ZADÁNÍ BAKALÁŘSKÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Fedorova** Jméno: **Daria** Osobní číslo: **492216**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra počítačů**

Studijní program: **Softwarové inženýrství a technologie**

II. ÚDAJE K BAKALÁŘSKÉ PRÁCI

Název bakalářské práce:

Vysoká dostupnost v prostředí kontejnerizovaných aplikací

Název bakalářské práce anglicky:

High availability in containerized applications

Pokyny pro vypracování:

V rámci spolupráce s externím subjektem proveďte analýzu existující aplikace, provozované v kontejnerovém prostředí. Vyhodnoťte míru její dostupnosti a navrhněte její zvýšení. Zaměřte se na využití více serverů a jejich orchestraci. Postupujte následovně:
1) Definujte pojem "vysoká dostupnost" a pojmy s ním spojené.
2) Definujte metriky, které se v této oblasti používají, a popište způsoby, jakými lze vyšší dostupnost zajistit.
3) Zaměřte se na oblast kontejnerizace a popište ji.
4) Popište "nejčastěji" používané technologie, využívané pro orchestraci kontejnerů.
5) Proveďte analýzu vybrané aplikace pro našeptávání a validaci územních identifikací adres, nemovitostí, parcel a úřadů. Navrhněte možnosti zvýšení její dostupnosti pomocí kontejnerizace.
6) Vyberte a následně realizujte vhodné zvýšení dostupnosti, vycházející z analýzy.
7) Zvolte vhodné metriky, kterými lze zvýšení dostupnosti změřit, a ve spolupráci se zadavatelem pomocí těchto metrik přínosy kontejnerizace vyhodnoťte.

Seznam doporučené literatury:

1. Marko Lukša. Kubernetes in Action, Second Edition. Manning Publications, 2018.
2. Jaroslaw Krochmalski. Docker and Kubernetes for Java Developers. Packt Publishing, 2017.
3. Brendan Burns, Joe Beda, and Kelsey Hightower. Kubernetes : up and running: dive into the future of infrastructure. O'Reilly Media, 2019.

Jméno a pracoviště vedoucí(ho) bakalářské práce:

Ing. Pavel Náplava, Ph.D. Centrum znalostního managementu FEL

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) bakalářské práce:

Datum zadání bakalářské práce: **02.02.2022** Termín odevzdání bakalářské práce: **15.08.2022**

Platnost zadání bakalářské práce: **30.09.2023**

Ing. Pavel Náplava, Ph.D. podpis vedoucí(ho) práce	podpis vedoucí(ho) ústavu/katedry	prof. Mgr. Petr Páta, Ph.D. podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

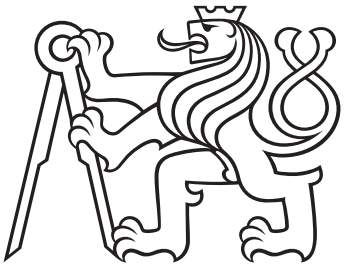
Studentka bere na vědomí, že je povinna vypracovat bakalářskou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací. Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v bakalářské práci.

. Datum převzetí zadání

Podpis studentky

České vysoké učení technické v Praze
Fakulta elektrotechnická

Studijní program: Softwarové inzenýrství a technologie
Katedra počítačů



Vysoká dostupnost v prostředí kontejnerizovaných aplikací

BAKALÁŘSKÁ PRÁCE

Vypracovala: Fedorova Daria
Vedoucí práce: Ing. Pavel Náplava, Ph.D.
Rok: 2022

Prohlášení

Prohlašuji, že jsem svou bakalářskou práci vypracovala samostatně a použila jsem pouze podklady (literaturu, projekty, SW atd.) uvedené v přiloženém seznamu.

V Praze dne ....

Fedorova Daria

Poděkování

Chtěla bych poděkovat vedoucímu mé bakalářské práce p. Ing. Pavlu Náplavovi za jeho konzultace, cenné rady a čas, který mi věnoval po celou dobu řešení dané problematiky. Chtěla bych také poděkovat společnosti Trixi a zejména vedoucímu svého týmu Janu Heroldovi za to, že mi poskytli hardware, nechali mě věnovat část mého pracovního času této práci a neustále přinášeli nové nápady na vylepšení.

Fedorova Daria

Název práce:
Vysoká dostupnost v prostředí kontejnerizovaných aplikací

Autorka: Fedorova Daria

Studijní program: Softwarové inzenýrství a technologie

Druh práce: Bakalářská práce

Vedoucí práce: Ing. Pavel Náplava, Ph.D.

Abstrakt: Tato bakalářská práce je věnována problému vysoké dostupnosti v kontejnerovém prostředí. Analytická část se zaměřuje na analýzu vysoké dostupnosti jako klíčového požadavku systému, jeho metriky a metod implementace. Tato část se zabývá virtualizačními a kontejnerizačními technologiemi, analyzovat koncepci orchestrace a porovnat dvě nejpopulárnější orchestrační platformy – Kubernetes a Docker Swarm. Praktická část bude analyzovat stávající infrastrukturu systému Smartform, identifikovat její slabiny a eliminovat tyto problémy přechodem na Kubernetes. Další částí je nasazení vysoce dostupného clusteru Kubernetes pomocí 3 serverů a samotné nasazení systému Smartform pomocí různých objektů Kubernetes. Na závěr budou pomocí vybraných indikátorů vyhodnoceny výhody orchestrace.

Klíčová slova: Vysoká dostupnost, Docker, Kubernetes, kontejnerizace, orchestrace kontejnerů

Title:
Název anglicky

Author: Fedorova Daria

Abstract: This bachelor thesis is dedicated to the problem of high availability in a container environment. The analytical part focuses on the analysis of high availability as a key requirement of the system, its metrics and methods of implementation. This part deals with virtualization and containerization technologies, analyze the concept of orchestration and compare the two most popular orchestration platforms – Kubernetes and Docker Swarm. The practical part will analyze the existing infrastructure of the Smartform system, identify its weaknesses and eliminate these problems by switching to Kubernetes. The next part is the deployment of the highly accessible Kubernetes cluster using 3 servers and the actual deployment of the Smartform system using various Kubernetes objects. Finally, the advantages of orchestration will be evaluated using selected indicators.

Key words: High availability, Docker, Kubernetes, containerization, container orchestration

Obsah

Seznam použitých zkratek **xi**

Seznam obrázků **xiii**

Úvod **1**

1 Vysoká dostupnost **3**
1.1 Úvod . . . 3
1.2 Co je to vysoká dostupnost? . . . 4
1.2.1 Environmentální vrstva . . . 4
1.2.2 Hardwarová vrstva . . . 5
1.2.3 Softwarová vrstva . . . 5
1.2.4 Síťová vrstva . . . 6
1.2.5 Aplikační vrstva . . . 6
1.3 Metriky dostupnosti . . . 6
1.4 Realizace vysoké dostupnosti . . . 8

2 Technologie používané pro zvýšeni dostupnosti **11**
2.1 Virtualizace . . . 11
2.2 Jak virtualizace funguje? . . . 12
2.3 Kontejnerizace . . . 13
2.4 Rozdíl mezi virtualizaci a kontejnerizaci . . . 13

3 Docker **15**
3.1 Komponenty Dockeru . . . 15
3.1.1 Image nebo obraz . . . 15
3.1.2 DockerFile . . . 15
3.1.3 Docker kontejnery . . . 16
3.1.4 Docker volumes . . . 17
3.1.5 Docker Engine . . . 17
3.1.6 Docker registry . . . 17
3.2 Architektura Dockeru . . . 18

4 Orchestrace kontejnerů **19**
4.1 Platformy pro orchestraci . . . 20
4.2 Docker Swarm . . . 20
4.2.1 Komponenty . . . 21
4.3 Kubernetes . . . 21
4.3.1 Architektura . . . 22
4.3.2 Komponenty . . . 22
4.4 Porovnaní platforem pro orchestraci . . . 25
4.4.1 Instalace . . . 26
4.4.2 Konfigurace clusteru . . . 26

4.4.3 Řízeni stavu prostředí pomoci GUI[1] . . . 26
4.4.4 Škálovatelnost . . . 27
4.4.5 Směrovaní dotazu uvnitř clusteru a ze vnějšku . . . 27
4.4.6 Síť . . . 28
4.4.7 Monitorování . . . 29
4.4.8 Rolling updates and rollbacks . . . 29
4.5 Shrnutí . . . 30

5 As-Is infrastruktura zadavatele 31
5.1 Popis systému . . . 31
5.2 Slabé stránky existující infrastruktury . . . 31

6 Příprava na vytvoření clusteru Kubernetes 35
6.1 Modely clusterů Kubernetes . . . 35
6.1.1 Single node cluster . . . 35
6.1.2 Single master cluster . . . 36
6.1.3 Multi-master cluster . . . 37
6.1.4 Výběr vhodného modelu . . . 38
6.2 Distribuce kubernetes . . . 38
6.3 Docker registry . . . 39
6.4 Databáze PostgreSQL v Kubernetes . . . 40

7 Nasazení clusteru na serveru 43
7.1 Technická specifikace serverů . . . 43
7.2 Konfigurace uzlů . . . 43
7.2.1 Instalace container runtime . . . 44
7.2.2 Instalace Kubeadm, Kubelet a Kubectl . . . 44
7.2.3 Inicializace control-plane komponent . . . 45
7.2.4 Instalace pluginu CNI . . . 46
7.2.5 Připojení zbývajících uzlů . . . 46
7.2.6 Ověření stavu clusteru . . . 47
7.2.7 Připojení k soukromému Docker registru . . . 48

8 Nasazení částí aplikací v clusteru 49
8.1 Kubernetes objekty . . . 49
8.2 Nasazení aplikací pomocí objektu Deployment . . . 50
8.2.1 ApiVersion . . . 52
8.2.2 Kind . . . 53
8.2.3 Metadata . . . 53
8.2.4 Deployment Specifications . . . 53
8.2.5 Pod Template Specifications . . . 54
8.3 Poskytování dat pomocí objektu ConfigMap . . . 56
8.4 Nasazení objektu do clusteru . . . 57
8.5 Konfigurace PostgreSQL serveru . . . 57
8.5.1 Konfigurace úložiště pomocí Persistent Volume . . . 58
8.5.2 Vystavení přístupu pomocí Service . . . 59
8.6 Zajištění externího přístupu k systému pomocí objektu Ingress Controller . . . 61
8.6.1 Instalace Nginx controlleru v clusteru . . . 62
8.6.2 Nastavení pro zpracování TLS provozu . . . 62
8.6.3 Konfigurace směrování provozu pomocí objektu Ingress . . . 63
8.7 Zajištění vysoké dostupnosti systému v Kubernetes. . . . 63

[1] Graphic User Interface

9 Evaluace **65**
9.1 Vyhodnocení navržené infrastruktury pomocí metrik 65
9.2 Plán budoucího rozvoje 67

Závěr **69**

Bibliografie **71**

A Instalace cri-dockerd **75**

B Instalace kubeadm, kubectl a kubelet **77**

Seznam použitých zkratek

IT	Informační technologie
KKCG	Investiční skupina Karla Komárka
UPS	Uninterruptible Power Supply/Source
HA	High Availability/Highly available
DNS	Domain Name System
IDC	International Data Corporation
RTO	Recovery time objective
SPOF	Single point of failure
MTTR	Mean time to repair
RPO	Recovery point objective
RAM	Random Access Memory
CPU	Central Processing Unit
OS	Operační systém
QA	Quality assurance
ERP	Enterprise resource planning
VM	Virtuální stroj
CLI	Command Line Interface
API	Application Programming Interface
REST	Representational state transfer
HTTP	Hypertext Transfer Protocol
JSON	JavaScript Object Notation
YAML	Ain't Markup Language
SNSF	Cloud Native Computing Foundation
IP	Internet Protocol
IPVS	IP Virtual Server
CRI	Container Runtime Interface
GUI	Graphic User Interface
TLS	Transport Layer Security
SSL	Secure Sockets Layer

URL Uniform Resource Locator

TCP Transmission Control Protocol

QA Quality assurance

RMI Java remote method invocation

CI Continuous Integration

CD Continuous Deployment

Seznam obrázků

1.1 Vrstvy dostupnosti 4
1.2 Třídy dostupnosti [8] 8
1.3 Aplikace metod zvyšování dostupnosti na systém 9

2.1 Architektura tradičního a virtualizovaného systému [13] 12
2.2 Architektonické rozdíly mezi virtuálním strojem a kontejnerem [17] . 13

3.1 Vrstvy image 16
3.2 Vrstvy kontejneru 16
3.3 Architektura Dockeru [23] 18

4.1 Docker Swarm architektura [25] 20
4.2 Kubernetes architektura [28] 22
4.3 Ukázka YAML definice podu 23

5.1 Smartform architektura 32

6.1 Arhcitektura minikube clasteru [33] 36
6.2 Arhitektura single master clasteru [34] 36
6.3 Arhcitektura multi-master clasteru [35] 37
6.4 Patroni cluster spravující dvě vysoce dostupné instance PostgreSQL [39] 41

7.1 Kubernetes cluster 44

8.1 Příklad interakce objektů Kubernetes 50
8.2 Aktualizace podů 54
8.3 Persistent Volume a Persistent Volume Claim v clusteru [47] 60
8.4 Přístup k podům prostřednictvím Ingress [43] 62

9.1 Čas na opravu poruchy 66

Úvod

Moderní doba je typická tím, že se bezpočet záležitostí řeší pomocí informačních technologií. Různé aplikace a informační systémy jsou nedílnou součástí státního, soukromého a podnikatelského sektoru. Udržovat zákazníky spokojené a zapojené do systému je možné pouze v případě, že je systém funkční. Existuje přímá a smysluplná korelace mezi dostupností systému a spokojeností zákazníků. Problémy dostupnosti stojí společnosti peníze, důvěru zákazníků a loajalitu. Společnost nemůže přežít dlouho, má-li neustále problémy s dostupností.

Cílem naší bakalářské práce je prozkoumat problematiku dostupnosti systémů, zaměřit se na koncepci kontejnerizace a orchestrace a zlepšit dostupnost systému Smartform přechodem na orchestrační platformu.

V kapitole 1. bude popsána analýza vysoké dostupnosti, na jejímž základě provedeme klasifikaci podle místa možného výpadku systému. Následně popíšeme, jaké metriky se obvykle pro vysokou dostupnost používají a jakými způsoby je možné jí dosáhnout.

Kapitola 2. stručně popisuje technologie pro zvýšení dostupnosti, jako jsou virtualizace a kontejnerizace, to, jak fungují, a také čím se liší.

Kapitola 3. představuje nejpopulárnější kontejnerizační technologii „Docker“ a rozebírá nejdůležitější komponenty a architekturu.

V kapitole 4. bude podrobněji rozebrán koncept orchestrace, zejména orchestrační platformy Docker Swarm a Kubernetes, a bude provedeno široké srovnání těchto dvou nejpopulárnějších orchestračních nástrojů.

Kapitola 5. je základem pro praktickou část bakalářské práce. V této části bude představena analýza systému Smartform pro našeptávání a validaci územních identifikací adres, nemovitostí, parcel a úřadů. Dále budou odhaleny nedostatky současné infrastruktury, což se zlepší přechodem na Kubernetes.

Kapitola 6. je věnována přípravě před nasazením. Ukáže možné modely nasazení, jak poskytovat přístup do soukromých registrů Docker a jak používat databázi PostregSQL v kontejnerezovaném prostředí.

Sedmou kapitolou začíná praktická část. Ta představuje postupný návod, jak nakonfigurovat HA[2] cluster a propojit uzly do tohoto clusteru.

Kapitola 8. se zaměří na nastavení různých částí pomocí objektů Kubernetes. Na základě příkladů uvedených v této části bude dosaženo závěru, že platforma implementuje metody pro zvýšení dostupnosti, a to jak na úrovni podů, tak na úrovni fyzických nebo virtuálních uzlů.

Závěrem, a to v 9. kapitole, shrneme dosažené výsledky a vyhodnotíme přínosy využití Kubernetes pomocí metrik.

[2]High Availability/Highly available

Kapitola 1

Vysoká dostupnost

1.1 Úvod

Informační technologie (IT) se staly životně důležitou a nedílnou součástí každého podniku, a to od nadnárodních korporací, které udržují desítky systémů a databází, až po malé podniky, které vlastní jeden počítač. Hlavní obchodní procesy firmy zcela závisí na provozu systému, proto jakákoli, dokonce i malá přestávka v činnosti IT[1] systémů pro ni představuje ztrátu zákazníků, snížení příjmů a poškození pověsti. V takových případech se při modelování a navrhování systémů klade důraz na spolehlivý provoz a zdůrazňuje se požadavek na vysokou dostupnost (High availability) jako prioritní podmínku.

Analýza dopadů na podnikání neplánovaných výpadků IT systémů, které podporují různé obchodní procesy, poskytuje přehled o důsledcích. Byly zveřejněny různé zprávy dokumentující náklady na prostoje v průmyslu. Tyto náklady se pohybují od desítek tisíc dolarů za hodinu až po miliony dolarů za tutéž dobu [1].

Zatímco jde o ohromující čísla, důvody jsou zcela zřejmé. V současné době moderní IT systémy, které umožňují vytvářet, ukládat, zpracovávat informace podporující různé obchodní procesy, se staly důležitým faktorem podnikání. Jiné aspekty systému, jako jsou vysoký výkon a rychlost, odcházejí do pozadí, pokud servery nejsou vysoce dostupné.

Názorným příkladem je selhání Facebooku, což se stalo 4. října 2021. Americký ekonomický časopis Fortune odhaduje, že na základě čtvrtletních ukazatelů v průměru Facebook Inc vydělává za den 319,6 milionu dolarů, nebo 13,3 milionu dolarů za hodinu [2]. Vzhledem k tomu, že závada trvala 7,5 hodiny, škoda by mohla činit 99,75 milionu dolarů, aniž by se zohlednily reputační škoda a kolaps akcií. Ve stejný den oznámila konkurenční platforma Telegram nárůst počtu uživatelů o 70 milionů.

Nicméně vysoká dostupnost není pro každého. Investice do ní může snadno znamenat zdvojnásobení nákladů nejen ve fázi vývoje, ale i během dalšího provozu a údržby. Pokud však jde o větší e-shop, banku nebo jiný kriticky důležitý systém, kde každá hodina prostojů bude stát desítky tisíc ušlých zisků nebo více, náklady na udržování vysoce dostupných systémů se mnohonásobně vyplatí.

[1] Informační technologie

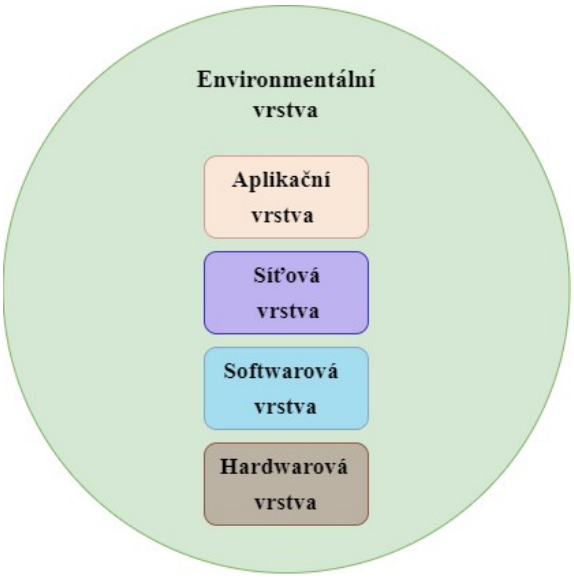
1.2 Co je to vysoká dostupnost?

Vysoká dostupnost znamená, že IT systém, komponenta, nebo aplikace může pracovat nepřetržitě, bez zásahu, po dané časové období. Infrastruktura s vysokou dostupností je nakonfigurována tak, aby poskytovala kvalitní výkon a zvládala různá zatížení i poruchy s minimálními nebo nulovými prostoji [3].

Systém se nazývá vysoce dostupný, pokud mezi jeho vlastnosti patří:

- minimální dopad na systém v případě chyb.
- všechny použité zařízení a hardware jsou spolehlivé, což znamená, že pravděpodobnost technické chyby je teoreticky nulová nebo téměř nulová.
- automatické obnovení částí systému po nekritických haváriích.
- včasné odhalení možných chyb.

Zatímco výše uvedené charakteristiky jsou abstraktní a liší se systém od systému, subjektivní hodnocení dostupnosti končí tam, kde začínají reálná čísla. V tomto případě potřebujeme metriky – kvalitativní nebo kvantitativní ukazatele, na jejichž základě je možné provést spolehlivé posouzení. O metrikách specifických pro vysokou dostupnost bude pojednávat další kapitola.



Obrázek 1.1: Vrstvy dostupnosti

Zkrácení doby prostoje je hlavním cílem zvýšení dostupnosti systému. Na základě toho lze dostupnost analyzovat a řešit na různých vrstvách (obrázek 1.1), podle toho, kde může dojít k výpadku.

1.2.1 Environmentální vrstva

Do této vrstvy patří všechny možné vnější podmínky provozního prostředí, které mohou způsobit selhání systému. Může to být přírodní katastrofa, jako jsou zemětřesení nebo tsunami. Pokud jsou všechny servery obsluhovány v jedné geografické zóně,

důsledky mohou být katastrofální a skončí dlouhodobým výpadkem. Zároveň podmínky prostředí, jako jsou vlhkost nebo teplota, mohou ovlivnit fungování systému. Například nadměrná vlhkost může způsobit zkrat.

Názorným příkladem, jak může společnost zachovat provoz zákaznických služeb za kritických přírodních podmínek, je situace na Moravě 24. června 2021. Tohoto dne tornádo o síle F3–F4[2] udeřilo v Lužicích vpodvečer, bez varování. Specialistovi dohledu datového centra společnosti DataSpring, spadajícího pod investiční skupinu KKCG[3], se ohlásilo přes alarmy výpadků napájení, pak ale události nabraly tak rychlý spád, že zachraňoval sám sebe útěkem do strojovny [4].

Prvním krokem datacentrum zahájilo aktivaci krizového plánu a dohodlo se s klienty na odpojení nekritických systémů. Přímý zásah tornáda poškodil generátory motorů a v první hodině se datové centrum muselo spoléhat pouze na UPS[4] nebo zdroj nepřerušovaného napájení, který zajistil nepřetržitou dodávku elektřiny. Včas se ale podařilo motorgenerátory zprovoznit, problémy ale pokračovaly i s turbokompresory pro chlazení, toto tornádo bylo opravdu silné. Pro minimalizaci rizik bylo k dispozici další geograficky oddělené datové centrum v Praze, vše ale zůstalo primárně na běhu v Lužicích. Další den probíhala oprava turbokompresorů, motorgenerátorů a dalších rozbitých hardwarových zařízení, ale už večer 26. června se spustily všechny, i nekritické systémy. Za necelých 24 hodin se tak firmě podařilo odstranit většinu následků a spustit kompletní provoz centra.

1.2.2 Hardwarová vrstva

Ačkoliv poruchy hardwaru představují jen malé procento neplánovaných výpadků systému, aktualizace poškozené části hardwaru může trvat dlouho. Mezi možné příčiny selhání hardwaru náleží:

- poškozené komponenty
- nekompatibilita komponent
- zanedbávaná aktualizace
- výrobní chyby
- komponenty, které jsou na konci své životnosti

Aby se předešlo takovým chybám, doporučuje se pravidelně udržovat hardware a monitorovat systém. Při zajištění dostupnosti zařízení je často nutné poskytnout i toleranci chyb. Odolnost proti chybám nebo tolerance chyb (Fault-tolerant) je schopnost systému plnit svou funkci správně i za přítomnosti poruch [5]. V případě selhání tohoto typu se používá specializované zařízení pro detekci hardwarové závady a okamžité přepnutí na záložní hardware komponent – ať už je to procesor, paměťová karta, nebo subsystém úložiště.

1.2.3 Softwarová vrstva

Nejzávažnějším typem selhání softwaru je selhání operačního systému, protože ten zastaví celý počítačový systém. Jelikož mnohé problémy se softwarem jsou jen

[2]Dostupné z: `https://cimss.ssec.wisc.edu/oakfield/Fscale.html` (01.08.2022)
[3]Investiční skupina Karla Komárka
[4]Uninterruptible Power Supply/Source

dočasné, restartování často řeší problém. To zahrnuje restart operačního systému, spuštění softwaru obnovujícího stav disku, který by se stal nekonzistentní v důsledku výpadku, obnovu komunikace s jinými systémy a restart všech aplikačních programů. Všechny tyto kroky prodlužují střední dobu opravy, čímž se snižuje i úroveň dostupnosti. Vzhledem k nevyhnutelnosti lidských chyb budou některé programy nasazeny s chybami. Chyby softwaru se mohou objevit v jeho specifikaci, designu nebo implementaci, což může v budoucnu způsobit selhání.

1.2.4 Síťová vrstva

Dostupnost sítě je jedním z nejdůležitějších parametrů každého IT systému. Na její zajištění firmy vynakládají nemalé prostředky. Dostupnost všech ostatních vrstev ztrácí svůj význam, pokud není vybudována vysoce dostupná síťová vrstva. IT systém může fungovat bez přerušení s téměř nulovou dobou prostoje, ale bude to zbytečné, pokud kvůli chybám v síti bude izolován bez možnosti komunikace s jinými systémy.

K faktorům, jež mohou mít vliv na dostupnost sítě, patří fyzické poruchy, nesprávná konfigurace zařízení (DNS[5] atd.), poruchy komponent a nebezpečné útoky. Zatímco některým příčinám výpadku sítě nelze zabránit, většině se dá vyhnout, nebo je minimalizovat. Vysoká dostupnost sítě vyplývá z pozornosti věnované třem klíčovým bodům [6]. Jsou to:

1. výběr síťových prvků, hardwaru a softwaru s vysokou dostupností
2. vytvoření a údržba prostředí s vysokou dostupností (včetně bezpečnosti)
3. použití postupů návrhu a provozu sítě, které kladou důraz na vysokou dostupnost

1.2.5 Aplikační vrstva

Dostupnost aplikací je měřítko sloužící k vyhodnocení toho, zda aplikace správně funguje a je použitelná pro splnění požadavků zákazníků nebo podniku. Systémy s vysokou dostupností jsou navrženy konkrétně tak, aby omezily prostoje a zamezily jedinému místu výskytu chyb. Jedním z běžných vzorů vysoce dostupné architektury jsou mikroslužby, které na rozdíl od monolitu snižují riziko selhání celého systému. Architektura mikroslužeb umožňuje izolovat selhání pomocí dobře definovaných hranic služeb. Dostupnost aplikace však nespočívá pouze v dobře modelovaném systému a schopnosti aplikace zvládnout chyby, ale také v řízení a celém aplikačním prostředí.

1.3 Metriky dostupnosti

Dostupnost lze obecně vyjádřit v procentech a počítat jako poměr pracovní doby bez přerušení k celkovému času. Když mluvíme o vysoké dostupnosti, tento podíl má obvykle podobu řady devítek. Například „čtyři devítky“ odkazují na 99,99% dostupnost, což znamená 52 minut výpadku za rok. Následující tabulka 1.1 ukazuje převod z procentuální dostupnosti na odpovídající dobu v jednotkách času:

[5]Domain Name System

Dostupnost (%)	Prostoje za den	Prostoje za měsíc	Prostoje za rok
90%	2.40 hodiny	73.05 hodin	36.53 dny
95%	1.20 hodiny	36.53 hodin	18.26 dny
99%	14.40 minut	7.31 hodin	3.65 dny
99.9%	1.44 minuty	43.83 minut	8.77 hodin
99.99%	8.64 sekund	4.38 minuty	52.60 minut
99.999%	864.00 milisekund	26.30 sekund	5.26 minut
99.9999%	86.40 milisekund	2.63 sekundy	31.56 sekund
99.99999%	8.64 milisekund	262.98 milisekund	3.16 sekundy

Tabulka 1.1: Vyjádření dostupnosti v jednotkách času

Systémovou dostupnost určuje spolehlivost komponent, které obsahují aplikaci (jak často selhávají), a druhým je, jak dlouho trvá obnovení aplikace, jakmile dojde k selhání. Metriky, jež se běžně používají k pochopení a měření vysoké dostupnosti, jsou následující:

1. **Single point of failure (SPOF[6])**: část nebo konkrétní součást systému, která zastaví celý systém; když selže, celý systém přestane fungovat,

2. **Recovery time objective (RTO[7])** je maximální přípustná doba trvání jakéhokoli výpadku, určuje toleranci obchodního procesu, pokud systém není k dispozici. Pro systém s vysokým provozem uživatelů a kritické podniky by RTO měla být nulová nebo téměř nulová,

3. **Recovery point objective (RPO[8])** se vypočítá jako celková ztráta dat, pokud systém není k dispozici. Měří se z hlediska času. Vyjadřuje, do jakého stavu v minulosti lze obnovit data,

4. **Mean time to repair (MTTR[9])** je průměrná doba oprav. Zahrnuje jak dobu opravy, tak i dobu testování. Tato metrika je nejužitečnější při sledování toho, jak rychle jsou pracovníci údržby schopni problém opravit.

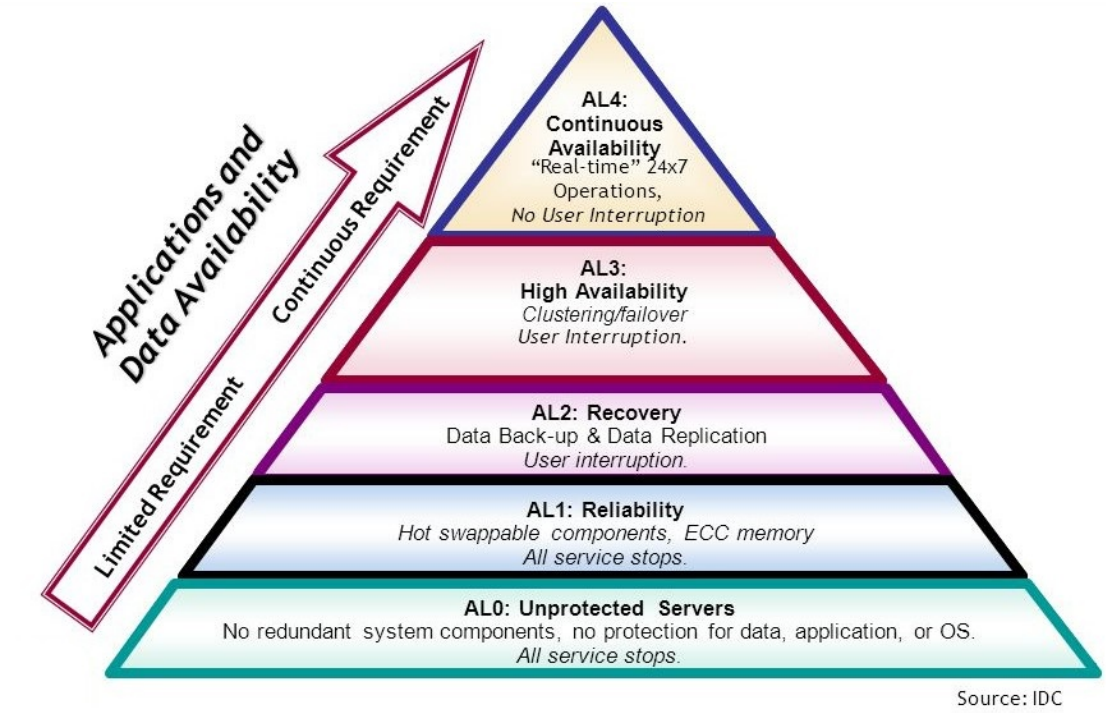
Na základě těchto metrik americká analytická společnost IDC[10], specializující se na výzkum trhu informačních technologií, vyvinula koncept tříd dostupnosti pro různé aplikace, jenž je uveden na obrázku 1.2 [7].

Jaké jsou třídy dostupnosti?

Třida spolehlivosti (Reliable) - vstupní úroveň mnoha aplikací, nevyžaduje ochranu, nebo jednoduše zvyšuje spolehlivost systému konfigurací redundantních komponent pro klíčové komponenty systému, což obecně znamená přidání dodatečných komponent pro duplikování. Příkladem této varianty je zrcadlení disků (disk mirroring). Vyžaduje obnovu, ale proces řízení nezávisí na těchto aplikacích.

Třida obnovitelnosti (Recoverable) - úroveň dostupnosti, která používá zálohování některých součástí infrastruktury, jako jsou webové služby, servery DNS

[6]Single point of failure
[7]Recovery time objective
[8]Recovery point objective
[9]Mean time to repair
[10]International Data Corporation



Obrázek 1.2: Třídy dostupnosti [8]

s automatickým obnovením po havárii. V řadě případů však selhání těchto služeb není povoleno.

Třída vysoké dostupnosti (Highly Available) - úroveň dostupnosti pro ERP[11] systémy, databáze, poštovní a další služby, které poskytují výrobní procesy. Stanou-li se služby nedostupnými, může dojít ke ztrátě dat, což může výrazně ovlivnit cenu výpadku.

Třída neustále dostupnosti (Continuously Available) je zapotřebí pro malý počet úkolů, jako jsou kritické systémy, např. aplikace pro dopravu, trading systémy a banky.

Porovnáním RTO z obrázku pyramidy a prostojů z tabulky číslo 1.1 získáme kategorizaci tříd pro mezinárodní měřítko. Systémy třídy spolehlivosti mají 95% až 99% dostupnosti. Třída obnovitelnosti odpovídá 99,5% dostupnosti. Systémy třídy vysoké dostupnosti vyžadují 99,95% dostupnosti. Třída neustálé dostupnosti má v ideálním případě nulový prostoj, což je nejčastěji vyjádřeno v 99,9999999% dostupnosti. V praktické části se ukáže aplikovanost a použitelnost výše uvedených metrik na existující systém.

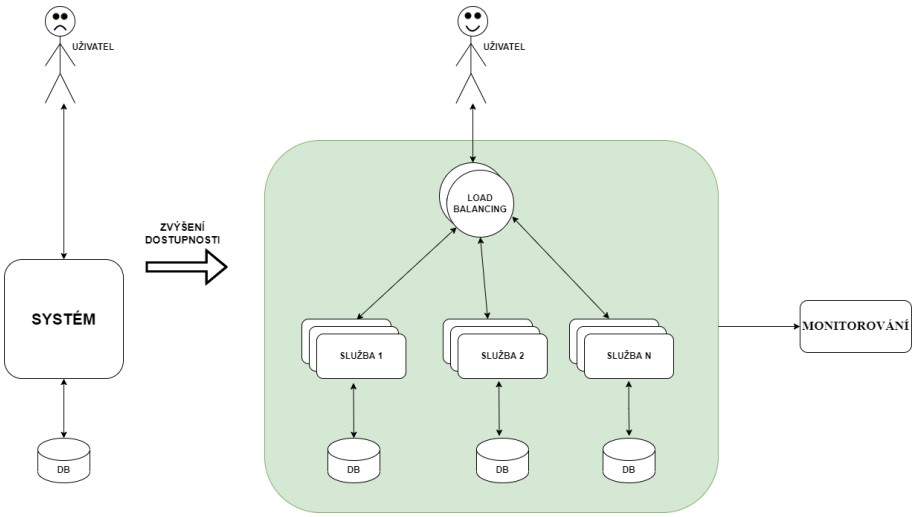
1.4 Realizace vysoké dostupnosti

Hlavní metody zvyšování úrovně dostupnosti systému na různých úrovních jsou následující::

1. Škálování

   Škálovatelnost je schopnost systému, sítě nebo procesu zvládnout rostoucí objem práce nebo jeho potenciál rozšířit tak, aby mohl tento nárůst zpracovat [9]. Existují dva typy škálování:

[11] Enterprise resource planning



Obrázek 1.3: Aplikace metod zvyšování dostupnosti na systém

- Horizontální škálování, jehož je dosaženo přidáním více strojů (kvantitativní změna),
- Vertikální škálování, toho je dosaženo zvýšením kapacity strojů (CPU[12], RAM[13]) (kvalitativní změna).

Škálování je možné provést zvýšením počtu jader CPU a paměti na serveru (vertikální škálování), nebo můžeme zvýšit počet serverů (horizontální škálování). Takovým způsobem škálování lze pomoci zabránit nedostatku zdrojů ovlivňujících výkon a dostupnost.

2. Monitorování

   Pomáhá sledovat stav celého systému, rychle identifikovat a vyhodnocovat problémy na základě řady ukazatelů. Pomocí nástroje monitorování je možné zjistit aktuální situaci na serveru a například v případě, že indikátory ukazují na problém s nedostatkem paměti, tomu včas zabránit.

3. Vyvažování zátěže (Load Balancing)

   Zvyšuje a optimalizuje výkon systému a distribuuje provoz mezi více stroji. Během přesměrování provozu load balancery také sledují zdravotní stav každého serveru či instance. Díky tomu v případě zjištění selhání serveru přesměrují provoz na jiný server a zajistí, že systém zůstane přístupný.

4. Klasterování (Clustering)

   Cluster je skupina propojených systémů, které pracují společně, na straně uživatele se cluster zdá být jednotný systém. Na každém fyzickém počítači může být v clusteru jeden nebo více logických uzlů. Clustering umožňuje serverům vzájemně si pomáhat: pokud jeden server spadne, jiný uzel clusteru může přebrat pracovní zatížení, aniž by došlo k přerušení.

Obrázek 1.3 obecně ukazuje, jak tyto metody mohou změnit systém: horizontální škálování zvýšilo počet systémových služeb, byl přidán load balancer mezi

[12]Central Processing Unit
[13]Random Access Memory

těmito instancemi a monitorovací nástroj je připojen k celému systému. Výsledkem těchto akcí může být spokojenost klienta.

V dalších kapitolách přiblížíme, jakým způsobem využívá technologie orchestrace kontejnerů tyto metody pro zvýšení dostupnosti.

Kapitola 2

Technologie používané pro zvýšeni dostupnosti

Organizace dříve provozovaly aplikace na fyzických serverech. Kvůli neschopnosti definovat hranice zdrojů pro aplikace na fyzickém serveru došlo k problémům s distribucí zdrojů. Spuštěním dvou programů na jednom serveru s různými požadavky se strávila spousta času. Pokud je na fyzickém serveru spuštěno více aplikací, mohou existovat případy, kdy jedna z nich zabírá většinu zdrojů, a v důsledku toho pak ostatní aplikace fungují hůře. Řešením bylo spustit každou aplikaci na jiném fyzickém serveru, ovšem situace se nezlepšila, protože zdroje nebyly plně využity, což způsobilo, že organizace byly nuceny udržovat mnoho fyzických serverů. Právě za účelem řešení těchto problémů se zrodila geniální myšlenka virtualizace serverů.

Tato kapitola slouží jako úvod do problematiky. V následujících sekcích prozkoumáme problém virtualizace a kontejnerizace i jejich hlavní rozdíly.

2.1 Virtualizace

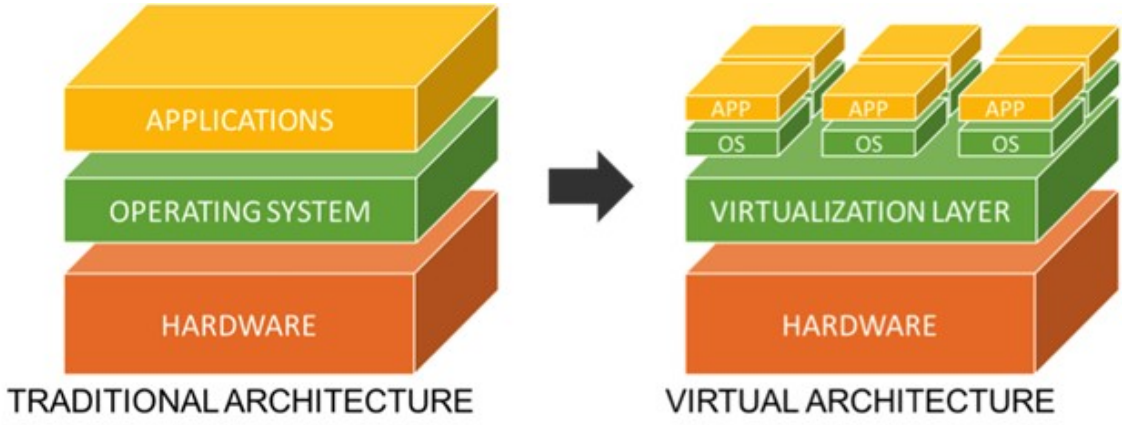
Virtualizace umožňuje, aby na jednom fyzickém serveru (na jednom hardwaru) běželo více oddělených serverů s vlastním operačním systémem. Fyzický server každému takovému virtuálnímu serveru emuluje virtuální hardware (procesor, paměť, disk, síťová karta, mechaniky, periferní zařízení a další) [10].

Koncept virtualizace má své počátky v období přelomu 60. a 70. let, kdy společnost IBM[1] investovala mnoho času i úsilí do vývoje robustních řešení pro sdílené využívání počítačo vých zdrojů mezi velkou skupinou uživatelů [11]. V té době šlo o to, aby se dosáhlo možnosti provedení několika programů – Multics byl jedním z prvních systémů umožňujících současnou práci více uživatelů na jednom počítači díky sdílení výpočetního času základního procesoru. Multics našel své uplatnění, třebaže měl své nevýhody: byl pomalý, nespolehlivý a nebezpečný. Vědci jej chtěli vylepšit, a dokonce věděli, jak se toho dá dosáhnout, avšak možnosti vybavení byly omezené [12].

Masová distribuce začala až počátkem roku 2000. Virtualizace však měla řadu nevýhod, jež vedly ke vzniku vylepšené metody virtualizace – kontejnerizace. Přestože kontejnery a kontejnerové technologie byly známy již dlouho, získaly skutečnou popularitu až v roce 2013, kdy byla zveřejněna platforma Docker.

[1] Dostupné z: `https://www.ibm.com/th-en` (07.08.2022)

2.2 Jak virtualizace funguje?



Obrázek 2.1: Architektura tradičního a virtualizovaného systému [13]

Software zvaný hypervizor odděluje fyzické zdroje od virtuálních prostředí – věcí, které tyto zdroje potřebují. Hypervizory mohou být umístěny nad operačním systémem nebo přímo instalovány na hardware. Hypervizory vezmou fyzické zdroje a rozdělí je tak, aby je virtuální prostředí mohlo využívat. Obrázek 2.1 znázorňuje rozdíl v architektuře tradičního a virtualizovaného systému.

Zdroje jsou rozděleny podle potřeby z fyzického prostředí do mnoha virtuálních prostředí. Operačnímu systému, na kterém virtuální stroj běží, tj. ten, který běží na reálném počítači, se říká hostitel (host). Virtuálnímu operačnímu systému se říká hostující (guest) [14].

Hypervizor poskytuje vzájemnou izolaci operačních systémů, ochranu a zabezpečení, oddělení zdrojů mezi běžícími operačními systémy. V závislosti na typu použité virtualizace může hypervizor pracovat buď přímo s hardwarem bez hostitelského systému, nebo prostřednictvím hlavního operačního systému nainstalovaného na hostitelském počítači. V prvním případě se používá hardwarová virtualizace, ve druhém softwarová. Druhý typ je běžně používán na domácích počítačích. Na rozdíl od instalace dvou operačních systémů na jeden počítač je virtualizace mnohem bezpečnější metodou.

Virtualizace nejen poskytuje možnost lépe využít hardware jednotlivých serverů, ale také umožňuje zajistit redundanci pomocí klastrování. Díky tomu je možné rychle, či dokonce automaticky přesouvat virtuální servery mezi jednotlivými fyzickými servery, a vytvořit tak platformu v podstatě odolnou proti fyzickým výpadkům. Z těchto důvodů se cluster považuje za nedílnou součást při stavbě virtuální serverové infrastruktury.

Virtualizační software platformy, konkrétně emulátory a hypervizory, je software, který emuluje celý fyzický počítačový stroj a poskytuje více virtuálních strojů na jedné fyzické platformě. Nejpopulárnější produkty, jež poskytují řešení pro virtualizaci serverové infrastruktury, jsou [15]:

- Citrix Hypervisor
- Red Hat Virtualization
- VMware vSphere

- Proxmox VE
- Microsoft Hyper V

2.3 Kontejnerizace

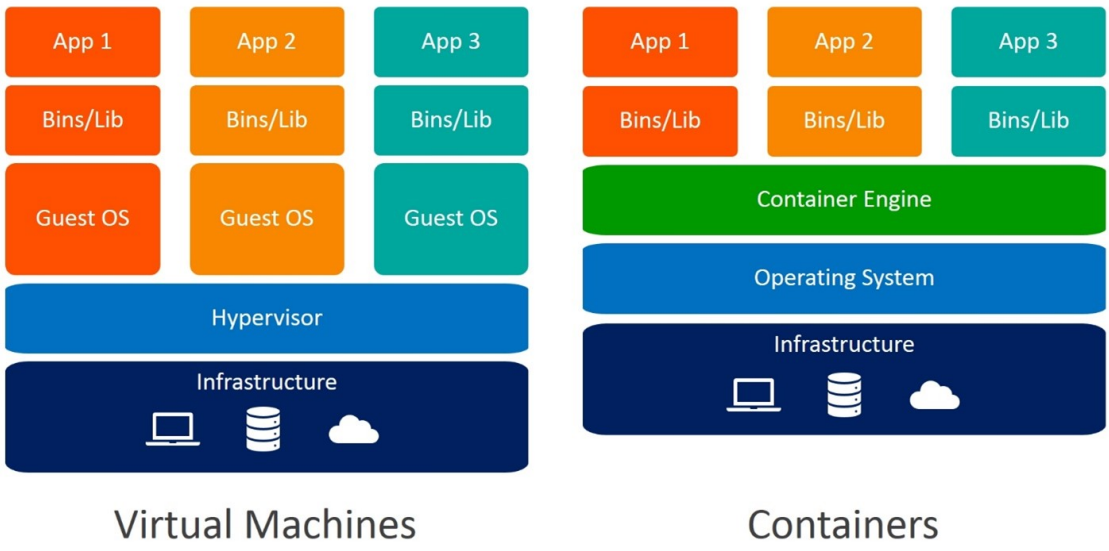
Kontejnerizace je definována jako forma virtualizace operačního systému, jejímž prostřednictvím jsou aplikace spouštěny v izolovaných uživatelských prostorách zvaných kontejnery, přičemž všechny aplikace používají stejný sdílený operační systém (OS[2]).[16]

Kontejner je soubor umístěný na virtuálním disku, který zapouzdřuje aplikaci se všemi nezbytnými závislostmi: kódem aplikace, systémovými nástroji, knihovnami a nastaveními. Je spuštěn pomocí kontejnerového enginu. Kontejnerový engine je část softwaru, která přijímá požadavky uživatelů, vytahuje images z registru a z pohledu koncového uživatele kontejner spustí. Existuje mnoho enginů, včetně Dockeru, RKT, CRI-O a LXD.

Aplikace běžící v kontejneru je izolovaná a má oddělený prostor, což znamená, ze běžící systém ani jiné kontejnery nemohou ovlivnit běh aplikací. Při spuštění kontejneru je nižší režie a není třeba pro každou aplikaci nastavovat samostatné hostující OS, protože všechny mají stejné jádro OS.

Stejně jako virtualizace využívá klastrování k vytvoření vysoce dostupného systému, kontejnerizace pomocí orchestrace může dosáhnout lepších výsledků, protože platformy pro orchestraci nejen vytvářejí cluster uzlů, ale také umožňují škálovat kontejnery, monitorovat jejich stav a mít z krabice vnitřní load balancer. Více se orchestraci budeme věnovat v další kapitole.

2.4 Rozdíl mezi virtualizaci a kontejnerizaci



Obrázek 2.2: Architektonické rozdíly mezi virtuálním strojem a kontejnerem [17]

[2]Operační systém

Kontejnery bývají často srovnávány s virtuálními stroji, protože obě technologie poskytují značnou efektivitu a umožňují běh několika typů softwaru v jednom prostředí. Kontejnerizace se může jevit jako lehčí verze virtualizace, ale v praxi kontejnery ukazují řadu výhod oproti virtuálním strojům. Rozdíly v architektuře mezi virtuálním strojem a kontejnerem jsou znázorněny na obrázku 2.2.

Kontejnery bývají často označovány jako „lehké“, jelikož sdílí hostitelský operační systém, nemusí spouštět zvláštní operační systém ani načítat knihovny. Díky tomu mohou být kontejnery mnohem efektivnější a jednodušší. Kontejnerizované aplikace je možné spustit během několika sekund a počítače mohou obsahovat mnohem více instancí aplikací než ve scénářích s virtuálními počítači [18]. Tak tomu není v případě virtualizace prostřednictvím virtuálních strojů (VM[3]). VM běží na vrcholu hypervizoru (viz obrázek 2.2). Prostřednictvím hypervizoru má každý VM přiřazen virtualizovaný zásobník hardwaru, včetně procesorů, úložišť a síťových adaptérů. Vzhledem k tomu, že každý VM obsahuje OS a virtuální kopii veškerého hardwaru, který OS vyžaduje, vyžadují VM významné zdroje RAM a CPU v porovnání s kontejnery.

Nedá se jednoznačně říci, že kontejnerizace je ve všem rozhodně lepší. Při výběru mezi kontejnerizací a virtualizací je nutné zvažovat určitý systém s jeho specifickými požadavky. V současné době systém, na který bude zaměřena celá praktická část bakalářské práce, používá kontejnerizační platformu Docker v infrastruktuře, proto tím pojednání o virtualizaci končíme a další část práce se zaměří pouze na kontejnerizaci a specifickou technologii pro zavádění a správu aplikací v kontejnerech – Docker.

[3]Virtuální stroj

Kapitola 3

Docker

Docker je opensourcový projekt pro automatizaci nasazení aplikací jako přenositelných, samoobslužných kontejnerů, které mohou běžet v cloudu nebo lokálně [19].

Hlavními výhodami Dockeru jsou:

- automatické vytváření kontejnerů: Docker může automaticky sestavit image kontejneru na základě zdrojového kódu,
- verzování kontejneru: Docker může sledovat verze image kontejneru, vrátit se k předchozím verzím a sledovat, kdo a jak verzi sestavil,
- opětovné použití kontejnerů: stávající kontejnery lze použít jako základní image – v podstatě jako šablony pro stavbu nových kontejnerů,
- sdílené knihovny kontejnerů: vývojáři mají přístup k registru s otevřeným zdrojovým kódem, který obsahuje tisíce kontejnerů vložených uživatelem.

3.1 Komponenty Dockeru

Než přejdeme k architektuře Dockeru, je nutné se seznámit se základními komponentami a pojmy.

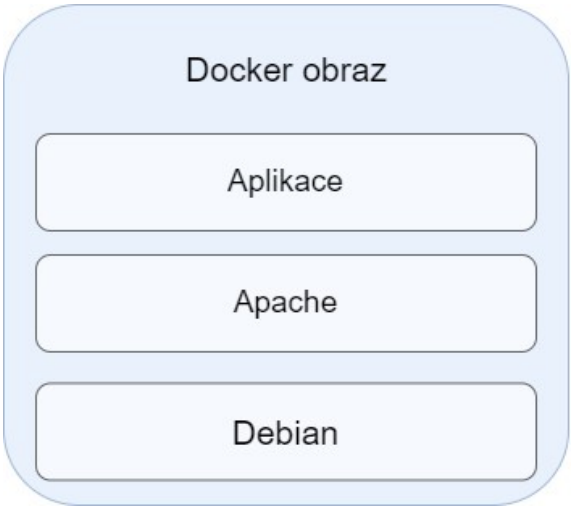
3.1.1 Image nebo obraz

Image je šablona pouze pro čtení s instrukcemi pro vytvoření kontejneru Docker. Často je obraz založen na jiném obrazu, ale s dodatečnou customizací. Image Dockeru se skládá z vrstev. Každý příkaz v souboru obrazu přidá novou vrstvu, která překrývá předchozí [20].

Konečná image je kombinací všech vrstev v jedné. Lze například vytvořit image, která je založena na Debianu, ale nainstaluje webový server Apache, naši aplikaci a také konfiguraci, jež jsou potřebné pro spuštění aplikace. Takové nastavení je znázorněno na obrázku 3.1.

3.1.2 DockerFile

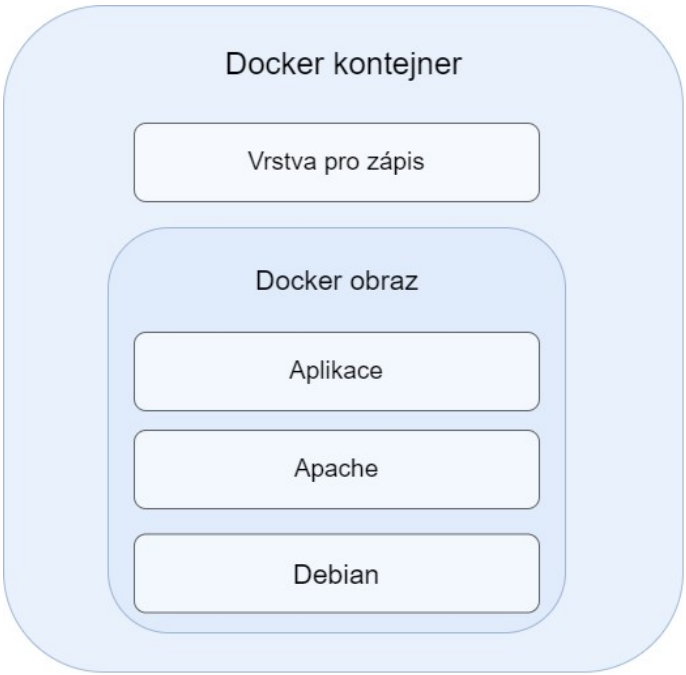
Každý kontejner Dockeru začíná jednoduchým textovým souborem obsahujícím pokyny, jak sestavit image kontejneru Docker. DockerFile automatizuje proces vy-



Obrázek 3.1: Vrstvy image

tváření obrázků v Dockeru. Je to v podstatě seznam příkazových řádkových instrukcí (CLI[1]), které Docker Engine spustí, aby sestavil image [20].

3.1.3 Docker kontejnery



Obrázek 3.2: Vrstvy kontejneru

Image je souhrn vrstev, jež se používají jen pro čtení. Kontejner je stejná image, jež má další vrstvu pro zápis, viz obrázek 3.2. Kontejner používá přidanou vrstvu, když potřebuje uložit informace v průběhu své práce: logy, dočasné soubory a tak dále. Pokud je kontejner zničen, ztratí se i tato vrstva a všechny informace na ní. Když je nový kontejner vytvořen ze stejného obrázku, bude mít novou a prázdnou vrstvu pro zápis. Kontejner je možné vytvořit, spustit, zastavit, přesunout nebo

[1]Command Line Interface

odstranit pomocí rozhraní Docker API[2] nebo CLI. Kontejner je definován svým obrazem stejně jako možnostmi konfigurace, které mu poskytneme při jeho vytvoření nebo spuštění. Po vyjmutí kontejneru zmizí všechny změny jeho stavu, které nejsou uloženy v trvalém úložišti [20].

3.1.4 Docker volumes

Důležitou vlastností Docker kontejnerů je proměnlivost. Kontejner může být kdykoliv restartován. V takovém případě se všechna data, která se v něm nahromadí, ztratí. Jak ale Docker může spouštět aplikace, které by měly ukládat informace o stavu? Existují tedy způsoby, aby důležitá modifikovatelná data nebyla závislá na kontejnerech. Docker volumes je jeden z takových způsobů.[21] Volumes jsou vytvářeny a spravovány Dockerem. V kontejneru jsou volumes vidět jako běžný katalog, který definujeme v Dockerfile. Důvody, proč se používají volumes v Dockeru, jsou následující:

- sdílení dat mezi několika běžícími kontejnery,
- vzdálené úložiště dat,
- zálohování nebo přenesení dat do jiného hostitele Dockeru.

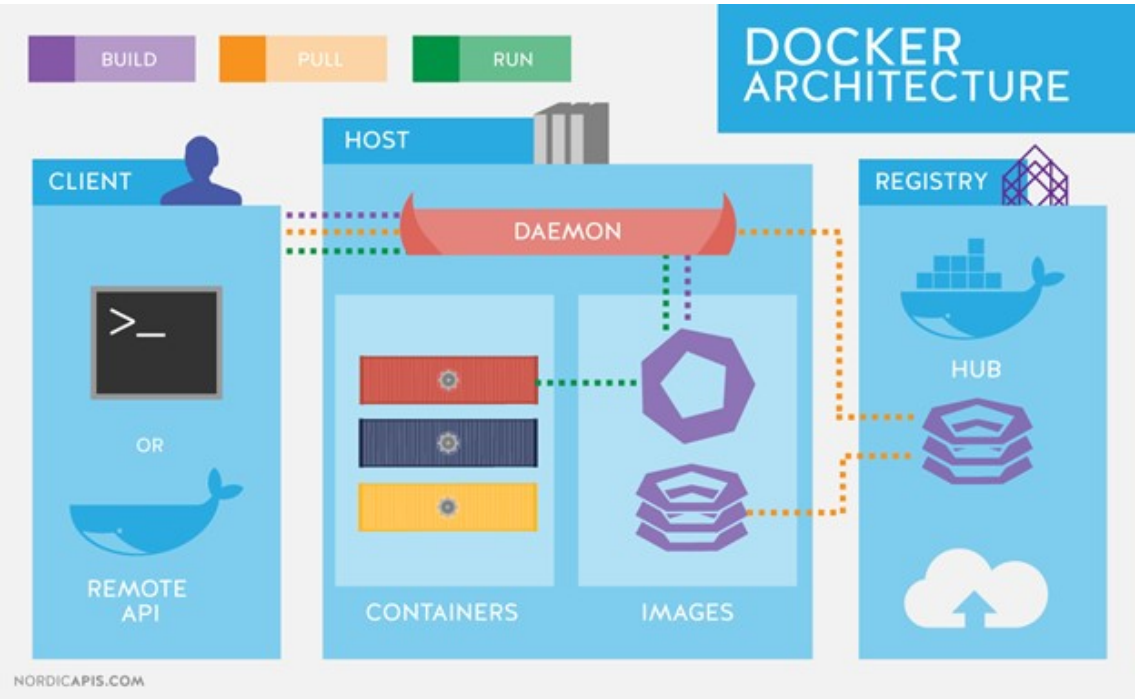
3.1.5 Docker Engine

Docker Engine umožňuje vyvíjet, kontrolovat a spouštět aplikace s použitím těchto komponent:

- Docker Daemon: trvalý proces na pozadí, který spravuje images, kontejnery, sítě a data volumes. Docker Daemon neustále poslouchá požadavky Docker API a zpracovává je [22],
- Docker Engine REST[3] API: API používané aplikacemi pro interakci s Docker Daemonem; může k němu přistupovat HTTP[4] klient [22],
- Docker CLI: Klientské rozhraní příkazového řádku pro interakci s Docker Daemonem. Výrazně zjednodušuje způsob správy instancí kontejnerů a je jedním z klíčových důvodů, proč vývojáři rádi používají Docker [22].

3.1.6 Docker registry

Registry poskytují možnost uchovávat a sdílet images. Docker Hub je veřejný registr, který může použít každý, a Docker je nakonfigurován tak, aby ve výchozím nastavení hledal image v Docker Hubu. Lze si dokonce spustit vlastní soukromý registr. Při použití příkazů `docker pull` nebo `docker run` jsou požadované obrazy staženy z nastaveného registru. Při použití příkazu `docker push` se image přesune do nastaveného registru.



Obrázek 3.3: Architektura Dockeru [23]

3.2 Architektura Dockeru

Uprostřed obrázku 3.3, Docker host představuje fyzický stroj nebo VM, ve kterém jsou nasazeny Docker Ddaemon a kontejnery. Instalace Dockeru závisí na operačním systému stroje. Postupy pro instalaci na různé OS jsou k nalezení na adrese oficiální dokumentace `https://docs.docker.com/engine/install/`. Docker Host poskytuje kompletní prostředí pro spouštění aplikací. Skládá se z Docker Daemonu, obrázků, kontejnerů, sítí a úložiště.

Docker Klient je na obrázku 3.3 zleva. Komunikuje s Docker Daemonem pomocí RESTful API nebo CLI. Docker Klient má za úkol ovládat hostitele, vytvářet obrazy, publikovat, spouštět a spravovat kontejnery odpovídající instanci těchto obrazů. Klient může být spuštěn na stejném stroji jako Daemon nebo se připojit k Daemonu na vzdáleném stroji.

[2]Application Programming Interface
[3]Representational state transfer
[4]Hypertext Transfer Protocol

Kapitola 4

Orchestrace kontejnerů

Jeden kontejner obvykle zpracovává jednu aplikaci nebo službu. Pro organizace s obrovskými distribučními systémy jeden kontejner nestačí. K rozmístění celého systému jsou potřeba desítky a stovky kontejnerů. Tento architektonický návrh představuje výzvu pro řízení životního cyklu a škálování. Bylo by nemožné ručně spravovat velkou sadu nezávisle rozmístěných kontejnerů.

Kontejnerové orchestry mohou tyto problémy řešit a automatizovat.

Orchestrace je metodika, která umožňuje získat kompletní informace o kontejnerech, vidět jejich umístění kontejnerů ia rozdělení pracovních úkolů na kontejnerech a sledovat takové umístění a distribuci. Orchestrace je rozhodující pro nasazení více kontejnerů. Bez orchestrace bude zapotřebí každý kontejner spravovat ručně [24].

Mezi hlavní funkce orchestrace kontejneru patří:

- přidělování zdrojů mezi kontejnery,
- škálování kontejnerů podle pracovní zátěže,
- směrování požadavku,
- monitorování stavu kontejneru a obnovení služby,
- load balancing

Většina nástrojů pro orchestraci kontejnerů podporuje deklarativní konfigurační model: uživatel vytvoří YAML[1] nebo JSON[2] soubor, který popisuje konfiguraci a stav aplikace. Pomocí těchto souborů orchestr ví, jakým způsobem má získávat image kontejnerů, jak vytvářet kontejnery, a kde jsou umístěny, jak vytvářet a zabezpečovat síť mezi kontejnery a jak poskytovat datové úložiště kontejnerům. Jakmile jsou kontejnery nasazeny, nástroj pro orchestraci řídí životní cyklus kontejnerového prostředí.

Kromě výše uvedených funkcí mají orchestry také řadu dalších výhod, které se liší od zvolené platformy. V následující kapitole budou popsány populární platformy pro orchestraci kontejnerů.

[1] Ain't Markup Language

[2] JavaScript Object Notation

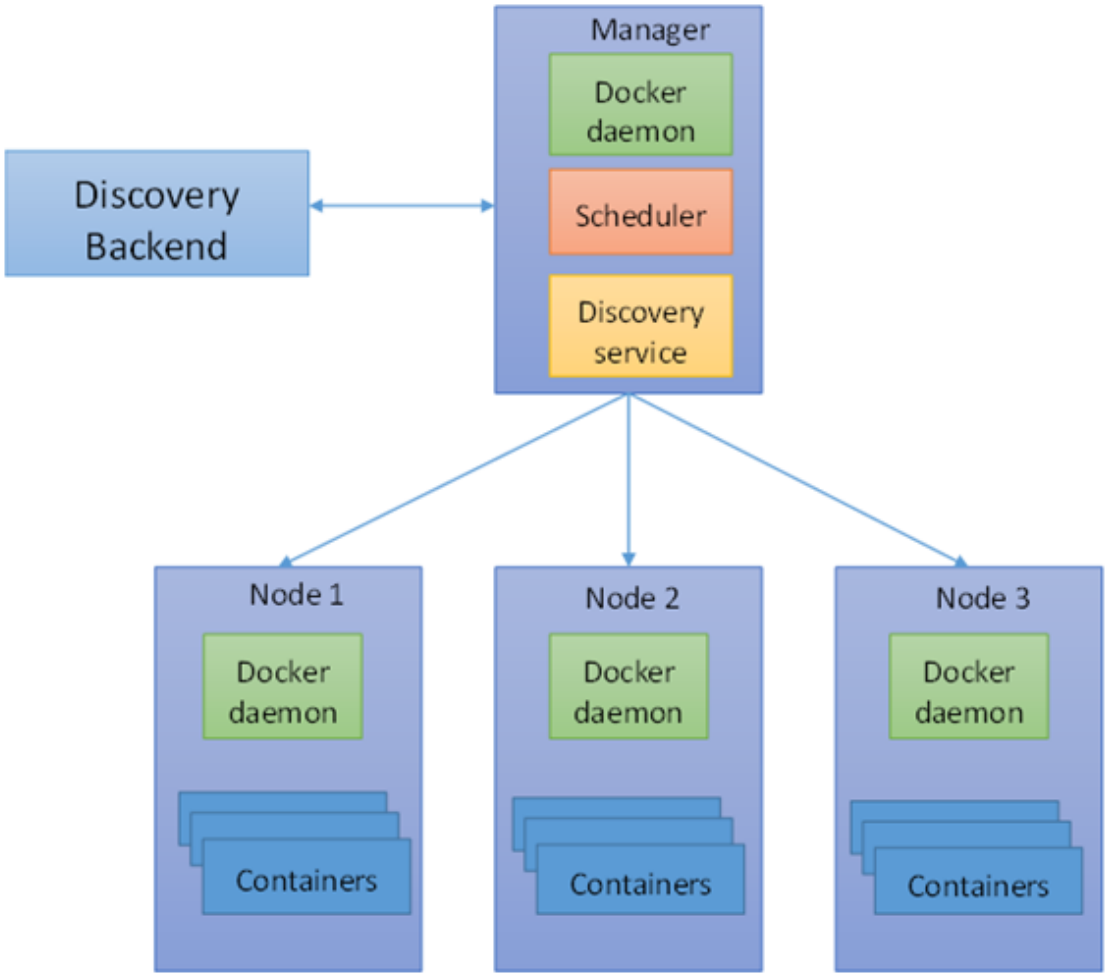
4.1 Platformy pro orchestraci

Existují různé platformy pro orchestraci kontejnerů. Nejznámější systémy jsou Kubernetes, Docker Swarm a Apache Mesos. Existují taktéž Nomad, Fleet, Aurora, Amazon EC2 Container Service, Microsoft Azure Container Service, ale ty jsou méně populární. Pro detailní analýzu jsme si vybrali dvě platformy: Docker Swarm a Kubernetes, se kterými máme základní zkušenosti. Tato kapitola se zaměří na jejich komponenty, architekturu a nabízené služby.

4.2 Docker Swarm

Docker Swarm je nativní nástroj pro orchestraci od Dockeru. Ve verzi 1.12 Docker Engine zavádí Docker Swarm, který umožní vyvářet a spravovat clustery z více virtuálních nebo fyzických strojů.

Architekturu Docker Swarmu znázorňuje obrázek 4.1:



Obrázek 4.1: Docker Swarm architektura [25]

4.2.1 Komponenty

Hlavní komponentou architektury Docker Swarm je uzel. Uzel je Docker instance. Jeden nebo více uzlů může pracovat s jedním fyzickým zařízením nebo cloudovým serverem. Swarm používá stejné rozhraní CLI jako Docker, což velmi zjednodušuje spuštění Swarmu po instalaci Dockeru. To také znamená, že ostatní nástroje používající Docker API (např. Compose) mohou být použity beze změn. Swarm může být ideálním řešením pro aplikace s nižší pracovní zátěží.

Existují dva typy uzlů: manažerský a pracovní. Každý uzel v clusteru může být manažerem nebo pracovním uzlem, občas v malém testovacím prostředí pracovní uzel může být také manažerem.

Manažersky uzel

Manažerský uzel je klíčovou komponentou pro řízení a uložení stavu, odesílání úkolů (tašků) do pracovních uzlů a má na starosti:

- udržování stavu clusteru,
- plánování,
- obsluhování HTTP API koncových bodů.

Když je cluster založen, pomocí algoritmu Raft[3] pro dosažení shody je jeden z nich přiřazen jako „vedoucí uzel“. Tento uzel zajišťuje veškeré řízení a organizování úkolů.

Důvodem, proč režim Docker Swarm používá konsenzuální algoritmus Raft, je ujištění se, že všechny nadřízené uzly v clusteru mají stejný konzistentní stav. To znamená, že v případě selhání může každý manažer provádět úlohy a obnovit služby do stabilního stavu. Pokud například neočekávaně zemře vedoucí uzel, který je zodpovědný za plánování úkolů v clusteru, kterýkoli jiný manažerský uzel se může stát vedoucím [26].

Pracovní uzel Pracovní uzly jsou také instancemi Docker Enginu, jejichž jediným účelem je spouštění kontejnerů. Tyto uzly se nezúčastňují v distribuovaných volbách a nečiní rozhodnutí. Pracovní uzel přijímá a provádí úkoly, které mu manažer delegoval, informuje manažera o stavu uzlu, na němž běží, prostřednictvím REST API.

4.3 Kubernetes

Kubernetes, známý také jako K8, je open-source systém pro automatizaci zavádění, škálování a správu kontejnerových aplikací [27]. Tato platforma byla vyvinuta Googlem z interního systému správy clusterů Borg. Vývojový tým společnosti Google měl za úkol vytvořit open-source software pro orchestraci kontejnerů. Aplikace byla napsána v jazyce Go.

V roce 2014 byly zveřejněny zdrojové kódy, o rok později se objevila první verze programu Kubernetes 1.0. Veškerá práva na produkt byla předána nekomerčnímu fondu Cloud Native Computing Foundation (SNSF[4]), kam patří Google, The Linux Foundation a řada největších technologických společností.

Mezi hlavní funkce Kubernetes patří:

[3]Dostupné z: `https://docs.docker.com/engine/swarm/raft/` (31.07.2022)

[4]Cloud Native Computing Foundation

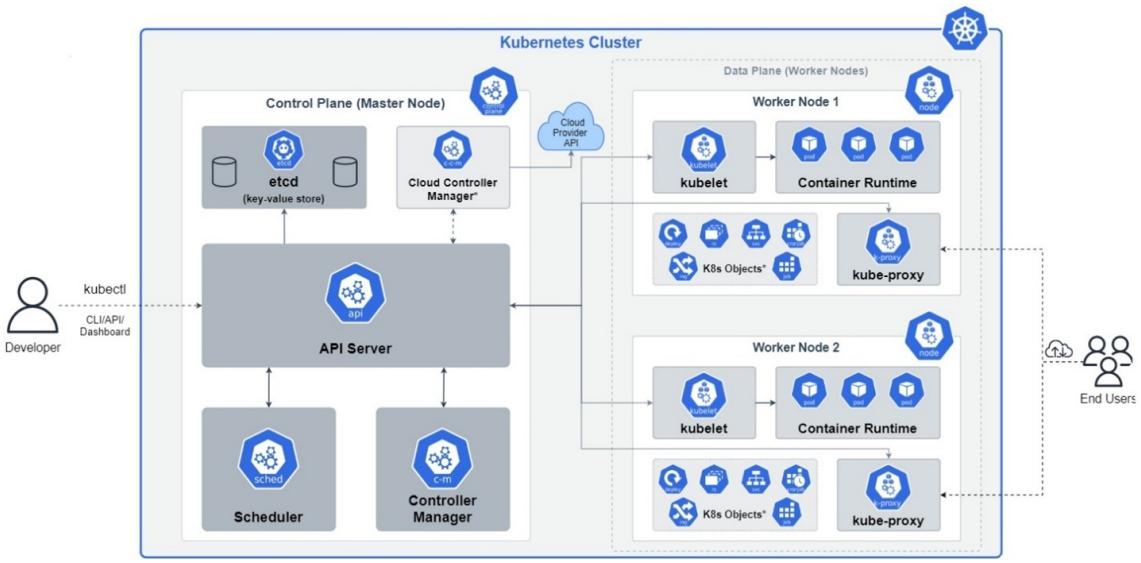
- nasazení kontejnerů a spuštění požadované konfigurace. Patří mezi ně restart zastavených kontejnerů, přidělování zdrojů na nové kontejnery atd.,
- škálování a spouštění více kontejnerů,
- monitorování a automatické restartování kontejneru v případě chyby.

Koncept „požadovaný stav“ versus „skutečný stav“ Požadovaný stav je základní koncept Kubernetes. Znamená to, že pomocí deklarativního nebo imperativního API popíšeme stav objektů, které budou spouštět kontejnery.

Kubernetes je schopen zvládat stálé změny a bude se neustále snažit přizpůsobit skutečný stav požadovanému stavu napsanému v konfiguračním souboru. Potenciálně možná nikdy nedosáhne požadovaného stavu, ale na tom nezáleží, protože ovládače clusteru by měly být neustále spuštěny a pracovat na opravě a zotavení z chyb, pokud je to možné.

4.3.1 Architektura

Nasazené prostředí Kubernetes se nazývá Kubernetes cluster. Jak lze vidět na obrázku 4.2, cluster má dvě části: Control Plane nebo řídicí panel a worker nodes nebo pracovní uzly. Řídicí panel a instance uzlů mohou být fyzická zařízení, virtuální stroje nebo instance v cloudu.



Obrázek 4.2: Kubernetes architektura [28]

4.3.2 Komponenty

Pod

Pod je kolekce jednoho nebo více kontejnerů, jejich konfigurací a některých sdílených zdrojů pro tyto kontejnery [27]. Mezi tyto zdroje patří:

- Sdílené úložiště
- Síťové zdroje, jako sdílená adresa IP[5] clusteru

[5]Internet Protocol

- Informace o tom, jak spouštět jednotlivé kontejnery, jako je verze image kontejneru nebo konkrétní porty, které se mají použít

Jedná se o nejmenší výpočetní jednotku, kterou lze vytvořit, nasadit a spravovat pomocí platformy Kubernetes.

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
  - name: nginx
    image: nginx:1.14.2
    ports:
    - containerPort: 80
```

Obrázek 4.3: Ukázka YAML definice podu

Pod se může nacházet v následujících fázích [27]:

- **vyčkávající (Pending)** - již byl přijat systémem, ale všechny jeho kontejnery ještě nebyly vytvořeny, nebo stále nebyl naplánován,
- **běžící (Running)** - již byl přiřazen pracovnímu uzlu a veškeré kontejnery byly už vytvořeny a minimálně jeden kontejner stále běží nebo se nachází ve stavu startování nebo restartování,
- **úspěšný (Succeeded)** - všechny kontejnery úspěšně skončily a nebudou nadále restartovány,
- **neúspěšný (Failed)** - alespoň jeden z běžících kontejnerů skončil s chybou,
- **neznámý (Unknown)** - z nějakého důvodu se nepodařilo zjistit stav. Tato fáze obvykle nastává v důsledku chyby při komunikaci s uzlem, na kterém by měl pod běžet.

Pody se řídí definovaným životním cyklem, začínají ve fázi Pending, postupují přes Running, pokud alespoň jeden z primárních kontejnerů začíná pracovat, a pak buď fází Succeeded, nebo Failed, podle toho, zda některý kontejner v podu skončil chybou [27].

Zatímco běží pod, kubelet (viz podkapitolu 4.3.2, sekce „Základní komponenty uzlu“) je schopen restartovat kontejnery, aby zvládl nějaké závady. V rámci podu sleduje Kubernetes různé stavy kontejnerů a určuje, jaké kroky učinit, aby byl pod opět zdravý.

Workload resources

Pro automatické řízení podu Kubernetes poskytuje řadu zdrojů, které se nazývají workload resources. Tyto zdroje konfigurují ovládače, které se ujistí, že je spuštěno správné číslo správného typu podu, aby odpovídalo zadanému stavu.

Kubernetes poskytuje několik vestavěných zdrojů:

1. **Deployment a ReplicaSet** (nahrazení staršího ReplicationController)

   ReplicaSet neboli replikační sada zajišťuje, že určitý počet podů bude v clusteru Kubernetes spuštěn kdykoli. Deployment je abstrakce vyšší úrovně, která spravuje replikační sady a poskytuje pokročilé funkce správy kontejnerů. Vývojáři Kubernetes doporučují použít Deployment, pokud není potřeba konkrétní nastavení orchestru. Deployment se stará sám o správu replikačních sad.

2. **StatefulSet**

   StatefulSet je používán ke spouštění stavové aplikace v clusteru Kubernetes. StatefulSets přiřazují každému podu identitu – pořadové číslo začínající od nuly – místo přiřazování náhodných ID pro každou repliku podu. Informace o stavu a další data pro libovolný pod v StatefulSetu jsou udržovány v trvalém diskovém úložišti spojeném se StatefulSetem.

3. **DaemonSet**

   Hlavním účelem sady je spustit pody na všech uzlech clusteru: pokud je uzel přidán/odebrán, DaemonSet automaticky přidá/odebere pod. Sada DaemonSet je vhodná pro spouštění aplikací, které by měly být spuštěny na všech uzlech, jako jsou monitorování, sběr logů atd.

4. **Job a CronJob**

   Job v Kubernetes je navržen tak, aby vytvořil pod, v němž bude proveden pouze jeden úkol, a pak bude zastaven. Úloha může vytvořit jeden nebo několik podů, spustit úlohu souběžně v několika podech, provést určený počet operací a pak bude dokončena. CronJob je podobný, ale pomocí plánovače může být úloha naplánována.

Základní komponenty řídicí panely

Součásti řídicího panelu rozhodují o clusteru a také detekují události v clusteru a reagují na ně. Kubernetes se opírá o několik administrativních služeb běžících v panelu. Tyto služby spravují aspekty, jako jsou komunikace součástí clusteru, plánování pracovní zátěže a zachovávání perzistence clusteru.

API Server (kube-apiserver)

API server je frontendová cast řídící roviny Kubernetes. Vystavuje HTTP API, které umožňuje komunikaci mezi koncovými uživateli, externími součástmi a různými částmi clusteru. Kubernetes API umožňuje dotazovat se a manipulovat se stavem API objektů, jako jsou pody, ConfigMaps, události a jmenné prostory [29].

Etcd (klíč-hodnota úložiště)

Jedná se o konzistentní, distribuované a vysoce dostupné úložiště, které je navrženo tak, aby nemělo jediný bod selhání a tolerovalo selhání hardwaru a sítí. Je to jediný zdroj pravdy pro celý cluster, ve kterém se uchovávají všechna data. Etcd je součást řídicího panelu, ale může být nakonfigurován externě [27].

Plánovač (kube-scheduler)

Jedná se o část, která sleduje nově vytvořené pody bez přiřazeného uzlu a vybírá pro ně uzly, které tam budou spuštěny. Nejsou-li vhodné uzly, pod je převeden do čekacího stavu.

Mezi faktory, které se berou v úvahu při rozhodování o plánovači, patří:

- požadavky na zdroje
- lokalizace dat
- specifikace affinity a anti-affinity pravidel[6] (umožňuje omezit, na které uzly je možné naplánovat pod na základě labelu v uzlu).
- omezení hardwaru/softwaru

Správce (kube-controller-manager)

Sleduje požadovaný stav objektů a jejich aktuální stav prostřednictvím serveru API. Provede opravné kroky pro převodu do požadovaného stavu. Logicky je každý správce samostatný proces, ale aby se snížila složitost, jsou všechny zkompilovány do jednoho binárního souboru a běží v jednom procesu [27].

Některými typy správců jsou:

- ovládač uzlu: odpovídá za zaznamenání a odezvu při vypnutí uzlů,
- ovládač koncových bodů (endpointů): odpovědný za zajišťování koncových bodů,
- ovládač replikací: odpovídá za údržbu replikací objektů v clusteru.

Základní komponenty uzlu

Základní komponenty běží na každém uzlu, udržují pody a poskytují runtime prostředí Kubernetes.

Kubelet

Kubelet je proces, který běží na každém uzlu Kubernetes a vytváří, ničí nebo aktualizuje pody a jejich Docker kontejnery pro daný uzel, když dostane pokyn.

Kube-proxy

Kube-proxy je síťová proxy, která běží na každém uzlu clusteru a implementuje část konceptu Kubernetes Service. Kube-proxy udržuje síťová pravidla na uzlech. Tato síťová pravidla umožňují síťovou komunikaci s pody uvnitř nebo vně clusteru.

Runtime kontejneru

Runtime kontejneru je software zodpovědný za provoz kontejnerů v podech. Ke spuštění kontejnerů má každý pracovní uzel kontejnerový runtime engine. Spuštění se provádí vytažením image z registru.

4.4 Porovnaní platforem pro orchestraci

Tato podkapitola bude věnována podrobnějšímu srovnání popsaných platforem pro orchestraci kontejnerů. Každá z výše uvedených platforem má svá „pro“ a „proti“, ale pokusíme se porovnat rozdíly různých vlastností (např. konfigurace clusteru, síťové nastavení atd) Docker Swarmu a Kubernetes.

[6]Dostupné z: `https://kubernetes.io/docs/concepts/scheduling-eviction/assign-pod-node/` (07.08.2022)

4.4.1 Instalace

DockerSwarm

DockerSwarm je považován za lehký orchestrační nástroj s jednoduchou a rychlou instalací. DockerSwarm je také projekt Docker, takže pokud je Docker již nainstalován na požadovaném hostiteli, lze konfiguraci clusteru spustit okamžitě. DockerSwarm má stejné API a CLI rozhraní, což je velká výhoda pro společnosti s malými nebo téměř neexistujícími oddíly DevOps s menšími technickými znalostmi orchestračních technologií.

Kubernetes

Před verzí 1.21 byla instalace Dockeru prvním a povinným krokem pro instalaci Kubernetes, ale vývojáři Kubernetes oficiálně oznámili, že ve verzi 1.23 a vyšší zcela opustí Docker jako runtime prostředí kontejnerů. Místo Dockeru bude použito nativní rozhraní CRI[7]. Kubernetes vyžaduje řadu manuálních konfigurací pro spojení jeho součástí. Existuje celá řada způsobů instalace, od plně manuální metody až po automatizovanou instalaci, které budou podrobněji rozebrány v kapitole 6.2.

4.4.2 Konfigurace clusteru

DockerSwarm

Po dokončení kroků nastavení jsou Docker hosty připraveny pro vytvoření clusteru. Celá konfigurace clusteru přes DockerSwarm může být provedena v několika krocích:

1. inicializace manažera na jednom z uzlu a vytvoření clusteru na tomto uzlu. Až manažer bude nakonfigurován, vygeneruje se token,
2. konfigurace pravidel firewallu pro povolení provozu Docker Swarmu,
3. pomocí tokenu z kroku č. 1 do clusteru se přidají nové pracovní a manažerské uzly,
4. vytvoření a spuštění služeb v Docker Swarmu. Při vytvoření služby je možné také nakonfigurovat počet replik této služby.

Kubernetes

Konfigurace Kubernetes clusteru je podobná konfiguraci Docker Swarmu, ale kvůli složitější architektuře vyžaduje další kroky, jako je nastavení podů a deployment pro řízení instancí podů. Ačkoli oficiální dokumentace nabízí podrobné pokyny, instalace může vyžadovat více technických znalostí a více času na konfiguraci.

4.4.3 Řízeni stavu prostředí pomoci GUI

DockerSwarm

Docker Swarm nemá vestavěné GUI. Existuje však několik nástrojů třetích stran.

Kubernetes

Kubernetes přichází s vlastním vybudovaným GUI. Kubernetes Dashboard umožňuje snadno škálovat a zavádět jednotlivé aplikace, a také ovládat a monitorovat různé clustery.

[7]Container Runtime Interface

4.4.4 Škálovatelnost

Jak Kubernetes, tak Docker Swarm umožní snadněji škálovat infrastrukturu nahoru nebo dolů v závislosti na potřebách. Zatímco Swarm vyžaduje, aby veškeré škálování probíhalo ručně, Kubernetes přidává automatizované škálování na základě provozu a metrik využití uzlů. V Kubernetes existují tři možnosti automatického škálování: Horizontální Pod Autoscaler, Vertikální Pod Autoscaler a Cluster Autoscaler [30].

4.4.5 Směrovaní dotazu uvnitř clusteru a ze vnějšku

DockerSwarm Docker Swarm load balancer běží na každém uzlu a dokáže zpracovat a přesměrovat požadavky na libovolný kontejner na libovolném hostiteli v clusteru. V případě nasazení Docker Swarmu bez NGINX Swarm load balancer obsluhuje příchozí požadavky klientů i interní požadavky služeb, ale jen na 4. vrstvě (TCP[8]). Vyrovnávač zatížení 4. vrstvy rozhoduje o směrování na základě zdrojových a cílových adres IP a portech zaznamenaných v hlavičce paketu, aniž by bral v úvahu obsah paketu.

Ovšem mnohé aplikace vyžadují dodatečné funkce, jako jsou:

- SSL[9]/TLS[10],
- směrování na základě obsahu (založené například na URL[11]),
- kontrola a autorizace přístupu,
- přepisování (rewrite) a přesměrování (redirects).

Vnitřní load balancer Dockeru neposkytuje tyto možnosti, ale integrováním externího vyrovnávače, jako například NGINX, lze zajistit potřebnou funkčnost.

Kubernetes

Důležitými komponentami v architektuře **Kubernetes**, které odpovídají za směrování dotazu a jež nejsou uvedeny v kapitole Kubernetes, jsou Service a Ingress Controller.

Service je abstrakce, která definuje logickou množinu podů a zásadní pravidla, podle kterých k nim lze přistupovat. U Kubernetes není nutné upravovat aplikaci, aby bylo možné použít mechanismus vyhledávání služeb (Discovery Service). Kubernetes dává podům jejich vlastní IP adresy a jediný název DNS pro sadu podů a může přes ně vyvazovat zátěž [27]. Tyto služby, stejně jako v Dockeru, jsou objekty čtvrté úrovně modelu OSI.

Service může být pěti typů:

- ClusterIP
- NodePort
- LoadBalancer
- ExternalName

[8] Transmission Control Protocol
[9] Secure Sockets Layer
[10] Transport Layer Security
[11] Uniform Resource Locator

- ExternalIPs

Ingress je objekt, který spravuje externí přístup ke službám v clusteru, obvykle HTTP/HTTPS. Na rozdíl od služeb je objektem 7. úrovně. Tento objekt je pod, na kterém běží kontejner s aplikací (například NGINX, HProxy) distribuující provoz zvenku uvnitř clusteru. Může poskytovat load balancing, SSL offloading (rozšifrování a opětovné zašifrování SSL komunikace) a virtuální hosting na základě názvu [27]. Pro skutečné vyrovnávání zatížení nejpopulárnější metodou v Kubernetes je Ingress, fungující jako controller ve specializovaném modulu. Ovládač obsahuje soubor pravidel, jimiž se řídí provoz – a Daemona, který tato pravidla uplatňuje. Ovládač má vlastní vestavěné funkce pro vyrovnávání zatížení s několika přiměřeně sofistikovanými schopnostmi.

4.4.6 Síť

DockerSwarm
Pro Docker Swarm jsou důležité tyto tři síťové koncepty:

- Overlay nebo překryvné sítě spravují komunikaci mezi Docker Daemony v clusteru.
- Ingress síť je speciální překryvná síť, která usnadňuje load balancing mezi uzly služby. Když jakýkoliv uzel clusteru obdrží request na publikovaném portu, předá tuto žádost modulu nazvanému IPVS[12]. Ten sleduje všechny IP adresy účastnící se této služby, vybere jednu z nich a požadavek k ní přesměruje přes tuto síť. Ingress síť je vytvořena automaticky při inicializaci nebo připojení do clusteru.
- Docker_gwbridge je mostová (bridge) síť, která spojuje překryvné sítě (včetně Ingress sítě) s fyzickou sítí démona Docker

Kubernetes Síťování je ústřední součástí Kubernetes. V dokumentaci jsou uvedeny čtyři síťové modely, které je třeba znát [31]:

- "Container-to-Container"komunikace. Container-to-Container“ komunikace. Kontejnery uvnitř podů jsou jako procesy běžící v rámci VM nebo hostitele – kontnejnery běží ve stejném síťovém jmenném prostoru a sdílejí IP a MAC adresu [32]. Veškerá komunikace mezi těmito kontejnery může probíhat přes localhost, protože jsou všechny součástí stejného jmenného prostoru.
- „Pod-to-Pod“ komunikace. Kubernetes nespravuje konfiguraci sítě pod-to-pod sám. Síť je instalována správcem systému, nebo může být připojena pluginem CNI (Container Network Interface). V Kubernetes se CNI integruje s kubeletem a umožňuje automatickou konfiguraci sítě pomocí podkladové (underlay) nebo překryvné (overlay) sítě. Příklady pluginů ČNI:
  - Flannel
  - Calico
  - Romana

[12] IP Virtual Server

  - AWS VPC CNI

  Praktická instalace v clusteru bude uvedena v kapitole 7.2.4.

- "Pod-to-Service"a "Internet-to-Service"komunikace. Každý pod má svou IP adresu, přes kterou je možné s ním komunikovat, ale ne vždy je to vhodné. Jednou z hlavních abstrakcí sítě je Service, která pomáhá vyhnout se spoustě nepříjemností tím, že poskytuje jediný bod vstupu do skupiny podů. Jak je tento objekt realizován a jakým způsobem přesměrován provoz uvnitř i vně clusteru, lze najít v článku `https://opensource.com/article/22/6/kubernetes-networking-fundamentals`.

4.4.7 Monitorování

Monitorování pomáhá identifikovat problémy a proaktivně řídit clustery kontejnerů. Efektivní sledování clusterů usnadňuje správu kontejnerové infrastruktury sledováním provozu, využíváním prostředků clusteru (jako jsou paměť, CPU a úložiště) a interakcí mezi součástmi clusteru.

Kubernetes nabízí několik nativních řešení pro monitorování nasazených služeb v rámci clusteru. Tato řešení používají Node Problem Detector a monitorují výkon aplikací pomocí:

- pozorování stavu celého clusteru,
- kontroly služeb, podů a kontejnerů.

Na rozdíl od Kubernetes Docker Swarm nenabízí monitorovací řešení. Ale jak v prostředí Docker Swarm, tak i v Kubernetes lze používat nástroje třetích stran pro komplexní sledování, jako jsou:

- Grafana
- Prometheus
- Zabbix

4.4.8 Rolling updates and rollbacks

DockerSwarm

Ve světě Dockeru se nasazení softwaru nebo nasazení nových verzí provádí prostým nahrazením aktuálně spuštěného kontejneru novým kontejnerem s novým zdrojovým kódem. Oba kontejnery by měly mít stejné rozhraní a zobrazovat stejné API. Během tohoto procesu budou oba kontejnery po určitou dobu zastaveny a veškeré requesty, které do kontejneru přijdou, budou zamítnuty – to způsobí prostoje u příslušné služby. Délka prostoje závisí také na metodě použité pro výměnu kontejnerů – pokud byla výměna provedena ručně, budou prostoje značné.

Naštěstí Docker Swarm poskytuje řešení tohoto problému. Řešením je jednoduše použít Docker služby místo ručního spouštění a zastavování Docker kontejnerů. Objekt služby může být definován a konfigurován nejen pro ovládání a zaručení nulové doby výpadku během výměny, ale také pro ovládání v případě selhání a vrácení zpět (rollback).

Kubernetes

Kubernetes podporuje nasazení nových verzí s nulovým prostojem a nabízí až šest různých strategií:

- Recreate: při tomto typu velmi jednoduchého nasazení jsou všechny staré pody zabity najednou a jsou nahrazeny novými,
- Ramped nebo rolling-update: takové nasazení je standardním výchozím nasazením pro Kubernetes. Funguje to tak, že pomalu, jeden po druhém, nahrazujeme pody předchozí verze aplikace pody nové verze bez jakýchkoliv odstávek clusteru,
- Blue/green: ve strategii nasazení modré/zelené se zároveň nasazuje stará verze aplikace (zelená) a nová verze (modrá). Při nasazení obou těchto funkcí mají uživatelé přístup pouze k zelené; zatímco modrá je k dispozici týmu QA12 pro testovací automatizaci na samostatné službě nebo prostřednictvím přímého port-forwardingu,
- Canary: tento typ se částečně podobá nasazení blue/green, ale je více kontrolován a využívá více postupného zavádění,
- A/B testování: Testovací nasazení A/B spočívá ve směrování podskupiny uživatelů na novou funkčnost za specifických podmínek. Obvykle se jedná spíše o techniku pro přijímání obchodních rozhodnutí na základě statistik než o strategii nasazení.

4.5 Shrnutí

Vysoká popularita těchto dvou technologií často vyvolává dlouhé debaty o tom, co je vlastně lepší používat. Při porovnávání různých funkčních oblastí, které jsou popsány výše, uvádíme následující shrnutí:

Docker Swarm je built-in nástroj pro orchestraci. Snadno se používá a konfiguruje, ale není příliš flexibilní. Hlavní výhodou je jednoduchost a rychlost nastavení a konfigurace clusteru, a také plná kompatibilita s Dockerem. To je ideální pro malé společnosti a projekty, které vyžadují od organizace minimální zlepšení.

Kubernetes je všestranný nástroj pro vytváření distribuovaných systémů. Je to složitý systém se spoustou funkcí. Hodí se pro středně velké projekty, které plánují rozvoj a rozšíření své funkčnosti a architektury, a pro velké společnosti vyžadující pružnou konfiguraci celé infrastruktury.

Docker Swarm je ve srovnání s Kubernetes snadno použitelný orchestrační nástroj, avšak s omezenou nabídkou. Oproti tomu Kubernetes je komplexní, ale je samoregenerační, nabízí více možností pro zajištění vysoké dostupnosti a odolnosti vůči poruchám. Na základě výše uvedené analýzy se praktická část této práce zaměří na vybudování plnohodnotné systémové infrastruktury s využitím technologie Kubernetes pro orchestraci.

Kapitola 5

As-Is infrastruktura zadavatele

5.1 Popis systému

Základem pro praktickou část této práce je komerční projekt Smartform české firmy Trixi software s. r. o. Společnost Trixi déle než 15 let vyvíjí software na zakázku, ale Smartform je jeden z prvních projektů, jež jsou stále na trhu. Smartform začínal jako služba pro pohodlné vložení poštovní adresy pomocí našeptávačů a zkontrolování, zda je zadaná adresa platná. V současné době je nabídka rozšířena a zahrnuje tyto webové služby:

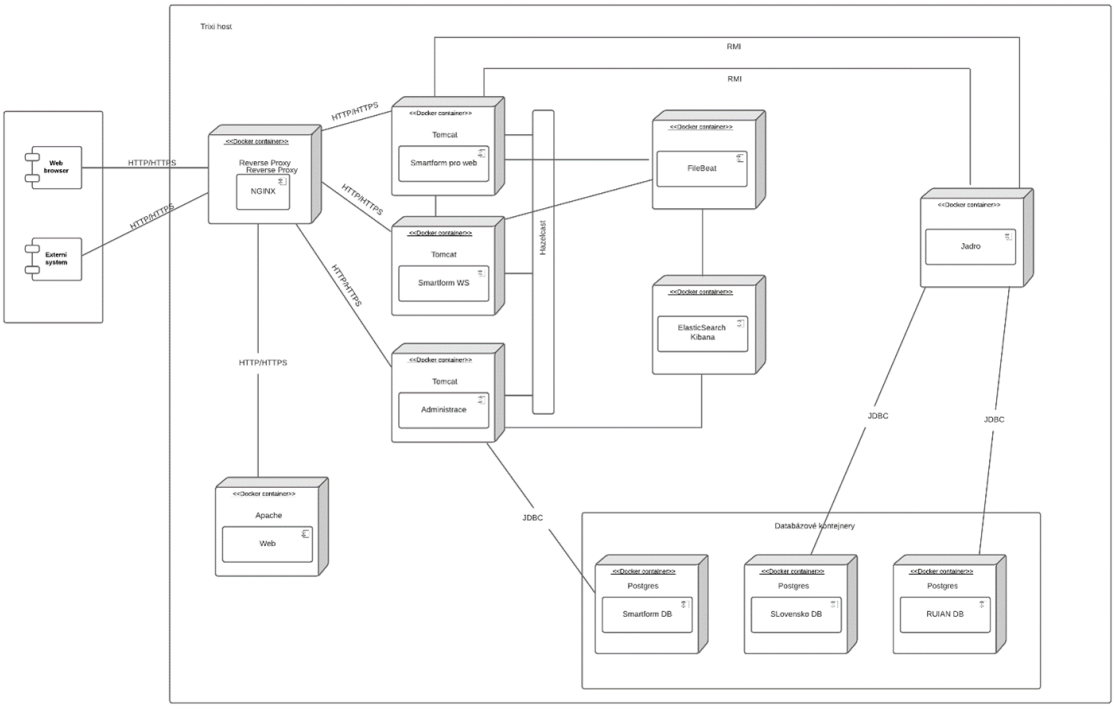
- kontrola poštovních adres, doplnění souřadnic a dalších podrobností k adrese,
- našeptávání poštovních adres,
- hledání adres a parcel podle souřadnic (reverse geocoding),
- hledání parcel,
- ověření sídla obecního úřadu na dané poštovní adrese,
- kontrola křestních jmen a příjmení osob, doplnění 5. pádu jmen, určení pohlaví.

Integrace se systémem je možná dvěma způsoby: přidáním skriptů do frontendové části webového klienta nebo integrací do firemního backendu. Tento projekt se neustále zlepšuje, nových zákazníků neustále přibývá, stejně tak se ale zvyšuje zatížení systému.

S cílem zajistit zákazníkům bezproblémový provoz služby na svých webových stránkách a automatizovat časově náročné procesy v rámci projektu bylo rozhodnuto o zlepšení infrastruktury. Toto zlepšení bude zaměřeno na zvýšení přístupnosti odstraněním problémů z kapitoly 5.2.

5.2 Slabé stránky existující infrastruktury

Architektura systému Smartform je založena na principu distribuovaných služeb. Charakteristickým rysem tohoto typu je zpracování informací rozdělených mezi několik částí systémů, které poskytují omezenou funkčnost. Distribuované systémy umožňují různé způsoby komunikace služeb.



Obrázek 5.1: Smartform architektura

Architektura systému je znázorněna na obrázku 5.1. Služby komunikují třemi různými způsoby: HTTP/ HTTPS, RMI[1] a Hazelcast. Gateway a reverzní proxy pro všechny příchozí požadavky klientů je kontejner NGINX, který je distribuuje mezi čtyři hlavní kontejnery v závislosti na obsahu požadavku. Data systému jsou uložena ve třech objektově-relačních databázích PostgresSQL, z nichž dvě představují úložiště českého a slovenského registru územní identifikace, adres a nemovitostí.

V současné infrastruktuře společnosti všechny aplikace běží na fyzickém serveru v kontejnerovém prostředí pomocí nástroje Docker. Tato infrastruktura dává plnou kontrolu nad spuštěnými aplikacemi, ale má své slabé stránky.

1. Každá část systému je prezentována v jedné instanci bez dalších replik. Takový systém netoleruje selhání kontejneru. Třebaže zbytek systému může fungovat i při výpadku jednoho kontejneru, je to stále považováno za prostoj, což činí zákaznickou zkušenost méně příjemnou.

2. Systém nemá možnost automatické opravy v případě poruchy, z toho důvodu je ruční zásah jedinou strategií nápravy. V takové situaci se výrazně prodlužuje doba odezvy na selhání, stejně jako celková doba výpadku služby.

3. Fyzický server je jediný bod selhání. Pokud se z nějakého důvodu server vypne (důvody, proč se to může stát, byly analyzovány v kapitole 1.2.2), bude celý systém nedostupný. Bez záložního serveru, jako v případě Failover strategie, může být tento výpadek opravdu dlouhý.

4. Dalším bodem selhání je reverzní proxy Nginx, který přesměruje dotazy od zákazníků. V případě selhání serveru proxy to ovlivní celý systém – všechny služby budou fungovat, ale nebudou k dispozici zákazníkovi.

[1] Java remote method invocation

5. Absence průběžné kontroly. Problém v tomto případě může nastat přímo v aplikaci bez selhání kontejneru. Bude to vypadat, že služba funguje, ale nemusí být k dispozici zákazníkům.

6. Průběžná aktualizace (Rolling Update) během nasazení není podporována. Takže při nové verzi aplikace je potřeba nejprve vypnout současnou verzi a poté nasadit novou, služba v tomto okamžiku nebude k dispozici.

Z těchto důvodů současná infrastruktura neposkytuje vhodnou úroveň odolnosti proti chybám, a proto nepředstavuje nejvhodnější variantu. Pro zvětšení úrovní dostupnosti a škálování bylo rozhodnuto o vytvoření infrastruktury, založené na platfromě orchestrací kontejneru. Taková platforma pro orchestrace kontejnerů umožní budovat distribuovaný systém s automatickým řízením životního cyklu kontejnerů a udržením stabilního stavu systému. Ke zlepšení dostupnosti dojde především na softwarové úrovni a ke stávajícímu serveru přibudou další dva.

V předchozí kapitole při porovnání dvou platforem bylo rozhodnuto použít Kubernetes, takže v další části ukážeme, jaké jsou možnosti nasazení clusteru Kubernetes, jak jsme použili tento cluster pro nasazení systému na novou infrastrukturu s cílem zlepšit dostupnost a eliminaci výše uvedených problémů, a k jakému výsledku jsme nakonec dospěli.

Kapitola 6

Příprava na vytvoření clusteru Kubernetes

V předchozí kapitole jsme zformulovali hlavní problémy, které plánuji minimalizovat v rámci praktické části této bakalářské práce. Tato kapitola se zaměří na přípravu před samotným nasazením.

Nedílnou součástí, která může trvat dostatečné množství času, je plánování. Pokud je cluster Kubernetes navržen tak, aby vykonával kritická pracovní zatížení, musí být nakonfigurován na odolnost vůči chybám a výpadku. V této kapitole popisuji hlavní úkoly, na které jsem musela myslet při plánování clusteru v Kubernetes. Budeme analyzovat příklady clusterových modelů, jak je možné nasadit zvolenou infrastrukturu, a také problém databází v kontejnerizovaném prostředí.

6.1 Modely clusterů Kubernetes

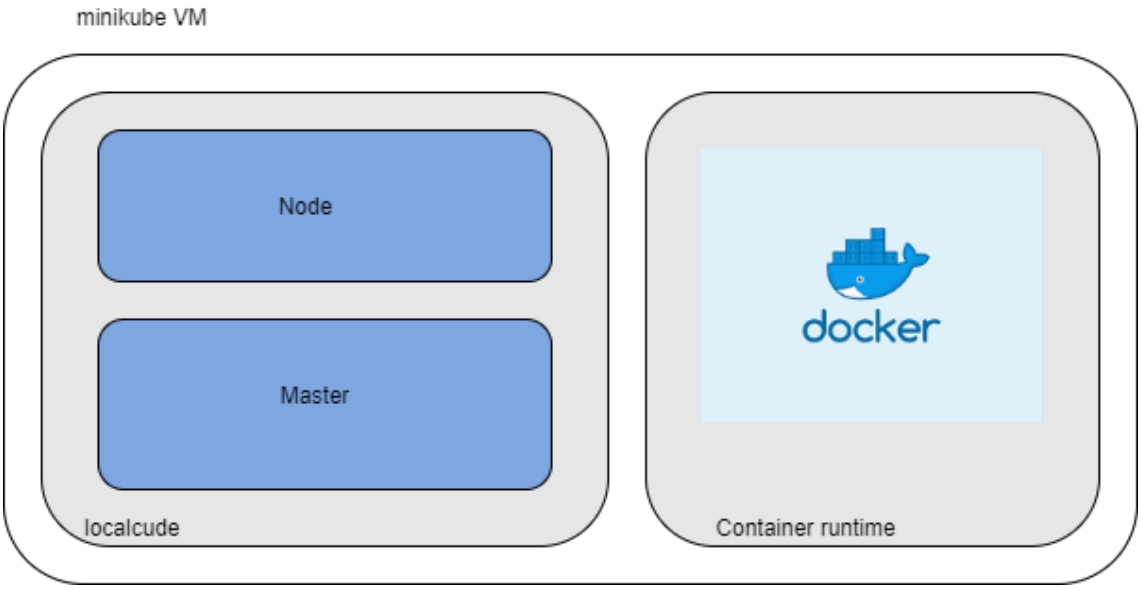
V kapitole 4.3.1 jsme zmínili, že cluster Kubernetes se skládá ze dvou hlavních částí: Control Plane, který běží na master uzlech, a worker uzly, na které přichází základní pracovní zátěž. V závislosti na velikosti budované infrastruktury bude záviset umístění a počet těchto dvou hlavních spojení. Modely nasazení pro Kubernetes sahají od jednoho serveru, vykonávajícího funkce masteru a workeru, až po víceuzlové clustery ve více datových centrech. V této kapitole se zaměříme na existující modely nasazení, jejich rozdíly a to, jaká infrastruktura je nejlepší pro různá prostředí, a zvolíme vhodný model pro implementaci nové infrastruktury.

6.1.1 Single node cluster

V tomto modelu se používá jediný uzel pro hostování jak služeb Control Plane, tak i pracovního uzlu. Tato architektura je nejsnazší na nasazení, ale neposkytuje vysokou dostupnost služeb. V případě, že uzel není dostupný, nemůže docházet k žádné interakci se serverem minimálně do doby, než bude server vrácen do provozu.

Jednoduché nasazení, často s využitím jediného serveru, je ideální pro tým vývojářů nebo QA[1] tým. Tato architektura může být užitečná pro testování, kvalifikaci, ověření koncepce a další testovací využití, nicméně by neměla být používána pro nasazení v produkcím prostředí. Tento model nasazení s jedním serverem využívají

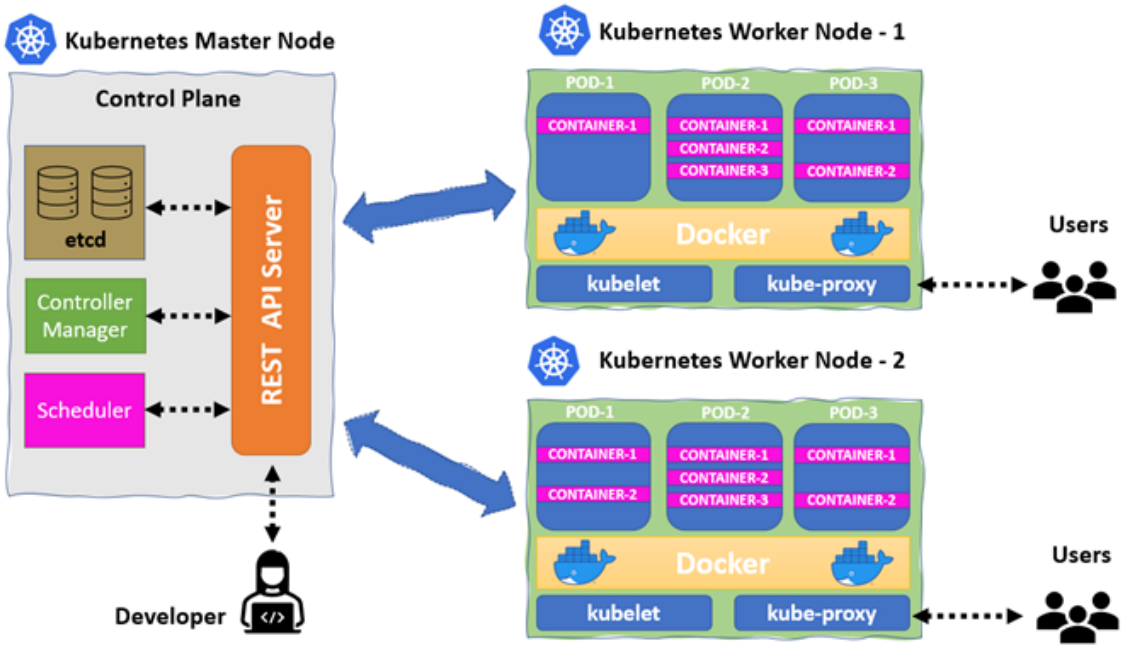
[1] Quality assurance



Obrázek 6.1: Arhcitektura minikube clasteru [33]

distribuci, jako je minikube. Obrázek 6.1 ukazuje architekturu minikube, kde lze vidět, že na jednom virtuálním stroji běží obě komponenty.

6.1.2 Single master cluster



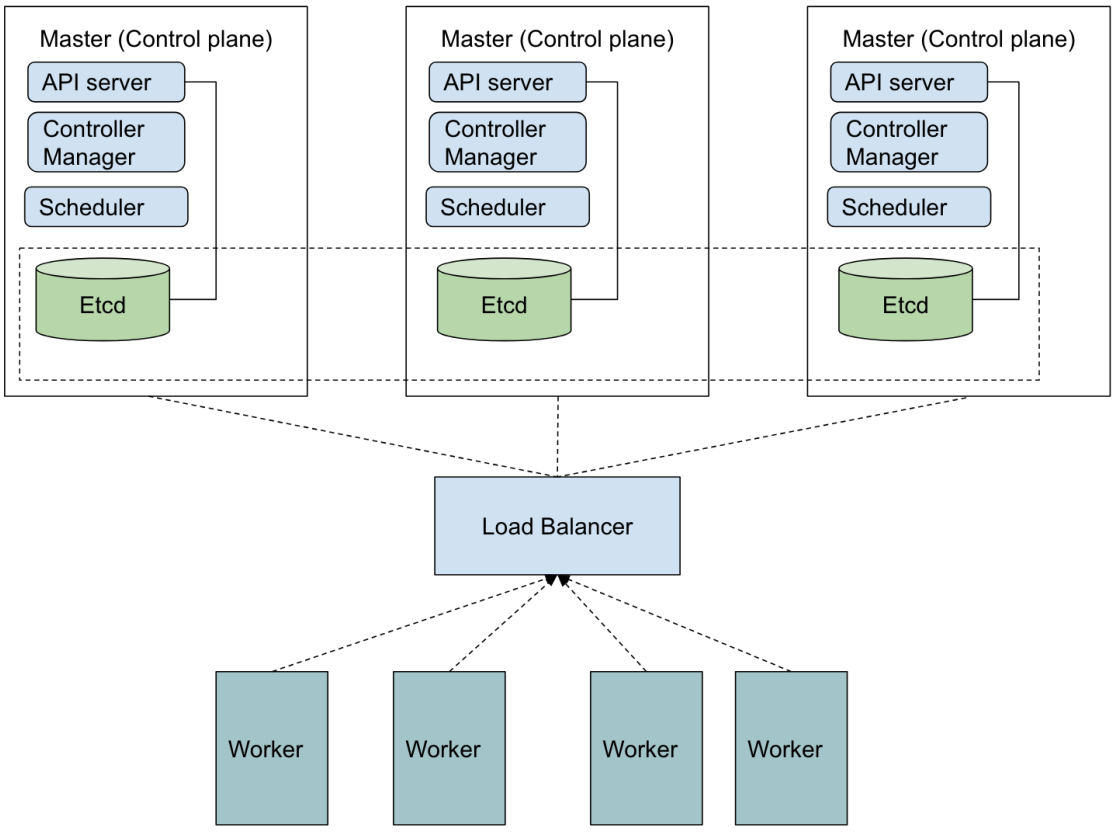
Obrázek 6.2: Arhitektura single master clasteru [34]

Jedná se o nejběžnější model clusteru, zahrnující dva nebo více fyzických nebo virtuálních strojů. V této konfiguraci je jeden master uzel a alespoň jeden worker uzel. Master uzel spravuje databázi etcd, server, správce ovládačů, plánovač a pracovní uzly. Pokud však tento hlavní uzel selže, ztratíme možnost spravovat cluster, což znamená, že tento model také nezajistí vysokou dostupnost, stejně jako model single node.

Porucha hlavního uzlu však neznamená, že cluster a všechny pracovní uzly jsou ztraceny. Cluster bude i nadále běžet s naprosto stejnou konfigurací jako před poruchou. Aplikace spuštěné v clusteru Kubernetes budou i nadále použitelné. Bez hlavního uzlu však není možné vytvářet nová nasazení nebo se zotavovat z poruch uzlů.

Jedním ze způsobů, jak se vypořádat se selháními hlavního uzlu, je nastavení clusteru s vysokou dostupností, jemuž bude věnována pozornost v následující podkapitole. Přístup High Availability Cluster není vždy lepší než nastavení jednoho hlavního serveru: jeho konfigurace je složitější, více času zaberou údržba a monitorování. Proto se v případě výpadku master uzlu občas používá Disaster Recovery[2] strategie, založená na zálohování etcd pomocí snapshotu a následném obnovení.

6.1.3 Multi-master cluster



Obrázek 6.3: Arhcitektura multi-master clasteru [35]

Existence více master uzlů zajišťuje, že služby zůstanou dostupné i v případě selhání jednoho z master uzlů. V případě multi-master prostředí jsou důležité komponenty replikovány na více serverech, a pokud některý z master uzlů selže, ostatní servery udržují cluster v provozu.

Hlavní uzly by měly být nasazeny s lichým počtem (například 3, 5, 7, 9 atd.), takže kvórum (většina master uzlů) může být zachováno, pokud jeden nebo více nadřazených serverů selže. Ve scénáři HA bude Kubernetes udržovat kopii etcd databází na každém masteru, ale uspořádá volby, aby vybral lídra mezi instancemi

[2]Dostupné z: `https://documentation.commvault.com/v11/essential/144609_disaster_recovery_for_kubernetes.html` (07.08.2022)

kube-controller-manager a kube-scheduler na různých master uzlech, aby se předešlo konfliktům. Worker uzly mohou komunikovat s jakýmkoli serverem API prostřednictvím load balanceru.

6.1.4 Výběr vhodného modelu

Shrneme-li analýzu, dá se říct, že každý model má své klady i zápory, v závislosti na potřebách firem může být použit kterýkoli z výše uvedených modelů. Single node cluster je ideální pro lokální vývoj a testování, pro single master cluster se rozhodují malé a střední projekty, u nichž nemusí být velké infrastruktury a prostředky pro udržení HA clusteru, ve velkých firmách bývá nejvhodnější variantou cluster s použitím většího počtu master uzlů.

Jedním z hlavních cílů této práce bylo vytvořit takový cluster, aby splňoval požadavek vysoké dostupnosti, proto se na základě analýzy existujících modelů a analýzy projektu Smartform budu při vývoji clusteru držet třetího modelu Multi-master. Pro testování nové infrastruktury budeme mít k dispozici tři virtuální servery: jeden ze serverů bude mít pouze roli mastera, u dalších dvou, které mají nejlepší technické vlastnosti, jako jsou paměť a výkon, budeme kombinovat roli master a worker uzlů.

6.2 Distribuce kubernetes

Jako open source projekt poskytuje Kubernetes svůj zdrojový kód veřejně přístupný na GitHubu. Kdokoliv si může pomocí tohoto zdrojového kódu stáhnout, zkompilovat a nainstalovat Kubernetes na infrastrukturu podle svého výběru. Z `https://github.com/kelseyhightower/kubernetes-the-hard-way` lze najít tzv. „the hard way“, jak nainstalovat Kubernetes cluster bez použití skriptů a instalátorů. Tento návod vytvořil hlavní inženýr Googlu Kelsey Hightower. Pomohl vyvinout a vylepšit mnoho cloudových produktů Googlu, včetně enginu Google Kubernetes, cloudových funkcí a API brány Apigees. Tato varianta je skvělá pro trénink, ale tento návod není pro ty, kteří hledají plně automatizovaný příkaz ke spuštění clusteru Kubernetes [36].

Většina lidí, kteří chtějí nainstalovat Kubernetes, si však nikdy nestahuje nebo nezkompiluje zdrojový kód z mnoha důvodů: zabere to spoustu času a úsilí, Kubernetes má řadu komponent, které je potřeba správně naistalovat atd.

Z tohoto důvodu existují distribuce Kubernetes, které obsahují instalátory, ovládací panely a dodatečné moduly, jež nejsou v Kubernetes. Distribuce také umožňuje modifikovat komponenty, jako jsou síťové služby, prostředí pro spouštění kontejnerů, aniž by se šlo do interních komponent Kubernetes.

Distribuce lze rozdělit do čtyř typů [37]:

- instalace s jedním uzlem: tento typ instalace se hodí pro ty, kteří chtějí zobrazit náhled Kubernetes, nebo je vhodný pro testy a vývojové účely. Příkladem jsou k3s[3] nebo minikube[4],

- manuální instalace: používá se pro nasazení minimálně fungujícího clusteru. Některé části instalace by měly být provedeny ručně. Je to preferovaný způsob,

[3]Dostupné z: `https://k3s.io/` (21.06.2022)
[4]Dostupné z: `https://minikube.sigs.k8s.io/docs/start/` (21.06.2022)

jak poprvé nasadit cluster Kubernetes. Kubeadm[5] je distribuce tohoto typu,

- automatická instalace clusteru: tento typ se provádí pomocí automatizačních nástrojů, skriptů nebo distribuovaných instalačních programů od poskytovatelů. Je to preferovaný způsob pro ty, kteří chtějí nasadit produkčně kvalitní clustery Kubernetes v prostředí jedné provozovny, nebo chtějí spravovat životní cyklus clusteru ručně. Příkladem jsou kops[6] nebo kubespray[7],
- spravované clustery: životní cyklus clusterů je spravován poskytovateli. V tomto typu instalace může být nasazen produkčně kvalitní cluster s minimálními uživatelskými akcemi. Poskytovatelé jsou zodpovědní za správu celého clusteru i základní infrastruktury.

Za nejvhodnější variantu pro tuto bakalářskou práci považujeme použití manuální instalace clusteru. Na rozdíl od jednouzlové instalace tento typ zahrnuje konfiguraci clusterového multimasteru. Také v porovnání s automatizovaným nastavením, které spouští většinu nastavení automaticky jediným příkazem, instalátoři jako Kubeadm vytvoří pouze minimální konfiguraci a umožňují manuálně ovlivňovat konfiguraci clusteru.

V praktické části při instalaci clusteru použijeme instalátor Kubeadm. Tento instalátor je vyvíjen a udržován oficiální komunitou Kubernetes, proto je nejdoporučovanější variantou. Na oficiálních stránkách je uveden manuál použití Kubeadm, který lze nalézt na `https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/create-cluster-kubeadm/`.

6.3 Docker registry

V kapitole 3.1.1 jsme zmiňovali, že Image je hlavním stavebním kamenem pro vytvoření a spouštění Docker kontejnerů. Registr je bezstavová, vysoce škálovatelná serverová aplikace, která ukládá a umožňuje distribuovat Docker obrazy kontejnerů [38]. Hlavním účelem registru je zjednodušit distribuci obrazů. Docker registr může být buď veřejný, nebo soukromý.

Docker Hub[8], nejpopulárnější veřejný registr, má více než milion obrazů a může uchovávat vaše vlastní Docker obrazy. Avšak v případě, že systém je soukromý, není možné z bezpečnotních důvodů, aby byl váš obraz veřejně dostupný. Obrazy obvykle obsahují veškeré kódy potřebné ke spuštění aplikace, takže při nasazení proprietárního softwaru je vhodnější použít soukromý registr.

Tyto registry je možné samostatně spravovat na vlastním hardwaru a v cloudu nebo využít předpřipravené řešení. V kapitole 7.2.7 bude ukázán způsob získání přístupu k self-hosted registru. I když spuštění registru není součástí této práce, rádi bychom zanechali odkazy na to, jak je možné vytvořit a zabezpečit takový registr (`https://docs.docker.com/registry/deploying/`) a jaké jsou populární platformy třetích stran (`https://bluelight.co/blog/how-to-choose-a-container-registry`).

[5]Dostupné z: `https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/create-cluster-kubeadm/` (21.06.2022)

[6]Dostupné z: `https://kops.sigs.k8s.io/` (21.06.2022)

[7]Dostupné z: `https://kubernetes.io/docs/setup/production-environment/tools/kubespray/` (21.06.2022)

[8]Dostupné z: `https://hub.docker.com/` (07.08.2022)

6.4 Databáze PostgreSQL v Kubernetes

Kubernetes byl původně navržen pro aplikace bez datového úložiště a stavu. Jakýkoliv kontejner nasazený na Kubernetes má vysokou pravděpodobnost restartu a selhání. Je to způsobeno tím, že pody jsou dočasné a jsou navrženy tak, aby se v případě problémů deaktivovaly a restartovaly.

Proto nemůžeme naše data uchovávat v podu. Naopak, aplikace se zachováním stavu by měly data ukládat a mít je k dispozici mezi restarty. Databáze jako Postgres nebo MySQL jsou typické aplikace se zachováním stavu. Přestože Kubernetes takové aplikace podporuje, pro optimální provoz bude nutné nakonfigurovat další komponenty.

V kapitole 4.3.2 jsme uvedli definici StatefulSet s tím, že tento objekt je určen pro organizování „stateful“ aplikací. Pomocí sady StatefulSet mohou být data uložena v trvalém úložišti, čímž dojde k oddělení databázové aplikace od trvalého úložiště, takže když je pod znovu bude restartovat, všechna data tam stále budou. Navíc když je pod znovu vytvořen v sadě StatefulSet, ponechá si stejné jméno, takže máme konzistentní endpoint, ke kterému se lze připojit. Konzistentní data a konzistentní jména jsou dvě z největších výhod sady StatefulSets.

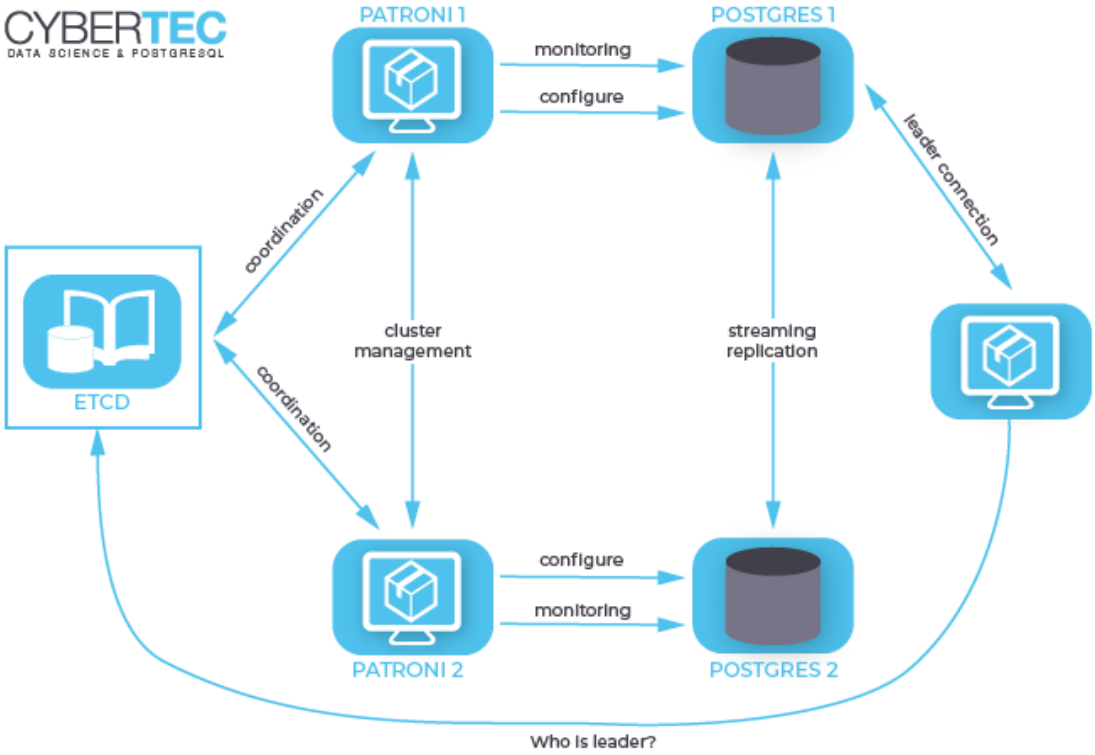
Pro uložení databázových dat Kubernetes zajišťuje připojení souborového systému uvnitř kontejneru pomocí Persistent Volume. Persistent Volumes mohou být zcela odlišné: od lokálních disků po externí clusterové systémy. Dvě běžné metody jsou vázání Persistent Volumes na lokální souborový systém hostitele nebo ukládání dat do vzdáleného úložiště (například NFS – distribuovaný souborový systém). Výhodou použití vzdáleného úložiště je, že je nezávislé na clusteru, a pokud dojde ke ztrátě uzlů, neztratí se na něm uložená data. Ve velkých systémech, kde je větší počet uzlů, existuje pravděpodobnost, že uzel selže. Když je infrastruktura malá a uzly jsou udržovány přímo ve firmě nebo pronajímány, je tato pravděpodobnost extrémně nízká. Ale s lokálním souborovým systémem se objeví další minus, že restart na jiném uzlu je nemožný, protože jeho úložiště tam není vytvořeno.

Kubernetes je znám tím, že téměř všechny komponenty mají přídavné moduly a rozšíření, které automatizují a zjednodušují práci. Pokud jde o databáze v Kubernetes, používají se controllery, jimž se říká operátoři. Jsou to softwarová rozšíření Kubernetes, která využívají vlastní zdroje ke správě aplikací a jejich součástí [27]. Operátor Postgres, například Stolon, Crunchy Data PostgreSQL Operator nebo Zalando Postgres Operator, spravuje clustery PostgreSQL na Kubernetes. Tyto operátory mohou obsahovat další funkce, jako jsou sdílení a replikace dat, volba lídra a funkce failoveru, potřebné pro úspěšné nasazení MySQL nebo PostgreSQL v Kubernetes.

V závislosti na infrastuktuře aplikace lze nasadit PostgreSQL Server dvěma způsoby:

1. jednodušší způsob – nasadit jen jednu instanci. V takovém případě není nutné se starat o replikaci dat nebo výběru hlavního serveru, ale tato možnost zcela nepodporuje vysokou dostupnost. V okamžiku, kdy padne jediná instance, dojde také ke ztrátě jakéhokoli přístupu k databázi,

2. 2. pokročilé nasazení – pomocí PostgreSQL clusteru. V tomto případě PostgreSQL cluster bude mít architekturu master-slave. Jedna hlavní databáze, která umožňuje jak zapisovat nová data, tak i číst. Mnoho replik, které replikují data z masteru a používají se pouze pro čtení. Tento typ nasazení zahrnuje



Obrázek 6.4: Patroni cluster spravující dvě vysoce dostupné instance PostgreSQL [39]

poměrně složitou kombinaci problémů, jež je potřeba vyřešit: vyvažování zatížení, synchronizace mezi instancemi, automatické zálohování, obnovení ze zálohy, monitorování a tak dále. Složité nastavení takových clusterů může být automatizováno operátory, o kterých již dříve byla zmínka. Na webu oficiální dokumentace je možné najít návod k rozhození clusteru (`https://kubernetes.io/blog/2017/02/postgresql-clusters-kubernetes-statefulsets/`).

Příkladem architektury clusteru PostgreSQL je obrázek 6.4, který zobrazuje organizaci clusteru vytvořeného pomocí Patroni[9].

Vysoká dostupnost databází v Kubernetes by měla být vždy pečlivě zvážena, stejně jako samotný provoz databází v clusteru.

Projekt Smartform používá dva databázové servery PostgreSQL. Do jednoho serveru se ukládají všechny informace o uživatelích a je často používána k zápisu. V tomto případě nelze jednoduše zvýšit počet instancí, protože je nezbytné postarat o replikaci a konzistenci dat. Nasazení clusteru PostgreSQL může být komplikovaná a časově náročná práce, takže spolu se zadavatelem bylo rozhodnuto, že to nebude součástí této bakalářské práce. Nasazení tohoto serveru bude provedeno podle první varianty, to znamená, že v clusteru bude přítomna pouze jedna instance.

Druhý server představuje úložiště českého a slovenského registru územní identifikace, adres a nemovitostí a je read-only. Tento PostgreSQL server se nepoužívá k zápisu, její obsah se také často nemění, takže replikace dat je nadbytečná. Vzhledem k tomu, že tento server je většinou jen ke čtení, je možné snadno zvýšit počet instancí bez použití PostgreSQL clusteru. Tímto způsobem bude nasazení serveru horizontálně škálováno s více instancemi, čímž se zvýší dostupnost této části systému. Pro rovnoměrné přesměrování provozu bude použit load balancer, který bude

[9]Dostupné z: `https://patroni.readthedocs.io/en/latest/` (06.07.2022)

rovnoměrně přesměrovávat požadavky mezi instancemi. Také v konfiguraci tohoto serveru bude přidána vlastnost, díky níž budou instance rovnoměrně rozmístěny na uzlech. Na každý pracovní uzel bude připojena jedna instance, která bude fungovat nezávisle na jiných instancích. Tím získáme dvě repliky serveru na dvou uzlech. Tato varianta je optimální a nabízí dostatečnou úroveň přístupnosti vzhledem k tomu, že toleruje nejen selhání jedné z replik, ale i pád celého uzlu.

Kapitola 7

Nasazení clusteru na serveru

V předchozí kapitole jsme popsali teoretický základ pro instalaci clusterů, zkoumali různé modely clusterů a možné varianty jejich instalace. Jedním z cílů a záměrů této práce je zajistit odolnost nejen aplikací proti chybám a výpadkům, ale také nakonfigurovat samotný cluster tak, aby toleroval chyby. Pomocí modelu nasazení „multi-master“ je možné tento cíl realizovat.

Tato část popisuje všechny kroky potřebné k nasazení clusterů multi-master modelu pomocí nástroje pro automatizaci a konfiguraci Kubeadm: jak nainstalovat container runtime a ostatní nutné nástroje a pluginy, jak probíhá inicializace komponent masteru a připojení ostatních uzlů a jakým způsobem získat přístup do privátního Docker registru.

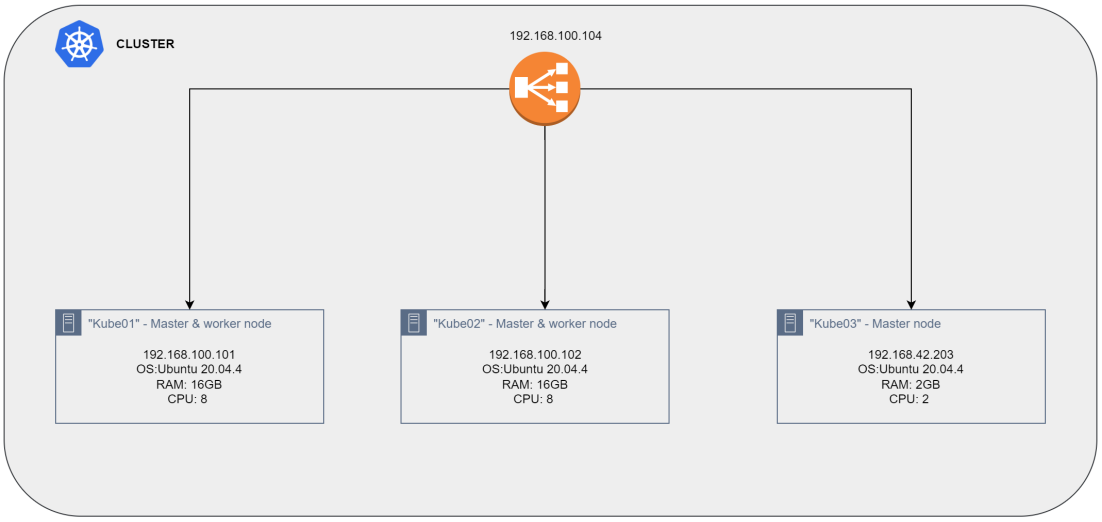
7.1 Technická specifikace serverů

V předchozích kapitolách jsme uváděli, že v současné době je aplikace nasazena na jednom fyzickém serveru. Pro dosažení vysoké dostupnosti bylo rozhodnuto o přidání dalších serverů. Pro nasazení a testování byly vytvořeny celkem tři virtuální servery s operačním systémem Ubuntu 20.04.4, viz obrázek 7.1. Tyto tři servery budou spojeny do jednoho multi-master clusteru. Jeden ze serverů bude mít pouze roli master, další dva budou kombinovat role master a worker uzlů.

Dva virtuální stroje, které budou použity jako pracovní a master uzly, mají 16 GB operační paměti, 8 CPU. Třetí server, jehož úkolem bude v clusteru fungovat pouze jako master uzel, je co do technických vlastností horší než první dva, má pouze 2GB paměť a 2 CPU. Každý uzel je spojen do veřejné (192.168.42.x) a privátní sítě (192.168.100.x). Privátní síť slouží ke komunikaci mezi nody, u níž je nežádoucí, aby probíhala na vnější síti, kde je komunikace viditelná pro ostatní. Každý ze serveru splňuje minimálně požadavky na instalaci Kubernetes clusteru přes Kubeadm, více informací lze najít na oficiální stránce Kubernetes na odkazu (`https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/install-kubeadm/`).

7.2 Konfigurace uzlů

Před propojením uzlů do clusteru je nutné je připravit a nainstalovat všechny nástroje, zejména v této části instalujeme kontejnerový runtime, stejně jako nástroje Kubernetes, jako jsou Kubeadm, Kubelet a Kubectl.



Obrázek 7.1: Kubernetes cluster

7.2.1 Instalace container runtime

Pro běh podů v kontejnerech používá Kubernetes kontejnerový runtime. Ve výchozím nastavení používá Kubernetes Container Runtime Interface (CRI) pro interakci se zvoleným runtime kontejnerem. Pokud není nespecifikován runtime, kubeadm se automaticky pokusí detekovat nainstalovaný runtime kontejner prohledáním seznamu známých endpointů. Pokud je detekováno více runtime kontejnerů nebo žádný, kubeadm vyhodí chybu a požádá uživatele o specifikování runtimu [40].

V této práci jsme se rozhodli pro Docker jako kontejnerizované prostředí, více informací o dalších možných prostředích je možné najít na oficiálních stránkách Kubernetes (`https://kubernetes.io/docs/setup/production-environment/container-runtimes/`).

Aby bylo možné Docker runtime používat, je potřeba na každém uzlu mít:

1. naintalovany Docker Engine[1].

2. adaptér pro ovládání Docker Engine přes Kubernetes. Komponenta dockershim Kubernetes, která umožňovala použít Docker jako kontejner runtime pro Kubernetes, je od 1.20 a později nedostupná. Docker Engine neimplementuje CRI, takže je potřeba ručně nainstalovat službu cri-dockerd. Příkazy pro instalaci cri-dockerd jsou dostupné v příloze A.

7.2.2 Instalace Kubeadm, Kubelet a Kubectl

Nakonec instalujeme Kubernetes na všechny stroje. To zahrnuje instalaci tří různých nástrojů, včetně Kubeadm, Kubelet a Kubectl. Kubectl je nástroj příkazového řádku pro správu klastrů Kubernetes. Kubelet je agent pracující na každém uzlu v clusteru, který sleduje provoz kontejnerů v podech. Kubeadm je nástroj pro automatizaci instalace a konfigurace komponentů clusteru. Příkazy pro instalaci jsou dostupné v příloze B.

[1]Dostupné z: `https://docs.docker.com/engine/install/#server` (13.06.2022)

7.2.3 Inicializace control-plane komponent

Po provedení výše uvedených kroků na všech uzlech můžeme inicializovat master uzel. Inicializace clusteru na master uzlu se provádí příkazem „kubeadm init“:

```
$ kubeadm init
—control-plane-endpoint="192.168.100.104:6443"
—upload-certs
—pod-network-cidr=10.244.0.0/16
—cri-socket=unix:///var/run/cri-dockerd.sock
```

Seznam parametrů, jejichž pomocí lze customizovat cluster:

1. –control-plane-endpoint. Tento příznak by měl být nastaven na adresu nebo DNS load-balanceru,
2. –pod-network-cidr. Určuje rozsah IP adres pro síť podů, viz sekci 7.2.4.
3. –cri-socket. Pokud je v uzlu nainstalováno více než jeden container runtime, je potřeba určit jeden konkrétní. Pro cri-dockerd, který jsme nainstalovali, specifikujeme –cri-socket=unix:///var/run/cri-dockerd.sock,
4. –apiserver-advertise-address. IP adresa, na které poslouchá API Server. Tento parametr není nutný pro nasazení multi-master modelu.

Tento krok se provádí jen na jednom master uzlu. Tím se vytvoří všechny komponenty hlavního uzlu, popsané v kapitole 4.3.2, včetně etcd, kube-apiserveru, kube-controller-manageru, kube-proxy a kube-scheduleru.

Pokud je vše provedeno správně, zobrazí se na obrazovce text „Your Kubernetes control-plane has initialized successfully!“. Poté budou vypsány příkazy, které je třeba provést pro použití clusteru a nastavení kubectl na master uzlu: :

```
To start using your cluster, you need to run the following as a regular user:

  mkdir -p $HOME/.kube
  sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
  sudo chown $(id -u):$(id -g) $HOME/.kube/config

Alternatively, if you are the root user, you can run:

  export KUBECONFIG=/etc/kubernetes/admin.conf
```

Ve stejné zprávě o úspěšné inicializaci bude příkaz, který umožňuje připojit zbývající uzly clusteru k řídicímu uzlu:

```
You can now join any number of the control-plane node running
the following command on each as root:

  kubeadm join 192.168.100.101:6443 --token 3uxmrb.nz6xeb6tym5qvu40 \
--discovery-token-ca-cert-hash sha256:371ee76fbefre4df0fcf95d
4452a6e9513e8d21d44c12c5c26e5deb1abcd12 \
--control-plane --certificate-key d0cd35dab3a096aewe563f2cbbccd1cac0
38ac830827d674fd0bdcc2
```

```
Please note that the certificate-key gives access to cluster sensitive
data, keep it secret! As a safeguard, uploaded-certs will be
deleted in two hours; If necessary, you can use
"kubeadm init phase upload-certs --upload-certs" to reload
certs afterward.

Then you can join any number of worker nodes by running the
following on each as root:

kubeadm join 192.168.100.101:6443 --token 3uxmrb.nz6xeb6tym5qvu40 \
        --discovery-token-ca-cert-hash sha256:371ee76fbefre4df0fcf95d
4452a6e9513e8d21d44c12c5c26e5deb1abcd12
```

7.2.4 Instalace pluginu CNI

V kapitole jsme uváděli různé síťové modely používané v Kubernetes. Pro správné fungování „Pod-to-Pod“ komunikace je nutné naistalovat síťový plugin[2]. Při použití CNI může hladce integrovat s Kubernetes a umožnit použití překryvné nebo podkládací sítě pro automatickou konfiguraci sítě mezi pody. V teto bakalarske praci budu pouzivat Flannel plugin na základě svých předchozích zkušeností.

Flannel vytváří virtuální síť, která běží přes hostitelskou síť a nazývá se překryvná (overlay) síť. Z této sítě se přiřazují jedinečné IP adresy podům. Na každém hostitelském uzlu běží agent „flanned“, který přiděluje každému uzlu jedinečnou podsíť IP adres z většího adresního prostoru. Poté všechny pody v uzlu používají IP z tohoto rozsahu. Flannel má výchozí rozsah podsítě 10.244.0.0/16, který musí být uveden při inicializaci clusteru parametrem „pod-network-cidr“.

Instalace Flannelu probíhá pomocí yml souboru z git repozitáře [41]: [41]

```
$ kubectl apply -f https://raw.githubusercontent.com/flannel-io/
flannel/master/Documentation/kube-flannel.yml
```

Tento příkaz vytvoří několik různých objektů, které umožní spuštění Flannelu. Pokud chceme použít vlastní rozsah IP adresy, tedy odlišný od 10.244.0.0 / 16, musíme nejprve stáhnout výše uvedený manifest a upravit konfiguraci.

Pro ověření, že pody Flannel jsou v provozu, je zapotřebí spustit následující příkaz:

```
$ kubectl get pods - n kube-flannel
```

Pod s názvem kube-flanel-ds-* musí být ve stavu Ready. To tedy potvrzuje, že naše clusterová síť je správně nakonfigurována.

7.2.5 Připojení zbývajících uzlů

Poté, co na každém zbývajícím uzlu byly instalovány container runtime, kubeadm, kubelet a kubectl, je možné začít přidávat tyto uzly do clusteru. Připojení jak pracovních, tak master uzlů je automatizováno v kubeadm a provádí se jedním

[2]Dostupné z: `https://kubernetes.io/docs/concepts/extend-kubernetes/compute-storage-net/network-plugins/` (01.08.2022)

příkazem „kubeadm join“, který poskytuje všechny potřebné informace včetně adresy IP, portu serveru API hlavního uzlu a zabezpečeného tokenu. Tento příkaz má následující formát pro připojení pracovního uzlu:

```
# Worker join
$ kubeadm join --token <token> <master-ip>:<master-port>
--discovery-token-ca-cert-hash sha256:<hash>
```

Pro připojení master uzlů je potřeba poskytnout několik dalších parametrů:

```
# Master join
$ kubeadm join --token <token> <master-ip>:<master-port>
--discovery-token-ca-cert-hash sha256:<hash> --control-plane
--certificate-key <key>
```

Tento příkaz pro připojení zbývajících uzlů je možné nalézt v konzole pro úspěšném provedení příkazu „kubeadm init“ nebo může být zreprodukován pomocí „kubeadm token create –print-join-command“.

Ve výchozím nastavení se cluster skládá z masterů a workerů, které jsou různými fyzickými nebo virtuálními uzly. Jsou situace, kde infrastruktura potřebuje multi-master nastavení clusteru pro vysokou dostupnost, ale zároveň je nedostatek vybavení pro takovou implementaci. Jak již bylo zmíněno, pro nasazení infrastruktury budou použity pouze tři virtuální servery, dva ze tří uzlů budou také sloužit jako workery.

Cluster Kubernetes nebude rozmísťovat pody na master uzly, protože master uzly mají specifický label, který se v terminologii Kubernetes nazývá taint[3]. Proto je nutné ručně povolit clusteru využívat master uzly jako workery a odstranit taint z požadovaných uzlů. Koncept taints pomáhá dosáhnout selektivního plánování tím, že zabraňuje plánování podů do nežádoucích uzlů, jako je master.

Spuštěním tohoto příkazu se odstraní taint z master uzlu:

```
$ kubectl taint nodes <node-name> node-role.kubernetes.io/master-
$ kubectl taint nodes <node-name> node-role.kubernetes.io/control-plane:
NoSchedule-
```

7.2.6 Ověření stavu clusteru

Pro dokončení inicializace je doporučeno zkontrolovat stav uzlů. Nástroje kubectl můžeme používat nejen pro nasazení aplikací, ale i pro kontrolu zdrojů a stavu clusteru.

Níže budou uvedeny nejčastěji používané příkazy pro kontrolu, další informace, včetně kompletního seznamu operací kubectl, je možné najít v dokumentaci kubectl (`https://kubernetes.io/docs/reference/kubectl/`).

Příkaz „kubectl get nodes“poskytne informace o uzlech a jejich stavech. Více informací o každém uzlu je dostupných prostřednictvím příkazu „kubectl describe nodes <node-name>“.

Spuštěním příkazu „kubectl cluster-info“ můžeme zjistit informace o clusteru Kubernetes, například na jaké adrese cluster běží, nebo odkud běží Core DNS. Příklad spuštění příkazu:

[3]Dostupné z: `https://kubernetes.io/docs/concepts/scheduling-eviction/taint-and-toleration/` (01.08.2022)

```
$ kubectl cluster-info
Kubernetes control plane is running at https://192.168.100.104:6443
CoreDNS is running at https://192.168.100.104:6443/api/v1/namespaces/
kube-system/services/kube-dns:dns/proxy
```

7.2.7 Připojení k soukromému Docker registru

V kapitole 6.3 jsme zmiňovali, že používání veřejných Docker registrů může být nevhodnou a nezabezpečenou variantou, proto má systém Smartform vlastní registr. Tento registr je self-hosted s využitím certifikátu podepsaného vlastnoručně a je chráněn přístupem pomocí přihlašovacích údajů. Dále bude uveden návod, jak lze nastavit přístup ke clusteru pro tento typ registru.

Krok 1. Certifikát

Protože používáme certifikáty podepsané vlastnoručně, musíme nastavit Docker Daemon tak, aby věřil takovému certifikátu. Tento krok není povinný, pokud byl certifikát vydán certifikační autoritou – CA. Pro nastavení máme zkopírovat certifikát domain.crt do /etc/docker/certs.d/<REGISTRY-DOMAIN>:<PORT>/ca.crt na každém hostiteli. Způsob, jak to provést, závisí na typu OS. Zkoušíme na Linux serverech, proto tento návod platí jen pro Linux OS.

Krok 2. Login

Login do registru se provádí pomocí příkazu[4]:

```
$ docker login -u <username> -p <password> <IP ADDRESS>:5000
```

Místo IP adresy lze použít doménu registru.

Krok 3. Vytvoření Secret na základě existujícího pověření Cluster Kubernetes používá Secret typu kubernetes.io/dockerconfigjson k autentizaci registru k načtení soukromé image. Existuje příkaz kubectl pro vytvoření Secretu Docker-registry, který lze použít pro stahování imagů ze soukromých registrů. Pokud jsme již spustili „docker login“, můžeme toto pověření zkopírovat do Kubernetes pomocí [27]:

```
$ kubectl create secret generic regcred \
  --from-file=.dockerconfigjson=<path/to/.docker/config.json> \
  --type=kubernetes.io/dockerconfigjson
```

nebo je možné ručně předat přihlašovací údaje:

```
$ kubectl create secret docker-registry regcred \
  --docker-server=<your-registry-server> \
  --docker-username=<your-name> \
  --docker-password=<your-pword> \
  --docker-email=<your-email>
```

V další kapitole ukážeme, jak vytvořit pod, který používá tento Secret ke stažení soukromé image.

Nyní, když máme náš cluster připraven, všechny uzly jsou v provozu a mají přístup k Docker registru, je čas nasadit systém na Kubernetes cluster.

[4]Dostupné z: https://docs.docker.com/engine/reference/commandline/login/ (24.06.2022)

Kapitola 8

Nasazení částí aplikací v clusteru

Poté, co je nastaven HA cluster Kubernetes, je možné začít postupně nasazovat každou část systému. Nejmenší a základní části Kubernetes je pod, ale nedoporučuje se definovat každý pod samostatně, místo toho budeme definovat vhodné abstrakce vyšší vrstvy – StatefulSet, Deployment a ReplicaSet. V kapitole 4.3.2 jsme se již s těmito pojmy setkali, tato sekce bude věnována konfiguraci těchto zdrojů pomocí YAML souborů.

Aby aplikace v podech fungovaly, je nutné jim poskytnout potřebnou konfiguraci a úložiště, proto bude také popsáno, jak pomocí objektů Volumes a ConfigMap dosáhnout funkčnosti aplikace.

Dále musíme definovat každou komponentu našeho systému jako službu. Tímto způsobem bude Kubernetes automaticky implementovat service discovery, které je vyžadováno v mikroservisní architektuře našeho systému. Navíc prostřednictvím vystavení služeb vnějšímu světu umožníme koncovým uživatelům přístup k příslušným částem systému.

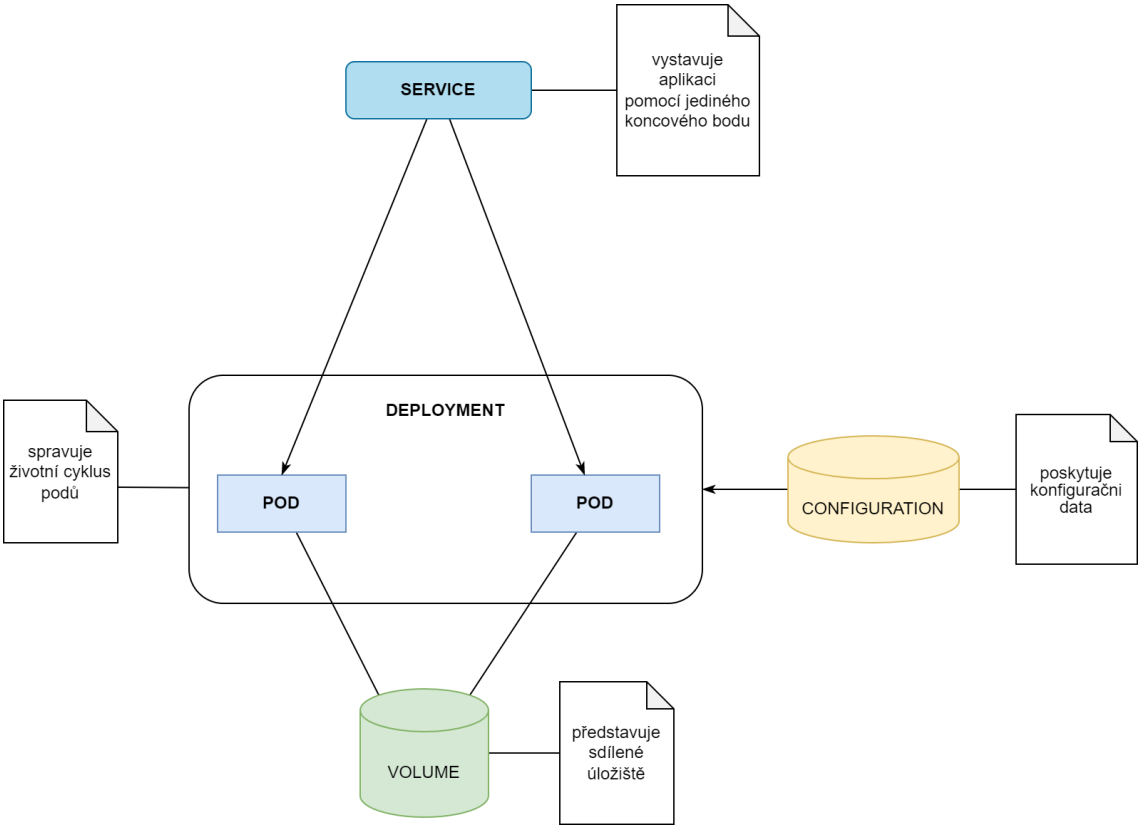
Na konci kapitoly se podíváme, jak Kubernetes pomohl zvýšit odolnost proti poruchám a eliminovat nebo minimalizovat problémy popsané v kapitole 5.

8.1 Kubernetes objekty

Kubernetes objekty jsou Kubernetes entity poskytované pro nasazení, údržbu a škálování. Pro vytvoření objektů v Kubernetes je zapotřebí poskytnout definici, která popisuje jeho požadovaný stav, a také některé základní informace o objektu (například název). Tato definice se následně odesílá do Kubernetes API, který se stará o spuštění. Je možné přímo odeslat požadavek do rozhraní ve formátu JSON, ale více se doporučuje možnost prostřednictvím kubectl v .yaml souboru. Kubectl převádí informace do JSON formátu při vytváření požadavků.

Všechny objekty lze rozdělit do čtyř kategorií:

- **Workload resources.** Jedná se o zdroje Kubernetes, jako jsou Deployment nebo ReplicaSet, které vytvářejí a spravují pody,
- **Service a networking resources.** Mezi tyto objekty patří Service a Ingress, které vystavují aplikace a různými způsoby spravují síťový provoz v clusteru Kubernetes.
- **Volumes.** Používá se k zajištění dlouhodobého i dočasného úložiště v clusteru,



Obrázek 8.1: Příklad interakce objektů Kubernetes

- **Configuration.** Objekty umožňující předávat konfigurační data do aplikace během spuštění.

Na obrázku 8.1 je znázorněno, jak objekty z výše uvedených kategorií vzájemně interagují v clusteru.

Vytvořit úložiště a konfigurační data, nasadit aplikaci a dát jí možnost komunikovat – to je algoritmus, který se ve většině případů používá ke spuštění aplikace v clusteru. Po provedení tohoto algoritmu pro každou aplikaci můžeme spustit celý systém. Proto v následujících částech podrobně popíšeme, jak lze krok za krokem při vytváření potřebných objektů Kubernetes tento algoritmus implementovat.

8.2 Nasazení aplikací pomocí objektu Deployment

Pro nasazení aplikace jsou použity objekty vyšších abstrakcí z kategorie „workload resourses“. Použití objektu Deployment zajišťuje, že požadovaný počet replik podu bude vždy fungovat a poskytuje jednoduché horizontální škálování podů. Pro ilustraci konfigurace objektu Deployment jsme se rozhodli pro Grails aplikace, pracující v základním obrazu pro Java 8, má přístup k databázi a v rámci clusteru bude mít dvě repliky.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: admin
  namespace: dev
```

```
spec:
  replicas: 2
  selector:
    matchLabels:
      app: admin
  strategy:
    rollingUpdate:
      maxSurge: 1
      maxUnavailable: 1
    type: RollingUpdate
  template:
    metadata:
      labels:
        app: admin
    spec:
      imagePullSecrets:
       - name: regcred
      containers:
        - image: 84.242.100.238:5000/java8_custom:0.1
          name: admin
          workingDir: /opt/admin/lib
          command: [ "java" ]
          args:
            - java
            - Xms500M -Xmx500M
            - Dgrails.env=docker_tester
            - Dfile.encoding=UTF-8
            - XX:+UseG1GC
            - XX:MaxGCPauseMillis=100
            - XX:+PrintGCDateStamps -verbose:gc
            - XX:+PrintGCDetails
            - Xloggc:/opt/admin/logs/gc-%t.log
            - XX:+UseGCLogFileRotation
            - XX:NumberOfGCLogFiles=10
            - XX:GCLogFileSize=50M
            - XX:+HeapDumpOnOutOfMemoryError
            - jar admin.war
          envFrom:
            - configMapRef:
                name: admin-enviromental-variables
          ports:
            - containerPort: 80
            - containerPort: 443
          livenessProbe:
            httpGet:
              path: /
              port: 8080
            initialDelaySeconds: 60
            periodSeconds: 60
```

```
        affinity:
          podAntiAffinity:
            prefferedDuringSchedulingIgnoredDuringExecution:
              - labelSelector:
                    matchExpressions:
                      - key: "app"
                        operator: In
                        values:
                          - admin
        resources:
          requests:
            cpu: 10m
            memory: 500Mi
          limits:
            cpu: 100m
            memory: 700Mi
        volumeMounts:
          - mountPath: /opt/admin/lib
            name: data
          - mountPath: /opt/admin/logs
            name: logs
      volumes:
        - name: data
          hostPath:
            # directory location on host
            path: /var/docker/data/admin
            # this field is optional
            type: Directory
        - name: logs
          hostPath:
            # directory location on host
            path: /var/docker/logs/admin
            # this field is optional
            type: Directory
...
```

Fragment kódu 8.1: Příklad konfigurace objektu Deployment

Každá konfigurace objektu obsahuje čtyři povinná pole: ApiVersion, Kind, Metadata a Specifications. Podrobnosti o každém poli a jeho vlastnostech budou popsány níže.

8.2.1 ApiVersion

Definuje použitou verzi Kubernetes objektu [42]. Existuje několik verzí a s každou z nich je zavedeno několik objektů. Při aktualizaci verzí Kubernetes je vytvořena nová apiVersion. Doporučená verze závisí na typu objektu[1].

[1]Dostupné z: `https://matthewpalmer.net/kubernetes-app-developer/articles/kubernetes-apiversion-definition-guide.html` (28.06.2022)

8.2.2 Kind

„Kind“ uvádí typ objektu, který chceme vytvořit – Ingress, ReplicaSets, CronJobs, StatefulSet atd. Na obrázku 8.1 „Kind: Deployment“ představuje objekt Kubernetes Deployment. Příkaz „kubectl api-resources“ lze použít k zobrazení dostupných typů zdrojů.

8.2.3 Metadata

Pro jednoznačnou identifikaci objektů se používají metadata. Povinně musí obsahovat název (name), UID a namespace [42]. Pokud nebude uvedeno UID a namespace, Kubernetes to provede sám. Takže metadata mohou obsahovat anotaci[2] a labely[3]. V příkladu 8.1 v poli metadat jsme poskytli informace o názvu (name : admin) a namespace, ve kterém bude objekt (namespace : dev).

8.2.4 Deployment Specifications

Obsahuje podrobnosti o požadovaném stavu objektu. Přesný formát specifikace objektu je pro každý objekt Kubernetes jiný a obsahuje vnořená pole specifická pro daný objekt. Níže uvedená specifikace polí platí pro Deployment, dokumentace (`https://kubernetes.io/docs/reference/kubernetes-api/`) může pomoci najít formát pro všechny ostatní objekty.

Replicas
Počet požadovaných replik podu. Výchozí hodnota je 1. Pro zvýšení úrovně dostupnosti jsme tento parametr nastavili na dvě repliky. Load balancer bude regulovat provoz mezi těmito instancemi.

Selector
Pole „selector“ definuje, které pody musí spravovat Deployment. Musí odpovídat štítku (spec.template.metadata.labels) šablony podu. Dále bude pod označen labelem „app: admin“, proto v matchLabels používáme stejnou značku.

Strategy
Strategie nasazení je způsob, jak při změně nebo upgradování nahradit existující pody novými. **Strategy.type** definuje typ nasazení. Může být „Recreate“ nebo „RollingUpdate“. Defaultní hodnota je „RollingUpdate“. Pokud jsme zvolili typ „RollingUpdate“, pak je třeba zadat další atributy vnořené vlastnosti rollingUpdate, jak je uvedeno níže:

```
strategy:
  rollingUpdate:
```

[2]Dostupné z: `https://kubernetes.io/docs/concepts/overview/working-with-objects/annotations/` (29.06.2022)

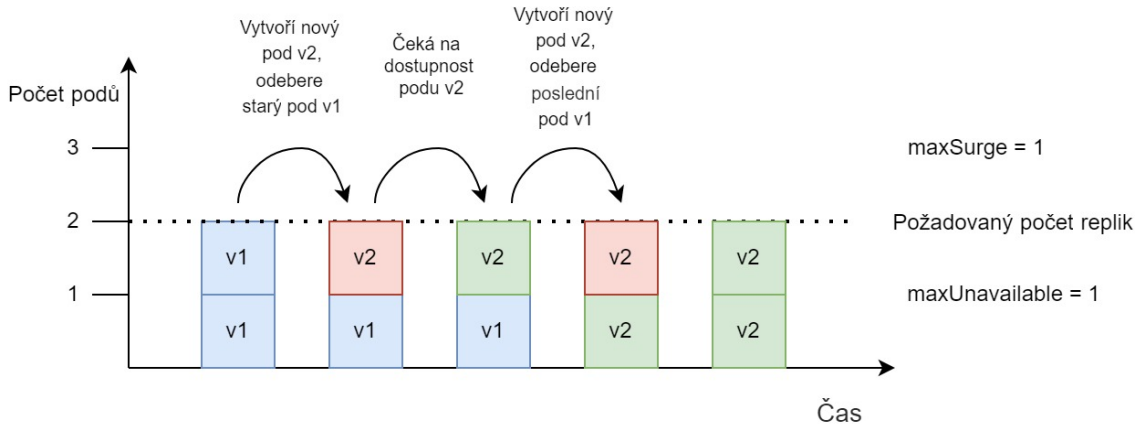
[3]Dostupné z: `https://kubernetes.io/docs/concepts/overview/working-with-objects/labels/` (29.06.2022)

```
    maxSurge: 1
    maxUnavailable: 1
  type: RollingUpdate
```

MaxSurge (strategy.rollingUpdate.maxSurge) definuje maximální počet podů, jež lze naplánovat nad požadovaný počet, který je uveden ve vlastnosti „replicas“. Hodnota může být absolutní číslo (např. 1) nebo procento požadovaných podů (např.: 10 %).

MaxUnavailable (strategy.rollingUpdate.maxUnavailable) definuje maximální počet podů, které mohou být během aktualizace nedostupné. Hodnota může být absolutní číslo (např. 1) nebo procento požadovaných podů (např. 10 %). Nemůže být 0, pokud je hodnota MaxSurge 0.

Strategie RollingUpdate umožňuje zajistit trvalou dostupnost této části systému i při upgradu nebo jakékoli změně konfigurace. Protože požadovaný počet replik ve našem případě činí dvě a obě tyto vlastnosti jsme nastavili na 1, vlastnost maxSurge umožnila dosáhnout počtu tří podůa vlastnost maxUnavailable povolila mít maximálně jeden nedostupný pod, jinými slovy jeden pod měl být kdykoli k dispozici. Obrázek 8.2 znázorňuje, jak bude Kubernetes odebírat a přidávat nové pody při aktualizaci z verze v1 na v2.



Obrázek 8.2: Aktualizace podů

Existuje několik dalších vlastností objektu Deployment, jež jsme nepoužili, např. minReadySeconds a revisionHistoryLimit, které si lze přečíst v dokumentaci. Většinu konfigurace tvoří popis podů ve vlastnosti **template**, kterými se budeme podrobněji zabývat v další kapitole 8.2.5.

8.2.5 Pod Template Specifications

Pod Template je konfigurace pro vytváření podů a je součástí definice workload resources. Každý objekt z kategorií „workload resources“ používá Pod Template uvnitř objektu k vytváření skutečných podů. Stejně jako u každého objektu jsou povinnými poli apiVersion, kind a metadata, která byla již zmíněna v předchozí kapitole. Dále se zaměříme právě na vlastnost textbfSpecifications (spec).

spec.imagePullSecrets V kapitole 7.2.7 jsme ukazovali způsob připojení k soukromému Docker registru. V kroku č. 3 byl vytvořen objekt typu Secret s názvem „regcred“. Pro stahování obrazu z tohoto soukromého registru použijeme tento

Secret ve vlastnosti imagePullSecrets, které určuje, kde by Kubernetes měl získat pověření. Pak je ve specifikaci image kontejneru pomocí DNS jména nebo IP adresy potřeba specifikovat registr, ze kte- rého Kubernetes bude stahovat image (např. image: 84.242.100.238:5000/java8_custom:0.1).

spec.volumes Seznam úložišť, ke kterým má pod přístup. Volume lze považovat za adresář, který je přístupný kontejnerům v podu. V definici Deploymentu 8.1 jsou definovány dva volumes s názvem data a logs typu hostPath. Tento typ připojí soubor nebo adresář ze souborového systému uzlů do podu. Kompletní seznam všech dostupných typů úložišť je uveden v dokumentaci Kubernetes[4].

spec.containters
Seznam kontejnerů patřících do podu. Musí být alespoň jeden kontejner. **Vlastnosti spec.containters**:

- name. Název kontejneru - name : admin
- image. Název obrázu kontejneru - image : 84.242.100.238:5000/java8_custom:0.1. Protože image je ze soukromého registru, před názvem obrazu upřesníme, z jakého registru je potřeba vzít image,
- workingDir. Pracovní adresář kontejneru - workingDir : /opt/admin/lib
- command a args. Definuje příkaz a argumenty pro kontejnery, které běží v podu,
- envFrom. Proměnné prostředí je možné předat dvěma způsoby : ve vlastnosti **env** vypsat každou proměnnou s použitím párů name a value, nebo odkaz na objekt typu ConfigMap ve vlastnosti envFrom. Více o definování ConfigMap v kapitole 8.3.
- ports. Seznam portů, které se mají z kontejnerů vystavit. Vystavení portů zde poskytuje systému dodatečné informace o síťových spojeních, které kontejner používá, ale je primárně informativní. Neurčení portů zde nepřekáží tomu, aby byl tento port vystaven [27]
- livenessProbe. Z parametrů pro řízení životního cyklu podu existují také **readinessProbe** a **startupProbe**.[43] Kubernetes automaticky znovu vytvoří pody, když dojde k selhání hlavního procesu kontejnerů. Existují však situace, kdy aplikace uvnitř kontejneru nefunguje, ale nevede k pádu kontejneru. Například Java aplikace může způsobit chybu kvůli nedostatku paměti, to ale nezastaví samotný proces JVM. Proto se v Kubernetes uskutečňuje kontrola aplikace v kontejneru třemi typy: kontrola přes **HTTP GET**, kontrola **TCP socketu** a kontrola **Exec**, jež spustí libovolný příkaz uvnitř kontejneru.

  Pro otestování zdraví aplikace v kontejneru jsme nastavili liveness probe na kontrolu přes HTTP v intervalech každé minuty. Kontejner bude restartován, pokud kontrola selže. Ve výchozím nastavení Kubernetes počká na tři neúspěšné pokusy před restartováním, počet pokusů je možné nastavit přes pole **failureThreshold**,

[4]Dostupné z: `https://kubernetes.io/docs/concepts/storage/` (01.07.2022)

- affinity. Kubernetes umožňuje ovlivnit, kam pody budou přiřazeny [44]. Rozlišují se tři typy affinity: **Node Affinity** – pravidla plánování podů na základě popisků (labels) na uzlech; **Pod affinity / anti-affinity** – pravidla plánování na základě popisků (labels) na jiných podech.

  V definici 8.1 jsme použili pod anti-affinity, což zajistí, že dva pody budou pracovat na různých nodech, při výpadku jednoho nodu pod na druhém bude dostupný. Ale není to striktní pravidlo: **„preferredDuringSchedulingIgnoredDuringExecution“** znamená, že plánovač se pokusí najít uzel, který splňuje toto pravidlo. Není-li odpovídající uzel k dispozici, plánovač stále plánuje pod [44],

- resources. Výpočetní požadavky (cpu, memory). **resources.requests** je minimální požadované množství výpočetních zdrojů, **resources.limits** je maximální povolené množství výpočetních zdrojů,

- volumeMounts. Pole, které určuje, že Volume s názvem **volumeMounts.name** je připojen k adresáři **volumeMounts.mountPath** uvnitř kontejneru. Pro každý kontejner tady uživatel definuje dostupný volume ze specifikace **spec.volumes**. Například v definici Deploymentu 8.1 do adresáře /opt/admin/lib a /opt/admin/logs jsou předána dvě úložiště s názvy data a logs.

8.3 Poskytování dat pomocí objektu ConfigMap

Konfigurační mapa nebo ConfigMap je objekt Kubernetes sloužící k ukládání dat ve dvojicích klíčů a hodnot. Pody mohou používat konfigurační mapy jako proměnné prostředí, argumenty příkazového řádku nebo jako konfigurační soubory ve volumes [45].

Konfigurační mapa byla použita pro předání proměnných prostředí v definici Deployment 8.2 ve vlastnosti **template.spec.envFrom.configMapRef**. Následující yaml konfigurace mapy je zkrácená verze. Produkční verze všech konfiguračních souborů je možné najít ve zdrojovém kódu k této bakalářské práci

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: admin-enviromental-variables
  namespace: dev
data:
  TZ: "Europe/Prague"
  dataSource.url: "jdbc:postgresql://postgres_smartform/smartform"
  dataSource.user: "${SMARTFORM_USER_PASSWORD}"
  dataSource.password: "${SMARTFORM_DB_PASSWORD}"
...
```

Fragment kódu 8.2: Příklad konfigurace objektu ConfigMap

ApiVersion, kind a metadata jsou stejná povinná políčka jako v definici Deployment, ale místo pole „spec“ konfigurační mapy používají pole „data“. Toto pole je seznam dvojic „klic-hodnota“, tzn. uvnitř kontejneru bude proměnná s názvem „TZ“ a hodnotou „Europe/Prague“.

8.4 Nasazení objektu do clusteru

Vytváření objektů přes kubectl probíhá dvěma způsoby: pomocí příkazů „kubectl apply“ a „kubectl create“. Rozdíl mezi **apply** a **create** spočívá v tom, že **apply** vytváří objekty Kubernetes prostřednictvím deklarativní syntaxe – s použitím souboru manifestu, zatímco příkaz **create** je imperativní na příkazovém řádku.

Jakmile jsou manifesty nasazení Deployment a ConfigMap napsány, můžeme spustit příkaz, který vytvoří Kubernetes objekty:

```
$ kubectl apply (-f FILENAME | -k DIRECTORY)
```

Poté je možné zkontrolovat stav vytvořeného objektu. Pro zjištění stavu a informací objektů se používá operace „kubectl get“. Například následujícím příkazem je možné získat informaci o Deploymentu:

```
$ kubectl get deployment
NAME      READY     STATUS    RESTARTS   AGE
admin     1/1       Running   0          7m
```

Výsledkem je, že Deployment funguje, protože stav je Running, počet objektů je jeden a byl vytvořen před sedmi minutami. Objekt může být také ve stavu Pending. Nejčastěji je to poprvé po vytvoření, kdy Kubernetes vytváří potřebné pody, ale může doiít k chybě, kvůli níž se objekt nikdy nedostane do stavu Running. K získání podrobnějších informací ohledně objektů lze použít **kubectl describe**.

8.5 Konfigurace PostgreSQL serveru

V kapitole 6.4 jsme popsali, jak plánujeme nasadit dva PostgreSQL servery dvěma různými způsoby: pomocí jedné instance a škálováním instance na uzly. V této kapitole chceme popsat, jak se doporučuje nasadit databáze v clusteru Kubernetes.

Pro nasazení stavových aplikací v Kubernetes existuje předpřipravené řešení – objekt StatefulSet. K hlavním rysům objektu StatefulSet patří:

1. pody vytvořené StatefulSet nejsou přesné repliky. Každý může mít své úložiště (a tedy konstantní stav), který ho odlišuje od jeho sousedů,

2. názvy podů jsou předvídatelné. Každý pod vytvořený sadou StatefulSet má přiřazen pořadový index. Název podu se skládá z názvu sady a tohoto pořadového indexu. Například pokud jsme pojmenovali StatefulSet „web“, pak první pod bude mít název „web-0“a tak dále,

3. 3. na rozdíl od podu v Deploymentu, kde všechny repliky mají jedno sdílené úložiště, každá replika v StatefulSetu bude mít svůj vlastní stav a každý z podů si vytvoří vlastní PVC (Persistent Volume Claim). StatefulSet se třemi replikami vytvoří tři pody, z nichž každý má své vlastní úložiště [46],

4. vzhledem k tomu, že se pody od sebe navzájem odlišují, StatefulSet vyžaduje, aby byla vytvořena headless služba (tj. služba bez cluster IP), která poskytuje skutečnou síťovou identitu každému podu – záznam DNS. Ten může být vyžadován, aby jiné pody v clusteru mohly přistupovat k podu podle jeho hostitelského jména (hostname).

Hlavní objekty, které je potřeba vytvořit pro spuštění PostgreSQL serveru, jsou StatefulSet, PersistentVolume a Service. Konfigurace StatefulSetu je poměrně podobná konfiguraci nasazení, a proto nebude uvedena, ale lze ji najít ve zdrojovém kódu.

8.5.1 Konfigurace úložiště pomocí Persistent Volume

Persistent Volume je objekt, který umožňuje podu dynamicky žádat o úložiště a skrýt před vývojářem informace o skutečné infrastruktuře síťového úložiště. Například pro vytvoření úložiště s podporou NFS musí vývojář znát skutečný server, na kterém se nachází export NFS. To je v rozporu se základní myšlenkou Kubernetes, která si klade za cíl skrýt skutečnou infrastrukturu, umožňuje se nestarat o specifika infrastruktury a zajistit přenositelnost aplikací napříč širokou škálou cloudových poskytovatelů a lokálních datových center [43].

Místo toho, aby vývojář specifikoval úložiště při vytvoření podu, správce clusterů definuje nějaké úložiště, a pak přes Kubernetes API zaznamená jako zdroj clusteru. Při vytváření zdroje Persistent Volume se určuje jeho velikost a typ přístupu. V příkladu 8.3 je definice lokálního úložiště v adresáři /var/data, které má kapacitu 30 GB.

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: smartform-db
  namespace: dev
spec:
  capacity:
    storage: 30Gi
  accessModes:
    - ReadWriteMany
  persistentVolumeReclaimPolicy: Retain
  storageClassName: local-storage
  local:
    path: /var/data/
  nodeAffinity:
    required:
      nodeSelectorTerms:
        - matchExpressions:
            - key: kubernetes.io/hostname
              operator: In
              values:
                - kube01
```

Fragment kódu 8.3: Příklad konfigurace objektu PersistentVolume

Když uživatel clusteru potřebuje použít trvalé úložiště v jednom ze svých podů, nejprve vytvoří manifest PersistentVolumeClaim, který funguje jako žádost o úložiště. Uživatel pak odešle tento manifest do API-serveru, přičemž Kubernetes pak najde vhodné úložiště PersistentVolume a spojuje je se žádostí PersistentVolumeClaim. Pomocí vlastnosti spec.volumeName v manifestu žádostí je možné explicitně

přiřadit potřebné úložiště. Žádost i úložiště jsou v relaci 1:1, proto vyžadované úložiště nemůže být přiřazeno k nějaké jiné žádosti. Příklad 8.4 ukazuje manifest PersistentVolumeClaim s odpovídajícím úložištěm z minulého příkladu.

```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: smartform-db-pvc
  namespace: dev
spec:
  accessModes:
    - ReadWriteMany
  storageClassName: local-storage
  volumeName: smartform-db
  resources:
    requests:
      storage: 20Gi
```

Fragment kódu 8.4: Příklad konfigurace objektu PersistentVolumeClaim

Po vytvoření manifestu je potřeba jej odeslat do API pomocí příkazu „kubectl apply“. Zobrazit všechny aplikace lze příkazem „kubectl get pvc“:

```
$ kubectl get pvc
NAME              STATUS VOLUME     CAPACITY ACCESSMODES AGE
smartform-db-pvc Bound smartform-db 20Gi      RWX         3s
```

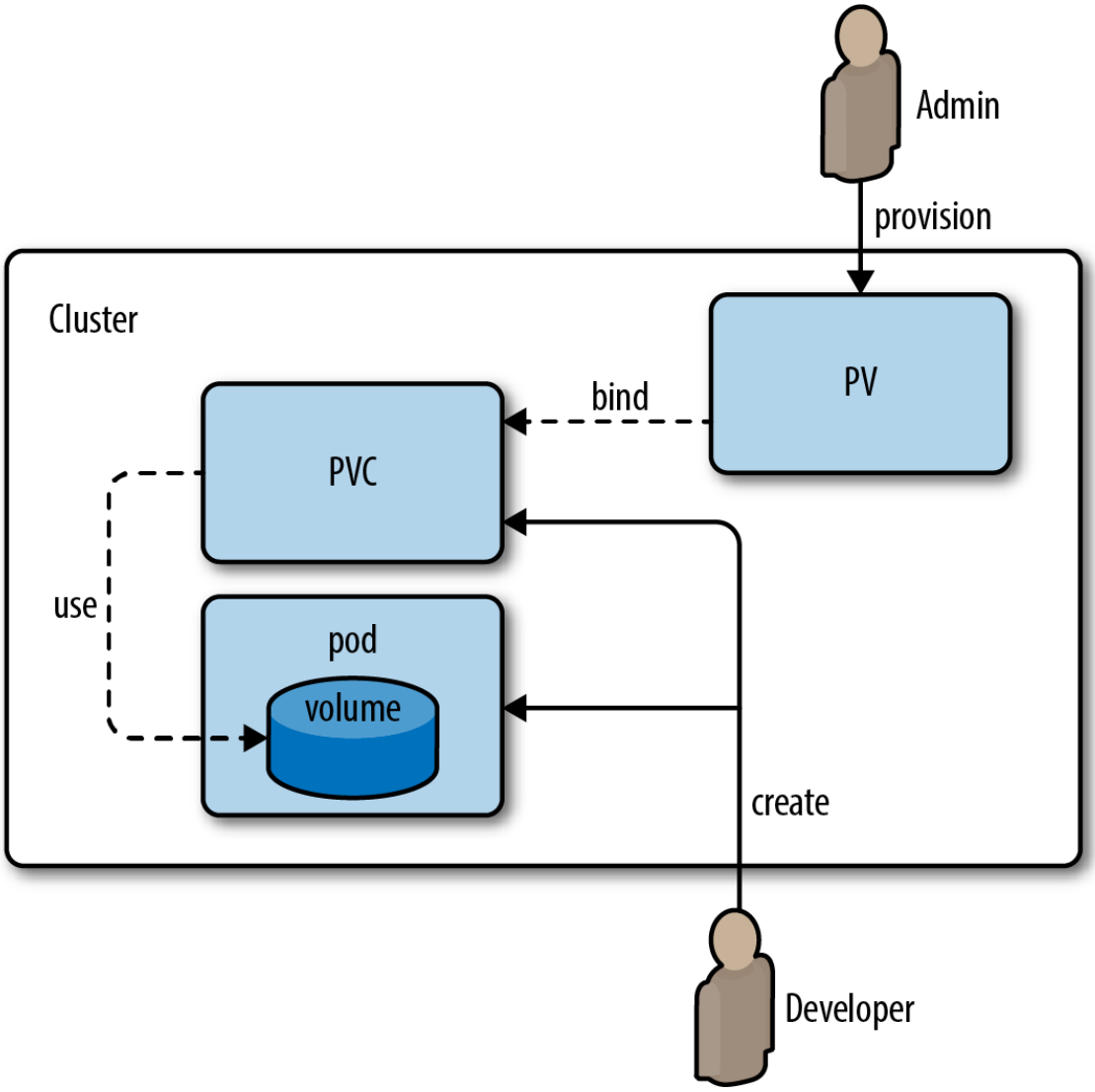
Poté v požadovaném objektu propojíme vytvořené datové úložiště „smartform-db-pvc“ s podem pomocí vlastnosti „volumesMount“. Obrázek8.3 je zjednodušené vizuální vysvětlení Persistent Volume, Persistent Volume Claims v clusteru Kubernetes.

Vytvořený objekt pro ukládání dat má typ **Local Persistent Volume**, protože úložiště je přidruženo k adresáři /var/data/. Zdá se, že vazba hostitelského adresáře na kontejner již byla popsána v Deployment manifestu pomocí hostPath, nicméně tyto dvě metody mají své odlišnosti [48]:

- **hostPath.** Svazek připojí soubor nebo adresář ze souborového systému hostitelského uzlu do podu. Pokud tedy máte víceuzlový cluster, pod z nějakých důvodů může být restartován a přiřazen k jinému uzlu, je možné, že data v novém uzlu nebudou relevantní nebo nebudou vůbec,
- **Local Persistent volume.** Takový typ úložiště Kubernetes pomáhá překonat omezení a může pracovat ve víceuzlovém prostředí bez problémů. Kubernetes si pamatuje, který uzel byl použit pro úložiště, čímž se ujistí, že restartující pod vždy najde své datové úložiště ve stavu, v jakém jej opustil před restartem. Jakmile však uzel zanikne, data jak hostPath, tak Local Persistent Volume tohoto uzlu budou ztracena.

8.5.2 Vystavení přístupu pomocí Service

Po vytvoření úložiště a spuštění objektu StatefulSet je nutné udělit přístup k vytvořeným podům. Každý pod má svou IP adresu, ale vzhledem k tomu, že pod může být restartován s jinou adresou, je přístup k nim poskytován objektem Service.



Obrázek 8.3: Persistent Volume a Persistent Volume Claim v clusteru [47]

V kapitole 4.4.5 jsme se již zmínili, že existuje několik typů abstrakcí služeb, ale v tomto příkladu budeme potřebovat standardní variantu ClusterIP. Pomocí tohoto typu je služba dostupná pouze z clusteru.

Jako každý objekt lze službu vytvořit imperativně:

```
$ kubectl expose (-f FILENAME | TYPE NAME) [--port=port]
[--protocol=TCP|UDP] [--target-port=number-or-name]
[--name=name] [--external-ip=external-ip-of-service]
[--type=type]
```

Nebo je možné využít obyčejné pomoci yaml souboru. Příklad manifestu pro vytvoření služby k StatefulSet databáze je uveden v příkladu 8.5.

```
apiVersion: v1
kind: Service
metadata:
  name: smartform-db
spec:
  ports:
```

```
  - port: 5432
  selector:
    app: smartform-db
```

Fragment kódu 8.5: Příklad konfigurace objektu Service

Tento manifest vytvoří službu, která cílí na TCP port 5432 (obyčejný port PostgreSQL serveru) na libovolný pod, který má label „app: smartform-db“.

Ke službě lze přistupovat dvěma způsoby:

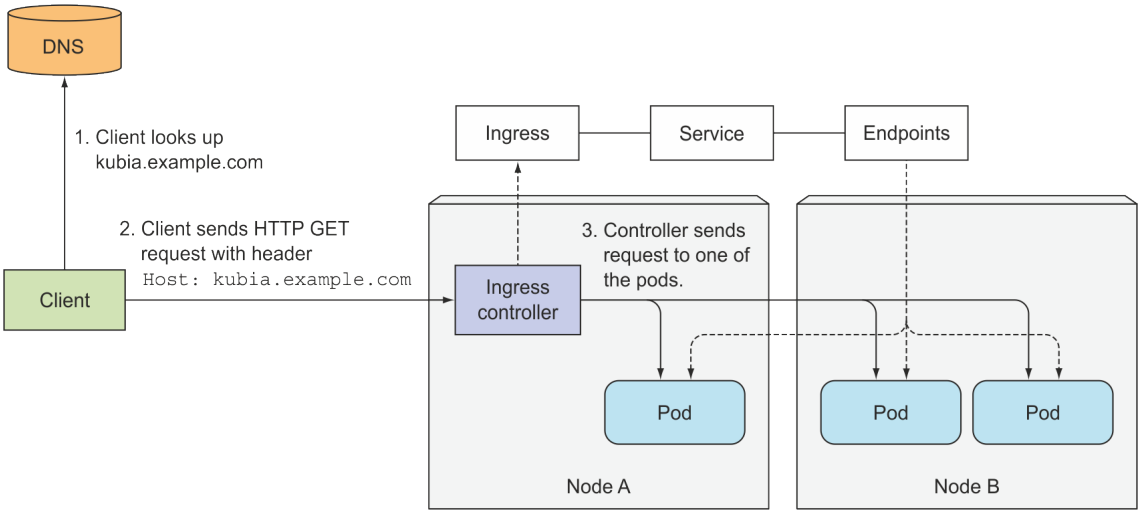
- DNS (nejběžnější): metoda DNS je doporučenou metodou zjišťování služeb. Pro každou službu a endpoint DNS vytvoří záznam. Formát záznamu pro službu je „service-name.namespace-name.svc.cluster.local“. Díky těmto záznamům je přístup ke službě možný prostřednictvím názvu služby. V rámci stejného namespace je služba dostupná pomocí názvu <service-name>, v rámci jiného namespace je možné k ní přistupovat pomocí <service-name>.<namespace-name>. V našem případě je možné z jiných podů přistupovat k databázi s názvem „smartform-db“.
- Proměnné prostředí: Tato metoda spoléhá na to, že kubelet přidá proměnné prostředí pro každou aktivní službu pro každý uzel, na kterém je pod spuštěn.

Aby části systému mohly komunikovat mezi sebou, je ke každému objektu Deployment a StatefulSet vytvořena vlastní služba.

8.6 Zajištění externího přístupu k systému pomocí objektu Ingress Controller

Vytvořením služeb pro každou část aplikace jsme nastavili komunikaci uvnitř clusteru. Kubernetes umožňuje přístup k službám clusteru třemi způsoby:

- NodePort. Tato služba na každém uzlu clusteru otevře statický port a přesměruje provoz na požadovanou službu. Kontaktovat službu mimo cluster je možné pomocí dotazu na <NodeIP>:<NodePort>.
- LoadBalancer. Při použití load balanceru poskytovatel cloudových služeb vytvoří před uzly load balancer. Takový typ služby poskytuje veřejnou IP adresu nebo název DNS, ke kterému se mohou externí uživatelé připojit.
- Ingress. Kubernetes Ingress navazuje na Kubernetes Services a zajišťuje vyrovnávání zátěže na aplikační vrstvě, mapování HTTP a HTTPS požadavků s konkrétními doménami nebo URL na služby Kubernetes [49]. Tato metoda se liší od NodePortu a load balanceru. Ingress není typ služby, ale je to samostatný objekt, který je na úrovni abstrakce vyšší než služby. Na rozdíl od služeb Kubernetes, které jsou zpracovávány na síťové vrstvě (L3-4), Ingress pracuje na aplikační vrstvě (L5-7). Jedním z důležitých důvodů je to, že dvě výše uvedené služby poskytují přístup jen k jednomu typu služby, zatímco Ingressfunguje jako API gateway a přesměruje provoz na desítky služeb. Když klient odešle požadavek HTTP ke vstupu, hotname a cesta (path) v dotazu určí, na kterou službu musí tento požadavek být přesměrován. Obrázek 8.6 ukazuje, jak funguje připojení klienta k jednomu z podů prostřednictvím controlleru Ingress.



Obrázek 8.4: Přístup k podům prostřednictvím Ingress [43]

Pro systém Smartform má smysl používat právě Ingress. Konkrétní implementace Ingressu závisí na tom, který Ingress controller[5] vyberete. Tři nejpopulárnější controllery nasazené na Kubernetes jsou Nginx, Trafika a HAProxy.

8.6.1 Instalace Nginx controlleru v clusteru

Aby ovládač Ingress fungoval, je třeba vytvořit určité objekty Kubernetes – jmenný prostor „ingress-nginx“, service account, konfigurační mapy atd. Všechny zmíněné objekty Kubernetes můžete vytvořit pomocí souboru yaml z oficiálního githubu Ingress pomocí příkazu:

```
$ kubectl apply -f https://raw.githubusercontent.com/kubernetes/
ingress-nginx/master/deploy/static/mandatory.yaml
```

Ověřit, že Ingress Nginx Controller běží, lze takto:

```
$ kubectl get svc -n ingress-nginx
NAME                       TYPE           CLUSTER-IP      EXTERNAL-IP
PORT(S)
ingress-nginx-controller   LoadBalancer   10.96.229.38    129.146.214.219
80:30756/TCP,443:30118/TCP
```

V současné architektuře Nginx je SPOF, jeho selhání přímo povede k nedostupnosti všech ostatních služeb, protože se nedostanou ke klientským dotazům. K tomu v infrastruktuře Kubernetes provedeme horizontální škálování této části systému a zvýšíme počet replik na třipomocí příkazu:

```
$ kubectl scale deployment nginx-ingress-controller
-n ingress-nginx --replicas=3
```

8.6.2 Nastavení pro zpracování TLS provozu

Když klient otevře TLS spojení s Ingress ovládačem, Ingress ukončí toto TLS spojení. Komunikace mezi klientem a ovládačem je šifrovaná, zatímco komunikace

[5]Dostupné z: `https://kubernetes.io/docs/concepts/services-networking/ingress-controllers/` (13.07.2022)

mezi ovládačem a pody šifrovaná není. Pro povolení této funkcionality je nutné k Ingressu připojit certifikát **tls.cert** a soukromý klíč **tls.key**. Ty by měly být uloženy ve zdroji Kubernetes s názvem Secret, na který pak odkazuje manifest Ingress [43].

Vytvoření objektu Secret z klíče a certifikátu se provádí následujícím způsobem:

```
$ kubectl create secret tls secret-tls --cert=tls.cert
--key=tls.key
```

8.6.3 Konfigurace směrování provozu pomocí objektu Ingress

Jakmile je v clusteru aktivován controller, můžeme vytvořit zdroj Ingress. Celý manifest je uveden ve zdrojovém kódu, pro ilustraci bude uvedena část souboru:

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: nginx-smartform
  namespace: dev
spec:
  ingressClassName: nginx
  tls:
  - hosts:
    - secure.smartorm.cz
    secretName: secret-tls
  rules:
  - host: "secure.smartorm.cz"
    http:
      paths:
        - pathType: Prefix
          path: "/admin"
          backend:
            service:
              name: admin
              port:
                number: 80
```

Fragment kódu 8.6: Příklad konfigurace objektu Ingress

Stejně jako ve všech manifestech jsou povinná pole apiVersion, kind, metadata a speč. V části 8.6.2 jsme vytvořili Secret s názvem „secret-tls“ a použili ho v manifestu Ingress. Nyní Ingress ovládač může přijímat https provoz na doméně secure.smartform.cz. Přístup ke službě „admin“ je tedy možný zvenčí a klienti k ní mají přístup podle URL secure.smartform.cz/admin.

8.7 Zajištění vysoké dostupnosti systému v Kubernetes.

Nakonec, poté, co jsme popsali, jak je možné nasadit aplikaci v clusteru a dát jí možnost komunikovat v rámci clusteru a přijímat provoz z vnějšího světa, se nyní

podívejme, jak jsme pomocí konfigurací odstranili problémy z kapitoly 5.2, které se vyskytují v současné architektuře, a zvýšili úroveň dostupnosti.

1. Zvýšení počtu replik jedné aplikace. Nyní se selhání jednoho kontejneru nezobrazí jako úplná nedostupnost služby pro zákazníky, protože v systému bude fungovat ještě minimálně jedna replika. Počet replik lze nastavit ve většině objektů kategorie „Workload resourses“ v poli „replicas“. Z tohoto důvodu se nedoporučuje ručně vytvářet každý pod, ale nechat objekty spravovat pody samy. Zvýšením počtu replik jsme eliminovali problém reverzního proxy Nginx, který jsme považovali za SPOF.

2. Automatizace oprav. Považujeme to za největší plus přechodu na platformu Kubernetes. Odstraněním nutnosti ručně opravovat každou chybu výrazně snižujeme dobu prostoje. Pokud jsou komponenty masteru v pořádku, api-server si ve velmi krátké době všimne, zda se stav podu změnil, a ihned upozorní požadovaný objekt Kubernetes, který tento pod ovládá.

3. Affinity nebo distribuce aplikace napříč uzly. Způsob, který minimalizuje dopad havárie uzlu. Díky této funkci jsme při nasazení naší aplikace rozdělili repliky mezi uzly tak, aby ani výpadek jednoho z uzlů neovlivnil chod systému. Bohužel ne všude se podařilo použít metodu škálování a například jeden ze serverů PostgreSQL nemá další repliky. Pokud tedy spadne server nebo uzel, na kterém pracuje, stane se nedostupným.

4. Průběžná kontrola pomocí liveness probes. Zajistí zvýšenou dostupnost na úrovni podů. Jejich pomocí bude kontejner chráněn nejen před neočekávanými poruchami, ale také před nadcházejícími na základě výsledků těchto kontrol.

5. Rolling updates. Způsob zavádění nové verze a zároveň zkrácení času plánovaného prostoje aplikace. Dříve aktualizace fungovala tak, že se všechny kontejnery zabíjejí a pak spouštějí nové. Strategie Rolling Updates, která je popsána v kapitole 8.2, nahrazuje tento zastaralý způsob.

V další kapitole se pokusíme kvantifikovat toto zlepšení pomocí vybraných metrik.

Kapitola 9

Evaluace

9.1 Vyhodnocení navržené infrastruktury pomocí metrik

Vysoká dostupnost systému se skládá ze dvou kritérií: toleruje-li systém výpadek nějaké části, a jak dlouho se systém z výpadku zotavuje. Pro vyhodnocení těchto kritérií jsme společně se zadavatelem vybrali dva ukazatele z těch, které jsme popsali v kapitole 1.3, protože jsou jednoznačné a umožňují jasně měřit zlepšení vysoké dostupnosti: SPOF a Mean time to repair.

Toleranci systému k výpadkům jsme zvýšili pomocí eliminace **SPOF**. Jak již bylo popsáno, v současné infrastruktuře jsme odvodili dva body, fyzický server a reverzní proxy, jejichž pád může vést k zastavení celého systému, a novým řešením se podařilo tyto dva problémy odstranit. V kapitole 1.4 jsme uvedli příklady, které mohou být realizovány pro zvýšení dostupnosti, z toho dvě metody byly použity k řešení tohoto problému.

- Clustering. Ke stávajícímu serveru jsme přidali dva virtuální servery a umístili jsme je do clusteru. Clustering serverů chrání služby před hardwarovým selháním výpočetního zdroje – serveru. Takže při selhání jednoho serveru se aplikace automaticky restartují na jiných uzlech clusteru bez ručního zásahu správce systému.

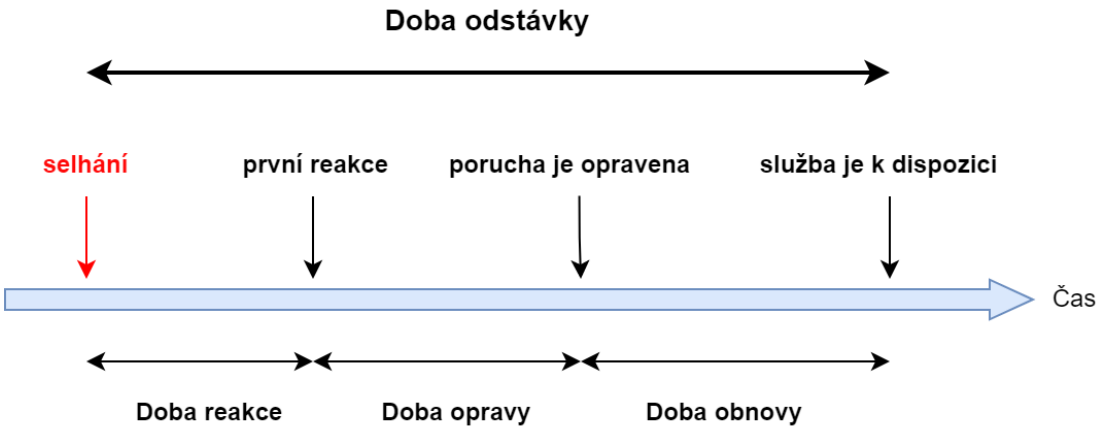
- Horizontální škálování. Systém Smartform se skládá z deseti částí, při přesunu do clusteru se podařilo zvýšit počet replik pro sedm z nich. Bylo rozhodnuto ponechat tři služby v původní verzi: zvýšení počtu replik pro dvě aplikace je kvůli funkčnosti samotné aplikace nemožné a do budoucna nemá smysl se tím zabývat, nicméně jedna aplikace (PostgreSQL server) nebyla replikována kvůli složité konfiguraci a v budoucnu, jak píšeme v kapitole 9.2, je vhodné vyřešit tento problém přesunem do PostgreSQL clusteru.

Druhá metrika, **Mean time to repair**, ukazuje přínos použití Kubernetes a vyjadřuje dobu potřebnou k opravě. Tu lze rozdělit do následujících podkategorií [50]:

- doba reakce: doba mezi událostí poruchy a první reakcí, která vyjadřuje, že událost selhání byla detekována,

- doba opravy: doba mezi první reakcí na selhání, a kdy je porucha opravena,

- doba obnovy: doba mezi první reakcí na událost selhání, a kdy je služba opět k dispozici,
- doba odstávky: doba, po kterou nebyla služba k dispozici. Představuje součet doby reakce a doby obnovy, jak je znázorněno na obrázku 9.1.



Obrázek 9.1: Čas na opravu poruchy

Abychom pomocí kvantitativního indikátorů ukázali, že s Kubernetes bylo možné zkrátit průměrnou dobu opravy, budeme pro každou metriku analyzovat situaci selhání podu a uzlu v současné i navrhované infrastruktuře a svou analýzu shrneme v tabulce 9.1. Některé věci mohou být při měření této doby velmi subjektivní, zejména při hodnocení současné infrastruktury, vzhledem k neustálé potřebě ručního zásahu při takových poruchách.

Metrika	Navržená instrastruktura	Současnosná infrastruktura
Doba reakce na selhani podu	1-5 sekund	několik minut až několik hodin
Doba reakce na selhání uzlu	1 minuta	několik minut až několik hodin
Doba opravy podu	5 sekund až 5 minut	15 minut a víc
Doba opravy uzlu	5-10 minut	30 minut a víc
Doba obnovy	5 sekund až 5 minut	5 sekund

Tabulka 9.1: Srovnání doby oprav v různých infrastrukturách.

Doba reakce

V průměru Kubernetes reaguje až pět sekund na selhání podu a až minutu na selhání uzlu. Pokud jde o současný systém, doba odezvy se může značně lišit. Podle toho, jak rychle se správce dozví o poruše, tak může trvat několik minut až několik hodin.

Doba opravy

Když pod spadne v clusteru, Kubernetes ho opraví tak, že odstraní poškozený pod ze seznamů koncových bodů a vytvoří nový. Když vytváří, může potřebovat další čas na stažení obrazu kontejnerů a konfiguraci úložiště. Proto to také zpravidla trvá od 5–10 sekund do 5 minut v závislosti na konfiguraci samotného podu. Spadne-li uzel, všechny pody, které byly na rozbitém uzlu, Kubernetes restartuje na jiných uzlech. Proto to také v závislosti na počtu podů trvá v průměru až 5–10 minut.

Pokud jde o současnou infrastrukturu, hodně záleží na důvodu selhání podu a na tom, co administrátor udělá: může jednoduše restartovat kontejner, nebo například pokud chyba nastala na úrovni aplikace kvůli nové verzi, administrátor může nahradit aplikaci stabilní verzí. Proto to v průměru trvá 15 minut a déle. V případě poruchy uzlu se čas na opravu prodlužuje. Rozbitý uzel je pro tuto infrastrukturu kritický. Obnovení může trvat v nejlepším případě až půl hodiny, zatímco administrátor restartuje server a znovu spustí celý systém.

Doba obnovy

Po dokončení opravy a spuštění kontejneru může nějakou dobu trvat, než se spustí samotná aplikace uvnitř kontejneru (načtení konfigurace, připojení databáze atd.). Pokud je readiness probe správně nastaven, pod nebude v seznamu koncových bodů, dokud se aplikace úspěšně nespustí. Kubernetes díky tomu sice prodlužuje dobu obnovy, ale neumožňuje zákazníkům přistupovat do aplikace, která ještě není připravena přijímat dotazy. To je stále plus nové infrastruktury – i když se to z časového hlediska zdá být horší, je to jen proto, že Kubernetes provádí dodatečné kontroly a čeká na kompletní a úspěšné spuštění aplikace.

Na základě toho je vidět, že na dvou ze tří indikátorů vykazují Kubernetes výrazně lepší časové výsledky, tudíž se oproti stávající infrastruktuře zkrátí celá doba oprav.

9.2 Plán budoucího rozvoje

Ve své práci jsme se snažili zlepšit kritérium dostupnosti v infrastruktuře, ale navrhované řešení nepokrývá celý systém před selháním 100%. V této kapitole navrhujeme plán dalšího rozvoje a zkvalitnění infrastruktury v Kubernetes, který dále zlepší dostupnost.

1. 1. Výměna lokálního úložiště na vzdálené. Oddělení míst pro ukládání dat od míst pro provádění aplikací zlepší infrastrukturu a zvýší dostupnost celého systému. Díky tomu nebudou části systému pevně spojeny s uzlem: při rozbití uzlu zůstanou data dostupná a neporušená a všechny části aplikací lze snadno restartovat na jiném uzlu. Přestože mohou nastat problémy s použitím vzdáleného úložiště, je toto řešení lepší než použití lokálního úložiště, protože poskytuje větší flexibilitu clusteru a minimalizuje situace, kdy aplikace nemůže být restartována kvůli nedostatku potřebného datového úložiště.

2. CI\CD[1]. Kontinuální integrace (CI[2]) a nepřetržité doručování (CD[3]) je soubor zásad a postupů, jejichž pomocí je možné zavádět změny softwaru rychleji, častěji a spolehlivěji.

3. Monitorování. Není součástí navrženého řešení, nicméně je velice důležité a umožňuje ověřit fungování uzlů v clusteru a sledovat zdraví celého clusteru Kubernetes. Mezi oblíbené nástroje pro sledování patří Kubernetes Dashboard, Prometheus a EFK Stack.

[1]Dostupné z: `https://docs.gitlab.com/ee/ci/introduction/` (20.07.2022)
[2]Continuous Integration
[3]Continuous Deployment

4. Další škálování systémů. Při hodnocení jsme zmínili, že ne všechny části systému byly replikovány, takže v budoucnu bude nutné zvětšit úroveň dostupnosti pro server PostgreSQL a provést migrace z jednotné instance serveru na cluster.

5. Vnější etcd úložiště[4]. Pomocí této topologie rozdělujeme řídicí rovinu a úložiště etcd, čímž poskytujeme konfiguraci HA, kde má ztráta instance řídicí roviny nebo prvků etcd menší dopad a nemá vliv na redundanci clusteru.

[4]Dostupné z: `https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/high-availability/#external-etcd-nodes` (20-07-2022)

Závěr

Hlavním cílem této bakalářské práce bylo vyhodnotit míru dostupnosti existující infrastruktury systému Smartform a navrhnout její zvýšení pomocí zvolené technologie. Pro splnění tohoto cíle bylo potřeba práci rozdělit na teoretickou a praktickou část.

V rámci teoretické části byla prozkoumána problematika vysoké dostupnosti informačních systémů, byly vypracovány klíčové metriky pro posouzení dostupnosti každého systému a běžné techniky využívané pro zvětšení dostupnosti. Byla provedena rešerše technologií orchestrace, existujících platforem pro orchestraci kontejnerů a na základě porovnání zvolených platforem vybrán Kubernetes jako nejvhodnější platforma pro praktickou část.

V rámci praktické části byla provedena analýza systému Smartform, provozovaného v kontejnerovém prostředí, byly odhaleny slabé stránky této infrastruktury a byly shromážděny požadavky na novou infrastrukturu. Na základě poznatků z teoretické části bylo rozhodnuto o realizaci zvýšení dostupnosti pomocí konfigurace clusteru v orchestrační platformě Kubernetes. Implementace byla rozdělena na dvě etapy: na konfiguraci a nasazení vysoce dostupného multi master clusteru a následné nasazení systému Smartform do vytvořeného clusteru. Nakonec byla provedena evaluace podle zvolených metrik z teoretické části, která vyjádřila přínos migrace systému do Kubernetes.

Zadatel je spokojen s výsledky dosaženými v důsledku této bakalářské práce, hodnotí práci zcela pozitivně a považuje ji za přínosnou pro další rozvoj projektu. Po zkušebním nasazení a samotném testování byla obdržena zpětná vazba od zaměstnance společnosti Trixi: "Řešení, které bylo v rámci bakalářské práce vyvíjeno, zatím nemáme produkčně nasazeno. Po shlédnutí prezentace toho, co bylo zatím uděláno, si myslím, že nám vyvíjené řešení významně pomůže eliminovat výpadky způsobené havárií aplikace nebo celého serveru. Stejně tak očekávám, že nám ubyde výpadků, ke kterým dochází při nasazování nových verzí našich aplikací, a že díky tomu budeme moci nasazovat častěji a celý vývojový proces tak bude mnohem plynulejší".

Za výstup této práce považujeme návod, jak zapojit servery do HA clusteru a jak používat objekty Kubernetes pro nasazení aplikace a konfigurační soubory, jejichž pomocí byla provedena samotná migrace do Kubernetes. Tato bakalářská práce bude užitečná pro čtenáře, kteří nemají základní znalosti orchestrálních nástrojů, ale i pro zkušenější uživatele, kteří jsou s prací na Kubernetes již seznámeni. Tato práce nám pomohla lépe porozumět tématům, která nás zajímala, a aplikovat teoretické poznatky získané z článků a knih na skutečný projekt.

Bibliografie

1. LERNER, Andrew. *The Cost of Downtime*. 2014. Dostupné také z: https://blogs.gartner.com/andrew-lerner/2014/07/16/the-cost-of-downtime/. [cit. 2021-12-11].
2. MORRIS, Chris. *Facebook's outage cost the company nearly 100 million dollars in revenue*. Fortune, 2021. Dostupné také z: https://fortune.com/2021/10/04/facebook-outage-cost-revenue-instagram-whatsapp-not-working-stock/. [cit. 2021-11-24].
3. *What is high availability?* Cisco, 2021. Dostupné také z: https://www.cisco.com/c/en/us/solutions/hybrid-work/what-is-high-availability.html. [cit. 2021-12-01].
4. *Cloud, Který Neodfoukne Ani Tornádo*. [B.r.]. Dostupné také z: https://www.dataspring.cz/cloud-ktery-neodfoukne-ani-tornado/. [cit. 2021-12-04].
5. SYSEL, Martin; LUKAŠÍK, Petr. Fault tolerance systém v prostředí výpočetního Gridu. *trilobit*. 2015, s. 1–2. [cit. 2021-11-24].
6. *Planning for network availability*. [B.r.]. Dostupné také z: https://www.ibm.com/docs/en/power5?topic=communications-planning-network-availability. [cit. 2021-12-01].
7. *Levels Of Availability*. 2009. Dostupné také z: https://deinoscloud.wordpress.com/2009/07/28/levels-of-availability/. [cit. 2021-12-01].
8. KYPRIANIDES, Martin. *Next Generation Availability*. Martin Kyprianides, 2018. Dostupné také z: https://present5.com/marathon-ever-run-next-generation-availability-martin-kyprianides/. [cit. 2021-03-04].
9. *Co je to Škálovatelnost?* [B.r.]. Dostupné také z: https://it-slovnik.cz/pojem/skalovatelnost/?utm_source=cp&utm_medium=link&utm_campaign=cp. [cit. 2021-12-01].
10. *Co to je virtualizace?* [B.r.]. Dostupné také z: https://tech-lib.eu/tema/1859/co-to-je-virtualizace. [cit. 2021-12-01].
11. *Brief history of virtualization*. [B.r.]. Dostupné také z: https://docs.oracle.com/cd/E26996_01/E18549/html/VMUSG1010.html. [cit. 2022-08-01].
12. FEDOSEENKO, Victoria. *ISPSystem*. 2019. Dostupné také z: https://www.ispsystem.com/news/brief-history-of-virtualization. [cit. 2022-07-01].
13. *Difference between virtualization and Emulator*. [B.r.]. Dostupné také z: https://tipsmake.com/difference-between-virtualization-and-emulator. [cit. 2021-03-12].

14. TYLEČEK, Viktor. *Properties comparison of SW for local virtualization*. 2017. Dostupné také z: https://dspace.cvut.cz/bitstream/handle/10467/68585/F3-BP-2017-Tylecek-Viktor-Porovnani_vlastnosti_SW_produktu_pro_lokalni_virtualizaci.pdf?sequence=1&isAllowed=y. [cit. 2021-12-04].

15. *Best server virtualization software of 2022: ENP*. 2022. Dostupné také z: https://www.enterprisenetworkingplanet.com/guides/server-virtualization-software/. [cit. 2022-07-21].

16. *What is containerization? – containerization definition - citrix*. [B.r.]. Dostupné také z: https://www.citrix.com/solutions/app-delivery-and-security/what-is-containerization.html. [cit. 2021-12-04].

17. *A practical guide to choosing between Docker Containers and VMS*. [B.r.]. Dostupné také z: https://www.weave.works/blog/a-practical-guide-to-choosing-between-docker-containers-and-vms. [cit. 2021-03-03].

18. *Co Je Kontejner?* [B.r.]. Dostupné také z: https://azure.microsoft.com/cs-cz/overview/what-is-a-container/#overview. [cit. 2021-11-02].

19. NISHANIL. *Learn docker*. [B.r.]. Dostupné také z: https://docs.microsoft.com/cs-cz/dotnet/architecture/containerized-lifecycle/what-is-docker. [cit. 2022-07-31].

20. *Docker Overview*. 2021. Dostupné také z: https://docs.docker.com/get-started/overview/. [cit. 2021-11-02].

21. ŠVALOV, Alexsandr. *Uložiště dat v dockeru*. Habr, 2021. Dostupné také z: https://habr.com/ru/company/southbridge/blog/534334/. [cit. 2021-11-12].

22. *Docker architecture*. 2021. Dostupné také z: https://www.aquasec.com/cloud-native-academy/docker-container/docker-architecture/. [cit. 2021-11-02].

23. MERSCH, Vassili van der. *API-driven devops: Spotlight on Docker: Nordic apis*. 2016. Dostupné také z: https://nordicapis.com/api-driven-devops-spotlight-on-docker/. [cit. 2021-02-25].

24. CONTAINERIZATION IS A METHODOLOGY THAT IS DRIVING EFFICIENT, performant. *What are containers in cloud computing?* [B.r.]. Dostupné také z: https://www.intel.co.uk/content/www/uk/en/cloud-computing/containers.html. [cit. 2021-11-02].

25. *Docker Swarm*. [B.r.]. Dostupné také z: https://subscription.packtpub.com/book/cloud-&-networking/9781788394383/2/ch02lvl1sec17/docker-swarm. [cit. 2021-02-23].

26. *Raft consensus in swarm mode*. 2021. Dostupné také z: https://docs.docker.com/engine/swarm/raft/. [cit. 2021-11-02].

27. *Production-grade container orchestration*. [B.r.]. Dostupné také z: https://kubernetes.io/docs/concepts/_print/#pg-4d68b0ccf9c683e6368ffdcc40c838d4. [cit. 2021-11-02].

28. PATEL, Ashish. *Kubernetes Overview*. DevOps Mojo, 2021. Dostupné také z: https://medium.com/devops-mojo/kubernetes-architecture-overview-introduction-to-k8s-architecture-and-understanding-k8s-cluster-components-90e11eb34ccd. [cit. 2021-02-23].

29. MAAYAN, Gilad David. *Securing the kubernetes API server: Critical best practices: Nordic apis /*. 2022. Dostupné také z: https://nordicapis.com/securing-the-kubernetes-api-server-critical-best-practices/. [cit. 2022-08-07].

30. 16, Kevin Casey March. *Kubernetes Autoscaling, explained*. [B.r.]. Dostupné také z: https://enterprisersproject.com/article/2021/3/kubernetes-autoscaling-explanation. [cit. 2022-07-11].

31. *Cluster networking*. 2022. Dostupné také z: https://kubernetes.io/docs/concepts/cluster-administration/networking/. [cit. 2022-11-07].

32. SOOKOCHEFF, Kevin. *A guide to the kubernetes networking model*. [B.r.]. Dostupné také z: https://sookocheff.com/post/kubernetes/understanding-kubernetes-networking-model/#kubernetes-networking-model. [cit. 2022-11-07].

33. GUNAWARDANA, Mananu. *Run A kubernetes cluster locally on windows*. Medium, 2020. Dostupné také z: https://medium.com/@mananu/run-a-kubernetes-cluster-locally-on-windows-c15e685113bb. [cit. 2021-04-01].

34. *How to setup a kubernetes (k8s) cluster from scratch?* 2022. Dostupné také z: https://www.armosec.io/blog/setting-up-kubernetes-cluster/. [cit. 2021-05-01].

35. *Kubernetes*. 2019. Dostupné také z: https://www.rancher.cn/learning-paths/introduction-to-kubernetes-architecture/. [cit. 2021-05-11].

36. HIGHTOWER, Kelsey. *Bootstrap kubernetes the hard way on google cloud platform.* [B.r.]. Dostupné také z: https://github.com/kelseyhightower/kubernetes-the-hard-way. [cit. 2022-03-20].

37. BOSTANDOUST, Saeid. *Kubernetes Installation Methods The Complete Guide*. ITNEXT, 2021. Dostupné také z: https://itnext.io/kubernetes-installation-methods-the-complete-guide-1036c860a2b3. [cit. 2022-03-20].

38. *Docker Registry*. 2022. Dostupné také z: https://docs.docker.com/registry/.

39. MARKWORT, Julian. *Patroni : Setting up a highly available postgresql cluster*. 2019. Dostupné také z: https://www.cybertec-postgresql.com/en/patroni-setting-up-a-highly-available-postgresql-cluster/. [cit. 2021-06-01].

40. DOCS, Kubernetes. *Installing kubeadm*. ITNEXT, 2021. Dostupné také z: https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/install-kubeadm/. [cit. 2022-05-21].

41. *Installing Kubernetes with the Flannel Network Plugin on centos 7*. [B.r.]. Dostupné také z: https://gist.github.com/rkaramandi/44c7cea91501e735ea99e356e9ae7883. [cit. 2022-07-11].

42. KUBERNETES. *API Conventions*. 2022. Dostupné také z: https://github.com/kubernetes/community/blob/master/contributors/devel/sig-architecture/api-conventions.md#metadata. [cit. 2022-06-22].

43. LUKSA, Marko. *Kubernetes in action*. Manning, 2018. [cit. 2022-07-01].

44. *Assigning pods to nodes*. 2022. Dostupné také z: https://kubernetes.io/docs/concepts/scheduling-eviction/assign-pod-node/. [cit. 2022-06-15].

45. *Configmaps Kubernetes Documention*. 2022. Dostupné také z: https://kubernetes.io/docs/concepts/configuration/configmap/. [cit. 2022-06-12].

46. BAELDUNG. *Kubernetes deployment vs. StatefulSets*. 2021. Dostupné také z: https://www.baeldung.com/ops/kubernetes-deployment-vs-statefulsets#1-understanding-statefulsets-the-basics. [cit. 2022-07-05].

47. HAUSENBLAS, Michael; SÉBASTIEN, Goasguen. *Kubernetes Cookbook: Building cloud native applications*. O'Reilly Media, Inc., 2018. [cit. 2021-06-28].

48. *What's the difference between hostpath volume and the local persistent volume?* 2020. Dostupné také z: https://www.bswen.com/2020/12/others-local-vs-hostpath-k8s.html. [cit. 2022-07-10].

49. *About kubernetes ingress*. [B.r.]. Dostupné také z: https://projectcalico.docs.tigera.io/about/about-kubernetes-ingress. [cit. 2022-07-13].

50. VAYGHAN, Leila Abdollahi. *Kubernetes as an availability manager for Microservice based applications*. 2019. Dostupné také z: https://spectrum.library.concordia.ca/id/eprint/985906/. [cit. 2022-07-16].

Příloha A

Instalace cri-dockerd

```
# Go installation. Need version 1.17 and above
        $ wget https://dl.google.com/go/go1.17.11.linux-amd64
.tar.gz
        $ mv go /usr/local
        $ source ~/.profile
        $ export PATH=$PATH:/usr/local/go/bin


# Cri-dockerd installation
        $ git clone https://github.com/Mirantis/cri-dockerd.git
        $ cd cri-dockerd
        $ mkdir bin
        $ go get && go build -o bin/cri-dockerd
        $ mkdir -p /usr/local/bin
        $ install -o root -g root -m 0755 bin/cri-dockerd
/usr/local/bin/cri-dockerd
        $ cp -a packaging/systemd/* /etc/systemd/system
        $ sed -i -e 's,/usr/bin/cri-dockerd,/usr/local/bin/
cri-dockerd,' /etc/systemd/system/cri-docker.service
        $ systemctl daemon-reload
        $ systemctl enable cri-docker.service
        $ systemctl enable --now cri-docker.socket
```

Příloha B

Instalace kubeadm, kubectl a kubelet

```
# Update the apt package index and install packages needed
to use the Kubernetes apt repository
        $ sudo apt-get update
        $ sudo apt-get install -y apt-transport-https
ca-certificates curl


# Download the Google Cloud public signing key
        $ sudo curl -fsSLo /usr/share/keyrings/
kubernetes-archive-keyring.gpg
https://packages.cloud.google.com/apt/doc/apt-key.gpg


# Add the Kubernetes apt repository
        $ echo "deb [signed-by=/usr/share/keyrings/
kubernetes-archive-keyring.gpg] https://apt.kubernetes.io/
kubernetes-xenial main" | sudo tee
/etc/apt/sources.list.d/kubernetes.list


# Install kubelet, kubeadm and kubectl,
        $ sudo apt-get update
        $ sudo apt-get install -y kubelet kubeadm kubectl
        $ sudo apt-mark hold kubelet kubeadm kubectl
```