

UNIVERZITA HRADEC KRÁLOVÉ
FAKULTA INFORMATIKY A MANAGEMENTU
KATEDRA INFORMAČNÍCH TECHNOLOGIÍ

DIPLOMOVÁ PRÁCE

Implementace hybridního multi-cloud konceptu pro
běh distribuovaných aplikací v kontejnerech a
virtuálních serverech do enterprise prostředí

Autor: Jan Cach

Studijní obor: Aplikovaná informatika

Vedoucí práce: Ing. Jakub Pavlík, MSc.

Hradec Králové, 2019

Prohlašuji, že jsem diplomovou práci vypracoval samostatně a uvedl jsem všechny použité prameny a literaturu.

V Hradci Králové dne April 24, 2019

Jan Cach

Rád bych poděkoval Ing. Jakubu Pavlíkovi, MSc. za odborné vedení práce, podnětné rady a čas, který mi věnoval.

Anotace

Tato bakalářská práce pojednává o kontejnerové virtualizaci, zabývá se problematikou nasazení kontejnerů do standardního firemního prostředí. První část je věnována představení virtualizace obecně a její porovnání s kontejnerovou virtualizací. Následně je řešena problematika orchestrace a ovládání kontejnerového prostředí. Čtvrtá a pátá kapitola je věnována analýze a převodu aplikace do kontejnerů, součástí kapitoly pět je praktická ukázka funkčnosti orchestrátoru. Cílem této práce je seznámení čtenáře s konceptem kontejnerů a zjistit jejich využitelnost v produkčním prostředí a možnost převodu klasické aplikace do kontejnerů.

Klíčová slova

Cloud computing, Cloud native, Multi cloud, Kubernetes

Annotation

Title: Evaluation of Container-based Virtualization for standard company environment

This Bachelor's thesis describes container-based virtualization especially the issues in its use in standard company environment. The first part is dedicated to introducing general virtualization and its comparison to container-based virtualization. Afterward is tackled the troubleshooting of orchestration and control of the container-based environment. The fourth and fifth chapters are devoted to the analysis and conversion of application into containers. Practical example of the orchestration's functionality is included in chapter five. The aim of this work is to familiarize the reader with the concept of containers and to find out their applicability in the production environment as well as the ability to convert a classic application into containers.

Key words

Cloud computing, Cloud native, Multi cloud, Kubernetes

Contents

1 Úvod	1
2 Cloud computing	3
2.1 Dělení cloud computingu	5
3 Cloud-Native infrastruktura a aplikace	7
3.1 Stavební bloky cloud native aplikace	8
3.1.1 Architektura mikroslužeb	9
3.1.2 Kontejnery	11
3.1.3 Orchesrátory	15
3.2 Edge cloud	24
4 Existující řešení pro hybrid a multi cloud	27
4.1 Red Hat Cloudforms/Manage IQ	29
4.2 Cloud foundry	30
4.3 Mist.io	31
4.4 Kubernetes federation	33
4.5 Spinnaker	34
5 Návrh a implementace systému pro běh distribuovaných aplikací	36
5.1 Definice požadavků na běh distribuované aplikace	36
5.2 Návrh aplikace	39
5.3 Implementace prototypu aplikace	41
6 Testování aplikace	48
6.1 Definice prostředí pro testování aplikace	48
6.2 Testování distribuce Kubernetes objektů	48
6.2.1 Distribuce Kubernetes node objektů	49
6.2.2 Distribuce Kubernetes pod objektů	50
6.2.3 Distribuce Kubernetes deployment objektů	52
6.2.4 Distribuce Kubernetes statefulset objektů	57
7 Závěry a doporučení	60
Literatura	62
Přílohy	62

1 Úvod

V současné době dochází na poli návrhu a vývoje počítačových systémů k výraznému posunu. Příchod cloud computingu a později cloud native přístupu transformoval základní principy návrhu a běhu aplikací. Namísto fyzické infrastruktury, vývojáři implementují služby nad sofistikovanou virtualizovanou platformou, která přináší výhody oproti klasickému přístupu. První generace systémů, které běželi v cloudovém prostředí se nazývaly “cloud enabled” systémy. Tyto systémy, typické pro cloud computing éru, byly přeneseny do prostředí cloudu, ale byly provozovány velice podobně jako v období před cloud computingem. Z tohoto důvodu aplikace plně nevyužívaly výhody, které jim cloud computing nabízel. Pro překonání těchto problémů se vytvořily nové přístupy a technologie v tvorbě aplikací, souhrně nazývané cloud native [1].

Cloud native přístup k tvorbě a správě aplikací zasahuje do celého životního cyklu aplikace. Architektura cloud native aplikací je založena na mikroslužbách, které rozdělují aplikace na menší vzájemně nezávislé celky. Tyto celky se lépe rozšiřují, udržují a škálují oproti monolitickým aplikacím. Dalším stavebním blokem cloud native přístupu jsou kontejnery. Kontejnery v sobě zapouzdřují závislosti aplikací a napomáhají přenositelnosti aplikací mezi testovacím a produkčním prostředím. Pro správu cloud native aplikací jsou používány orchestrátory kontejnerů, které přináší jednoduchou a automatizovanou práci s kontejnery.

Cloud native přístup má více oblastí použití. První oblastí jsou aplikace, které vyžadují vysokou dostupnost a škálovatelnost. Vysoké dostupnosti může být dosaženo s využitím orchestrátorů kontejnerů, které sledují stav jednotlivých mikroslužeb a v případě výpadku jsou schopné nahradit chybné kontejnery. Architektura mikroslužeb dovoluje škálovat pouze potřebnou část aplikace. S rozvojem Internetu věcí (IoT) se cloud native přístup prosazuje i v prostředí Edge computingu. Cílem Edge computingu je zpracovávat data, co nejbližší k jejich zdroji a umožnit tak jejich rychlé zpracování a snížení doby odezvy. Stejná situace panuje i v prostředí multi-cloud. Společnosti mají zájem jednotně spravovat zdroje a využívat služby napříč více poskytovateli cloudových služeb. Společnosti jako Google, Amazon, Red Hat a spousta dalších se snaží přijít s vlastním řešením, které nabídne jednotnou správu zdrojů a aplikací. Z těchto důvodů se tato oblast stále rozvíjí a nabízí tak prostor pro nové technologie.

Cílem této diplomové práce je analyzovat a navrhnout platformu, která bude umožňovat běh distribuovaných aplikací v prostředí multi-cloudu. Řešení práce spočívá ve vytvoření požadavků pro takovou platformu, navrhnutí architektury dané platformy a vytvoření prototypu systému, který bude společně s jejím testováním výstupem této práce.

Diplomová práce se v teoretické části zaměřuje na vývoj cloud computingu. V první kapitole je popsán princip cloud computing. Druhá kapitola navazuje na cloud computing a představuje cloud native computing jako nové cloudové paradigma. V kapitole jsou představeny základní principy a technologie cloud native computingu a v čem se odlišuje od předchozího přístupu. Kapitola dále představuje oblasti využití cloud native přístupu v podobě edge cloudu a hybrid cloudu. V podkapitole hybrid cloud jsou dále rozebrány existující nástroje pro tuto oblast. Poznatky z teoretické části jsou dále využity pro definici požadavků na aplikaci, která umožní běh distribuovaných aplikací. Praktická část práce se dále zabývá návrhem takového systému a implementací prototypu. V poslední části práce jsou uvedeny výsledky z testování vytvořené aplikace, které mají za úkol ověřit funkčnost navržené aplikace.

2 Cloud computing

Označení cloud computing se začalo objevovat okolo roku 2008[2]. Někteří odborníci považovali cloud computing model za nové paradigma, někteří dokonce mluvili o nové technologii, která umožňuje přístup k výpočetním zdrojům a službám přes internet [3]. Cloud computing, často také označovaný pouze jako cloud, tak dovoluje jednotlivcům, malým firmám a dalším subjektům jednoduchý přístup k výpočetnímu výkonu z pohodlí domova či kanceláře za přijatelnou cenu, bez nutnosti nakupovat, obměňovat a spravovat celou výpočetní infrastrukturu. Uživatelé jsou tak odstíněni od konfigurace serverů, síťových zařízení a služeb samotných.

Podle [4] lze na cloud computing nahlížet jako na využívání elektřiny. Elektřinu využíváme jako službu. Nezajímáme se o to jak se elektřina vyrábí a jak je dodávána do jednotlivých elektrických zásuvek v našem pokoji. My pouze připojíme zařízení do elektrické zásuvky a očekáváme, že například nabije náš telefon či počítač. Pokud tento příklad přeneseme do oblasti IT, znamená to, že uživatelé jsou odstíněni od vnitřního fungování nějaké služby nebo použitých technologií. Jako příklad může být brána služba cloudového úložiště fotografií. Pro uživatele je důležité, že si může fotografie prohlížet z různých zařízení přes internet odkudkoliv a kdykoliv. Ostatní problémy jako jsou uložení a záloha těchto dat, webové rozhraní pro práci s fotografiemi a další je ponecháno na poskytovateli služby.

Některé instituce, akademičtí pracovníci či IT inženýři vytvořili definice a charakteristiky popisující cloud computing. Například Americký národní institut standardů a technologií (NIST) [5], definuje cloud computing následovně. Cloud computing je model pro pohodlný síťový přístup ke skupině konfigurovatelných výpočetních zdrojů, jako jsou počítačové sítě, servery, datová úložiště, aplikace a služby, které jsou dostupné odkudkoliv a okamžitě na vyžádání. Tyto zdroje mohou být rychle vytvořeny a uvolněny s vyvinutím minimálního úsilí nebo minimální interakce od provozovatele dané služby.

Definice ze zdroje [6] zmiňuje, že uživatelé platí pouze za zdroje, které skutečně využívají podle sjednaných podmínek. Tento postup se nazývá pay-per-use model. Znamená to žádné pevně stanovené poplatky bez ohledu na to zda zákazník využívá polovinu přidělených zdrojů nebo se využití zdrojů blíží 100%. Díky tomu je možné si například vypočítat náklady na provozování služby u různých firem a rozhodnout se

tak pro nejvhodnější platformu.

Armbrust a jeho kolegové z Kalifornské Univerzity v Berkeley [7], shrnují cloud computing do 3 bodů. Prvním bodem je zdání, že uživatel má k dispozici nekonečný objem výpočetních zdrojů. Z tohoto důvodu není potřeba dopředu plánovat zda bude k dispozici dostatek zdrojů. Zdroje jsou dynamicky přidávány a odebírány na požádání. Druhým bodem je skutečnost, že není potřebné uzavírat žádné předběžné závazky ze strany uživatele. Toto dovoluje firmám začít s malým počtem zdrojů a postupně navyšovat zdroje s rostoucími potřebami. Třetím a posledním bodem je zde platit za využívané zdroje ve velice krátkém horizontu, např. hodin nebo dní a poté je uvolnit podle potřeb.

Z výše zmíněných definic vyplývá, že cloud computing umožňuje vzdálený přístup k výpočetním zdrojům. Uživatelé tak mohou přistupovat k aplikacím a datům z různých zařízení a lokací. K tomu využívají jediný účet a na základě definovaných práv mají přístup do různých částí systému. Dalším plusem je jednoduché přidání a odebrání zdrojů podle potřeby, s tím spojené placení pouze za skutečně využívané zdroje a jenom po dobu používání. Tato vlastnost cloudu je velký posun. Společnosti nemusí dlouhodobě plánovat nákup a výměnu hardwaru. Využívají zdroje, které aktuálně potřebují. Náklady za zdroje se odvíjejí od momentálního vytížení. Pokud je potřeba zvýšit počet zdrojů, stane se tak automaticky podle předem připraveného scénáře. Například vytvoření dalších webových serverů pro rozložení zátěže a odbavení uživatelů při špičce. Poté zase vypnutí nevyužitých zdrojů v době mimo špičku. Spolehlivost a vysoká dostupnost takového řešení je další výhodou cloudu. Poskytovatel služby má spoustu odborníků na danou problematiku, pravidelně zálohuje data, obnovuje hardware a celkově se stará o chod a vylepšování služeb, které nabízí. Cloud computing při všech těchto výhodách není pouze pro velké a bohaté společnosti. Firmy přecházející do prostředí cloudu nemusí řešit velké počáteční investice. Když se rozhodnou, že chtějí vybranou službu začít využívat mohou začít téměř ihned, stejně tak jako přestat službu využívat ze dne na den.

Tyto prvky cloud computingu jsou možné díky technologiím jako je virtualizace, automatizace a orchestrace zdrojů a řešení pro vysokou dostupnost. Virtualizace je technika, která umožňuje běh více virtuálních serverů na jednom fyzickém serveru. Fyzický server emuluje pro každý virtuální server hardware, to znamená procesor, RAM paměť, síťovou kartu a další. Uživatel vnímá virtuální server jako hardwarový server a nepozná rozdíl. Přitom jednotlivé virtuální servery jsou od sebe navzájem izolované. Virtualizace umožňuje lepší využití zdrojů a automatizaci vytváření virtuálních serverů. Virtualizace se ale netýká pouze virtuálních serverů, ale také počítačových sítí a síťových prvků a také datového uložení. Jednotlivé koncepty jsou představeny v [8].

Aby mohli uživatelé dynamicky pracovat se zdroji, je potřebné automatizovat správu

jednotlivých zdrojů. Každý poskytovatel cloudového řešení představuje vlastní přístup k tomuto problému. Uživatelé mohou poté přes webový portál či API dynamicky spravovat zdroje bez zásahu poskytovatele cloudového řešení. Přidání, změna či odebrání zdrojů se děje bez zásahu poskytovatele. Jednotliví uživatelé jsou účtováni na základě využívání jednotlivých zdrojů.

2.1 Dělení cloud computingu

Cloud computing můžeme rozdělit do tří kategorií podle služeb, která má daný model poskytovat. Zmíněnými modely jsou Infrastructure as a service (IaaS), Platform as a service (PaaS) a Software as a service (SaaS). Dalším aspektem podle kterého může být cloud computing dělen je způsob nasazení daného servisního modelu. Servisní modely určují jaké služby budou využívány. Tyto modely se mohou dále lišit podle způsobu nasazení. Zde jsou 3 základní modely nasazení. Private cloud vlastněný organizací, public cloud sdílený více organizacemi a hybrid cloud kombinující private a public cloud.

Takovéto dělení cloud computingu bylo relevantní v době svého vzniku. Dnes se jednotlivé rozdíly smazávají, dělení není tak striktní a trendem je propojení více modelů dohromady tak, aby uživatelům cloudů nabídlo výhody každého z řešení. Příkladem může být například banka HSBC. Tato banka se snaží integrovat nejmodernější technologie, které jsou dostupné tak aby z toho měla co největší přidanou hodnotu pro svoje zákazníky. Tato společnost využívá AWS (Amazon web services) cloud. Amazon například pomohl s přesunem dat z privátního datacentra. Toto spojení pomáhá HSBC nasadit aplikace v rámci sekund místo měsíců. IaaS a PaaS modely také ztrácí na svém významu. Např. HSBC používá AWS Lambda službu. Tato služba abstrahuje uživatele od správy serverů a zaměřuje se pouze na vykonání funkce. Uživatel tak pouze pošle data do Lambdy a zpět obdrží vypočtená data nebo relevantní odpověď. Takovéto řešení velice dobře škáluje a umožňuje souběžný výpočet pro velké množství transakcí [9]. Dalším aspektem je využití multi cloud řešení. Využití více jak jednoho cloudového řešení. Příkladem může být opět banka HSBC, která využívá kombinaci AWS a GCP (Google Cloud platform). GCP nabízí nástroje pro analýzu dat a strojové učení, které HSBC používá pro podporu vnitřních rozhodovacích procesů a analýzu velkého objemu dat [10]. Z toho můžeme vyvodit, že dnes společnosti využívají výhody jednotlivých poskytovatelů služeb a kombinují je tak, aby bylo dosaženo co nejefektivnějšího řešení, které bude dané společnosti vyhovovat, protože jeden poskytovatel nepokryje všechny specifiky. Společnosti tak mají svobodu ve výběru technologií od různých poskytovatelů cloudových služeb.

Dalším důvodem pro využití multi cloud řešení je i legislativa, nezávislost pouze

na jednom poskytovateli služby a dostupnost aplikací. Legislativa například omezuje v jaké geografické lokaci mohou být data občanů dané země uloženy. Toto nařízení se týká především nadnárodních společností, které působí po celém světě. Například podle přehledu v tabulce 2.1 vidíme jednotlivé regiony cloudových poskytovatelů AWS, Azure, Google cloudu a Alibaba cloud. Z tabulky je patrné, že každý poskytovatel má své datové centrum ve velkých evropských zemích jakou jsou Německo, Francie a Velká Británie. Dále každý poskytovatel přidává lokaci v menší zemi. AWS má datové centrum ve Švédsku, Azure ve Švýcarsku a Google cloud ve Finsku. Vyjímkou je Alibaba cloud. Tento čínský poskytovatel cílí hlavně na asijský trh a v Evropě tak zatím nemá tak velké zastoupení. Multi cloud řešení minimalizuje závislost na jednom poskytovateli. Je to ochrana před radikální změnou politiky daného poskytovatele, případně zmenšuje škodu způsobenou výpadkem služeb daného poskytovatele.

AWS	AZURE	GCP	Alibaba cloud
Irsko	Německo	Londýn	Londýn
Londýn	Francie	Hamina, Finsko	Asie
Paříž	Londýn	Emshaven, Nizozemsko	
Stockholm, Švédsko	Nizozemsko	Belgie	
	Švýcarsko		

Tabulka 2.1: Tabulka lokací datových center různých poskytovatelů

3 Cloud-Native infrastruktura a aplikace

Cloud native strategii můžeme považovat za nástupce cloud computingu. Cílem cloud native není pouze běh aplikací v prostředí cloudu, ale zaměřuje se na celý životní cyklus aplikace od architektury aplikace, nasazení aplikace, škálovatelnost, doručování nových verzí a také monitoring aplikace a statistiky používání. Tento přístup by měl umožňovat správu velkých a komplexních aplikací, které jsou ovšem přizpůsobené prostředí cloudu a využívají možnosti, které cloud nabízí. Součástí Cloud native je také přechod od velkých monolitických aplikací na menší microservice. Monolitické aplikace byly často migrovány do cloudového prostředí pouze jako monolitické virtuální servery. Takováto aplikace není odolná vůči výpadkům v dostupnosti a také nárokům na škálovatelnost [11] a nevyužívá tak plně možností které prostředí cloudu nabízí. Cloud native přístup se snaží všechny tyto nevýhody odstranit.

S vývojem cloud computingu se měnil i pohled na servery (fyzické servery, virtuální servery nebo kontejnery) a jejich správu. Podle analogie s Pets and Cattle [12] pokud vidíme server jako něco, co může být kdykoliv zničeno a nahrazeno, potom mluvíme o stádu. Pokud vidíme server jako nepostradatelný, tedy jeho ztráta by byla kritická poté mluvíme o mazlíčkovi (Pet). Pokud se podíváme na tuto zvířecí analogii blíže, za pets, neboli mazlíčky považujeme servery, které jsou v naší infrastruktuře nepostradatelné nebo to jsou unikátní systémy, které musí vždy fungovat. Takovéto servery jsou manuálně nakonfigurovány a spravovány, z toho důvodu o ně nechceme přijít a děláme vše pro to aby nepřestaly fungovat. Do této skupiny patří mainframy, loadbalancery, firewally a také master-slave databázové systémy. Na druhé straně máme servery, které jsou vytvořeny pomocí nástrojů pro automatizaci. Pokud jeden nebo více serverů přestane fungovat, můžeme je nahradit novými servery se stejnou konfigurací. Zde již není potřeba starat se o každý server samostatně, pokud nastane problém server je restartován nebo nahrazen novým serverem. Sem můžeme zařadit webové servery případně clustrované databáze. V tomto případě máme například více webových serverů, pokud jeden vypadne ostatní převezmou kontrolu. Poškozený server je poté nahrazen nově vytvořeným webovým serverem se stejnou konfigurací.

Podle Cloud native computing foundation (CNCF) dovoluje cloud native přístup organizacím vytvářet a provozovat škálovatelné aplikace v moderním a dynamickém prostředí jako je public, private nebo hybrid cloud. Technologie jako jsou kontejn-

ery, service meshes, microservices, neměnné infrastruktury a deklarativní API ilustrují cloud native přístup. Tyto techniky umožňují volně provázané systémy, které jsou flexibilní, spravovatelné a dají se monitorovat. V kombinaci se solidní automatizací dovolují vývojářům dělat předvídatelné, velké a časté změny s minimální námahou [13].

CNCF je organizace, která se snaží definovat a sjednocovat cloud native standardy a zastřešovat jednotlivé open source technologie celého cloud stacku. To znamená, že zde nalezneme technologie runtime kontejnerů, nástroje pro orchestraci kontejnerů, technologie pro komunikaci mezi jednotlivými službami, bezpečnost, uložení pro cloud native prostředí a nástroje pro logování a monitoring služeb. CNCF také podporuje komunitu okolo cloud native technologií a snaží se také podporovat spolupráci mezi vývojáři, koncovými uživateli a také mezi výrobci, kteří se mohou potkat a diskutovat nové technologie, trendy a postupy na pravidelných konferencích. Organizace také nabízí konzultace, tréninky a certifikace ať už jednotlivcům z řad vývojářů tak i zaměstnancům společností, kteří se mohou naučit pracovat s novými technologiemi a začít tyto nabyté znalosti uplatňovat v byznysu svých společností a šířit dále mezi své spolupracovníky.

Studie [14] se zabývá vysvětlením pojmu cloud native a jeho lepším porozuměním. Tato studie definuje cloud native aplikaci jako distribuovaný, flexibilní a horizontálně škálovatelný systém skládající se z microservices, které izolují stav aplikace v minimu stavových komponent. Aplikace a každá její část jsou navrženy podle vzorů zaměřených na prostředí cloudu na samoobslužné flexibilní platformě.

3.1 Stavební bloky cloud native aplikace

Pokud mají být cloud native aplikace dobře škálovatelné, rozšiřitelné a spravovatelné, potřebují být k tomuto účelu navrženy. S monolitickou architekturou není možné tyto požadavky naplno splnit. Cloud native přístup tak přichází s microservices architekturou, která rozděluje aplikaci na menší aplikace, které se lépe škálují, spravují a testují. Tato architektura je připravená plně využít všech možností prostředí cloudu. Jestliže dříve byli virtuální servery základní jednotkou nasazení aplikace do cloud prostředí, dnes v době cloud native aplikací jsou to kontejnery. Kontejnery jsou méně náročné na zdroje a startují rychleji v porovnání s virtuálními servery. Poskytují izolaci pro jednotlivé microservisy a jsou základním jednotkou nasazení. Jelikož má aplikace všechny potřebné závislosti zabalené uvnitř kontejneru je možné aplikace nasadit v jiném prostředí bez komplikací. Dalším krokem je nasazení a správa aplikace. Pro tuto část se používají orchestrátory kontejnerů. Orchestrátory obstarávají spoustu věcí od spuštění jednotlivých kontejnerů, monitorování jejich stavu, doručování nových verzí přes komunikaci mezi kontejnery, load balancing, bezpečnost a řízení přístupu až po monitor-

ing a další statistiky celé aplikace. Orchestrátory nabízejí možnost nahradit jednotlivé služby z předchozího výčtu jinou technologií pomocí pluginů. V následující části jsou detailně představeny jednotlivé technologie celého cloud native stacku.

3.1.1 Architektura mikroslužeb

Microservice architektura vznikla jako reakce na monolitické aplikace, které se s rostoucí velikostí aplikace, počtem uživatelů aplikace a komplexností celého systému staly hůře škálovatelné, spravovatelné, rozšiřitelné a testovatelné. Velké společnosti jako přepravní služba Uber [15], online distributor hudby SoundCloud [16], společnost Groupon zabývající se online obchodováním [17] a také online distributor filmů a seriálů Netflix se rozhodli přejít od monolitické architektury na microservices architekturu a plně tak využít možností cloudu. Zmíněné společnosti se staly průkopníky v použití microservice architektury pro provozování aplikací a zároveň přinesly ostatním společnostem nové poznatky a technologie, které řeší problémy na které během vývoje narazily. Jedním z příkladů může být technologie ChaosMonkey vytvořená společností Netflix [18]. Tato technologie náhodně vypíná produkční servery a kontejnery a simuluje tak výpadek služeb. Tento mechanismus nutí zaměstnance navrhovat a upravovat aplikace tak, aby takovéto výpadky bez následků přečkaly.

Microservices architektura je ze své podstaty protiklad monolitické aplikace. Microservice rozdělují aplikaci na menší celky, které spolupracují, aby dosáhly výsledné funkcionality. Monolitická aplikace je zase tvořena jako jeden velký celek. Pro nastínění pojmu monolitická aplikace můžeme například využít článek od pracovníků ze společnosti Google [19]. Monolitická aplikace je doručena jako celek, například jeden WAR soubor v jazyce Java nebo .NET webová aplikace. Takové aplikace se obvykle skládají ze tří vrstev. První je databázová vrstva pro přístup k datům, dále vrstva logiky aplikace a prezentační vrstva zodpovědná za zobrazování dat uživatelům. Monolitické aplikace se drží objektově orientovaného přístupu, jsou komplexní a vykazují velkou vnitřní provázanost jednotlivých tříd systému.

Microservices je nová architektura cloud native aplikací navržená a optimalizovaná pro cloud, která zmíněné problémy řeší. Cloud native aplikace se skládají z malých služeb, které dohromady vykonávají funkcionalitu celého systému. Každá služba (microservice) vykonává pouze svou činnost, část logiky aplikace, pro kterou je určena a má definované API pomocí kterého mezi sebou jednotlivé služby komunikují [20].

Rozdělení aplikace na jednotlivé menší jednotky, microservices, dělá aplikace lépe implementovatelné a srozumitelnější. Jednotlivé části aplikace jsou na sobě nezávislé, mají oddělené zdrojové kódy a mohou být napsané v rozdílných programovacích jazycích. Díky tomu je vývoj rychlejší, stejně jako nasazení nové verze dané microservice. Přidání či změna funkcionality neovlivňuje celý systém, ale pouze malou část

systému. Jednotlivé služby mezi sebou komunikují pomocí definovaných API volání. Služba při volání na API ví jaká data musí poslat a jaká data má očekávat. Toho je dosaženo s využitím IDL nástrojů (Interface definition language). IDL dovoluje definovat nezávisle na použitém programovacím jazyku metody, které API nabízí, dále jaké parametry jednotlivá volání přijímají a jaký je formát odpovědi na dané API volání. IDL poté dovoluje vygenerování serverové části aplikace a klientské části aplikace pro libovolný podporovaný jazyk, které vychází z jednoho zdroje a jejich případná úprava se provádí pouze na jednom místě. Díky tomu jsou vyřešeny služby, které API nabízí a formát zpráv, které budou mezi klientem a serverem posílány. Mezi zástupce patří například Apache Thrift [21] nebo Protocol buffers.

V porovnání s monolitickou aplikací umožňuje microservice architektura rychlejší a častější doručování změn. Při změně jedné služby systému stačí pouze vytvořit a otestovat novou verzi změněné části. Na druhé straně změna v monolitické aplikaci vyžaduje vytvoření a otestování celého monolitu, což je časově náročnější a celý proces doručení záplat a nové funkcionality se časově prodlužuje.

Dalším benefitem microservices architektury je škálovatelnost jednotlivých částí aplikace. Při velké zátěži se efektivně škáluje pouze vytěžovaná část aplikace, tedy určitá microservice a ne celá aplikace. Například počet API naší služby, která přijímá požadavky se může měnit v závislosti na vytíženosti, ostatní části aplikace se mohou škálovat nezávisle na ostatních částech. Toto je velice ekonomické a flexibilní v prostředí cloudu v porovnání s monolitickou aplikací. Škálujeme pouze využívanou microservice a ne celý systém. Pokud chceme škálovat pouze malou část monolitické aplikace je nutné nastartovat celou novou aplikaci což je pomalé a neflexibilní. Přidání jedné instance microservice je efektivnější a méně náročné na zdroje, protože jedna microservice bude využívat znatelně méně výpočetních zdrojů než nová instance celé monolitické aplikace.

Jednou z dalších nevýhod monolitické aplikace je případná změna či využití nových technologií. Vnitřní provázanost monolitické aplikace velice ztěžuje použití jiných technologií. Vývojáři jsou tak většinou od nějakého bodu vývoje závislí na zvolené technologii a změna monolitu znamená rozsáhlé změny. Na druhé straně microservice dovolují změnu technologie podle potřeby modifikací pouze jediné služby. Je tedy možné rychleji reagovat na nové trendy a adoptovat nové technologie.

Jako každá jiná technologie i microservices sebou přináší řadu problémů, které je nutné řešit. Jedním z nich je otázka jak efektivně a nejlépe automatizovaně spravovat velké množství microservices. Další otázky, které tato architektura přináší jsou bezpečná výměna dat mezi jednotlivými službami, jak se budou řešeny pomalé či nedostupné služby a v neposlední řadě jak budou všechny služby monitorovány. Pro vyřešení těchto překážek se objevily a stále objevují nové technologie, které zmíněné

otázky řeší. Problematika jednotlivých otázek je řešena v dalších částech této kapitoly.

3.1.2 Kontejnery

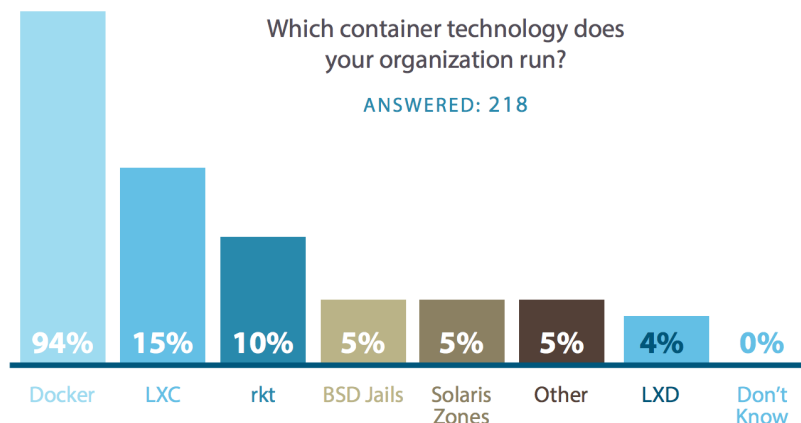
V předchozí kapitole byla představena architektura microservices, která rozděluje aplikaci na nezávislé menší části, kde každá část je nasazena jako samostatná aplikace. Každá část aplikace je zabalena v kontejneru. Kontejner, jako základní jednotka instalace, v sobě zabaluje všechny závislosti aplikace a poskytuje izolaci jednotlivých služeb. Tento koncept malých a přenositelných kontejnerů se skvěle hodí pro využití v cloud native aplikacích.

Kontejnerovou virtualizací, zkráceně pouze kontejnery, označujeme nenáročný mechanismus, který izoluje jednotlivé běžící procesy. Takto izolované procesy mohou interagovat pouze s definovanými procesy a využívat pouze přiřazené zdroje. Na jednom serveru, který řídí kontejnery, může být v kontejnerech spuštěno mnoho aplikací. Tyto aplikace nevidí ostatní aplikace, jejich procesy, soubory ani síťovou komunikaci a fungují nezávisle na ostatních aplikacích [22].

Kontejnery jsou v porovnání s virtuálními servery méně náročné na zdroje. Virtuální servery emulují celý operační systém a všechny jeho části jako jsou CPU, RAM, disky, síťové prvky a další. Na druhé straně kontejnery sdílejí jednotlivé zdroje jako je jádro systému, RAM, ale také různé knihovny, a běží stejně jako ostatní procesy operačního systému s tím rozdílem, že jsou izolované. Díky tomu startují kontejnery rychleji a je možné na serveru provozovat větší množství kontejnerů než virtuálních serverů. Z toho vyplývá, že kontejnery se hodí pro rychlé a snadné škálování aplikace jsou také odolné vůči výpadkům. Pokud je potřeba zvýšit počet instancí aplikace stačí spustit další kontejnery a začít na ně směřovat uživatele. Na druhé straně pokud dojde zastavení aplikace v kontejneru, stačí kontejner zničit a nastartovat nový kontejner. Ztráta kontejneru není nijak zásadní, nově nastartovaný kontejner převezme jeho roli. V tomto přístupu jsou kontejnery viděny jako cattle neboli dobytek, pokud použijeme analogii s pets vs cattle příkladem ze začátku kapitoly cloud native.

Kontejnery napomáhají přenositelnosti aplikace mezi různými prostředími. Protože všechny závislosti aplikace jsou již nainstalované v kontejneru. Spuštění kontejneru bude stejné na pracovní stanici vývojáře, stejně tak jako v testovacím prostředí a na produkci. Tento přístup zjednodušuje vývoj a testování aplikace a také šetří čas a zrychluje doručování nových verzí. Jelikož si kontejner nese vše potřebné v sobě, technici nemusí kromě instalace prostředí pro běh kontejnerů instalovat další knihovny a závislosti.

Na trhu existuje několik kontejnerových řešení, mezi které patří LXC [23], Rkt[24] a Docker [25]. Jak je vidět na obrázku 3.1 nejpoužívanější kontejnerovou technologií je Docker. Docker se stal de facto standardem pro kontejnerovou technologii a termíny docker a kontejner jsou často považovány za totéž.

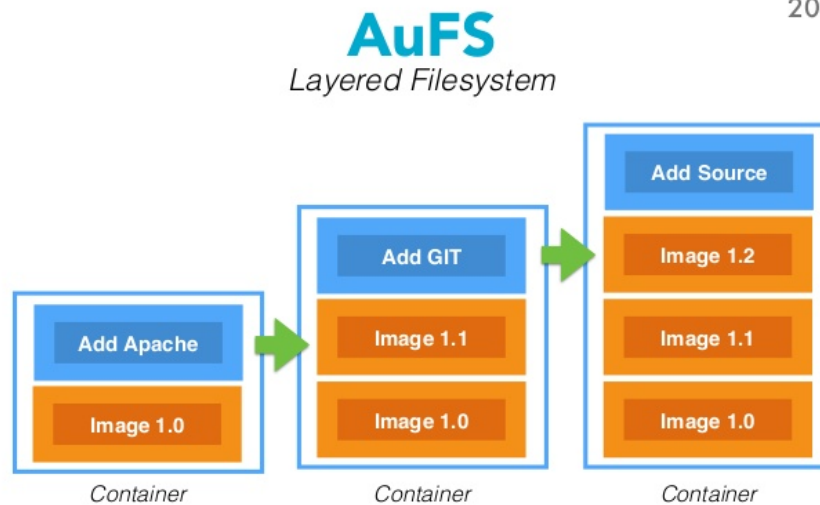


Obrázek 3.1: Nejpoužívanější kontejnerové technologie, zdroj: [26]

Docker

Součástí Dockeru je několik technologií. Mezi ně patří specifikace kontejnerů, runtime neboli běhové prostředí kontejnerů jehož součástí jsou Dockerfiles, které umožňují opakovatelné vytváření kontejnerů. Součástí je i technologie Docker Hub, která slouží jako úložiště kontejnerů. Docker projekt byl vytvořen v roce 2013 v programovacím jazyce Golang. Následující kapitola je vypracována s využitím zdroje [27].

Docker image je označení pro obraz dané aplikace se všemi spustitelnými soubory, závislostmi, knihovnami a také konfiguračními soubory. Docker image se skládají z vrstev společně s metadaty. Každá vrstva obsahuje informace o změně, která byla provedena vůči předchozí vrstvě. Na obrázku 3.2 je znázorněn systém přidávání jednotlivých vrstev. Vše začíná base, základním, image ze kterého potom vychází další vrstvy. Na base image navazuje přidání Apache serveru, poté přidání verzovacího systému Git a nakonec přidání souborů pro spuštění aplikace. Každá vrstva má svého předka a tím je předchozí vrstva. Vyjímkou je první vrstva, base image, která žádného předka nemá. Base image může být například klasická linuxová distribuce jako jsou Debian nebo CentOS a také minimalistický Alpine linux. Použití vrstev dovoluje doručovat image jako soubor modifikací nad určitým image.



Obrázek 3.2: Kontejnerové vrstvy, zdroj: [28]

Existují dva způsoby, jak je možné vytvořit docker image. Prvním z nich je spustit existující image a v něm provést změny. Například nainstalovat nový balíček, poté kontejner zastavit a vytvořit z něho nový kontejner s provedenými změnami. Tento proces není flexibilní a je nutné všechny kroky provést znovu při vytváření image. K automatizaci vytváření docker image slouží nástroj Dockerfiles. Tento nástroj vytvoří podle definice kontejneru se všemi potřebnými závislostmi a soubory. Formát a syntaxe DockerFile souboru na ukázce kódu 3.1 specifikuje base image ze kterého kontejner vychází. Dále odkud a jaký balíček má být nainstalovaný nebo také jaký příkaz se má vykonat při startu kontejneru.

Kód 3.1: Příklad docker file souboru

```
FROM golang:1.12
ENV version=1.11
WORKDIR /go/src/app
COPY basicHttp .
RUN chown golang:golang basicHttp
ENTRYPOINT ["/basicHttp"]
EXPOSE 80
```

Docker běží jako daemon na serveru a spravuje kontejnery. Tento program spouští kontejnery, kontroluje úroveň izolace jednotlivých kontejnerů, ověřuje, že kontejnery využívají pouze přidělené zdroje. Jednou z činností, které daemon vykonává je sle-

dování stavu kontejnerů a vykonání adekvátních akcí, např. restart kontejnerů. Daemon je také zodpovědný za správu images, konkrétně stáhnutí z , případně nahrání image do vzdáleného docker registry. Docker daemon také zodpovídá za vytváření images přes Dockerfiles. Docker registry je repozitář, kam uživatelé mohou nahrávat své kontejnery a také odtud kontejnery stahovat. Příkladem Docker registry je DockerHub. Vývojáři se mohou zaregistrovat a využívat repozitář pro sdílení images.

Pro oddělení jednotlivých zdrojů kontejnerů používá Docker Namespaces, neboli jmenné prostory. Namespaces izolují procesy, síťová zařízení, mount pointy a uživatele uvnitř kontejneru před ostatními procesy zdroji na serveru. Jedním typem je PID namespace. Tento namespace izoluje skupinu procesů. Tyto procesy jsou izolované a nevidí ostatní procesy mimo namespace. Z pohledu kontejneru jsou všechny procesy potomkem procesu s PID 1 a čísla procesů se mohou v jednotlivých namespace opakovat [29]. Druhou významnou technologií, kterou Docker používá jsou Cgroups. Cgroups je mechanismus jak omezit využití zdroje pro proces nebo skupinu procesů. Není tak možné aby jeden kontejner zabral všechny zdroje serveru pro sebe a blokoval tak ostatní kontejnery. Cgroups tak dovolují přiřadit kontejneru pouze omezené zdroje jako jsou čas na procesoru, RAM, využití sítě či disku [30]. Obě technologie jsou součástí linuxového jádra od verze 2.6.24.

Rkt

Rkt projekt byl vytvořený společností CoreOS a jeho hlavním účelem bylo odstranit nedostatky dockeru. Rkt je open source bezpečnější alternativa k Dockeru. Rkt také umožňuje spustit více izolovaných images, které sdílejí jádro systému. Rkt poskytuje větší bezpečnost než docker images. Například při stahování image z registry, docker nekontroluje image. Na druhé straně Rkt ověřuje podpis vydavatele dané image [31]. Rkt také podporuje řízení přístupu SELinux, případně umožňuje spustit kontejner s aplikací ve virtuálním serveru [32]. Rkt používá ACI (Application container format) image format a jako základní jednotku nasazení využívá Pod. Pod je seskupení jednoho nebo více images aplikace (ACI images). Jednotlivé images uvnitř podu sdílejí přidělené zdroje, například networking. Rkt je schopný spouštět jak ACI formát images tak i Docker formát. Rkt také implementuje OCI standard, který je představen v další kapitole.

Open container initiative

Open container initiative (OCI) je projekt, který má za cíl vytvořit standardy týkající se formátu kontejnerů a také běhového prostředí kontejnerů. Tento projekt byl vytvořen v roce 2015 lidry v oboru kontejnerových technologií Docker a Rkt. OCI aktuálně

zahrnuje dvě specifikace [33].

První je runtime specifikace, která specifikuje konfiguraci, spouštěcí prostředí a životní cyklus kontejneru. Konfigurace se specifikuje ve formátu JSON a obsahuje informace o verzi OCI specifikace, procesy, které mají být spuštěné, uživatelé, proměnné prostředí a další metadata. Standardizované spouštěcí prostředí zajišťuje, že aplikace běžící uvnitř kontejneru poběží stejně v různých runtime prostředích. Poslední částí je specifikace příkazů, které je možné na daném kontejneru provést [34].

Druhou specifikací je formát images. Tento formát definuje, že image splňující OCI specifikaci se skládají z manifestu, sadu vrstev souborového systému, konfiguraci a nepovinný image index. Manifest soubor obsahuje informace o image jako jsou konfigurace a sada vrstev pro jeden kontejner. Ve specifikaci je popsáno jak vytvářet a používat jednotlivé vrstvy souborového systému kontejneru. Konfigurace je popis změn provedených na jednotlivých vrstvách [35].

3.1.3 Orchestrátoři

Microservices architektura rozděluje aplikaci na menší části, které se lépe spravují. Technologie kontejnerů zabaluje jednotlivé microservices do samostatných kontejnerů společně s jejich závislostmi. Kontejnery dovolují spustit aplikaci se stejným výsledkem v různých prostředích bez nutnosti konfigurace aplikace. Pokud spravujeme dvě aplikace v deseti kontejnerech je možné veškerou správu aplikace a kontejnerů provádět ručně. Pokud ovšem provozujeme stovky aplikací v tisíci a více kontejnerech, manuální správa kontejnerů se stává velice obtížnou. Zde přicházejí na řadu orchestrátory kontejnerů. Orchestrátory provádějí následující úkony [36]:

- Orchestrátor se stará o stažení požadovaného image a jeho následné spuštění na vhodném serveru. Tento proces začíná předáním konfigurace orchestrátoru. Orchestrátor z konfigurace získá potřebné informace. Například odkud má stáhnout požadovaný image, na jakých portech by měla daná aplikace poslouchat, dále také jaké uložisko má být ke kontejneru připojeno, připojení kontejneru do sítě, aby mohl komunikovat a byl dostupný, vložení konfiguračních souborů pro aplikaci a přiřazení dalších metadat danému kontejneru. Orchestrátor také musí rozhodnout na jakém serveru kontejner spustit. Snahou je rovnoměrně zatěžovat jednotlivé servery, které hostí kontejnery a také přiřadit kontejnerům potřebné zdroje. Orchestrátor je i zodpovědný za správné ukončení aplikace. Například databáze bude vypnuta šetrně, aby nedošlo k poškození dat. Všechna potřebná data budou zapsána, transakce potvrzeny a poté dojde k uvolnění využívaných zdrojů.
- Dalším úkolem orchestrátoru je sledování běžících kontejnerů a jejich správa. Or-

Orchestrátor sleduje stav jednotlivých kontejnerů, jestli fungují jak mají. Pokud se například aplikace uvnitř kontejneru zastaví, kvůli chybě, orchestrátor to musí zjistit a namísto starého kontejneru vytvořit nový. Orchestrátor sleduje jednotlivé zdroje a udržuje takový stav jaký je definovaný v konfiguraci pro danou aplikaci. Jedním z aspektů, které orchestrátor kontroluje je i kolik zdrojů kontejner využívá a zda nepřekračuje přiřazené limity využívání. Orchestrátor nedohlíží pouze na stav kontejnerů, ale sleduje také stav serverů, které hostují kontejnerizovanou aplikaci. Pokud dojde k poruše serveru nebo jeho vypnutí, orchestrátor tuto akci zaznamená a všechny kontejnery, které na nedostupném serveru běžely jsou spuštěny na jiných běžících serverech. Na nedostupný server také nejsou posílány požadavky na spuštění kontejnerů, dokud není server obnoven.

- Jednou z výhod cloud native přístupu je škálovatelnost aplikací, kterou orchestrátory kontejnerů také zvládají. Orchestrátor umí na požádání přidat další instance aplikace. Například ve špičce při velkém zatížení jsou orchestrátory schopné automaticky navýšit počet kontejnerů dané komponenty. Orchestrátory též zajišťují load-balancing, neboli rozložení požadavků mezi všechny spuštěné instance aplikace. Příkladem může být webový server. V konfiguraci aplikace jsou specifikovány tři instance tohoto serveru. Uživatelské požadavky jsou rovnoměrně, případně podle jiného algoritmu, přesměrovány na jednotlivé instance webového serveru. Pokud se jeden z webových serverů porouchá, orchestrátor tuto informaci zjistí pomocí pravidelných kontrol a nepřesměrovává uživatele na tento kontejner.
- Pro přístup uživatelů k aplikacím z vnějšího světa, umožňují orchestrátory vystavit službu pomocí portu na veřejné adrese. Uživatelé se tak mohou jednoduše připojit na danou aplikaci a využívat služby, které nabízí.
- Další oblastí, kterou orchestrátory pokrývají je monitoring aplikací. Orchestrátory nabízejí různé statistiky o využití jednotlivých kontejnerů. Dále umožňují procházet logy jednotlivých služeb pro řešení problémů.

Orchestrátory kontejnerů dávají správcům aplikací mnoho výhod. Jednou z nich je jednotný přístup ke správě aplikací. Pro správu kontejnerizovaných aplikací používají správci stejné nástroje. Dalším aspektem je automatizace nasazení aplikace, přechod na novější verzi, škálování a dostupnost aplikace. Správci mohou automatizovat i jednotlivé kroky údržby aplikace. Tento přístup umožňuje cílit hlavně na vylepšování aplikace samotné v porovnání s manuálním přístupem ke správě kontejnerizované aplikace. Při manuální správě jsou zaměstnanci většinu času zabráni do udržování dostupnosti aplikace, nasazování nových verzí atd. Manuální řešení není vhodné pro větší

a dynamické projekty, protože zvětšování počtu lidí, kteří tuto práci provádějí není udržitelné. Orchestrátory zjednodušují a zrychlují správu větších a rostoucích projektů v dynamickém prostředí, které vyžaduje periodické přidávání a ubírání jednotlivých kontejnerů. Ke správě velkého počtu kontejnerů tak není zapotřebí armáda inženýrů, ale postačí malý tým operátorů pracujících s orchestrátorem.

Pokud si připomeneme Pets vs Cattle concept správy serverů z předchozí kapitoly, orchestrátory nakládají s kontejnery jako s Cattle, tedy dobyt看kem. Orchestrátor se nestará o jednotlivé kontejnery jako o Pets, neboli mazlíčky. Pokud je kontejner v chybovém stavu, nebo je nedostupný, dojde k jeho vypnutí a nahrazení jiným kontejnerem, který převeze jeho funkci. Společnost Sysdig ve své zprávě o využívání kontejnerů [37] uvádí, že ze zjištěných dat je životnost 95% kontejnerů maximálně týden. Z dat dále vyplývá, že nejvíce procent kontejnerů, konkrétně 27%, má životnost mezi pěti až deseti minutami. Tato data ukazují, že správa kontejnerů je velice dynamická. Orchestrátory dále dávají jasný pohled na běžící aplikaci a minimalizují tak manuální zásahy do systému o kterých ví pouze daný operátor. Manuální změny uvnitř kontejnerů nejsou perzistentní, při ztrátě kontejneru dojde i ke ztrátě manuální změny. Tento fakt napomáhá přenositelnosti aplikace mezi jednotlivými prostředími a zaručuje, že co funguje v testovacím prostředí bude fungovat v produkci. V dalších kapitolách budou představeny konkrétní orchestrátory kontejnerů. Sysdig zpráva o využívání kontejnerů [37] také uvádí hlavní tři nástroje na správu kontejnerů. Nejpoužívanějším nástrojem je Kubernetes, následované nástrojem Docker Swarm třetím nástrojem je Mesos Marathon. Tyto nástroje jsou představeny dále v kapitole.

Docker Swarm

Docker Swarm, jak už název napovídá, byl vytvořen společností Docker v roce 2015. Docker Swarm používá stejné standardní Docker API. To znamená, že všechny nástroje, které jsou schopné komunikovat s Dockerem (např. Docker CLI, Docker Compose) mohou stejně komunikovat i s Docker Swarmem. Výhodou je, že uživatelé mohou používat stejné nástroje, které už znají. Na druhé straně funkcionality je omezena pouze na Docker API.

Swarm spojuje jednotlivé servery, které jsou schopny spouštět Docker kontejnery do jednoho virtuálního serveru. Uživatel má dojem, že přistupuje pouze k jedné instanci Docker serveru, ve skutečnosti je na pozadí několik takovýchto serverů. Docker přidává další 2 nástroje Docker machine a Docker Compose. Docker machine slouží pro instalaci Docker engine na fyzické nebo virtuální servery, které budou hostovat Docker kontejnery. Pomocí Docker Compose je možné definovat pomocí YAML syntaxe chování kontejnerů a ty poté spustit. Příklad compose file na ukázce kódu 3.2.

Docker Swarm vyniká především svou jednoduchostí instalace i použití. Docker

Swarm je součástí Docker engine a používá stejné nástroje jako Docker. Swarm umožňuje spravovat skupinu serverů, které jsou schopné běžet kontejnery, jako jeden virtuální server.

Kód 3.2: Příklad docker compose souboru

```
version: '2'
services:
  nsqlookupd:
    image: nsqio/nsq
    command: /nsqlookupd
    ports:
      - "4160:4160"
      - "4161:4161"
  nsqd:
    image: nsqio/nsq
    command: /nsqd --lookupd-tcp-address=nsqlookupd:4160
    depends_on:
      - nsqlookupd
    ports:
      - "4150:4150"
      - "4151:4151"
  nsqadmin:
    image: nsqio/nsq
    command: /nsqadmin --lookupd-http-address=nsqlookupd:4161
    depends_on:
      - nsqlookupd
    ports:
      - "4171:4171"
```

Mesos Marathon

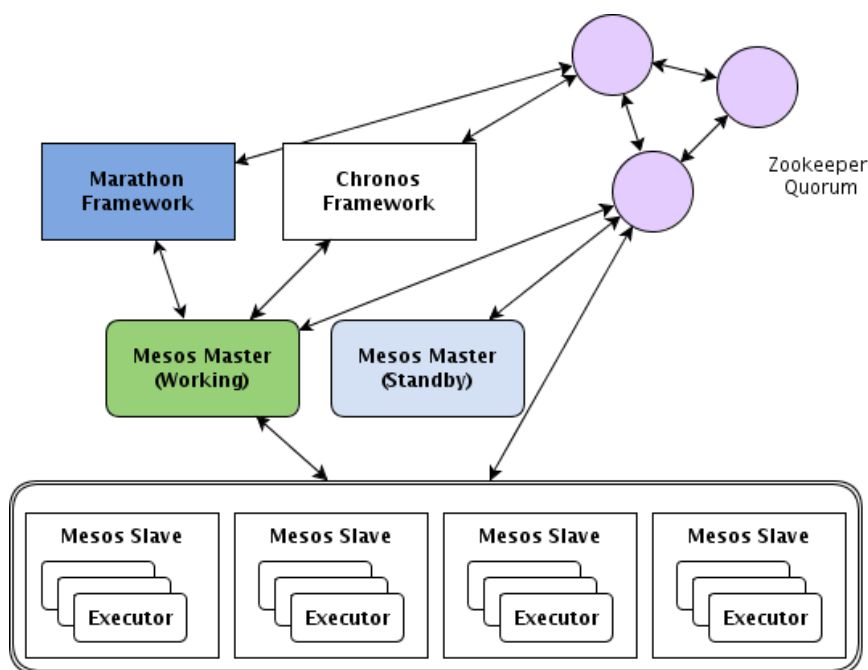
Mesos poskytuje vrstvu abstrakce nad fyzickými servery v datovém centru nebo nad velkými výpočetními zdroji. Mesos kombinuje zdroje jako jsou CPU, RAM a datové uložení do jednoho sdíleného prostoru, aby zdroje byly využívány co nejefektivněji. Tento prostor je abstrakce nad jednotlivými servery a jeví se jako jeden velký server. Odtud jsou zdroje přiřazovány jednotlivým běžícím aplikacím podle jejich požadavků. Aplikace běžící na Mesosu vidí tuto abstrahovanou vrstvu jako vysoce dostupný, odolný proti chybám a distribuovaný operační systém. Mesos je systém, který je schopný nabídnout škálovatelnost a odolnost vůči výpadkům velkým a náročným aplikacím. Apache mesos využívají například Apple pro svou hlasovou službu Siri, dále například společnost Bloomberg nebo PayPal. Funkce Mesosu mohou být rozšířeny

pomocí modulů [38].

Architekturu řešení Mesos můžeme vidět na obrázku [39]. Mesos se skládá z Masteru, který řídí agenty, dříve nazývané jako slaves. Agenti běží na fyzické serveru v datacentru. Jednotlivé moduly spouští s využitím master serveru úlohy na agentech. Moduly komunikují s masterem přes RestApi volání. Master servery pracují v active-passive módu. Pouze jeden master je aktivní. Pro výběr aktivního serveru používá Mesos nástroj Zookeeper. Obě komponenty Mesos master a Zookeeper běží v režimu vysoké dostupnosti, kdy při výpadku jednoho ze serverů je Mesos schopen fungovat dále.

Agenti oznamují primárnímu masteru zdroje, které mají k dispozici. Master poté nabízí tyto zdroje jednotlivým modulům. Modul si vybere z nabízených zdrojů, tak aby splňovaly požadavky na úlohu, kterou chce spustit. Modul poté oznámí aktivnímu master serveru na jakém agentovi vybranou úlohu spustit. Například orchestrátor kontejnerů pro systém Mesos se nazývá Marathon.

Marathon podporuje běhové prostředí kontejnerů Mesos containers a také Docker. Modul dále dovoluje připojení persistent storage pro stavové aplikace jako jsou databáze. Součástí je webové rozhraní pro práci s kontejnery, RestApi pro automatizaci a integraci s dalšími projekty, dále sada pravidel pro kontejnery. Marathon obsahuje nástroje pro service discovery, neboli katalog dostupných služeb, load-balancing, kontrolu dostupnosti služeb, metriky a statistiky 3.3.



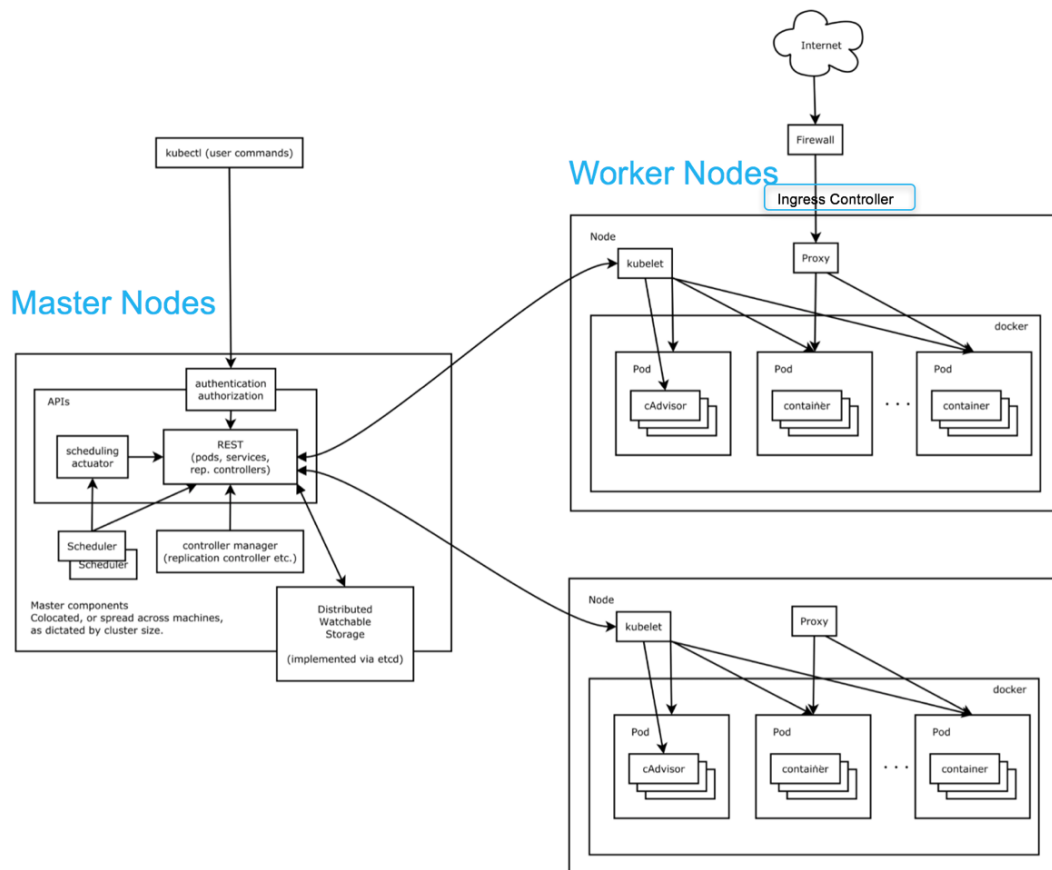
Obrázek 3.3: architektura Mesos orchestrátoru, zdroj:[39]

Kubernetes

Kubernetes je open source orchestrátor kontejnerů. Kubernetes, zkráceně k8s, nabízí širokou škálu funkcí. K8s bylo vytvořeno společností Google, ale poté bylo připojeno k CNCF. K8s se stalo de facto standardem pro orchestraci kontejnerů a to díky velké komunitě okolo tohoto projektu. O úspěchu projektu svědčí také podpora k8s napříč cloud providery jako jsou Google Cloud Engine, Microsoft Azure a AWS.

K8s je platforma pro orchestraci nasazení, škálování a správy aplikací běžících v kontejnerech. Kapitola o Kubernetes je vypracována s využitím zdroje [68]. Mezi funkce k8s patří možnost připojení souborového systému do kontejneru, distribuce citlivých informací, jako jsou hesla nebo tokeny, v podobě hashe mezi kontejnery pomocí tkz. Secrets. K8s umí také udržovat stanovený počet replik jednotlivých kontejnerů, škálovat aplikace, monitorovat zdroje, zpřístupnit logy jednotlivých kontejnerů a mnoho dalšího.

Architektura k8s je zobrazena na obrázku 3.4. K8s se skládá ze dvou základních částí, první je master role a druhá je node role a ty jsou dále tvořeny dalšími komponentami. Jednotlivé role mohou být nasazeny jak na fyzických tak i virtuálních serverch.



Obrázek 3.4: architektura Mesos orchestrátoru, zdroj: [41]

Master server tvoří control plane Kubernetes a je zodpovědný za nasazování a správu podů a také obsluhu různých událostí. Master server tvoří komponenty API server, scheduler a controller. API server vystavuje REST API, přes které může probíhat komunikace s Kubernetes. Ke Kubernetes API můžeme přistupovat pomocí nástroje Kubernetes CLI přímo z příkazové řádky, s využitím klientské knihovny a také přímo dotazy na API. Jednotlivé akce nabízené k8s tak mohou být automatizovány. API server jednoduše škáluje, protože data o běžících kontejnerech a stavu celého clusteru si uchovává v Etcd databázi. Controller je komponenta, která sleduje stav clusteru přes API server a řídí celý cluster k tomu aby byl v požadovaném stavu. Do controlleru patří pod controller, replica controller a také service controller. Například replica controller sleduje počet běžících podů a snaží se udržet definovaný počet instancí daného podu. Pokud existuje více podů, replica controller vypne přebývajících pody a pokud je naopak podů méně, replica controller nainstaluje další pod. Scheduler je komponenta, která je zodpovědná za spuštění podů na nodech. Scheduler musí rozhodnout, který node má dostatečnou kapacitu pro běh daného kontejneru nebo pokud node splňuje určité

požadavky jako například SSD disk nebo možnost připojení grafické karty a další.

Druhou komponentou je Node server, jehož úkolem je běh jednotlivých podů. Node server je řízený k8s master komponentou. Node interaguje s masterem z něhož získává informace jaký workload má spustit a zpět zasílá informace o workloadu do masteru. Kubernetes node je dále složen z kubeletu a kube proxy. Kube proxy je služba běžící na každém nodu, která může vykonávat jednoduché TCP/UDP směrování. Například vytváří virtuální adresy pro services, tak aby pody byly schopné komunikovat s ostatními pody v clusteru. Kubelet komunikuje s master komponentou, řídí a spravuje běžící pody. Mezi činnosti Kubeletu patří stahování secrets z API serveru, připojování uložiště do kontejnerů, provozování kontejnerů, oznamování API serveru svůj stav, stav nodu, a také stav běžících podů. Kubelet řídí kontejnery uvnitř podu pomocí CRI (Container runtime interface). CRI je soubor specifikací, požadavků a knihoven, které musí kontejnerové běhové prostředí (runtime) splňovat, aby mohlo být použito jako runtime kontejnerů pro k8s. K8s podporuje několik runtime prostředí, konkrétně Docker, Rkt, Cri-o nebo Containerd. Součástí k8s jsou i další koncepty a zdroje jako jsou pods, services, secrets, labels, replication controllers, volumes a další.

Základním stavebním kamenem k8s je pod. Pod obsahuje jeden nebo více kontejnerů. Kontejnery uvnitř jednoho podu jsou spuštěny najednou na stejném nodu. Všechny kontejnery v podu sdílejí jeden síťový prostor. To znamená, že mají stejnou ip adresu, sdílejí síťové porty a mohou komunikovat přes localhost adresu. Různé kontejnery v podu tedy musí používat rozdílné porty. Všechny kontejnery v podu mohou přistupovat ke sdílenému lokálnímu uložišti. Pody nejsou považovány za perzistentní, jsou vytvářeny a mazány na požádání. Při smazání podu jsou smazána i všechna data. K tomu aby se data uchovala se používá koncept volume.

Aby bylo možné vybírat specifické pody, například pody s určitou verzí aplikace, představuje k8s label, neboli popisek. Labels jsou popisky tvořené jako klíč a hodnota každý pod může mít více labels. Label používá ostatní zdroje pro identifikaci podů.

Dalším zdrojem je Replication Set, který spravuje skupinu podů, které jsou vybírány na základě label. Replication set zajišťuje, že definovaný počet podů je spuštěn. Pokud dojde k havárii jednoho nebo více z podů, Replication set vytvoří nový pod, aby byl jejich počet rovný definovanému stavu. Pokud je vytvořen pod s labelem, který se shoduje s definicí v replication setu, čímž dojde k překročení počtu instancí daného podu, je tento pod zastaven.

Pro load-balancing a redundancy podů poskytuje k8s services koncept. Services mají za úkol zpřístupnit uživatelům nebo dalším službám určitou funkcionalitu. Services používají label k výběru skupiny podů. Service poté rozděljuje požadavky mezi jednotlivé pody v dané skupině. Pomocí service můžeme například vystavit aplikaci na určité adrese a portu pro externí uživatele. Další použití může být pro komunikaci

jednotlivých microservices. Pody nekomunikují přímo s ostatními pody, ale přistupují na service, která load-balancuje mezi pody, jejichž label se shoduje s definicí v service.

Jak již bylo zmíněno data v podech nejsou persistentní, pokud je pod zničen všechna data jsou ztracena. Pokud potřebujeme, aby data zůstala i po zničení podu můžeme využít volume. Data je tak možné sdílet mezi kontejnery v podu a také přežijí restart podu. Pro stateful aplikace jako jsou například databáze nabízí k8s koncept stateful setu. Stateful set asociuje volume s daným specifickým podem. Pody řízené stateful setem mají stanovený neměnný hostname. Záleží na pořadí v jakém jsou spuštěny. Pokud definujeme 3 instance nejdříve bude spuštěn první s označení například mysql-0 a další v pořadí mysql-1 bude spuštěn až když mysql-0 je připraven a funkční. Na druhé straně při vypnutí je nejdříve vypnutý pod mysql-2, tedy poslední, po dokončení vypnutí se začne vypínat mysql-1 a tak dále. Stejně jako mají pody statefulsetu neměnný hostname, nemění ani ani volum, který mají připojený. To znamená, že mysql-1 bude mít vždy připojený stejný volume.

Pro oddělení jednotlivých podů se v k8s používá namespace. Namespace je virtuální cluster uvnitř fyzického clusteru. Jednotlivé zdroje uvnitř jednoho namespace jsou izolované od ostatních zdrojů v jiném namespace a komunikovat spolu mohou pouze přes veřejné rozhraní.

Kubernetes nabízí širokou škálu funkcí a možností jak spravovat aplikace v kontejnerech. K8s architektura je modulární a je možné vyměnit jakoukoliv část vlastním řešením a rozšířit tak funkcionalitu k8s. Například scheduler služba na master serveru může být nahrazena libovolnou službou, která bude implementovat jiný algoritmus pro vybírání nodů, na kterých bude workload spuštěn. Další velkou oblastí je networking. Pro k8s existuje mnoho nástrojů, které se starají o komunikaci mezi jednotlivými pody, services a také příchozí komunikaci z vnějšku. Mezi zástupce patří Project Calico [42], Flannel [43], nebo Weave Net [44].

3.2 Edge cloud

Edge computing, neboli edge cloud, je nový směr vývoje cloud computingu. Cílem edge computingu je přinést výpočetní zdroje blíže k místu, kde jsou potřeba. Cloud native aplikace mohou být využity pro edge cloud. Microservices služby zabalené v kontejnerech se všemi závislostmi mohou být použity pro nasazení aplikací na edge cloud. Kontejnery nejsou náročné na výpočetní zdroje, díky tomu není potřeba nasazovat do edge cloudu mnoho serverů, stačí pár serverů, které jsou optimalizované pro běh kontejnerů. S využitím orchestrátoru kontejnerů Kubernetes je možné nasadit stejnou aplikaci ve více edge cloudech. Aplikace vypadá pro všechny edge stejně a z tohoto důvodu se dobře udržuje a stejným způsobem se budou spravovat i jednotlivé k8s clustery napříč edge cloudy. K8s dále nabízí rychlé doručování nových verzí aplikací na jednotlivé edge cloudy a dovoluje tak reagovat na nové požadavky.

Cloud computing využívá centralizovaný model. Všechny zdroje jsou situovány v datovém centru s velkou kapacitou výpočetních zdrojů, ke kterému jednotlivé služby přistupují. Edge computing je na druhé straně distribuovaný model s omezeným objemem zdrojů a posouvá se blíže ke službám, které ho využívají. Centralizovaný cloud computing model není vhodný pro IoT (Internet of things) zařízení, které v posledních letech zažívají velký rozvoj a představují nové výzvy, které musí společnosti řešit. Jak uvádí [45], mezi otázky kterými se společnosti v souvislosti s edge cloudem a IoT zařízeními musí zabývat patří šířka pásma pro přenos dat, latence neboli zpoždění, stabilita, bezpečnost a omezené zdroje IoT zařízení. IoT zařízení produkující velké objemy dat mohou překročit šířku pásma pro dané přenosové médium. Příkladem mohou být samořiditelná auta, která mohou produkovat gigabajty dat za sekundu, a přenos takto velkého objemu dat do vzdáleného datového centra zpomaluje proces rozhodování o chování automobilu v dané situaci. Pro spolehlivou a rychlou komunikaci je nutné mít nízkou odezvu. Jedním z příkladů může být rozšířená realita. Pokud si přes brýle pro rozšířenou realitu prohlížíme stroje ve výrobní hale, očekáváme, že informace o objektech, které právě vidíme budou v rozumné době brýlemi zobrazeny. Nízká odezva od serveru, který zpracovává požadavky dovolí nabídnout uživatelům příjemný prožitek z používání brýlí. V obou těchto případech je užitečnější, aby jednotlivá zařízení komunikovala s edge technologií, která se nachází v bezprostřední blízkosti zařízení, než aby IoT zařízení komunikovalo se vzdáleným datacentrem. Komunikace s lokálním edge cloudem je také spolehlivější a méně náchylná na výpadku a rušení. IoT zařízení mají omezené výpočetní zdroje a energie pro svůj provoz. Z tohoto důvodu je například energeticky náročné přenášet přes internet data, případně provádět výpočetně náročnější operace. Edge tak může sloužit jako agregátor dat ze zařízení a na základě jejich analýzy vykonávat definované akce. Rozhodnutí o dalších

akcích tedy místo tradičního vzdáleného datacentra vykonává místní edge cloud. To ovšem neznamená, že tradiční datacentrum ztrácí smysl. Edge může odesílat data právě do vzdáleného datacentra, které působí jako agregátor dat z více edge cloudů, kde může například docházet k hlubší a detailnější analýze dat z více edge cloudů, případně dalším akcím které vyžadují použití většího objemu výpočetních zdrojů. Jednotlivá IoT zařízení v sobě nezahrnují dostatečné mechanismy, aby byly schopné ochránit sebe sama a také přenášená data. Data jsou do edge cloudu jsou přenášena nechráněně, ale odtud až do datacentra už putují přes zabezpečené kanály. Tímto přístupem se například minimalizuje zneužití zařízení pro vysílání napadených či chybných dat.

Edge cloud je poměrně nový přístup, který budí zájem společností z oblastí mobilních operátorů, rozšířené reality, autonomních a IoT zařízení. Jedním z projektů, který se zaměřuje na vytvoření technologie pro edge cloud je projekt Akraino Edge Stack [46]. Akraino je open-source projekt, který je součástí Linux Foundation. Mezi iniciátory tohoto projektu patří ATT a Intel. Projekt je veden jako komunitní projekt několika společností mezi něž patří Dell EMC, Ericsson, Huawei, Juniper Networks, Nokia, Qualcomm, Red Hat a další. Jeho cílem je vylepšení edge cloudové infrastruktury. Akraino projekt je seskupení technologií, které by měly poskytovat novou úroveň flexibility pro rychlé škálování edge služeb a zajistili spolehlivost služeb, které musí pořád běžet. Akraino komunita se zaměřuje na vytvoření edge API, vývojových nástrojů pro edge a umožní napojení na cloudová řešení třetích stran. Součástí projektu bude podpora pro vývoj edge aplikací a vytvoření virtual network function (VNF - virtuální síťové funkce jako jsou např. firewally, DNS servery) ekosystému.

Zatímco projekt Akraino je spíše v začátcích, americká síť fast-food občerstvení Chick-fil-A (dále jen Chick) využívá edge computing k efektivnímu řízení svých restaurací. Každá restaurace obsahuje tři malé servery na kterých je nasazený kubernetes orchestrátor. Společnost má okolo 2000 poboček, což znamená 6000 zařízení, která běží v edge cloudu. Pro nasazení k8s na každé zařízení si Chick vyvinulo vlastní řešení, které poskytuje automatickou instalaci zařízení, odolnost vůči výpadku a vysokou dostupnost. Díky tomu jsou schopni rychle reagovat na vznik nových poboček [47]. Chick využívá edge kvůli nízké latency, aplikacím, které jsou nezávislé na internetovém připojení, vysoké dostupnosti těchto aplikací, škálovatelnosti infrastruktury a také kvůli platformě, která umožňuje rychlé doručování nových funkcí do produkce. Chick restaurace se snaží vybudovat chytrou kuchyni, která bude poskytovat data o dění v kuchyni. Tato data jsou později použita ke zlepšování jednotlivých procesů, čímž mohou dosáhnout na lepší rozšiřitelnost celého byznysu. Data mohou být například použita pro předpověď objemu jídla, který má být v daném okamžiku připraven tak, aby zákazníci nečekali dlouho a pokaždé dostali čerstvě připravený produkt. Jednotlivé časovače, které rozhodují o tom zda připravená surovina již nesmí být podána zákazníkům, fun-

gují automaticky. Po naplnění nádoby čerstvě připravenými surovinami a jejím přesunutím pod kamerou, která načte kód umístěný na nádobě se automaticky podle kódu spustí časovač pro daný typ suroviny. Jednotlivé časy jsou poté vyobrazeny na dotykových obrazovkách a při vypršení limitu pro servírování jsou zaměstnanci upozorněni na vyhození dané suroviny. A právě data a procesy, které rozhodují o těchto akcích běží v edge cloudu, který je umístěný přímo v restauraci [49]. Páteří řešení je AWS cloud, který slouží jako autorizační server pro zařízení, sběr dat, centrální monitoring, alerting, tracing a správa instalací. Na straně edge cloudu Chick spravuje autentikaci, messaging s využitím MQTT, microservices vyřizující HTTP požadavky. Dále zde běží aplikace, které interagují se zařízeními v restauraci. Součástí edge řešení jsou i modely pro predikci budoucího chování na základě událostí ze zařízení v restauraci. Všechna data na straně edge jsou postupně přenesena do centrálního uložení a poté z edge uložení smazána. V případě výpadku sítě umí edge data agregovat a po opětovném obnovení spojení je nahrát do centrálního datového centra. Poslední vrstva se stará o připojení jednotlivých zařízení v restauraci. Chick používá kubernetes a kontejnery pro centrální AWS cloud tak i pro edge cloudy. Díky tomu mají stejné prostředí pro běh aplikací na obou stranách, což usnadňuje práci vývojařům a pomáhá rychlému dodávání změn [48].

4 Existující řešení pro hybrid a multi cloud

Hybrid cloud je typ cloudového prostředí, které kombinuje privátní a veřejný typ cloudu čímž dovoluje využít výhod obou řešení pro běh aplikací a sdílení dat. Každá část hybrid cloudu zůstává samostatná a unikátní, ale je pomocí standardizované nebo proprietární technologie napojená na ostatní nasazené modely. Toto spojení umožňuje jednotlivým aplikacím vzájemnou komunikaci. Hybrid cloud je tvořen spojením nejméně jednoho private cloudu a jednoho public cloudu [50].

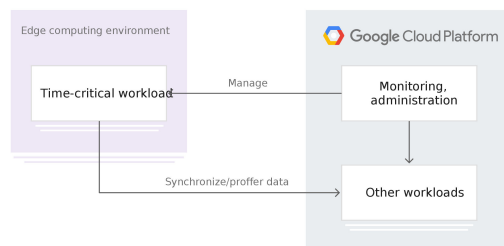
Hybrid cloud řešení snižuje CAPEX (Capital expenditure) náklady, protože část firemní infrastruktury je spravována poskytovatelem public cloud řešení. CAPEX náklady znamenají výdaje na pořízení výměnu a správu hardwaru a také pořízení vybavení datacentra. Hybridní řešení dále vylepšuje přiřazení zdrojů pro dočasné projekty. Využití public cloud řešení odstraňuje investice, které jsou potřeba během projektů. Odpadá tedy nutnost zakoupit hardware pro každý nový projekt. Tyto zdroje jsou alokovány na straně public cloudu. Po dokončení či zrušení projektu je snadné tyto zdroje opustit a společnost za ně nemusí nadále platit. Přidání zdrojů a odebrání zdrojů je snadné a napomáhá společnosti pokrýt sezónní nápor uživatelů. Tento přístup nabízí uživateli snadnou škálovatelnost a pomáhá optimalizovat náklady.

Cílem hybrid cloudu je kombinovat výhody public a private cloudu. Z modelu private cloudu si hybrid cloud bere především bezpečnost dat. Společnost tak může mít nasazenou aplikaci v prostředí public cloudu a data se kterými aplikace pracuje jsou uložena v privátním datacentru. Public cloud poskytne škálovatelné prostředí a privátní cloud zase zabezpečí, že data jsou uložena na serverech vlastněných a spravovaných společností v požadované zeměpisné lokaci. Společnost má s hybrid cloud modelem kritická data pod kontrolou.

Další oblastí využití modelu hybridního cloudu je oblast edge computingu. Edge computing je vhodný pro oblasti, které potřebují spolehlivou a rychlou síťovou konektivitu. Model hybridního edge cloudu se na tyto problémy zaměřuje a snaží se je řešit. Pro byznys důležité aplikace je užitečné provozovat lokálně v místě vzniku dat s kterými pracují. Díky tomu jsou splněny požadavky na rychlou odezvu. Pro hybrid edge model není internetové spojení mezi edge cloudem a centrálním cloudem kritické.

Používá se především pro správu edge prostředí a výměnu dat. Aplikace využívají lokální výpočetní zdroje. Synchronizace dat mezi edge a centrálním cloudem probíhá na pozadí, používají zabezpečené spojení. Při výpadku spojení mezi edge a centrálním cloudem zůstávají všechny aplikace funkční a data se ukládají lokálně, dokud není spojení obnoveno. Poté jsou přesunuty do centrálního cloudu k dalšímu zpracování. Edge tak není závislý na centrálním cloudu.

Architektura hybrid edge modelu je zobrazena na obrázku 4.1. Kritické a důležité aplikace běží v edge cloudu. V centrálním cloudu, v tomto případě GCP (Google Cloud Platform), běží centrální control plane pro jednotlivé edge cloudy a také monitoring, který agreguje data a poskytuje centrální správu a globální pohled na celé řešení. S využitím kontejnerů a například Kubernetes, které je podporované většinou poskytovatelé veřejného cloudu, odstranit rozdíly mezi jednotlivými edge lokacemi a centrálním cloudem. K8s zaručí stejné běhové prostředí pro kontejnery a podle potřeby dovoluje například přesouvat workload mezi edge cloudem a centrálním cloudem [51]. V následující kapitole jsou představeny jednotlivé nástroje, které je možné použít pro správu hybrid prostředí.



Obrázek 4.1: Google edge architektura, zdroj: [52]

Řešení pro správu hybrid cloud prostředí nabízí uživatelům možnost spravovat zdroje různých poskytovatelů z jednoho centralizovaného místa. Některá řešení jako jsou Red Hat Cloudforms (Manage IQ) a Mist.io se zaměřují primárně na správu infrastruktury u jednotlivých poskytovatelů. Jejich úkolem je připravit infrastrukturu, kam patří fyzické a virtuální servery, uložení a síťové prvky, pro běh aplikací napříč zdroji od více poskytovatelů. Na druhé straně řešení jako Spinnaker, Kubernetes Federation a Cloud foundry se primárně zaměřují na správu aplikací, které běží distribuovaně u více poskytovatelů cloudu. V následující sekci jsou jednotlivá řešení představena detailněji.

4.1 Red Hat Cloudforms/Manage IQ

Cloudforms je platforma pro jednotnou správu fyzických a virtuálních serverů a také kontejnerů napříč různými cloudovými prostředími. Pomocí CloudForms je možné ovládat zdroje v technologiích pro privátní cloudy OpenStack a také virtuální servery spravované technologií od společnosti VMware, Red Hat virtualization a Microsoft Hyper-V. CloudForms umožňuje spravovat zdroje ve veřejných cloudech AWS (Amazon web services), Microsoft Azure a také GCP (Google cloud platform). CloudForms umožňuje spravovat také kontejnery na platformě OpenShift. CloudForms technologie vychází z Manage IQ technologie. Manage IQ je jednotná open-source platforma pro správu cloudových prostředí. Cloudform je vyvíjeno, spravováno a podporováno přímo společností Red Hat, jedná se o komerční řešení. Manage IQ je stejná platforma s otevřeným kódem, která je spravována a rozvíjena komunitou napříč různými společnostmi [53]. Dále v textu je popisována Manage IQ verze.

Manage IQ přistupuje k jednotlivým systémům, označovaným jako poskytovatelé, přes API, které např. veřejné cloudy nebo OpenStack poskytují. Přes tato API si Manage IQ vyčítá informace o jednotlivých zdrojích jako jsou virtuální servery, kontejnery, load-balancery společně s jejich metadaty jako jsou jméno, typ a mnoho dalšího. Tyto zdroje jsou označovány jako managed elements, neboli spravovaný prvek, a jsou uloženy ve Virtual Managemet Database (VMDB). Manage IQ po počátečním získání informací z jednotlivých poskytovatelů stále sleduje jednotlivá API poskytovatelů a udržuje VMDB stále aktuální. Manage IQ poskytuje přehledné webové rozhraní ve kterém jsou zobrazeny jednotlivé zdroje poskytovatelů s detailními informacemi. Například při připojení na VMware je zobrazen seznam běžících virtuálních serverů společně s jejich atributy jako je počet jader, velikost RAM, operační systém, připojené sítě, nainstalované programy a tak dále.

Manage IQ dovoluje provádět akce nad objekty, které má zaregistrované ve VMDB. Například virtuální servery běžící ve VMware prostředí můžeme vytvářet, mazat, vypínat, zapínat a restartovat. Další oblastí správy zdrojů je sledování a provádění změn. Můžeme sledovat jednotlivé atributy zdrojů a také kdy došlo k jejich změně. Manage IQ umožňuje také porovnávat aktuální stav s počátečním stavem. Např. jaké změny byly provedeny na virtuálním serveru oproti jeho základní konfiguraci. Manage IQ může zaznamenat změnu sudo práv, přidání nového uživatele nebo nainstalování nového balíčku. Protože Manage IQ má data o všech zdrojích jednotlivých poskytovatelů, můžeme s jeho pomocí počítat náklady na provoz.

Další věcí, kterou Manage IQ nabízí jsou služby. Služby dovolují vytváření složitějších konfigurací jednoduchou cestou. Přes služby můžeme například vytvořit dva virtuální servery s nainstalovaným webovým serverem, jeden server poběží v prostředí Open-

Stacku a druhý na VMwaru. Dále virtuální server sloužící jako databáze a poté ještě load-balancer, který bude rozkládat zátěž mezi webové servery. Výběr těchto zdrojů může například sloužit vývojářům aplikace jako prostředí pro vývoj. Vytvořit takovéto prostředí bude možné na jedno kliknutí. Správce vytvoří podle požadavků skript, který tyto zdroje vytvoří a poté zdroje nastaví. Vývojáři mohou z webového rozhraní skript spustit a během chvíle dostanou připravené prostředí pro vývoj. Tento nástroj dovoluje vytvářet složitější konfigurace a jednoduše je nasadit. Vývojáři tak tráví více času programováním aplikace namísto nastavování serverů. Administrátoři mohou povolit přístup k určitým zdrojům pouze uživatelům, kteří splňují stanovená pravidla. Např. pouze uživatelé ve skupině vývojáři mohou spouštět skript, který vytvoří vývojářské prostředí z předchozího příkladu [54].

4.2 Cloud foundry

Cloud Foundry (CF) je open source cloudová PaaS platforma. Na této platformě mohou vývojáři jednoduše vytvořit, nasadit, běžet a škálovat aplikace. CF bylo původně vytvořeno společností VMware. Později došlo ke zveřejnění zdrojových kódů a stalo se komunitním projektem. Jedná se o Paas řešení, které je dobře nastavitelné a rozšiřitelné, uživatelé mohou používat širokou škálu frameworků a programovacích jazyků. CF může být nasazeno na vlastní hardware případně na jinou podporovanou cloud platformu jako je AWS nebo OpenStack. CF poskytuje seznam certifikovaných platform jako jsou IBM Cloud Foundry, Atos Cloud Foundry nebo Pivotal Cloud Foundry. Tyto certifikované platformy zaručují, že aplikace se budou na těchto platformách chovat stejně. Tento fakt minimalizuje vendor lock-in, tedy upoutání se pouze na jednoho poskytovatele. Další výhodou je možnost využití CF pro provozování aplikací na více cloudech, tedy multi-cloud [55].

CF se skládá z mnoha komponent a má otevřenou architekturu, které zahrnuje tkz. buildpack mechanismus. Tento mechanismus dovoluje přidávat nové frameworky, aplikační služby a také rozhraní pro cloudové poskytovatele. Prvním komponentou je router, který směruje příchozí požadavky na odpovídající komponenty. Další komponentou je UAA (user account and authentication) server, který vykonává správu identit obsažených v systému a přístup uživatelů k jednotlivým zdrojům. Další důležitou komponentou je Cloud Controller, který zajišťuje nasazení aplikace. Cloud controller řídí Diego Brain komponentu, aby spustila požadovanou aplikaci a společně obstarávají celý životní cyklus aplikace. Diego Brain je komponenta, která spouští jednotlivé aplikace a sleduje zda se aplikace nachází v požadovaném stavu. Diego Brain, implementuje control plane, rozhoduje o tom co a kde bude spuštěno. Pro samotné spuštění je zodpovědná komponenta Diego Cell. Cell jsou virtuální servery, na kterých běží

požadované aplikace. Aplikace jsou spouštěny v kontejnerech. CF používá jako běhové kontejnerové prostředí vlastní produkt Garden-runC [56], který splňuje OCI standard. Diego Cell poskytuje pro tyto kontejnery statistiky použití, logy a další informace. Další funkcí CF jsou služby. Služby fungují v CF jako katalog, ze kterého si vývojář vybírá. Příkladem může být MySQL, Redis, Jenkins, Rabbit MQ a mnoho dalších. Vývojář může využít předem definované zdroje, které společně s jeho aplikací vytvoří vybranou službu, bez nutnosti manuální instalace a konfigurace daného nástroje. CF dovoluje vytvářet a přidávat vlastní služby. Aby mohly být služby úspěšně spuštěny, musí implementovat definované API [57].

Aby mohlo CF spouštět aplikace na virtuálních serverech, potřebuje ovládat infrastrukturu, která hostuje CF aplikace. K tomuto účelu slouží nástroj BOSH, který zaručuje, že prostředí je správně nastavené a připravené hostovat uživatelské aplikace. BOSH se skládá ze tří vrstev. První vrstvou je Stemcell, který představuje univerzální operační systém, který abstrahuje aplikaci od operačního systému serveru, je stejný přes všechny poskytovatele a umožňuje jednotnou správu virtuálních serverů. Druhou vrstvou je vydání (Release), popisuje jaký software by měl být nasazen a jak by měl být nastavený. Druhá vrstva zaručuje opakované spuštění aplikace se stejným výsledkem. Poslední částí BOSH je manifest soubor, který popisuje jaký release by měl být nasazený a do jakého cloudu. BOSH také dovoluje spouštět aplikace v multi cloud prostředí. BOSH používá Cloud provider interface (CPI), aby abstrahoval rozdíly mezi jednotlivými poskytovateli. Díky tomu je možné přidávat nové poskytovatele. BOSH v současnosti může nasadit aplikaci na platformy AWS, Azure, GCC, OpenStack a také VMware [58].

4.3 Mist.io

Mist.io je open-source platforma, která zjednodušuje uživatelům správu cloudu. Mist.io poskytuje jednotné rozhraní pro správu cloudových řešení různých společností. Nástroj Mist.io může spravovat fyzické servery, virtuální servery na technologiích KVM, VMware, OpenStack nebo Rackspace. Mist.io ovládá také zdroje největších veřejných cloud poskytovatelů AWS, Google cloud engine a Azure. Mist.io dává uživatelům možnost monitorovat Docker kontejnery. Díky široké podpoře cloudových technologií může být Mist.io využito i pro správu multi-cloud infrastruktury a minimalizovat závislost pouze na jednom poskytovateli. Mist.io je možné využívat ve třech verzích. První verzí je Community edition, která je volně dostupná a uživatelé si ji musejí spravovat sami. Druhou verzí je Enterprise edition, která může být nasazena on premise na privátních serverech společnosti. Tato placená verze zahrnuje profesionální podporu od samotné společnosti Mist.io. Poslední možností je využití SaaS verze Mist.io pro-

duktu. Uživatelé této verze platí pouze za využití zdrojů a jedná se o nejjednodušší řešení jak s Minst.io začít. SaaS verze poskytuje stejně jako Enterprise edition profesionální podporu. Tato kapitola je vytvořena s využitím oficiální dokumentace k Mist.io nástroji [59].

Mist.io poskytuje webové rozhraní pro správu a monitoring jednotlivých zdrojů napříč různými poskytovateli. Součástí aplikace je i API, které umožňuje automatizovat jednotlivé postupy. Prvním krokem je přidání cloudových platforem, které budou spravovány. Každý poskytovatel vyžaduje rozdílnou konfiguraci pro správné fungování. Například pro správu fyzického serveru je nutné poskytnout ip adresu, ssh klíč, uživatele s root právy a číslo portu pro připojení. Pro připojení AWS poskytovatele je nutné vyplnit API klíč a přístupový klíč. Po přidání jednotlivých poskytovatelů je možné si prohlížet jednotlivé zdroje poskytovatelů a dále s nimi pracovat. Mist.io rozlišuje několik základních zdrojů jako jsou machines, images, networks, keys a scripts. Machines, neboli servery, jsou všechny fyzické a virtuální servery. Ve webovém rozhraní jsou servery jednotlivých poskytovatelů rozlišeny ikonkou. U serverů je možné si prohlížet jejich detaily jako je vytížení procesoru, využití RAM nebo zaplněnost disků. Dále je možné spravovat pravidla definovaná pro daný server. Pravidla umožňují definovat akci, která se má provést při splnění určitého pravidla. Například pokud služba na serveru neodpovídá, tak dojde k restartu služby. Jednotlivé servery je možné seskupovat do skupin podle tagů, neboli nálepek, a efektivně s nimi pracovat. Mist.io podporuje ssh připojení na vzdálený server přímo z webového prohlížeče. Každý server je možné ovládat pomocí definovaných akcí. Každý poskytovatel nabízí jinou sadu akcí, které je možné vykonávat. Mezi ty základní patří vypnutí, restart a smazání serveru. Jednotlivé servery pak mohou být vytvořeny přímo ve webovém rozhraní nebo přes API. Při vytváření serveru je možné specifikovat jméno serveru nebo poskytovatele, ve kterém bude server spuštěn. Druhým zdrojem jsou images, které slouží pro vytváření serverů nebo kontejnerů. Mist.io zobrazuje images, které objevil v jednotlivých public nebo private poskytovatelích a také z připojeného Docker engine. Součástí kontejner images jsou i základní docker images, které poskytuje přímo Mist.io. Pro další zdroj networks, umožňuje Mist.io zobrazit síť jednotlivých poskytovatelů společně s informacemi o nich a případně v nich vytvářet subnety. Aby mohli uživatelé přistupovat přes webové rozhraní na konzoli serverů, Mist.io nabízí možnost spravovat ssh klíče, ty poté přiřazovat jednotlivým zdrojům a získat tak přístup na jejich konzoli. Pokud uživatel klíč nemá, může si ho přes webové rozhraní vytvořit. Posledním zdrojem, který Mist.io nabízí je script. Zdroj script slouží k automatizaci opakujících se ukonů nebo konfiguraci serverů. Jednotlivé script zdroje mohou být vytvořeny v bashových skriptech nebo jako Ansible playbooks.

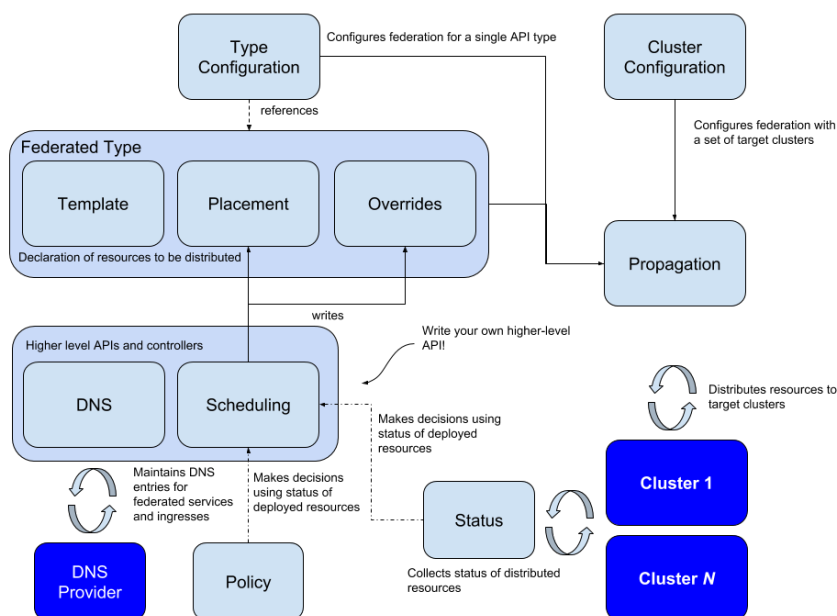
4.4 Kubernetes federation

Některé nástroje na správu multi cloud prostředí implementují vlastní řešení jak spravovat Kubernetes clusteru napříč různými lokacemi. Kubernetes komunita se také rozhodla vyvinout řešení pro správu více clusterů, která je zajímavá pro spoustu společností a přidává další možnosti využití. V porovnání s jedním clusterem je nasazení aplikace do více clusterů komplikovanější. Jedním z problémů je, jak bude workload jednotlivých aplikací rozdělován do různých clusterů. Existuje několik možností, replikovat zdroje aplikace mezi všechny clusteru, replikovat pouze mezi vybrané clusteru nebo zdroje rozdělit mezi jednotlivé clusteru. Dále je nutné vyřešit jak bude řízen přístup k jednotlivým clusterům nebo jak postupovat, jestliže vytvářené zdroje nebo jejich část již v nějakém clusteru existují. První verze řešení označovaná jako v1 využila koncept Kubernetes API, aby odstranila jakoukoliv přidanou složitost pro existující uživatele k8s. Tento přístup ovšem nebyl providitelný, protože zahrnoval spoustu problémů jako jsou složitost při znovuvytváření k8s api na úrovni clusteru, omezená flexibilita ve federovaných typech. Jedním z problémů byla také neustálená cesta ke stabilní veřejné dostupnosti (GA verze) a zmatek ohledně k8s API samotného. Například k8s deploymenty jsou veřejně dostupné v kubernetes projektu, ale ve v1 verzi kubernetes federace nebyly dostupné ani v beta verzi.

Původní myšlenka s federací specifické API architektury se dále rozvinula a v současnosti pokračuje jako verze v2. Zatím se jedná pouze o Alfa verzi, která není doporučena používat pro produkční použití. Nový návrh architektury je zobrazen na obrázku 4.2. Federace může být nastavena pomocí dvou typů informace. Prvním typem je Type configuration, který nám říká, jaké zdroje v API má federace propagovat. Druhým typem je Cluster configuraton, která určuje do kterých clusterů má federace propagovat jednotlivé zdroje. Informace Type configuration se skládá ze tří částí. Template, neboli šablona, obsahuje základní konfiguraci zdroje. Například zdroj FederationReplicaSet obsahuje informace o zdroji ReplicaSet, který má být distribuován do určených clusterů. Druhou částí je Placement, neboli umístění, který říká v jakých clusterech má být zdroj spuštěn. Poslední částí jsou Overrides, které určují specifické parametry pro jednotlivé clusteru. Tyto tři části obsahují nejn nutnější informace potřebné pro propagaci a jsou navrženy tak, aby splňovaly dynamické plánování spouštění zdrojů založené na pravidlech. Další důležité součásti federace k8s jsou Status, Policy a Scheduling. Status shromažďuje informace o všech zdrojích spuštěných napříč všemi federovanými clusteru. Policy, neboli politiky, rozhodují do jakých clusterů mohou být jednotlivé zdroje distribuovány. Scheduling, neboli plánování, zajišťuje rozdělení jednotlivých workloadů přes různé clusteru co nejlépe a neefektivněji.

Funkce, které by Kubernetes federace měla poskytovat patří propagace různých typů

zdrojů do vzdálených clusterů, CLI nástroj kubefed2 pro správu takovéto federace podobně jako nástroj kubectl, který slouží pro správu jednoho clusteru. Federace by také měla poskytovat DNS a Ingress služby pro více clusterů. Další funkcí by také mělo být pokročilé nastavení rozmístění jednotlivých zdrojů do clusterů podle uživatelské preference.



Obrázek 4.2: Architektura projektu Kubernetes Federation, zdroj: [61]

4.5 Spinnaker

Spinnaker je open source platforma, která umožňuje continuous delivery (CD) aplikací do multi-cloud prostředí. Spinnaker projekt byl vytvořen společností Netflix. Spinnaker je navržen s ohledem na snadnou rozšiřitelnost tak, aby bylo možné přidávat nové cloudové poskytovatele. Spinnaker podporuje nejpoužívanější cloudové poskytovatele jako jsou AWS, Google Cloud, Azure, OpenStack, Oracle Cloud Infrastructure a také technologii Kubernetes [62]. Mezi uživatele Spinnakeru patří například Waze, navigační aplikace pro mobilní telefony. Provozovatelé aplikace používají pro běh své aplikace AWS a Google cloud poskytovatele pro vysokou dostupnost své služby a také pro možnosti, které Spinnaker nabízí. Postup, který Waze používá začíná odesláním změn do github repozitáře, Jenkins software vytvoří balíček s požadovanou změnou a spustí další proces, jehož úkolem je nasazení aplikace. V obou cloudech zároveň je vytvořena image s požadovným balíčkem a jsou spuštěny integrační testy. Pokud

testy proběhnou bez problémů, aplikace je spuštěna v testovacím prostředí s reálnými daty. Jestliže aplikace v testovacím prostředí prošla testy je nasazena do produkčního prostředí. V případě chyby může být nová verze stažena z produkce a místo ní nasazena starší stabilní otestovaná verze [63].

Spinnaker obsahuje dvě základní sady funkcí správu aplikací a nasazení aplikací. S pomocí správy aplikací je možné spravovat a monitorovat zdroje v jednotlivých cloudech. Spinnaker vidí aplikace jako jednotlivé microservisy a takto s nimi pracuje. Spinnaker používá k popsání uživatelské aplikace, aplikace kterou chceme pomocí Spinnakeru nasadit do prostředí cloudu, koncepty jako jsou Applications, Clusters a Server Groups. Spinnaker dále používá koncepty Load balancer a Firewall pro popsání jak budou jednotlivé mikroslužby zpřístupněny uživatelům. Koncept Application je ve Spinnakeru reprezentace mikroslužby uživatelské aplikace, její konfigurace a infrastruktury na které bude aplikace provozována. Ač to Spinnaker nevyžaduje, dobrou praxí je vytvoření Spinnaker aplikace pro každou mikroslužbu uživatelské aplikace. Pojem Cluster je označení pro skupinu serverů ve Spinnakeru. Server Groups označení používá Spinnaker pro nasaditelné zdroje jako jsou obrazy virtuálních serverů nebo Docker image a konfiguraci (počet instancí zdroje, metadata, pravidla pro škálování, atd.). Pokud je Server Group nasazena, je kolekcí instancí dané mikroslužby (virtuální servery, pody v Kubernetes). Load balancer je použitý pro směrování požadavků mezi instance v jedné Server Group. Load balancer umí také kontrolovat dostupnost mikroslužby a v případě poruchy přestat posílat požadavky na poškozený Server Group.

Druhá skupina funkcí, kterou Spinnaker nabízí se týká nasazení aplikace a CD (continuous delivery) aplikace. Hlavní částí nasazení aplikace je Pipeline. Pipeline se skládá z posloupnosti akcí, které se označují jako Stages. Pipeline je možné spustit ručně nebo lze nastavit spouštěče. Jako spouštěč mohou být nastaveny různé podněty jako je nový Docker image v Docker registry, dokončení Jenkins úlohy, časový údaj atd. Pipeline také může uživatele upozorňovat o svém spuštění, průběhu, dokončení nebo chybovém hlášení. Stage je atomická část Pipeliny. Jednotlivé Stage se dají různě skládat. Například některá Stage musí být úspěšně dokončena předtím než je spuštěna další. V jiném případě může být spuštěno více Stagí najednou. V rámci jedné Stage může být vytvořen balíček s aplikací, vytvořen Docker image, spuštění testů aplikace nebo spuštění virtuálního serveru s novou verzí aplikace. Spinnaker využívá cloud native strategie pro správu aplikací. Mezi tyto strategie patří ověřování dostupnosti služeb, nasazení nové verze aplikace společně s blue/green a canary testováním a také možností vrátit předchozí verzi aplikace bez větších komplikací [64].

5 Návrh a implementace systému pro běh distribuovaných aplikací

5.1 Definice požadavků na běh distribuované aplikace

V předchozí kapitole jsou představeny projekty, které řeší otázku jak spravovat jednotlivá cloudová řešení a aplikace napříč různými cloudovými poskytovateli co nejeefektivněji z jednoho místa. Současným trendem jak provozovat aplikace v prostředí cloudu je využití kontejnerizace a nástroje Kubernetes pro správu těchto kontejnerů, viz. kapitola Cloud native. Kubernetes architektura a nástroje pro správu K8s jsou navrženy pouze pro provozování jednoho clusteru. Mnoho společností se zajímá o propojení jednotlivých clusterů tak aby tvořily jeden celek. Tento přístup přináší mnoho výhod. Uživatelé mají možnost spravovat několik clusterů z jednoho místa. Z tohoto centrálního uzlu by uživatelé měli být schopni vytvářet zdroje v jednotlivých clusterech a také odtud sledovat chování a stav jednotlivých clusterů. Další výhodou je využití několika poskytovatelů pro lepší dostupnost svých služeb. Při výpadku jednoho poskytovatele máme pořád běžící aplikaci u druhého poskytovatele. Dalším použitím více clusterů je Edge cloud koncept. K8s cluster, umístěné v místě potřeby snižují odezvu na události a zvyšují dostupnost pro kritické aplikace. Propojením jednotlivých k8s clusterů se zabývá spousta společností a také k8s komunita. Konkrétně se jedná o projekt Kubernetes Federation, který je představen v předchozí kapitole. Správa více k8s clusterů je komplexní úkol a musí splňovat několik požadavků, aby bylo možné jednoduše a z jednoho místa efektivně provozovat distribuovanou aplikaci:

- První požadavkem takového systému je možnost jednotně spravovat desítky až tisíce k8s clusterů. Systém by měl nabízet jeden vstupní bod pro obsluhu všech k8s clusterů z jednoho místa. Manuálně spravovat jednotlivé clustery není výhodné, protože toto řešení špatně škáluje. Pokud se například společnost, využívající edge cloud v každé pobočce, bude rozrůstat o nové pobočky, bude muset pro každé dvě až čtyři pobočky zaměstnat nového operátora k8s clusteru. Efektivnějším řešením je definovat aplikaci v centrálním systému a poté podle potřeby distribuovat danou aplikaci na jednotlivé clustery. Výhodou použití k8s v jednotlivých lokacích je stejné a konzistentní prostředí přes všechny clustery.

Odpadá tak nutnost ovládat technologie od různých společností. Spouštění a provozování aplikací bude v jednotlivých lokacích probíhat stejně jako v testovacím prostředí. Pro doručování nových verzí aplikace, nabízí k8s tkz. rolling updates. Tato funkce umožňuje bezvýpadkové nahrání nové verze aplikace. K8s postupně nahrazuje pody se starou verzí aplikace, jeden po druhém a zároveň spouští pody s novou verzí. Nové funkce a opravy tak mohou být rychleji aplikovány. Pokud dojde k nahrání špatné verze aplikace, která nebude fungovat, k8s umožní se rychle vrátit na předchozí funkční verzi.

- Dalším plusem je použití k8s architektury a všech zdrojů a technik, které k8s nabízí. K8s je robustní nástroj, který v sobě zahrnuje vysokou dostupnost, škálovatelnost, rozšiřitelnost, je to řešení odolné vůči chybám, vyvíjené, spravované a podporované širokou komunitou. Pro zmíněné atributy se k8s stalo de facto standardem pro orchestraci kontejnerů. Například použití Pod konceptu, namísto kontejneru. Pod dovoluje seskupit více kontejnerů, které na sobě úzce závisí a potřebují být spuštěny na stejném serveru nebo mezi sebou sdílejí zdroje, do jednoho logického celku. Pod dovoluje také využití tkz. Side-car konceptu. Side-car je označení pro kontejner, který běží ve stejném podu jako kontejner uživatelské aplikace a poskytuje další služby jako je sběr logů z uživatelského kontejneru a jejich odeslání na centrální server.
- Pro přístup více subjektů do systému je nutné izolovat zdroje jednotlivých subjektů tak, aby na sebe vzájemně neviděly a nemohly tak ovlivňovat aplikace ostatních zákazníků. Pro tento problém poskytuje k8s namespace. Namespace je virtuální cluster uvnitř fyzického clusteru. Jednotlivé zdroje uvnitř namespace jsou izolovány od ostatních. Uživatelé jsou schopni vytvářet, prohlížet, upravovat a mazat pouze zdroje v přiděleném namespace. Namespace může být využit pro oddělení jednotlivých aplikací, různá oddělení jedné společnosti mohou mít vlastní namespace, případně různé firmy mohou být odděleny pomocí namespace.
- Dalším požadavkem je spuštění různých aplikací v clusterech situovaných v různých lokacích. Např. aplikace, kterou zákazníci v obchodě používají pro výběr zboží budeme provozovat ve všech obchodech. Aplikaci pro interní zaměstnance, která ulehčuje správu skladů se zbožím budeme provozovat ve všech skladech a také v největších obchodech s vlastním skladem. Pro tento účel je možné využít labely, neboli popisky jednotlivých objektů. Labels v k8s mají podobu klíč a k němu přiřazená hodnota. Jednotlivé k8s deploymenty je možné popsat labely clusterů, ve kterých má být daný deployment spuštěný. Aplikace pro zákazníky by tak mohla obsahovat label lokace:praha a label typ:prodejna.

Pomocí labelů je tak možné jednoznačně určit lokace a typ prodejny na které má být aplikace spuštěna.

- Důležitým aspektem pro běh distribuovaných aplikací je řízení přístupu uživatelů k jednotlivým zdrojům. K8s obsahuje několik mechanismů pro ověření zda uživatel má právo provést danou akci či nikoliv. Administrátoři tak mohou kontrolovat jaké akce mohou uživatelé vykonávat. Více o problematice a typech k8s autorizace je uvedeno v [65].

5.2 Návrh aplikace

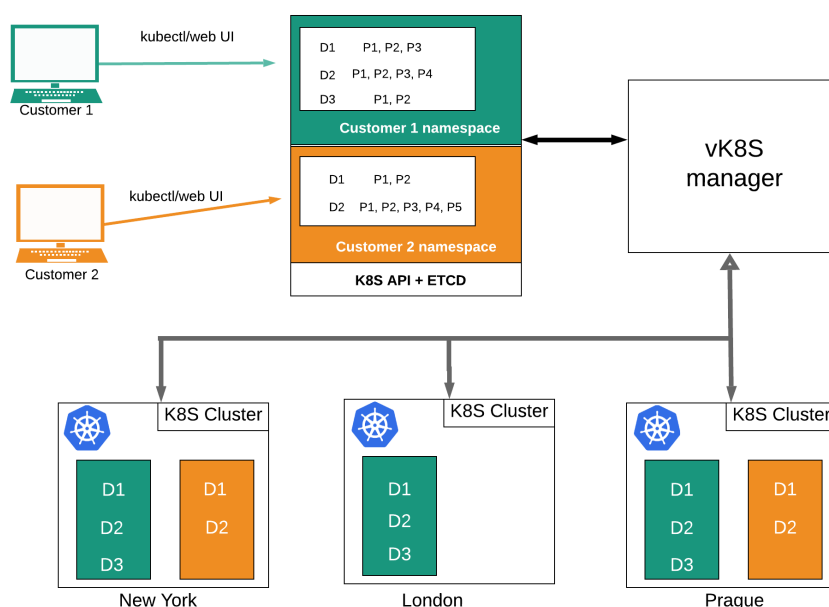
V následující kapitole je popsána architektura systému, který umožňuje běh cloud native aplikací v k8s clusterech podle požadavků definovaných v předchozí kapitole Definice požadavků na běh distribuované databáze. Architektura systému je zobrazena na obrázku 5.1. Systém se skládá ze tří menších celků. Prvním část se skládá z Kubernetes API a databáze Etcd. Uživatelé interagují s touto částí, s jejíž pomocí vytvářejí zdroje, které jsou dále distribuovány do jednotlivých clusterů. Druhá část systému se stará o distribuci uživatelských zdrojů do zvolených clusterů. Tato část se nazývá Controller. Třetí část je tvořena k8s clusterem, které jsou umístěné v různých lokacích a zajišťují běh zvolených aplikací.

První část, která se skládá pouze z k8s API a Etcd databáze, je vstupní bod pro uživatele, kteří budou vytvářet své aplikace. Tento koncept je převzatý z k8s Master serveru, ovšem bez služeb Controller Manageru a Scheduleru. K8s API umožňuje vytvářet nové definice zdrojů a manipulaci s nimi. Konfigurace je uložena v databázi Etcd. Uživatelé mohou použít existující řešení pro správu k8s zdrojů jako je webové rozhraní k8s nebo nástroj kubectl pro správu k8s z příkazového řádku. Tyto nástroje umožňují také monitorování jednotlivých zdrojů. Na obrázku můžeme vidět, že Customer 1 vytvořil v přiděleném namespace Deploymenty D1, D2, D3. Každý Deployment se skládá z N kontejnerů, které jsou označeny Pn. Zdroje, které se nachází v tomto centrálním API budou nazývány jako virtuální zdroje. Například virtuální Deployment D1 prozatím existuje pouze v centrálním API v podobě předpisu jak by měl být spuštěn a provozován. API také obsahuje mechanismy pro autentifikaci uživatelů na základě definovaných pravidel. Pro oddělení zdrojů jednotlivých uživatelů jsou použity namespacey. Uživatelé tak budou moci pracovat pouze se zdroji v přiděleném namespace. Tento přístup zajistí bezpečný a izolovaný přístup více uživatelů do systému. Jak je zobrazeno na obrázku, Customer 1, má možnost spravovat pouze zdroje v zeleném namespace a nikdy nemá možnost interagovat se zdroji Customera 2 v oranžovém namespace a naopak.

Úlohou Controlleru je distribuovat zdroje z centrálního API serveru do jednotlivých clusterů v různých lokacích podle definovaných pravidel. Controller sleduje databázi zdrojů v centrálním API. Pokud uživatel vytvoří nový Deployment, Controller zjistí tuto skutečnost a podle labelů, které jsou přiřazené tomuto Deploymentu vloží konfiguraci Deploymentu do k8s clusterů v požadovaných lokacích. Jak můžeme vidět na obrázku, všechny tři Deploymenty uživatele Customer 1 jsou spuštěny ve všech třech dostupných lokacích. Tato strategie je nejjednodušší. Deploymenty zákazníka Customer 2 jsou spuštěny pouze ve dvou ze tří lokací, konkrétně v New Yorku a Praze. Zákazník Customer 2 například nemá pobočku v Londýně a proto nepotřebuje spouštět

aplikaci v tomto regionu. Customer 2 má ovšem možnost v budoucnu spustit svou aplikaci v lokaci Londýn pokud to bude potřebovat. Uživatel tak přes centrální API přidá label, který popisuje lokaci v Londýně. Tuto změnu Controller zjistí a vloží definici do k8s clusteru v lokaci Londýn. Controller také podporuje spustit v lokaci Londýn pro uživatele Customer 2 pouze Deployment D1. Proto aby uživatelé mohli monitorovat své zdroje z centrálního API, Controller sleduje k8s clustery v jednotlivých lokacích a zpět do centrálního API vkládá informace o jednotlivých zdrojích. Uživatelé tak například vidí stav jednotlivých podů a další informace o Deploymentech.

V jednotlivých lokacích je spuštěn plnohodnotný k8s cluster. Tento cluster spouští aplikace podle konfigurace, kterou obdržel od Controlleru. Jednotlivé clustery jsou samostatné rozhodovací jednotky, které jsou na sobě nezávislé. Při výpadku spojení na centrální API, aplikace běžící v clusteru stále fungují a mohou tak obsluhovat uživatele. Po obnovení spojení jsou data o aplikacích nahrána zpět do centrálního API.

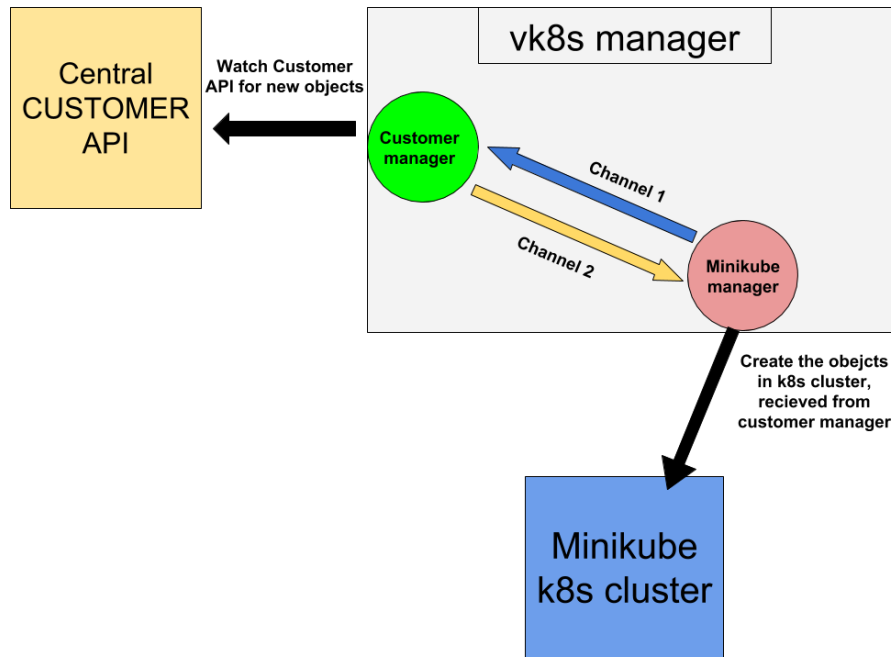


Obrázek 5.1: Architektura navrhované aplikace, zdroj: vlastní tvorba

5.3 Implementace prototypu aplikace

V předchozí kapitole je představena architektura systému, který umožňuje běh aplikací napříč více Kubernetes clustery. V této kapitole je představen prototyp aplikace, která tuto architekturu implementuje. Pro pojmenování jednotlivých částí aplikace jsou použity pojmy definované v architektuře aplikace. Pro vytvoření aplikace byl použit programovací jazyk Golang, zkráceně Go. Go programovací jazyk byl vytvořen zaměstnanci společnosti Google v roce 2009. Go je kompilovaný staticky typovaný jazyk, který umožňuje překlad zdrojových kódů do strojového kódu. Go se snaží být jednoduchý a dobře čitelný programovací jazyk. Go obsahuje na paměť nenáročná vlákna, nazývaná goroutines, které umožňují bez větších problémů v aplikacích spustit stovky až tisíce takovýchto goroutines. Pro komunikaci mezi jednotlivými goroutinami poskytuje jazyk Go tkz. Channely. Dalším plusem jazyka Go je rychlá kompilace, stejně jako běh samotné aplikace. Go obsahuje silnou typovou kontrolu a automatickou správu paměti. Součástí specifikace jazyka Go je i jednotný formát kódu. Pro formátování nabízí jazyk nástroj gofmt. Jazyk plně podporuje UTF-8 kódování a je plně open source [66]. V jazyce Go je vytvořeno mnoho projektů například Docker nebo Kubernetes. I tento fakt je jedním z důvodů proč byl vybrán právě jazyk Golang. Kubernetes nabízí pro jazyk Go klientské knihovny, které dávají vývojářům možnost jak pracovat s Kubernetes API.

Na obrázku 5.2 je zobrazena komunikace jednotlivých částí aplikace. Customer manager sleduje Customer API, složené z k8s API a Etcd databáze, ke kterému přistupují jednotliví uživatelé a vytvářejí v něm zdroje. Pokud Customer manager zjistí vytvoření nového zdroje, vytvoří událost, která tuto akci reprezentuje. Tato akce je zaslána Minikube manageru s využitím channelu. Minikube manager obdrží tuto událost a provede stejnou akci v Minikube clusteru. Příkladem takovéto akce může být vytvoření Podu s jedním kontejnerem uvnitř. Uživatel vytvoří definici Podu v Customer API. Customer manager zjistí tuto událost a zašle definici Minikube manageru. Minikube manager přijme z channelu akci a vytvoří tuto definici v k8s clusteru. V tomto clusteru je podle definice spuštěn Pod. Minikube manager sleduje Minikube cluster a odesílá informace. Například Minikube manager odesílá přes channel 1 Customer manageru informace o stavu vytvořeného Podu, který aplikuje tyto informace do Customer API. Uživatel tak vidí informace o podu v Customer API stejně jako by se připojoval přímo do Minikube clusteru.



Obrázek 5.2: Schéma vk8s manageru, zdroj: vlastní tvorba

Struktura aplikace je uvedena v příloze D. Funkcionalita aplikace je rozdělena do jednotlivých balíčků (packages). Soubor main.go obsahuje instrukce jak se má aplikace spustit. Jako první je vytvořen objekt Manager, který se skládá ze dvou dalších částí. Manager je součástí balíku cluster. Jedna část, nazývaná Customer manager slouží pro interakci s centrálním k8s API. Druhou částí je Minikube manager, který interaguje s k8s clusterem. Funkce implementující vytvoření nového Managera je zobrazena na ukázce kódu 5.1. Nejdříve jsou definovány jména namespace, které budou dále využity. Poté jsou vytvořeny dva managery, Customer Manager a Minikube Manager. Pokud vytvoření proběhlo úspěšně, metoda vrátí strukturu Manager.

Kód 5.1: funkce vytvoření nového managera

```
func New() *Manager {
    log.Println("Preparing new Manager")
    customerNamespace := "mynamespace"
    customerName := "customer1"
    minikubeNamespace := fmt.Sprintf("%s-%s", customerNamespace,
        customerName)
    m, err := vk8s.New(&manager.Config{KubeConfigPath:
        minikubeKubeconfig, Namespace: minikubeNamespace, Mode:
        manager.Minikube})
    if err != nil {
        log.Fatalf("Unable to setup new manager, %s", err)
    }
}
```

```

    }
    c, err := vk8s.New(&manager.Config{KubeConfigPath:
        customerKubeconfig, Namespace: customerNamespace, Mode:
        manager.Customer})
    if err != nil {
        log.Fatalf("Unable to setup new manager, %s", err)
    }
    return &Manager{m, c}
}

```

Implementace výše zmíněných Managerů se nachází v balíčku vk8s. Funkce vk8s.New() v ukázce kódu 5.2 přijímá jako vstupní parametr strukturu Config z balíčku manager. Tato Config struktura obsahuje informaci o cestě k souboru kubeconfig, název namespace a hodnotu Mode. Kubeconfig soubor se používá pro připojení a komunikaci s k8s API, obsahuje například ip adresu pro připojení k serveru nebo certifikáty k autorizaci. Hodnota Mode určuje zda bude vytvořen manager pro centrální customer API nebo pro k8s cluster minikube.

Kód 5.2: vk8s.New() funkce pro vytvoření customer a minikube managera

```

func New(cfg *manager.Config) (manager.Interface, error) {
    if cfg.Mode == "" || cfg.Namespace == "" ||
        cfg.KubeConfigPath == "" {
        return nil, fmt.Errorf("could not create new manager,
            incomplete configuration")
    }
    log.Printf("Preparing new %s manager", cfg.Mode)
    rc, err := k8s.NewRestConfig(k8s.Config{ConfigType:
        k8s.OutCluster, KubeConfigPath: cfg.KubeConfigPath})
    if err != nil {
        return nil, fmt.Errorf("Could not initialize rest config,
            err: %s", err)
    }
    clientset, err := kubernetes.NewForConfig(rc)
    if err != nil {
        return nil, fmt.Errorf("Could not initialize clientset,
            err: %s", err)
    }
    inf := informer.New(string(cfg.Mode))
    switch cfg.Mode {
        case manager.Customer:
            return &customer{vk8s{clientset, worker.New(clientset,
                inf), k8s.NewNamespaceObjectFromObject(clientset,

```



```

        &v1.Namespace{ObjectMeta: metav1.ObjectMeta{Name:
            cfg.Namespace}}, inf}}, nil
    case manager.Minikube:
        return &minikube{vk8s{clientset, worker.New(clientset,
            inf), k8s.NewNamespaceObjectFromObject(clientset,
            &v1.Namespace{ObjectMeta: metav1.ObjectMeta{Name:
            cfg.Namespace}}, inf}}, nil
    default:
        return nil, fmt.Errorf("unknown manager mode")
    }
}

```

Zmíněné managery implementují rozhraní `manager.Interface`, které obsahuje jednu metodu `Start(inEvent, outEvent)`, která přijímá dva channely jako vstupní parametry. Na ukázce kódu 5.3 je zobrazena metoda `Start()` pro `Customer` manager. Na začátku běhu funkce je ověřeno, že existuje potřebný namespace v `customer` API. Pokud neexistuje tak je vytvořen. V dalším kroku je vytvořen servisní účet, který bude aplikace používat pro práci s `k8s` API. Po vytvoření servisního účtu je spuštěna metoda `Watch(outEvent)`, která sleduje zdroje v centrálním `customer` API a je detailně popsána dále v textu. Posledním krokem funkce `Start()` je sledování akcí na vstupním channelu. Slovo `go` před definicí funkce značí, že funkce bude spuštěna v goroutině, neboli na paměť nenáročném vlákně. Uvnitř goroutiny se v nekonečném cyklu čeká na události, které jsou obdrženy z `inEvent` channelu. Channely v jazyce `Go` fungují pro komunikaci mezi goroutinami. V tomto případě jedna goroutina do channelu zapisuje data a druhá z tohoto channelu data čte. Pokud na channelu přijde nějaká událost, je tato událost předána do volání funkce `Run()`, která tuto událost aplikuje do `customer` API. Například vytvoření nového zdroje je reprezentováno jako struktura `Event` s typem události `Create` a objekt, který má být vytvořen. Do `inEvent` channelu zapisuje data `minikube` manager, který sleduje `k8s` cluster, díky tomu se informace z `k8s` clusteru dostanou do centrálního `customer` API.

Kód 5.3: funkce `Start` `Customer` managera

```

func (c *customer) Start(inEvent <-chan watch.Event, outEvent
    ...chan<- watch.Event) error {
    err := c.namespace.Apply()
    if err != nil {
        return fmt.Errorf("unable to start Customer k8s manager,
            %s", err)
    }
    log.Println("Customer k8s manager started")
    svc := &v1.ServiceAccount{ObjectMeta:

```

```

    metav1.ObjectMeta{Name: "default"}}
c.CoreV1().ServiceAccounts(c.namespace.GetName()).Create(svc)
err = c.Watch(outEvent)
if err != nil {
    return fmt.Errorf("unable to start Customer k8s manager,
        %s", err)
}
go func() {
    log.Println("Customer watch incoming events from hyperkube")
    for {
        select {
            case ev := <-inEvent:
                // apply event int vk8s
                c.w.Run(manager.Event{ev, manager.Customer},
                    worker.SetNamespace(c.namespace.GetName()))
        }
    }
}()
return nil
}

```

V rámci volání funkce `Start()` pro customer managera je spuštěna v goroutině funkce `watch`, která je zobrazena v ukázce kódu 5.4. Úkolem této funkce je sledovat zdroje customer API. Pokud jsou například přidány nové zdroje nebo dojde k modifikaci stávajících zdrojů, `Watch` funkce tuto událost zaznamená a odešle informaci do minikube clusterů. Na začátku funkce jsou vytvořeny watchery zdrojů v customer API. Watcher má za úkol sledovat jeden specifický zdroj v k8s API. Jestliže dojde k přidání, modifikaci nebo smazání zdroje je vygenerována událost, která je odeslána do channelu. Příkladem může být objekt `deploymentWatcher`, který sleduje zdroj `Deployment` v určitém namespace. Po vytvoření watcherů následuje spuštění goroutiny uvnitř které se v cyklu čeká na události zaslané od jednotlivých watcherů. Pokud je na channelu, který reprezentuje události spojené s deploymentem, obdržena událost je dále odeslána do všech `outEvent` channelů. Tímto mechanismem obdrží k8s cluster informace o novém deploymentu. Události z centrálního API jsou takto pomocí Manageru přeneseny do k8s clusterů. Minikube cluster obdrží na vstupním channelu informaci, kterou využijí a spustí v nich definovaný workload. Minikube manager funguje na stejném principu, s tím rozdílem, že sleduje k8s cluster a události z tohoto clusteru zasílá do customer manageru pomocí channelu, který tyto informace vkládá do centrálního customer API.

Kód 5.4: funkce Watch, která sleduje Customer API zdroje

```

func (c *customer) Watch(outEvent []chan<- watch.Event) error {
    log.Println("Watch for Customer cluster initiated")
    deploymentWatcher, err := c.Clientset.ExtensionsV1beta1().
        Deployments(c.namespace.GetName()).Watch(metav1.ListOptions{})
    if err != nil {
        return fmt.Errorf("unable to establish watch for Customer
            k8s, %s", err)
    }
    serviceWatcher, err := c.Clientset.CoreV1().
        Services(c.namespace.GetName()).Watch(metav1.ListOptions{})
    if err != nil {
        return fmt.Errorf("unable to establish watch for Customer
            k8s, %s", err)
    }
    configMapWatcher, err := c.Clientset.CoreV1().
        ConfigMaps(c.namespace.GetName()).Watch(metav1.ListOptions{})
    if err != nil {
        return fmt.Errorf("unable to establish watch for Customer
            k8s, %s", err)
    }
    statefulSetWatcher, err := c.Clientset.AppsV1beta2().
        StatefulSets(c.namespace.GetName()).Watch(metav1.ListOptions{})
    if err != nil {
        return fmt.Errorf("unable to establish watch for Minikube
            k8s, %s", err) }
    go func() {
        for {
            select {
            case ev := <-deploymentWatcher.ResultChan():
                // send deployment event into Minikube cluster
                for _, ch := range outEvent {
                    ch <- ev
                }
            case ev := <-serviceWatcher.ResultChan():
                // send service event into Minikube cluster
                for _, ch := range outEvent {
                    ch <- ev
                }
            case ev := <-configMapWatcher.ResultChan():
                // send config map event into Minikube cluster
                for _, ch := range outEvent {
                    ch <- ev
                }
            }
        }
    }()
}

```

```
    }  
    case ev := <-statefulSetWatcher.ResultChan():  
    // send statefull set event into Minikube cluster  
    for _, ch := range outEvent {  
        ch <- ev  
    }  
    }  
    }()  
    return nil  
}
```

6 Testování aplikace

Kapitola testování aplikace je rozdělena na dvě části. První část tvoří popis prostředí, které je potřebné pro spuštění aplikace. Druhá část testování je zaměřena na praktické testování fungování aplikace.

6.1 Definice prostředí pro testování aplikace

Testovací prostředí je vytvořeno na linuxové distribuci Ubuntu. První částí, kterou je potřeba spustit je k8s API a Etcd databáze, které představují Customer API se kterým uživatelé přímo komunikují. Kubernetes komunita vytvořila kód, který umožňuje spustit k8s cluster na lokálním počítači [67]. Spuštění k8s clusteru lokálně závisí na technologiích Docker, Etcd a programovacím jazyce Golang, které musí být na lokálním stroji nainstalovány. Dále musí být nainstalovány nástroje pro práci s certifikáty OpenSSL a CSFSSL, které k8s vyžaduje pro své fungování. V dalším kroku je spuštěn bashový skript, který vytvoří ze zdrojových kódů binární soubory jednotlivých služeb k8s, které jsou následně spuštěny. Skript také vytvoří Kubeconfig soubor pro komunikaci s k8s API. Pro běh aplikace jsou potřebné pouze komponenty k8s API a databáze Etcd, ostatní komponenty jako jsou scheduler a controller-manager mohou být vypnuty.

Pro reprezentaci k8s clusteru je využita technologie Minikube. Minikube je nástroj, který umožňuje snadno, pomocí jednoho příkazu, spustit k8s cluster. Minikube je možné spustit na Windows, macOS nebo Linuxovém operačním systému. Minikube na lokálním počítači spustí virtuální server ve kterém je již k8s připravené na použití. Minikube cluster je plnohodnotný k8s cluster, který běží všechny služby v jednom virtuálním stroji.

6.2 Testování distribuce Kubernetes objektů

Cílem testování je ověřit jak prototyp aplikace distribuuje zdroje z virtuálního clusteru do plnohodnotného k8s clusteru, který reprezentuje nástroj Minikube. Kubectl je nástroj příkazového řádku, který umožňuje vytvářet zdroje v k8s clusteru a vzdáleně tento cluster ovládat. Informace o jednotlivých clusterech jsou definovány v kubeconfig

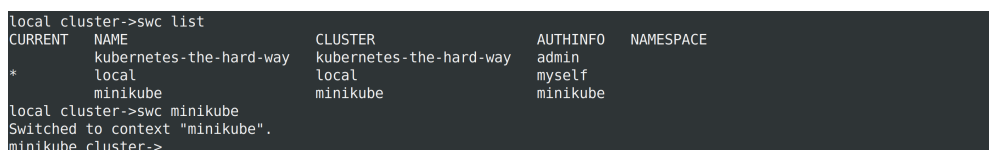
souborech. Kubectl nástroj dovoluje přepínat mezi jednotlivými kubeconfig soubory (kubectl používá kontexty pro rozlišení jednotlivých clusterů) a ovládat takto rozdílné k8s clustery. Pro snadnější přepínání mezi jednotlivými k8s clustery, je použit skript 6.1, který umí přepínat mezi jednotlivými kontexty. Přepnutí kontextu se vykoná pomocí příkazu `swc <název kontextu>`.

Kód 6.1: Bash skript pro práci s k8s kontexty

```
#!/bin/bash
if [ "$1" == "-h" ]; then
    echo "Usage: $0 [swc local - switch to local Kubernetes context]"
    exit 0
fi
if [ "$1" == "list" ]; then
    kubectl config get-contexts
    exit 0
fi

kubectl config use-context $1
```

Pro lepší přehlednost ve kterém clusteru byl znázorněný příkaz vykonán je upraven terminál tak, že zobrazuje aktuální název clusteru. Na obrázku 6.1 první řádek zobrazuje local cluster, to znamená že současný kontext je nastaven na virtuální k8s api. Dále je na prvním řádku spuštěn příkaz, který zobrazí seznam dostupných kontextů, kde hvězdičkou je označený aktivní kontext. Poté následuje `swc` příkaz 6.1, který přepne kontext na minikube cluster a změní se i popis řádku, který říká že aktuální kontext je minikube.



```
local cluster->swc list
CURRENT  NAME                CLUSTER                AUTHINFO  NAMESPACE
*        local              local                 myself    minikube
local cluster->swc minikube
Switched to context "minikube".
minikube cluster->
```

Obrázek 6.1: Vzhled terminálu, zdroj: vlastní tvorba

6.2.1 Distribuce Kubernetes node objektů

Aplikace umožňuje uživatelům sledovat jednotlivé zdroje a to i servery, které provozují jejich aplikace. V k8s jsou tyto servery označeny jako nody. Na obrázku 6.2 je zobrazen seznam dostupných nodů. První je lokální cluster, který reprezentuje virtuální k8s API a obsahuje jeden node s verzí 1.15.0-alpha a dalšími parametry. Druhý cluster minikube obsahuje rovněž jeden node, ovšem s rozdílnými parametry, např. jméno,

role nebo verze.

```

local cluster->k get nodes -o wide
NAME          STATUS    ROLES    AGE   VERSION          INTERNAL-IP  EXTERNAL-IP  OS-IMAGE
  KERNEL-VERSION  CONTAINER-RUNTIME
127.0.0.1     Ready    <none>    101m  v1.15.0-alpha.0.1222+d5a3db003916b1  127.0.0.1    <none>       Ubuntu 17.10
  4.13.0-46-generic  docker://18.6.3
local cluster->

local cluster->
local cluster->swc minikube
Switched to context "minikube".
minikube cluster->k get nodes -o wide
NAME          STATUS    ROLES    AGE   VERSION          INTERNAL-IP  EXTERNAL-IP  OS-IMAGE          KERNEL-VERSION  CON
TAINER-RUNTIME
minikube     Ready    master    87d   v1.12.4          192.168.122.8  <none>       Buildroot 2018.05  4.15.0          doc
ker://18.6.1
minikube cluster->

```

Obrázek 6.2: Seznam nodů v jednotlivých clusterech, zdroj: vlastní tvorba

Po spuštění aplikace dojde k inicializaci a spuštění managerů. Minikube manager informuje customer manager o dostupných zdrojích. V současné chvíli zde nejsou vytvořeny žádné zdroje. Minikube manager tak oznámí customer manageru informace o nodech, které má zaregistrované. Customer manager tuto událost obdrží a uloží tyto informace do virtuálního k8s API, kde si je uživatelé mohou prohlédnout. Výsledek této akce je uveden na obrázku 6.3. Uživatelé jsou zobrazeny nody v obou clusterech společně s jejich vlastnostmi.

```

local cluster->k get nodes -o wide
NAME          STATUS    ROLES    AGE   VERSION          INTERNAL-IP  EXTERNAL-IP  OS-IMAGE
  KERNEL-VERSION  CONTAINER-RUNTIME
127.0.0.1     Ready    <none>    119m  v1.15.0-alpha.0.1222+d5a3db003916b1  127.0.0.1    <none>       Ubuntu 17.
10  4.13.0-46-generic  docker://18.6.3
minikube     Ready    master    43s   v1.12.4          192.168.122.8  <none>       Buildroot
2018.05  4.15.0          docker://18.6.1
local cluster->

```

Obrázek 6.3: Seznam nodů po spuštění programu, zdroj: vlastní tvorba

6.2.2 Distribuce Kubernetes pod objektů

Dalším testovaným scénářem je testování distribuce podů z centrálního API do Minikube clusteru. Spouštění podů je základní a nezbytná funkce k8s clusterů. Nástroj kubectl umí vytvářet zdroje, které jsou definované s souborech, nazývaných manifesty. Definice podu je uvedena na ukázce kódu 6.2. Definice podu začíná parametrem apiVersion, který specifikuje verzi API, která bude použita pro vytvoření objektu. Parametr Kind uvádí typ objektu, v tomto případě se jedná o pod. Další částí jsou metadata, která připojují další informace o objektu jako jsou jméno nebo id objektu, případně namespace ve kterém má být objekt vytvořen. Posledním blokem jsou spec informace. Zde se nachází definice kontejneru, případně více kontejnerů, který má být v podu spuštěný. Nejdříve je nastaveno, že k8s nemá restartovat kontejner, dále název kontejneru uvnitř podu a definice kontejneru s aplikací. Konkrétně bude spuštěn kontejner BusyBox. Jedná se o malou linuxovou distribuci, která poskytuje základní

nástroje pro práci se systémem. Po spuštění kontejneru bude vypsána hláška a dojde k ukončení kontejneru.

Kód 6.2: pod.yml, definice podu

```
apiVersion: v1
kind: Pod
metadata:
  name: test-pod
  labels:
    app: testapp
spec:
  restartPolicy: Never
  containers:
  - name: test-container
    image: busybox
    command: ['sh', '-c', 'echo Hello world !']
```

Proces vytvoření zdroje pod je uvedena na obrázku 6.4. Po aplikování manifest souboru je spuštěn proces vytvoření zdroje. Customer manager zjistí vytvoření zdroje pod a vytvoří událost s odpovídajícími parametry, kterou odešle do minikube manageu. Ten vytvoří pod v minikube clusteru a o jeho stavu informuje customer manager. Pod se nejdříve nachází ve stavu Pending, během něhož je určeno na kterém nodu bude spuštěn a také dochází ke stažení potřebného kontejner image. Poté je kontejner ve stavu Running, neboli vykonává zadaný kód, výpis hlášky. Po úspěšném dokončení výpisu přechází kontejner do stavu Completed. Stejný výstup vypíše i minikube cluster. Stav podu je shodný v obou clusterech. Z minikube clusteru je možné vypsat průběh programu. Zde je vidět, že program vypsal hlášku “Hello world !”. Posledním krokem je odstranění podu. Zde bylo potřebné upravit kod aplikace, Aplikace totiž sleduje zdroje v určeném namespace a aplikuje všechny informace o těchto zdrojích do minikube clusteru v případě customer managera. Na druhé straně minikube manager aplikuje všechny změny z minikube k8s clusteru do centrálního API s využitím customer managera. Při tomto nastavení docházelo k zacyklení aplikace. Zdroj byl sice smazán v centrálním API ale než tato informace doputovala do minikube clusteru, minikube manager oznámil centrálnímu API informace o běžícím podu. Jako řešení je použit postup v customer manageru, který ověří příchozí události z minikube managera a dovolí vytvořit pouze objekt Node. Vytvoření je reprezentováno jako parametr type s hodnotou Added struktury Event, která reprezentuje událost.


```

local cluster->k create -f pod.yml
pod/test-pod created
local cluster->k get po
NAME      READY   STATUS    RESTARTS   AGE
test-pod  0/1     Pending   0           20s
local cluster->k get po
NAME      READY   STATUS    RESTARTS   AGE
test-pod  0/1     Completed 0           24s
local cluster->
local cluster->swc minikube
Switched to context "minikube".
minikube cluster->k get po
NAME      READY   STATUS    RESTARTS   AGE
test-pod  0/1     Completed 0           72s
minikube cluster->k logs test-pod
Hello world !
minikube cluster->swc local
Switched to context "local".
local cluster->k delete -f pod.yml
pod "test-pod" deleted
local cluster->k get po
No resources found.
local cluster->

```

Obrázek 6.4: Práce s pody, zdroj: vlastní tvorba

6.2.3 Distribuce Kubernetes deployment objektů

Deployments řídí bezstavové, stateless, aplikace běžící v clusteru. Deployment umí udržovat potřebný počet kontejnerů, zvyšovat nebo snižovat počet kontejnerů, provádět přechod na novější verzi aplikace, případně vrátit původní verzi aplikace v případě, že nová verze obsahuje chyby. Definice deploymentu je uvedena na ukázce kódu 6.3. V části spec je specifikován počet kontejnerů, neboli počet instancí aplikace. Další důležitou částí je selector, podle kterého jsou vybrány spravované kontejnery. Díky selectoru například udržuje deployment počet kontejnerů, pokud je počet kontejnerů s požadovaným labelem menší než je definováno, jsou vytvořeny nové kontejnery. V bloku template je definovaný kontejner a také jeho label a port na kterém aplikace poslouchá.

Kód 6.3: deployment.yml, definice deploymentu

```

apiVersion: apps/v1
kind: Deployment
metadata:
  name: diplomka-nginx-deployment
  labels:
    app: dnd
spec:
  replicas: 2
  selector:
    matchLabels:
      app: dnd
  template:
    metadata:
      labels:

```

```
    app: dnd
  spec:
    containers:
    - name: diplomka-nginx
      image: casek14/diplomka-nginx:v1
      ports:
      - containerPort: 80
```

Aplikace, která bude spuštěna pomocí výše definovaného deploymentu je jednoduchý webový server Nginx, který bude zobrazovat statickou stránku. Aplikace bude přijímat požadavky na portu 80. Tento port není přístupný pro uživatele. Pro přístup uživatelů bude ještě vytvořen zdroj Service, který umožní přístup uživatelů k vytvořené aplikaci. Service je opět definovaná pomocí manifest souboru 6.4. Důležitá část definice servicity je selector, který říká na které kontejnery má service směřovat požadavky. Typ service NodePort znamená, že na serveru s kontejnerem aplikace je připraven port pro přístup ke kontejnerům deploymentu. Service umí také Load-Balancing mezi spuštěnými kontejnery.

Kód 6.4: service.yml, definice service

```
apiVersion: v1
kind: Service
metadata:
  name: diplomka-nginx-service
spec:
  selector:
    app: dnd
  type: NodePort
  ports:
  - protocol: TCP
    port: 8080
    targetPort: 80
```

Na obrázku 6.5 je vidět postup nasazení deploymentu a service, která aplikaci zpřístupní uživatelům. K8s nasazení aplikace zabere určitý čas. Po přibližně 15 vteřinách je deployment ve fázi vytváření kontejnerů, což znamená stahování kontejneru. Proces spuštění kontejnerů zabral přibližně dvě a půl minuty. V produkčním prostředí s rychlým internetem by takový proces zabral okolo dvaceti vteřin.

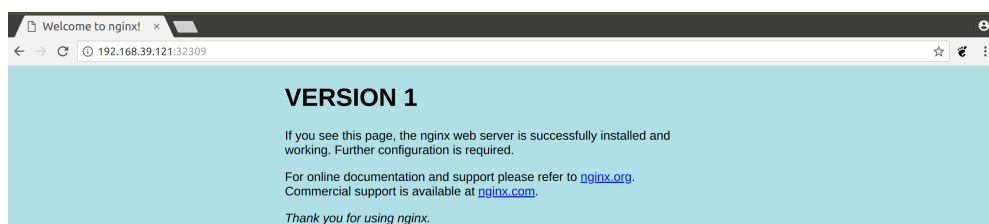
```

local cluster->k create -f deployment.yml
deployment.apps/diplomka-nginx-deployment created
service/diplomka-nginx-service created
local cluster->k get deployment
NAME READY UP-TO-DATE AVAILABLE AGE
diplomka-nginx-deployment 0/2 2 0 13s
local cluster->k get po
NAME READY STATUS RESTARTS AGE
diplomka-nginx-deployment-7dbfcc445-g4xx6 0/1 ContainerCreating 0 15s
diplomka-nginx-deployment-7dbfcc445-nwlfk 0/1 ContainerCreating 0 16s
local cluster->k get deployment
NAME READY UP-TO-DATE AVAILABLE AGE
diplomka-nginx-deployment 2/2 2 2 2m41s

```

Obrázek 6.5: Vytvoření deploymentu, zdroj: vlastní tvorba

Výsledná aplikace je zobrazena na obrázku 6.6. Aplikace je dostupná na portu, který minikube cluster alokoval. V definici deploymentu byla specifikována verze 1, která koresponduje s hlavním titulkem na obrázku 6.6.



Obrázek 6.6: Vzhled aplikace verze 1, zdroj: vlastní tvorba

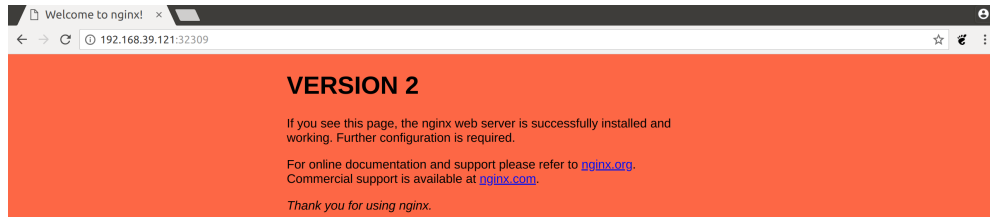
Jedním z rysů a výhod cloud native přístupu je doručování nových verzí aplikace. Cloud native aplikace nejsou statické co se týče nových funkcí a vylepšení. K8s dovoluje snadné nahrazení stávající verze novou verzí. V následujícím příkladu bude aplikace s verzí 1 nahrazena verzí aplikace 2. Postup přepnutí verze aplikace na verzi 2 je uveden na obrázku 6.7. Na prvním řádku je zobrazena verze aplikace před upgradem. Na druhém řádku je pomocí příkazu edit uvedena změněná verze v definici deploymentu, která je následně zobrazena. V tomto okamžiku již k8s provádí vypnutí kontejnerů verze 1 a začíná s postupným startováním kontejnerů s verzí 2. Aplikace s novou verzí je zobrazena obrázku 6.8, aplikace je dostupná na stejné adrese a portu.

```

local cluster->k describe deployment | grep Image
Image: casek14/diplomka-nginx:v1
local cluster->k edit deployment/diplomka-nginx-deployment
deployment.extensions/diplomka-nginx-deployment edited
local cluster->k describe deployment | grep Image
Image: casek14/diplomka-nginx:v2
local cluster->k get deployments
NAME READY UP-TO-DATE AVAILABLE AGE
diplomka-nginx-deployment 0/2 2 0 57m
local cluster->k get deployments
NAME READY UP-TO-DATE AVAILABLE AGE
diplomka-nginx-deployment 1/2 2 1 58m
local cluster->k get po
NAME READY STATUS RESTARTS AGE
diplomka-nginx-deployment-78b8bc7c94-szhhd 1/1 Running 0 2m13s
diplomka-nginx-deployment-78b8bc7c94-vn4xq 1/1 Running 0 2m5s

```

Obrázek 6.7: Úprava deploymentu, zdroj: vlastní tvorba

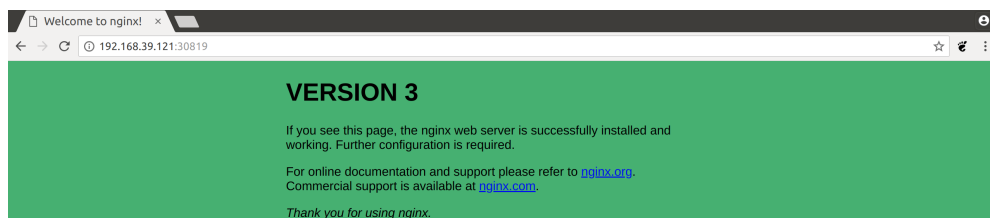


Obrázek 6.8: Vzhled apliakce verze 2, zdroj: vlastní tvorba

Deployment může být použitý také pro vrácení chybné verze aplikace. Tato technika je užitečná v okamžiku, že se do produkčního prostředí dostane verze aplikace, která nefunguje nebo obsahuje bug. Při testování této funkce, které je zobrazeno na obrázku 6.9, je nejdříve vytvořen deployment s aplikací verze 1. Následně byla verze aplikace změněna na verzi 2, tato akce byla uložena pomocí volby “--record”. K8s si tuto akci uloží, aby bylo možné se vrátit o krok zpět. Následně je změněna verze aplikace na v3. Třetí verze aplikace je zobrazena na obrázku 6.10. Po nasazení třetí verze je ovšem zjištěno, že barva pozadí aplikace neodpovídá zadání. Zřejmě došlo k pochybení programátora, a barva pozadí neodpovídá barvám, které společnost využívá. Proto je rozhodnuto vrátit aplikaci do verze 2. K tomuto účelu byla využita funkce “rollout”, která vrátí verzi aplikace o jeden krok zpět na požadovanou verzi 2.

```
local cluster->k create -f deployment.yml
deployment.apps/diplomka-nginx-deployment created
service/diplomka-nginx-service created
local cluster->k describe deployment diplomka-nginx-deployment | grep Image
Image: casek14/diplomka-nginx:v1
local cluster->k set image deployment/diplomka-nginx-deployment diplomka-nginx=casek14/diplomka-nginx:v2 --record
deployment.extensions/diplomka-nginx-deployment image updated
local cluster->k describe deployment diplomka-nginx-deployment | grep Image
Image: casek14/diplomka-nginx:v2
local cluster->k set image deployment/diplomka-nginx-deployment diplomka-nginx=casek14/diplomka-nginx:v3 --record
deployment.extensions/diplomka-nginx-deployment image updated
local cluster->k describe deployment diplomka-nginx-deployment | grep Image
Image: casek14/diplomka-nginx:v3
local cluster->k rollout undo deployment diplomka-nginx-deployment
^Cdeployment.extensions/diplomka-nginx-deployment
local cluster->k describe deployment diplomka-nginx-deployment | grep Image
Image: casek14/diplomka-nginx:v2
local cluster->
```

Obrázek 6.9: Vrácení verze aplikace pomocí rollout funkce, zdroj: vlastní tvorba



Obrázek 6.10: Vzhled aplikace verze 3, zdroj: vlastní tvorba

Jedním z testovaných aspektů je i škálování aplikací. Pokud současný počet instancí

aplikace nestíhá obsloužit všechny zákazníky je potřeba zvýšit jejich počet. K8s opět nabízí pro tento účel nástroje, které dovolují operátorům jednoduše zvýšit počet instancí. Service se postará o rozložení zátěže mezi všechny nové pody. Na obrázku 6.11 je vytvořen deployment, který se skládá ze dvou podů a také service směřující požadavky na tyto pody. Každý pod na portu 80 odpovídá na požadavky se svým jménem kontejneru. Zdrojový kód této aplikace je uveden v příloze A. Dále je spuštěn skript, který desetkrát pošle požadavek na service a vypíše odpověď. Z obrázku je patrné, že požadavky vyřizují pouze dva pody.

```
local cluster->k create -f go-deployment.yml
deployment.apps/diplomka-scale-deployment created
service/diplomka-scale-service created
local cluster->k get deployment
NAME                                READY    UP-TO-DATE    AVAILABLE    AGE
diplomka-scale-deployment          2/2      2              2            119s
local cluster->k get po
NAME                                READY    STATUS        RESTARTS    AGE
diplomka-scale-deployment-84d9d8878b-9rg5p  1/1      Running        0            2m10s
diplomka-scale-deployment-84d9d8878b-vjcqj  1/1      Running        0            2m10s
local cluster->./checkPods.sh
HOSTNAME: diplomka-scale-deployment-84d9d8878b-9rg5p
HOSTNAME: diplomka-scale-deployment-84d9d8878b-vjcqj
HOSTNAME: diplomka-scale-deployment-84d9d8878b-9rg5p
HOSTNAME: diplomka-scale-deployment-84d9d8878b-9rg5p
HOSTNAME: diplomka-scale-deployment-84d9d8878b-vjcqj
HOSTNAME: diplomka-scale-deployment-84d9d8878b-9rg5p
HOSTNAME: diplomka-scale-deployment-84d9d8878b-vjcqj
HOSTNAME: diplomka-scale-deployment-84d9d8878b-9rg5p
HOSTNAME: diplomka-scale-deployment-84d9d8878b-vjcqj
HOSTNAME: diplomka-scale-deployment-84d9d8878b-vjcqj
```

Obrázek 6.11: Test deploymentu se 2 pody, zdroj: vlastní tvorba

V dalším kroku je počet podů, které budou vyřizovat požadavky uživatelů zvýšen na deset 6.12. K8s ihned začne startovat nové kontejnery tak, aby bylo dosaženo požadovaného počtu deseti instancí aplikace. V přehledu podů je vidět, že dva původní pody zůstaly a k nim byly vytvořeny další pody. Patrné je to podle parametru “AGE” neboli stáří kontejneru.

```
local cluster->k scale deployment diplomka-scale-deployment --replicas=10
deployment.extensions/diplomka-scale-deployment scaled
local cluster->k get deployment
NAME                                READY    UP-TO-DATE    AVAILABLE    AGE
diplomka-scale-deployment          3/10     10             3            12m
local cluster->k get deployment
NAME                                READY    UP-TO-DATE    AVAILABLE    AGE
diplomka-scale-deployment          5/10     10             5            12m
local cluster->k get deployment
NAME                                READY    UP-TO-DATE    AVAILABLE    AGE
diplomka-scale-deployment          10/10    10             10           14m
local cluster->k get pods
NAME                                READY    STATUS        RESTARTS    AGE
diplomka-scale-deployment-84d9d8878b-6jblt  1/1      Running        0            3m2s
diplomka-scale-deployment-84d9d8878b-84sq6  1/1      Running        0            3m5s
diplomka-scale-deployment-84d9d8878b-9rg5p  1/1      Running        0            15m
diplomka-scale-deployment-84d9d8878b-dg4gg  1/1      Running        0            3m3s
diplomka-scale-deployment-84d9d8878b-gs4vv  1/1      Running        0            3m5s
diplomka-scale-deployment-84d9d8878b-hzcg1  1/1      Running        0            3m5s
diplomka-scale-deployment-84d9d8878b-jgvgt  1/1      Running        0            3m5s
diplomka-scale-deployment-84d9d8878b-mb8qm  1/1      Running        0            3m3s
diplomka-scale-deployment-84d9d8878b-qf7m6  1/1      Running        0            3m4s
diplomka-scale-deployment-84d9d8878b-vjcqj  1/1      Running        0            15m
```

Obrázek 6.12: Škálování deploymentu ze 2 na 10 podů, zdroj: vlastní tvorba

Service, která se stará o rozdělení zátěže, směřuje požadavky i na nově vytvořené

pody, které vybírá podle labelu `app=golang`. Na obrázku 6.13 je zobrazen výstup ze skriptu, požadavky směřují pokaždé na jeden z deseti podů, které obsahují aplikaci.

```
local cluster->./checkPods.sh
HOSTNAME: diplomka-scale-deployment-84d9d8878b-dg4gg
HOSTNAME: diplomka-scale-deployment-84d9d8878b-84sq6
HOSTNAME: diplomka-scale-deployment-84d9d8878b-qf7m6
HOSTNAME: diplomka-scale-deployment-84d9d8878b-9rg5p
HOSTNAME: diplomka-scale-deployment-84d9d8878b-gs4wv
HOSTNAME: diplomka-scale-deployment-84d9d8878b-jgvgt
HOSTNAME: diplomka-scale-deployment-84d9d8878b-6jb1t
HOSTNAME: diplomka-scale-deployment-84d9d8878b-84sq6
HOSTNAME: diplomka-scale-deployment-84d9d8878b-jgvgt
HOSTNAME: diplomka-scale-deployment-84d9d8878b-6jb1t
```

Obrázek 6.13: Test dostupnosti deploymentu s 10 pody, zdroj: vlastní tvorba

6.2.4 Distribuce Kubernetes statefulset objektů

Kubernetes umožňuje běh stateful aplikací, které uchovávají a spravují data. Příkladem stateful aplikací jsou databáze. Databáze potřebují uložení pro data, které vydrží restart kontejnerů. Tato funkcionality je v k8s reprezentována pomocí zdroje statefulset. Statefulset je obdobou deploymentu, ale pro stateful aplikace. Statefulset řídí počet replik, jejíž pody jsou vytvářeny jeden po druhém, mají definované pořadí a přiřazené stejné uložení. Pokud statefulset obsahuje tři repliky, jako první se vytvoří replika jedna a až po jejím úspěšném vytvoření a spuštění se začne spouštět druhá replika a poté následuje třetí replika. Uložení je do podu připojeno jako volume. Ovšem statefulset po restartování kontejneru tento volume nesmaže a tak nedojde ke ztrátě dat. Pro vytvoření volumu jsou použity tři k8s zdroje storageclass, persistentvolume (PV) a persistentvolumeclaim (PVC). Storageclass představuje reprezentaci uložení, kterou k8s cluster nabízí a obsahuje například parametr provisioner, který specifikuje jaký typ uložení bude použit. Může být použit NFS, CEPH a spoustu dalších. Příklad definice storage class je uveden v ukázce kódu 6.5. PV abstrahuje implementační detaily jednotlivých typů uložení a je spravováno administrátorem k8s clusteru. PVC je požadavek uživatele pro přidělení uložení pro určitý pod. PVC je alokováno z PV.

Kód 6.5: StorageClass definice pro minikube cluster

```
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: minikube-class
  provisioner: k8s.io/minikube-hostpath
  reclaimPolicy: Retain
```

Pro otestování distribuce stateful aplikací je použita jednoduchá aplikace, která po svém spuštění vytvoří ve specifickém adresáři soubor, jehož jméno je složeno z hostname daného kontejneru a časového údaje, kdy byl vytvořen. Aplikace následně po

vytvoření souboru zpřístupní tento adresář se souborem na portu 3000. Zdrojový kód aplikace je uveden v příloze B. Pro spuštění této aplikace je použitý manifest soubor uvedený v příloze C. Definice statefulsetu začíná definicí názvu této aplikace, následuje specifikace selektoru, tedy jaké pody patří k tomuto statefulsetu. Definice kontejneru a portu na kterém aplikace zpracovává požadavky je známá z předchozích příkladů. Definice “volumeMounts” reprezentuje trvalé uložisko, které bude připojené ke kontejneru a do něhož bude aplikace ukládat zmíněné soubory. Definice samotného uložiska je uvedena pod částí “volumeClaimTemplates”, která říká jaké uložisko bude použito. Parametr “storageClassName” se shoduje s názvem v definici pro storageclass 6.5. Velikost uložiska, které bude připojené k podu připojené má velikost 1GB. Poslední částí definice statefulsetu je definice systémové proměnné “DIRECTORY_PATH”, kterou aplikace využívá pro nastavení adresáře, kam jsou ukládány soubory, jejichž výpis je dostupný na portu 3000 daného podu. Tato systémová proměnná je definovaná s využitím dalšího k8s zdroje secretu. Secret slouží k uložení a správě citlivých informací jako jsou hesla, tokeny nebo ssh klíče. Definice secretu obsahuje název daného secretu, na který se poté odkazujeme v definici systémové proměnné v definici podu. Secret dále obsahuje data, která jsou použita pro nastavení hodnoty systémové proměnné. Hodnota je uložena jako klíč a k němu odpovídající hodnota. Aby nebyla cesta k adresáři se soubory uvedena pouze jako text, byla ještě zakódovaná pomocí nástroje base64 6.6. Výsledkem je hash, který reprezentuje cestu k adresáři. Tato cesta je shodná s cestou do které je připojené uložisko pro aplikaci, konkrétně je to adresář “/diplomka-serve-files”.

Kód 6.6: Zakódování textu pomocí base64 nástroje

```
local cluster->echo -n '/diplomka-serve-files' | base64
L2RpcGxvbWthLXNlcnZlLWZpbGVz
local cluster->echo 'L2RpcGxvbWthLXNlcnZlLWZpbGVz' |
base64 --decode
/diplomka-serve-files
```

Proces práce s statefulset aplikací je zobrazen na obrázku 6.14. Nejdříve jsou vytvořeny všechny potřebné zdroje, secret, statefulset a nakonec servica, která aplikaci zpřístupní uživatelům. Po stažení imagů jsou kontejnery v podu spuštěny. Ve výpisu podů je vidět, že druhý kontejner byl spuštěn pět vteřin po prvním kontejneru, to je vlastnost statefulsetu, který spouští kontejnery jeden po druhém na rozdíl od deploymentů, kde jsou všechny pody spuštěny současně. Obrázek 6.15 zobrazuje stav obou podů. Každý pod obsahuje právě jeden soubor, protože byly spuštěny pouze jednou. V další kroku dojde k okamžitému smazání podu fileserver-0. Tato akce donutí statefulset k vytvoření nového podu s aplikací. Ačkoliv se bude jednat o jiný pod, jeho název

bude stejný a stejně tak mu budou připojena již existující data vytvořená existujícím podem. Na obrázku 6.14 ve výpisu podů po smazání jednoho z nich je patrné, že nově vytvořený pod běží pouhé dvě minuty, kdežto druhý pod fileserver-1 běží od počátku bez restartu již minut šest. Obrázek 6.16 zobrazuje výpis dat. Pod fileserver-1 obsahuje pouze jeden soubor a pod fileserver-0 obsahuje právě dva soubory. Statefulset zdroj tedy aplikace byla aplikace schopná obsloužit, data se uchovala i mezi restartem podů.

```
local cluster->k create -f statefulset.yml
secret/secretenv created
statefulset.apps/fileserver created
service/serve-file-service created
local cluster->k get statefulset -o wide
NAME      READY   AGE    CONTAINERS   IMAGES
fileserver 1/2     18s    fileserver   casek14/diplomka-servefiles
local cluster->k get statefulset -o wide
NAME      READY   AGE    CONTAINERS   IMAGES
fileserver 2/2     39s    fileserver   casek14/diplomka-servefiles
local cluster->k get po
NAME      READY   STATUS    RESTARTS   AGE
fileserver-0 1/1     Running   0           2m41s
fileserver-1 1/1     Running   0           2m36s
local cluster->k delete po fileserver-0 --grace-period=0 --force
warning: Immediate deletion does not wait for confirmation that the running resource has been terminated. The resource may continue to run on the cluster indefinitely.
pod "fileserver-0" force deleted
local cluster->k get po
NAME      READY   STATUS    RESTARTS   AGE
fileserver-0 1/1     Running   0           2m4s
fileserver-1 1/1     Running   0           6m2s
local cluster->k get pvc
NAME                STATUS    VOLUME   CAPACITY   ACCESS MODES   STORAGECLASS   AGE
serverdir-fileserver-0 Bound      100Gi    100Gi      RWX           minikube-class 6m16s
serverdir-fileserver-1 Bound      100Gi    100Gi      RWX           minikube-class 6m11s
```

Obrázek 6.14: Vytvoření a otestování statefulsetu, zdroj: vlastní tvorba

```
local cluster->curl http://192.168.39.121:30280
<pre>
<a href="/fileserver-1-2019-04-18T15:07:22Z">fileserver-1-2019-04-18T15:07:22Z</a>
</pre>
local cluster->curl http://192.168.39.121:30280
<pre>
<a href="/fileserver-0-2019-04-18T15:07:18Z">fileserver-0-2019-04-18T15:07:18Z</a>
</pre>
```

Obrázek 6.15: Data stateful aplikace, zdroj: vlastní tvorba

```
local cluster->curl http://192.168.39.121:30280
<pre>
<a href="/fileserver-1-2019-04-18T15:07:22Z">fileserver-1-2019-04-18T15:07:22Z</a>
</pre>
local cluster->curl http://192.168.39.121:30280
<pre>
<a href="/fileserver-0-2019-04-18T15:07:18Z">fileserver-0-2019-04-18T15:07:18Z</a>
<a href="/fileserver-0-2019-04-18T15:11:22Z">fileserver-0-2019-04-18T15:11:22Z</a>
</pre>
local cluster->
```

Obrázek 6.16: Data stateful aplikace po restartu jednoho z podů, zdroj: vlastní tvorba

7 Závěry a doporučení

Cílem práce bylo analyzovat prostředí multi-cloud technologií a řešení, a následně navrhnout platformu, která bude umožňovat běh distribuovaných aplikací v prostředí multi-cloudu. Jak ukázala rešerše cloud prostředí, moderní cloud native prostředí využívá principů mikroslužeb, které rozdělují aplikace do menší autonomních částí. Tyto části se lépe testují, spravují, škálují a umožňují tak společně rychleji dodávat změny. Jednotlivé mikroslužby jsou doručovány jako kontejnery. Kontejnery umožňují přenositelnost aplikací mezi prostředími, protože závislosti aplikací jsou zabaleny s aplikací v kontejneru. Součástí cloud native přístupu jsou orchestrátory kontejnerů, které zjednodušují a automatizují správu kontejnerů. Ačkoliv na poli orchestrátorů existuje několik řešení, nejpoužívanějším orchestrátorem je Kubernetes, které se stalo synonymem pro orchestraci kontejnerů. K8s je velice dobrý nástroj, ovšem pro využití v multi-cloud prostředí je potřebné mít další nástroje, protože k8s se stará o správu pouze jednoho clusteru. Na trhu existují nástroje, které nabízejí jednotnou správu zdrojů napříč public cloudem, private cloudem a fyzickými servery. Projekt, který se zaměřuje na správu více k8s clusterů Kubernetes Federated, vytváří control plane pro centrální správu několika k8s clusterů.

Nabyté teoretické poznatky byly použity pro vytvoření architektury, která splňuje základní požadavky na správu více k8s clusterů. Pro vytvoření platformy je použita část architektury k8s, inspirována Kubernetes Federated projektem. Uživatelé interagují s virtuálním k8s API, které přijímá požadavky a ukládá je do Etcd databáze a funguje jako control plane. Toto centrální API obsahuje konfigurace aplikací, které mají být spuštěny ve vybraných clusterech. Distribuci těchto konfigurací do clusterů zajišťuje vk8s manager. Vk8s manager je aplikace vytvořená v jazyce Golang, stejně jako samotné k8s.

V rámci testování aplikace byly ověřeny základní operace, které k8s nabízí. Jako první byla testována distribuce nodů. Node je server na kterém jsou spouštěny uživatelské aplikace. Aplikace byla schopná synchronizovat nody z plnohodnotného k8s clusteru do centrálního API. Uživatelé tak mohou dohledat informace o názvech nodů, verzích komponent a také běhové prostředí kontejnerů, které daný node využívá. Druhým testovaným scénářem byla distribuce podů, které jsou základním k8s zdrojem. V tomto případě bylo ověřeno, že pody vytvořené v centrálním API jsou podle

konfigurace spuštěny k8s clusteru. Aplikace tento úkol bez problémů zvládla. Pod byl úspěšně spuštěn, vykonal svou ulohu, která spočívala ve vypsaní jednoduché hlášky, a poté byl ukončen. Všechny potřebné informace pro uživatele byly opět zapísány do centrálního API. Třetím testovaným případem byla práce s deploymenty. Distribuce a spuštění deploymentu v k8s clusteru fungovalo podle představ. Deployment byl spuštěn v minikube k8s clusteru a byl dostupný pro uživatele zvenčí s využitím dalšího k8s zdroje Servicy. Další funkcí deploymentu je možnost změnit verzi aplikace. Vytvářená aplikace zvládla změnu verze z v1 na verzi v2, která by dále změněna na verzi v3. Verze v3 ovšem obsahovala chybu a tak byla pomocí nástroje “rollout” vrácena zpět na funkční verzi v2. Dalším testem bylo škálování aplikace, která testovaná aplikace zvládla. Posledním testovaným scénářem byl statefulset. V rámci testu byly vytvořeny dva pody, každý s trvalým uožištěm pro soubory s kterými pracuje. Následně byl jeden pod smazán. Vytvořená data byla přístupná v nově vytvořeném podu podle předpokladů.

Aplikace pro distribuci konfigurací v testovaných scénářích obstála. Ve všech případech došlo k úspěšnému vytvoření zdrojů podle konfigurace v minikube k8s clusteru. Základní informace o zdrojích v minikube clusteru byly správně distribuovány do centrálního API, kde si je uživatelé mohli prohlížet. Během testování občas docházelo k chybnému zobrazování běžících zdrojů v centrálním API, ačkoliv byli dostupné a v minikube clusteru spuštěné.

Literatura

- [1] Remzi H a Andrea C. Arpaci-Dusseau. *Cloud-native file systems*. 10th Workshop on Hot Topics in Cloud Computing [online]. 2018, 2018, , 1 [cit. 2019-04-15]. Dostupné z: <https://www.usenix.org/system/files/conference/hotcloud18/hotcloud18-paper-arpaci-dusseau.pdf>
- [2] ZHANG, Shuai, Shufen ZHANG, Xuebin CHEN a Xiuzhen HUO. *Cloud Computing Research and Development Trend*. In: 2010 Second International Conference on Future Networks [online]. IEEE, 2010, 2010, s. 93-97 [cit. 2019-04-21]. DOI: 10.1109/ICFN.2010.58. ISBN 978-1-4244-5666-6. Dostupné z: <http://ieeexplore.ieee.org/document/5431874/>
- [3] BÖHM, MARKUS, STEFANIE LEIMEISTER, CHRISTOPH RIEDL a HELMUT KRCMAR. *Cloud Computing and Computing Evolution*. Technische Universität München (TUM), Germany [online]. 2010, 2010 [cit. 2018-12-15]. Dostupné z: https://s3.amazonaws.com/academia.edu.documents/7299069/05431874.pdf?AWSAccessKeyId=AKIAIWOWYYGZ2Y53UL3A&Expires=1555886954&Signature=4WW794UtzhnRNY7d21b0CaBPEWo%3D&response-content-disposition=inline%3B%20filename%3DCloud_computing_research_and_development.pdf
- [4] BUYYA, Rajkumar, James BROBERG a Andrzej GOSCINSKI. *Cloud computing: principles and paradigms*. Hoboken, N.J.: Wiley, c2011. ISBN 978-0-470-88799-8.
- [5] Mell, Peter and Grance, Tim and others. *The NIST definition of cloud computing*. Computer Security Division, Information Technology Laboratory, National Institute of Standards and Technology Gaithersburgb [online]. 2011, 2010 [cit. 2018-12-15]. Dostupné z: <http://faculty.winthrop.edu/domanm/csci411/Handouts/NIST.pdf>
- [6] Luis M. Vaquero, Luis Rodero-Merino, Juan Caceres a Maik Lindner. *A Break in the Clouds: Towards a Cloud Definition*. ACM SIGCOMM Computer Communication Review [online]. 2009, 2009, 39(1) [cit. 2018-12-17]. Dostupné z: <http://faculty.winthrop.edu/domanm/csci411/Handouts/NIST.pdf>

- [7] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz a Andy Konwinski. *Above the Clouds: A Berkeley View of Cloud Computing*. Electrical Engineering and Computer Sciences University of California at Berkeley [online]. 2009, 2009 [cit. 2018-12-17]. Dostupné z: <https://www2.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-28.pdf>
- [8] Alan Murphy. *WHITE PAPER Virtualization Defined - Eight Different Ways* [online]. 2007, 2007 [cit. 2018-12-17]. Dostupné z: <https://worldtechit.com/wp-content/uploads/2015/07/f5-white-paper-virtualization-defined-eight-different-ways-.pdf>
- [9] Leah Alger. *HSBC's journey with Amazon Cloud*. DevOps Online [online]. Velká Británie, 2018 [cit. 2018-12-20]. Dostupné z: <http://www.devopsonline.co.uk/hsbcs-journey-with-amazon-cloud/>
- [10] Edwin Yapp. *Culture shift imperative in cloud adoption: HSBC CIO*. Digital News Asia [online]. 2017 [cit. 2018-12-20]. Dostupné z: <https://www.digitalnewsasia.com/business/culture-shift-imperative-cloud-adoption-hsbc-cio>
- [11] , Sandro Brunner, Martin Blöchliger, Giovanni Toffetti, Josef Spillner, Thomas Michael Bohnert. *Experimental Evaluation of the Cloud-Native Application Design*. 2015 IEEE/ACM 8th International Conference on Utility and Cloud Computing (UCC) [online]. IEEE, 2015 [cit. 2019-01-07]. Dostupné z: <https://digitalcollection.zhaw.ch/bitstream/11475/7419/1/cnaeval-archive.pdf>
- [12] BIAS, Randy. *The History of Pets vs Cattle and How to Use the Analogy Properly*. Cloud scaling [online]. 29 Sep 2016 [cit. 2019-01-07]. Dostupné z: <http://cloudscaling.com/blog/cloud-computing/the-history-of-pets-vs-cattle/>
- [13] CNCF Cloud Native Definition v1.0. *Cloud native computing foundation github* [online]. 11 Jun 2018 [cit. 2019-01-07]. Dostupné z: <https://github.com/cncf/toc/blob/master/DEFINITION.md>
- [14] NANE KRATZKE a PETER-CHRISTIAN QUINT. *Understanding cloud-native applications after 10 years of cloud computing - A systematic mapping study*. The Journal of Systems and Software [online]. 2017, 126 [cit. 2019-04-22]. Dostupné z: https://www.researchgate.net/profile/Nane_Kratzke/publication/312045183_Understanding_Cloud-native_Applications_after_10_Years_of_Cloud_Computing_-_A

- Systematic_Mapping_Study/links/588202be4585150dde401522/Understanding-Cloud-native-Applications-after-10-Years-of-Cloud-Computing-A-Systematic-Mapping-Study.pdf*
- [15] HADDAD, Einas. *Service-Oriented Architecture: Scaling the Uber Engineering Codebase As We Grow*. Uber Engineering [online]. 2015 [cit. 2019-01-15]. Dostupné z: <https://eng.uber.com/soa/>
 - [16] CALCADO, Phil. *Building Products at SoundCloud —Part I: Dealing with the Monolith*. SoundCloud developers [online]. 2014 [cit. 2019-01-15]. Dostupné z: <https://developers.soundcloud.com/blog/building-products-at-soundcloud-part-1-dealing-with-the-monolith>
 - [17] GEITGEY, Adam. *I-Tier: Dismantling the Monolith*. Groupon Engineering [online]. 2013 [cit. 2019-01-15]. Dostupné z: <https://engineering.groupon.com/2013/misc/i-tier-dismantling-the-monoliths/>
 - [18] Netflix/ChaosMonkey. *Netflix GitHub* [online]. 2019 [cit. 2019-01-18]. Dostupné z: <https://github.com/Netflix/chaosmonkey>
 - [19] WU, Andy. *White Paper - Taking the Cloud-Native Approach with Microservices* [online]. In: . [cit. 2019-01-20]. Dostupné z: <https://cloud.google.com/files/Cloud-native-approach-with-microservices.pdf>
 - [20] ARMIN BALALAIE, ABBAS HEYDARNOOR a POOYAN JAMSHID. *Migrating to Cloud-Native Architectures Using Microservices: An Experience Report*. BALALAIE, Armin; HEYDARNOORI, Abbas; JAMSHIDI, Pooyan. *Migrating to cloud-native architectures using microservices: an experience report*. In: *European Conference on Service-Oriented and Cloud Computing* [online]. Springer, 2015 [cit. 2019-01-20]. Dostupné z: <https://arxiv.org/pdf/1507.08217.pdf>
 - [21] SLEE, Mark, Aditya AGARWAL a Marc KWIATKOWSKI. *Thrift: Scalable Cross-Language Services Implementation*. Facebook White Paper [online]. 2007 [cit. 2019-01-20]. Dostupné z: <https://users.cs.jmu.edu/bernstdh/Web/CS462/thrift-20070401.pdf>
 - [22] *OpenShift Enterprise Documentation*. Red Hat OpenShift [online]. 2019 [cit. 2019-01-26]. Dostupné z: https://docs.openshift.com/enterprise/3.0/architecture/core_concepts/containers_and_images.html
 - [23] *What's LXC?*. Linux containers [online]. 2019 [cit. 2019-01-26]. Dostupné z: <https://linuxcontainers.org/lxc/introduction/>

- [24] *Rkt overview. CoreOS [online].* 2019 [cit. 2019-01-26]. Dostupné z: <https://coreos.com/rkt/>
- [25] *Docker [online].* 2019 [cit. 2019-01-26]. Dostupné z: <https://www.docker.com/>
- [26] *Docker container use adoption devops clusterhq survey.* In: *Nanobox [online].* 16 Jun 2017 [cit. 2019-01-26]. Dostupné z: <https://content.nanobox.io/content/images/2017/06/docker-container-use-adoption-devops-clusterhq-survey.png>
- [27] COMBE, Theo, Antony MARTIN a Roberto DI PIETRO. *To Docker or Not to Docker: A Security Perspective.* *IEEE Cloud Computing [online].* 2016, 3(5), 54-62 [cit. 2019-01-23]. DOI: 10.1109/MCC.2016.100. ISSN 2325-6095. Dostupné z: <http://ieeexplore.ieee.org/document/7742298/>
- [28] *Docker Containers - Talk Linux Day.* In: *SlideShare [online].* 20 Jun 2016 [cit. 2019-02-10]. Dostupné z: <https://image.slidesharecdn.com/dockercontainerslinuxday2015-160620093427/95/docker-containers-talk-linux-day-2015-15-638.jpg?cb=1466415275>
- [29] *Namespaces - overview of Linux namespaces. Linux man pages online [online].* 2019 [cit. 2019-01-23]. Dostupné z: <http://man7.org/linux/man-pages/man7/namespaces.7.html>
- [30] *Cgroups - Linux control groups. Linux man pages online [online].* 2019 [cit. 2019-01-23]. Dostupné z: <http://man7.org/linux/man-pages/man7/cgroups.7.html>
- [31] MARTIN, John Paul, A. KANDASAMY a K. CHANDRASEKARAN. *Exploring the support for high performance applications in the container runtime environment.* *Human-centric Computing and Information Sciences [online].* 2018, 8(1) [cit. 2019-04-22]. DOI: 10.1186/s13673-017-0124-3. ISSN 2192-1962. Dostupné z: <https://hcis-journal.springeropen.com/articles/10.1186/s13673-017-0124-3>
- [32] *Rkt and SELinux. Rkt GitHub page [online].* [cit. 2019-01-25]. Dostupné z: <https://github.com/rkt/rkt/blob/master/Documentation/selinux.md>
- [33] *About Open Container Initiative. Open container initiative [online].* 2019 [cit. 2019-02-02]. Dostupné z: <https://www.opencontainers.org/about>
- [34] *Open Container Initiative Runtime Specification. OpenContainer GitHub page [online].* 2019 [cit. 2019-02-02]. Dostupné z: <https://github.com/opencontainers/runtime-spec/blob/master/spec.md>

- [35] *Image Format Specification. OpenContainer GitHub page* [online]. 2019 [cit. 2019-02-02]. Dostupné z: <https://github.com/opencontainers/image-spec/blob/master/spec.md>
- [36] ELDRIDGE, Isaac. *What Is Container Orchestration?*. New Relic [online]. 17 Jul 2018 [cit. 2019-02-08]. Dostupné z: <https://blog.newrelic.com/engineering/container-orchestration-explained/>
- [37] CARTER, Eric. *2018 Docker Usage Report. Sysdig* [online]. 29 May 2018 [cit. 2019-02-08]. Dostupné z: <https://sysdig.com/blog/2018-docker-usage-report/>
- [38] RENAN DELVALLE, PRADYUMNA KAUSHIK, ABHISHEK JAIN, JESSICA HARTOG, AND MADHUSUDHAN GOVINDARAJU. *Owards Efficient Resource Management on Heterogeneous Clusters with Apache Mesos*. 2017 IEEE 10th International Conference on Cloud Computing (CLOUD) [online]. IEEE, 2017, , 262-269 [cit. 2019-02-10]. Dostupné z: <http://cloud.cs.binghamton.edu/wordpress/wp-content/uploads/2017/07/electron-1.pdf>
- [39] LUPO, Gianluca. *Setup Mesos on a single node*. In: *Swiss push* [online]. 5 Dec 2014 [cit. 2019-02-10]. Dostupné z: <http://www.swisspush.org/assets/images/mesos/mesos-1.png>
- [40] *Marathon Docs. Mesos Sphere* [online]. [cit. 2019-02-20]. Dostupné z: <https://mesosphere.github.io/marathon/>
- [41] KAPOOR, Sandhya. *How Watson Health Cloud Deploys Applications with Kubernetes*. In: *Kubernetes blog* [online]. 14 Jul 2017 [cit. 2019-02-26]. Dostupné z: https://lh3.googleusercontent.com/EU3DgtFKagWp5S0UpKj-wRgx8WK2nvQ2BG-4dGio57pGNj42A7Lip9IARBba34hIm84-_7zwWt6iImQE8beSqLxpzXm-2w_84M_X2IHQ7jvpWtIDMF81hmq6N4hGSxp6DQoFW5qX
- [42] *Calico for Kubernetes. Project Calico* [online]. 2019 [cit. 2019-02-26]. Dostupné z: <https://docs.projectcalico.org/v2.0/getting-started/kubernetes/>
- [43] *Flannel. CoreOS GitHub page* [online]. [cit. 2019-02-26]. Dostupné z: <https://github.com/coreos/flannel>
- [44] *Run Weave Net with Kubernetes in Just One Line. Weaveworks* [online]. 27 Sep 2016 [cit. 2019-02-26]. Dostupné z: <https://www.weave.works/blog/weave-net-kubernetes-integration/>

- [45] BUYYA, Rajkumar a Satish Narayana SRIRAMA. *Fog and edge computing: principles and paradigms*. Hoboken, NJ, USA: John Wiley Sons, 2019. ISBN 9781119524984.
- [46] GRIMBERG, Andrew. *Akraino Wiki Page*. Akraino Edge stack [online]. 22 Feb 2019 [cit. 2019-03-06]. Dostupné z: <https://wiki.akraino.org/>
- [47] CHAMBERS, Brian, Caleb HURD a Alex CRANE. *Bare Metal K8s Clustering at Chick-fil-A Scale*. In: Medium [online]. 26 Jun [cit. 2019-03-07]. Dostupné z: <https://medium.com/@cfatechblog/bare-metal-k8s-clustering-at-chick-fil-a-scale-7b0607bd3541>
- [48] CHAMBERS, Brian, Caleb HURD, Alex CRANE, Morgan MCENTIRE, Jamie ROBERTS a Laura JAUHCH. *Edge Computing at Chick-fil-A*. In: Medium [online]. 30 Jul 2018 [cit. 2019-03-07]. Dostupné z: <https://medium.com/@cfatechblog/edge-computing-at-chick-fil-a-7d67242675e2>
- [49] SCHAEFER, Keith a Alain NICOLET. *Edge AI in a Smarter Chick-fil-A*. In: Medium [online]. 22 Oct 2018 [cit. 2019-03-07]. Dostupné z: <https://medium.com/@cfatechblog/edge-ai-in-a-smarter-chick-fil-a-6e56fe026154>
- [50] GOYAL, Sumit. *Public vs Private vs Hybrid vs Community - Cloud Computing: A Critical Review*. International Journal of Computer Network and Information Security [online]. 2014, 6(3), 20-29 [cit. 2019-04-22]. DOI: 10.5815/ijcnis.2014.03.03. ISSN 20749090. Dostupné z: <http://www.mecs-press.org/ijcnis/ijcnis-v6-n3/v6n3-3.html>
- [51] *Hybrid and Multi-Cloud Architecture Patterns*. Google cloud solutions [online]. 24 Oct 2018 [cit. 2019-03-15]. Dostupné z: <https://cloud.google.com/solutions/hybrid-and-multi-cloud-architecture-patterns>
- [52] *Hybrid multi cloud pattern edge hybrid*. In: Google cloud solutions [online]. [cit. 2019-03-15]. Dostupné z: <https://cloud.google.com/solutions/images/hybrid-multi-cloud-pattern-edge-hybrid.svg>
- [53] *ManageIQ Documentation*. ManageIQ [online]. 2019 [cit. 2019-03-20]. Dostupné z: <http://manageiq.org/docs/>
- [54] JANSEN, Geert. *Managing heterogeneous environments with ManageIQ*. LWM.net [online]. 16 Mar 2016 [cit. 2019-03-20]. Dostupné z: <https://lwn.net/Articles/680060/>
- [55] *Cloud Foundry Overview*. Cloud Foundry [online]. 2019 [cit. 2019-03-23]. Dostupné z: <https://docs.cloudfoundry.org/concepts/overview.html>

- [56] *Garden-runC Release. Cloud Foundry GitHub page [online]. 2019 [cit. 2019-03-23]. Dostupné z: <https://github.com/cloudfoundry/garden-runc-release>*
- [57] *Pivotal Cloud Foundry documentation. Pivotal Cloud Foundry [online]. 2019 [cit. 2019-03-23]. Dostupné z: <https://docs.pivotal.io/pivotalcf/2-4/pas/intro.html>*
- [58] *Cloud foundry BOSH get started. Cloud Foundry BOSH [online]. 2019 [cit. 2019-03-23]. Dostupné z: <https://www.cloudfoundry.org/get-started/>*
- [59] *Mist.io getting started. Mist.io [online]. 2019 [cit. 2019-03-25]. Dostupné z: <https://docs.mist.io/category/3-getting-started>*
- [60] REHMAN, Irfan Ur, Paul MORIE a Shashidhara TD. *Kubernetes Federation Evolution. Kubernetes blog [online]. 12 Dec 2018 [cit. 2019-03-28]. Dostupné z: <https://kubernetes.io/blog/2018/12/12/kubernetes-federation-evolution/>*
- [61] *Kubernetes federation. Kubernetes blog [online]. 2018 [cit. 2019-03-28]. Dostupné z: <https://raw.githubusercontent.com/kubernetes-sigs/federation-v2/master/docs/images/concepts.png>*
- [62] *Global Continuous Delivery with Spinnaker. Medium, Netflix technology blog [online]. 16 Nov 2015 [cit. 2019-03-30]. Dostupné z: <https://medium.com/netflix-techblog/global-continuous-delivery-with-spinnaker-2a6896c23ba7>*
- [63] FEINER, Tom, Katriel TRAUM, Nir TARCIC a Matt DUFTLER. *Guest post: Multi-Cloud continuous delivery using Spinnaker at Waze. Google cloud blog [online]. 16 Feb 2017 [cit. 2019-03-30]. Dostupné z: <https://cloud.google.com/blog/products/gcp/guest-post-multi-cloud-continuous-delivery-using-spinnaker-at-waze>*
- [64] *Spinnaker concepts. Spinnaker [online]. 2019 [cit. 2019-03-31]. Dostupné z: <https://www.spinnaker.io/concepts/>*
- [65] *Authorization Overview. Kubernetes documentation [online]. 2019 [cit. 2019-04-02]. Dostupné z: <https://kubernetes.io/docs/reference/access-authn-authz/authorization/>*
- [66] GIEBEN, Miek. *Learning Go [online]. 2018 [cit. 2019-04-07]. Dostupné z: <https://miek.nl/go/>*

- [67] *Getting started locally. Kubernetes community GitHub page [online]. 2017 [cit. 2019-04-07]. Dostupné z: <https://github.com/kubernetes/community/blob/master/contributors/devel/running-locally.md#starting-the-cluster>*
- [68] SAYFAN, Gigi. *Mastering Kubernetes [online]. Second edition. Birmingham: Packt publishing, 2018 [cit. 2019-04-05]. ISBN 978-1-78899-978-6.*

Seznam obrázků

3.1	Nejpoužívanější kontejnerové technologie, zdroj: [26]	12
3.2	Kontejnerové vrstvy, zdroj: [28]	13
3.3	architektura Mesos orchestrátoru, zdroj:[39]	19
3.4	architektura Mesos orchestrátoru, zdroj: [41]	21
4.1	Google edge architektura, zdroj: [52]	28
4.2	Architektura projektu Kubernetes Federation, zdroj: [61]	34
5.1	Architektura navrhované aplikace, zdroj: vlastní tvorba	40
5.2	Schéma vk8s manageru, zdroj: vlastní tvorba	42
6.1	Vzhled terminálu, zdroj: vlastní tvorba	49
6.2	Seznam nodů v jednotlivých clusterech, zdroj: vlastní tvorba	50
6.3	Seznam nodů po spuštění programu, zdroj: vlastní tvorba	50
6.4	Práce s pody, zdroj: vlastní tvorba	52
6.5	Vytvoření deploymentu, zdroj: vlastní tvorba	54
6.6	Vzhled aplikace verze 1, zdroj: vlastní tvorba	54
6.7	Úprava deploymentu, zdroj: vlastní tvorba	54
6.8	Vzhled aplikace verze 2, zdroj: vlastní tvorba	55
6.9	Vrácení verze aplikace pomocí rollout funkce, zdroj: vlastní tvorba	55
6.10	Vzhled aplikace verze 3, zdroj: vlastní tvorba	55
6.11	Test deploymentu se 2 pody, zdroj: vlastní tvorba	56
6.12	Škálování deploymentu ze 2 na 10 podů, zdroj: vlastní tvorba	56
6.13	Test dostupnosti deploymentu s 10 pody, zdroj: vlastní tvorba	57
6.14	Vytvoření a otestování statefulsetu, zdroj: vlastní tvorba	59
6.15	Data stateful aplikace, zdroj: vlastní tvorba	59
6.16	Data stateful aplikace po restartu jednoho z podů, zdroj: vlastní tvorba	59

Seznam tabulek

2.1	Tabulka lokací datových center různých poskytovatelů	6
-----	--	---

Seznam ukázek kódu

3.1	Příklad docker file souboru	13
3.2	Příklad docker compose souboru	18
5.1	funkce vytvoření nového managera	42
5.2	vk8s.New() funkce pro vytvoření customer a minikube managera	43
5.3	funkce Start Customer managera	44
5.4	funkce Watch, která sleduje Customer API zdroje	45
6.1	Bash skript pro práci s k8s kontexty	49
6.2	pod.yml, definice podu	51
6.3	deployment.yml, definice deploymentu	52
6.4	service.yml, definice service	53
6.5	StorageClass definice pro minikube cluster	57
6.6	Zakódování textu pomocí base64 nástroje	58

Přílohy

A. Zdrojový kód aplikace zobrazující hostname

```
package main

import (
    "fmt"
    "log"
    "net/http"
    "os"
)

func handler(w http.ResponseWriter, r *http.Request) {
    name, err := os.Hostname()
    if err != nil{
        log.Println("Unable to retrieve hostname")
    }
    fmt.Fprintf(w, "HOSTNAME: %s", name)
}

func main() {
    http.HandleFunc("/", handler)
    log.Fatal(http.ListenAndServe(":80", nil))
}
```

B. Zdrojový kód aplikace zobrazující soubory v definovaném adresáři

```
package main

import (
    "log"
    "net/http"
    "os"
    "time"
)

const directoryToServe = "/tmp"

func main() {
    path := os.Getenv("DIRECTORY_PATH")
    if path == "" {
        path = directoryToServe
    }
    log.Printf("Serving directory %s", path)
    hostname, err := os.Hostname()
    if err != nil {
        log.Printf("Cannot get hostname")
    }
    time := time.Now().Format(time.RFC3339)
    fileName := path+"/"+hostname+"-"+time
    newFile, err := os.Create(fileName)
    if err != nil {
        log.Printf("unable to create a file, %s", err)
    }
    log.Printf("New file created: %s", newFile.Name())

    fs := http.FileServer(http.Dir(path))
    http.Handle("/", fs)
    log.Println("Serving files on port 3000")
    http.ListenAndServe(":3000", nil)
}
```

C. Manifest pro vytvoření statefulsetu

```
apiVersion: v1
kind: Secret
metadata:
  name: secretenv
type: Opaque
data:
  dirpath: L2RpcGxvbWthLXNlcnZlLWZpbGVz
---
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: fileserver
spec:
  serviceName: "fileserver-service"
  replicas: 2
  selector:
    matchLabels:
      app: fileserver
  template:
    metadata:
      labels:
        app: fileserver
    spec:
      containers:
        - name: fileserver
          image: casek14/diplomka-servefiles
          ports:
            - containerPort: 3000
              name: servefiles
          volumeMounts:
            - name: serverdir
              mountPath: /diplomka-serve-files
          env:
            - name: DIRECTORY_PATH
              valueFrom:
                secretKeyRef:
```



```
        name: secretenv
        key: dirpath
volumeClaimTemplates:
- metadata:
    name: serverdir
  spec:
    accessModes: [ "ReadWriteOnce" ]
    storageClassName: "minikube-class"
    resources:
      requests:
        storage: 1Gi
---
apiVersion: v1
kind: Service
metadata:
  name: serve-file-service
spec:
  selector:
    app: fileserver
  type: NodePort
  ports:
  - protocol: TCP
    port: 8080
    targetPort: 3000
```

D. Struktura aplikace

```
/vk8s-master
main.go
pkg
  cluster
    cluster.go
  informer
    interfaces.go
  manager
    interface.go
    manager.go
  vk8s
    customer.go
    vk8s.go
    minikube.go
  worker
    configmap.go
    deployment.go
    interface.go
    node.go
    pod.go
    pvc.go
    service.go
    statefulset.go
    worker.go
README.md
```

\AM@currentdocname .pdf

.pdf