://www.techtarget.com/searchdatacenter/definition/edge-computing.
37. AUTHORS, KubeEdge Project. *KubeEdge* [online]. KubeEdge, 2023. [cit. 2023-04-19]. Dostupné z: https://kubeedge.io/en/.
38. TTLV. *servicebus.go* [online]. GitHub, 2021. [cit. 2023-03-19]. Dostupné z: https://github.com/kubeedge/kubeedge/blob/master/edge/pkg/servicebus/servicebus.go.
39. FOUNDATION, Cloud Native Computing. *Intro: KubeEdge - Cindy Xing, Futurewei & Dejan Bosanac, Red Hat* [online]. youtube.com, 2019. [cit. 2023-03-17]. Dostupné z: https://www.youtube.com/watch?v=pdqlANkpOMs.
40. FOUNDATION, Cloud Native Computing. *Intro to KubeEdge: Kubernetes Native Edge Computing Framework - Kevin Wang (Zefeng) & Yin Ding* [online]. youtube.com, 2022. [cit. 2023-03-19]. Dostupné z: https://youtu.be/XFaPZbOwgEM.
41. BETZ, Mark. *Understanding kubernetes networking: ingress* [online]. Google Cloud - Community, 2022. [cit. 2023-02-14]. Dostupné z: https://medium.com/@betz.mark/understanding-kubernetes-networking-ingress-1bc341c84078.
42. KUBERNETES, The Authors. *Network Plugins* [online]. Kubernetes, 2022. [cit. 2023-03-28]. Dostupné z: https://kubernetes.io/docs/concepts/extend-kubernetes/compute-storage-net/network-plugins/.
43. WOODS, Natasha. *A look back at KubeCon + CloudNativeCon North America 2017 – Part 1* [online]. Cloud Native Computing Foundation, 2018. [cit. 2023-04-07]. Dostupné z: ttps://www.cncf.io/blog/2018/01/10/look-back-kubecon-cloudnativecon-north-america-2017-part-1/.
44. FESL, Jan. *Aplikační vrstva, protokoly a služby*. BI-PSI. České vysoké učení technické v Praze Fakulta informačních technologií: Katedra počítačových systémů, 2021.
45. AMOANY, Evans. *Getting started with socat, a multipurpose relay tool for Linux* [online]. Enable Sysadmin, 2020. [cit. 2023-04-12]. Dostupné z: https://www.redhat.com/sysadmin/getting-started-socat.
46. HAYASHI, Tomofumi. *Multus-CNI* [online]. GitHub, 2019. [cit. 2023-03-28]. Dostupné z: https://github.com/k8snetworkplumbingwg/multus-cni.
47. AUTHORS, Kubernetes The. *Extending Kubernetes* [online]. Kubernetes, 2023. [cit. 2023-04-22]. Dostupné z: https://kubernetes.io/docs/concepts/extend-kubernetes/.
48. NGUYEN, Tu. *A deep dive into Kubernetes controllers* [online]. Understand Kubernetes Controller, 2017. [cit. 2023-04-06]. Dostupné z: https://docs.bitnami.com/tutorials/a-deep-dive-into-kubernetes-controllers/%5C#controller-pattern.

49. BÜHLER, Christoph. *KubeOps - Kubernetes Operator SDK* [online]. buehler.github.io, 2021. [cit. 2023-04-19]. Dostupné z: `https://buehler.github.io/dotnet-operator-sdk/kubeops.html`.
50. KAPLAN, Věroš. *ArgoCD: GitOps v Kubernetes*. 2022. Dis. pr.
51. VELICHKO, Ivan. *Exploring Kubernetes Operator Pattern* [online]. Learning Containers, Kubernetes, a Backend Development, 2021. [cit. 2023-04-02]. Dostupné z: `https://iximiuz.com/en/posts/kubernetes-operator-pattern/`.
52. FOUNDATION, Cloud Native Computing. *Tutorial: Zero to Operator in 90 Minutes! - Solly Ross, Google* [online]. youtube.com, 2020. [cit. 2023-03-27]. Dostupné z: `https://youtu.be/KBTXBUVNF2I`.
53. BÜHLER, Christoph. *KubeOps* [online]. GitHub, 2023. [cit. 2023-04-26]. Dostupné z: `https://github.com/buehler/dotnet-operator-sdk`.
54. STACKOVERFLOW. *2022 Developer Survey* [online]. StackOverflow, 2023. [cit. 2023-04-20]. Dostupné z: `https://survey.stackoverflow.co/2022/%5C#most-popular-technologies-webframe-prof`.

Obsah přiloženého archivu

- `ansible/` — zdrojové kódy pro ansible
  - `playbook/` — definice prostředí pomocí ansible
  - `inventory/` — seznam hosts pro andisble
  - `vars/` — proměnné pro ansible playbooks
- `code/` — zdrojové kódy
  - `EdgeOperator/` — zdrojové kódy operatoru
    - `EdgeOperator/` — operátor
    - `EdgeOperator.Tests/` — unit testy
    - `EdgeOperator.sln`
  - `sampleSrvers/` — zdrojové kódy pomocných programů pro testování
    - `tcpServer/`
    - `udpServer/`
- `doc/` — návod na instalaci prostředí a instalaci operatoru
- `manifests/` — manifesty pro Kubenretes
  - `lab/`
  - `operator/`
- `scripts/` — pomocná scripty pro instalaci prostředí
- `text/` — text bakalářské práce
- `vagrant/` — adresář obsahující definice virtuálního prostředí
- `.github/` — definice pipelines na GitHub
- `readme.md` — stručný popis repositáře