FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

ZADÁNÍ DIPLOMOVÉ PRÁCE

Název:	Automatizované nasazování databází do cloudového prostředí
Student:	Bc. Maroš Špak
Vedoucí:	Ing. Josef Gattermayer
Studijní program:	Informatika
Studijní obor:	Webové a softwarové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	Do konce zimního semestru 2019/20

Pokyny pro vypracování

1) Prozkoumejte stav existujících řešení pro automatizované nasazení databázových systémů v prostředí Kubernetes.
2) Připravte automatizované nasazení databázových systémů MongoDB, MySQL a Cassandra v prostředí Kubernetes. Řešení bude podporovat clusterování pro dosažení HA a bude kompatibilní s infrastrukturami Google Cloud, AWS a bare metal. Výstupem bude infrastructure as a code, neboli konfigurační soubory pro automatické nasazení navržené infrastruktury.
3) Navrhněte a implementujte prototyp aplikace, na které ukážete funkčnost nasazeného systému. Simulujte zátěž aplikace pomocí sekvence databázových dotazů. Doložte, že vytížení jednotlivých uzlů clusteru je obdobné, neboli že aplikace škáluje.

Seznam odborné literatury

Dodá vedoucí práce.

Ing. Michal Valenta, Ph.D.
vedoucí katedry

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
děkan

V Praze dne 21. února 2018

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

Diplomová práca

Automatizované nasazování databází do cloudového prostředí

Bc. Maroš Špak

Katedra softwarového inženýrství
Vedúci práce: Ing. Josef Gattermayer

8. mája 2018

Poďakovanie

Chcel by som poďakovať vedúcemu mojej práce Ing. Josefovi Gattermayerovi za cenné rady a pripomienky, ktoré uľahčili tvorbu práce. Tiež ďakujem svojim rodičom za ich podporu počas celého štúdia.

Prehlásenie

Prehlasujem, že som predloženú prácu vypracoval(a) samostatne a že som uviedol(uviedla) všetky informačné zdroje v súlade s Metodickým pokynom o etickej príprave vysokoškolských záverečných prác.

Beriem na vedomie, že sa na moju prácu vzťahujú práva a povinnosti vyplývajúce zo zákona č. 121/2000 Sb., autorského zákona, v znení neskorších predpisov. V súlade s ustanovením § 46 odst. 6 tohoto zákona týmto udeľujem bezvýhradné oprávnenie (licenciu) k užívaniu tejto mojej práce, a to vrátane všetkých počítačových programov ktoré sú jej súčasťou alebo prílohou a tiež všetkej ich dokumentácie (ďalej len „Dielo“), a to všetkým osobám, ktoré si prajú Dielo užívať.

Tieto osoby sú oprávnené Dielo používať akýmkoľvek spôsobom, ktorý neznižuje hodnotu Diela, a za akýmkoľvek účelom (vrátane komerčného využitia). Toto oprávnenie je časovo, územne a množstevne neobmedzené. Každá osoba, ktorá využije vyššie uvedenú licenciu, sa však zaväzuje priradiť každému dielu, ktoré vznikne (čo i len čiastočne) na základe Diela, úpravou Diela, spojením Diela s iným dielom, zaradením Diela do diela súborného či zpracovaním Diela (vrátane prekladu), licenciu aspoň vo vyššie uvedenom rozsahu a zároveň sa zaväzuje sprístupniť zdrojový kód takého diela aspoň zrovnateľným spôsobom a v zrovnateľnom rozsahu ako je zprístupnený zdrojový kód Diela.

V Praze 8. mája 2018 . . . .

České vysoké učení technické v Praze
Fakulta informačních technologií
© 2018 Maroš Špak. Všetky práva vyhradené.
Táto práca vznikla ako školské dielo na FIT ČVUT v Prahe. Práca je chránená medzinárodnými predpismi a zmluvami o autorskom práve a právach súvisiacich s autorským právom. Na jej využitie, s výnimkou bezplatných zákonných licencií, je nutný súhlas autora.

Odkaz na túto prácu

Špak, Maroš. *Automatizované nasazování databází do cloudového prostředí.* Diplomová práca. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2018.

Abstrakt

Kubernetes je open-source systém pre automatizáciu, nasadzovanie, škálovanie a správu kontajnerových aplikácií.

V súčastnosti na cloud platfome Kubernetes chýba predpripravená konfigurácia pre nasadenie databázových systémov, ako je napríklad MongoDB, Cassandra a MySQL, s využítím podpory clustrovania a zabezpečením vysokej dostupnosti. Táto práca sa zaoberá analýzou dostupných riešení pre nasadzovanie spomínaných databázových systémov na platformu Kubernetes a návrhom a implementáciou vlastného riešenia, ktoré zohľadňuje požiadavky na vysokú dostupnosť, ktoré vznikajú pri produkčnom nasadení.

Výsledkom práce sú konfiguračné súbory (vo formáte YAML) a scripty pre nasadenie do Kubernetes. Konfigurácia využíva zabudovanú funkcionalitu Kubernetes pre clustrovanie, vysokú dostupnosť a zvyšuje stabilitu systému v prípade havárie alebo nedostupnosti jedného z uzlov. Riešenie je možné použiť v prostredí Kubernetes bežiacom na Google Cloud, Amazon Web Services alebo na vlastnom hardware.

Implementovaný bol aj prototyp webovej aplikácie napísanej v jazyku Python, ktorá demonštruje prístup ku jednotlivým databázovým clustrom a na ktorej prebehol test priepustnosti požiadaviek pri zvyšovaní databázových uzlov.

Klíčová slova Kubernetes, Cassandra, MySQL, MongoDB, Docker, scaling, vysoká dostupnosť

Abstract

Kubernetes is an open-source system for automating deployment, scaling, and management of containerized applications.

Currently, there is lack of production-ready configuration for a deployment of database systems such as MongoDB, Cassandra and MySQL that uses a clustering capability of these systems and provides high availability setup backed by Kubernetes functionality.

The aim of this thesis is to analyze current solutions to an automatized deployment of database systems on the Kubernetes platform and to design and implement a custom solution that respects requirements for production deployments.

Outputs of the thesis are configurations files (in YAML format) and scripts used to deploy these databases on the Kubernetes. It supports deployment to multiple infrastructures - Google Cloud, AWS and bare metal.

Implemented web application, written in Python, provides an example on how to access deployed database systems. it was also used for load testing and to test the scaling capabilities of the clusters.

Keywords Kubernetes, Cassandra, MySQL, MongoDB, Docker, škálovateľnosť, high availability

Obsah

Úvod **1**

1 Analýza **3**
1.1 Kontajnery 3
1.2 Docker 3
1.3 Kubernetes 5
1.4 Škálovanie 15
1.5 MongoDB 17
1.6 Cassandra 20
1.7 MySQL 23

2 Prieskum sučasných riešení **27**
2.1 Cassandra 27
2.2 MongoDB 28
2.3 MySQL 31

3 Príprava vlastného riešenia **33**
3.1 Cassandra 33
3.2 MongoDB 37
3.3 MySQL 46
3.4 Prototyp web aplikácie 51

4 Nasadenie **55**
4.1 Príprava prostredia Kubernetes 55
4.2 Nasadenie Cassandra 62
4.3 Nasadenie MongoDB 62
4.4 Nasadenie MySQL 64

5 Testovanie **65**
5.1 MongoDB 65

5.2 Cassandra 70
5.3 MySQL 71

Záver **75**

Literatúra **77**

A Zoznam použitých skratiek **83**

B Obsah přiloženého CD **85**

Zoznam obrázkov

1.1 Znázornenie charakteristík IaaS, CaaS a PaaS. 6
1.2 Znázornenie architektúry behu Podov na Kubernetes uzloch (zdroj: [1]) 7
1.3 Diagram znazorňujúci komunikáciu vo vnútri shardovaného clustra. (zdroj: [2]) 19
1.4 Diagram znazorňujúci Casandra cluster. Coordinator je uzol, ktorý obsluhuje požiadavku (zdroj [3]) 22
1.5 Znázornenie high-level rozdielov medzi MySQL replikáciou a Galera clustrom. 25

3.1 Pohľad na architektúru nasadenia a využitie headless Service pri smerovaní požiadaviek k jednotlivým Cassandra uzlom. 35
3.2 Pohľad na Využitie headless `Service` pri smerovaní požiadaviek k jednotlivým uzlom replica setu. 42
3.3 Pohľad na celkovú architektúru nasadenia MongoDB s podporou shardingu 44
3.4 Pohľad na architektúru nasadenia mongos routerov. 45
3.5 Štruktúra vytváranej tabuľky pre MySQL 52
3.6 Pohľad na návrh nasadenia webovej aplikácie v kontexte databázových systémov 53

Zoznam tabuliek

5.1 Výsledky záťažového testu MongoDB 70
5.2 Výsledky záťažového testu Cassandra 71
5.3 Výsledky záťažového testu nasadeného PerconaXtraDB cluster . . 73

Úvod

Kubernetes sa v posledných rokoch dominantnou témou konverzácií v DevOps svete. Kubernetes je open-source systém typu kontajner ako služba (Container as a Service), ktorý zjednodušuje vývoj, testovanie, nasadenie a škálovanie aplikácií v prostredí cloudu. Aplikácie sú na Kubernetes nasadzované vo forme Docker kontajnerov.

Kontajnery prinášajú spôsob, akým vývojári môžu distribuovať identickú kópiu aplikácie na rôzne prostredia. Kontajner obsahuje zdrojový kód aplikácie, knižnice, závislosti, systémové nástroje a nastavenia, ktoré aplikácia potrebuje k svojmu behu. Kontajnerizácia zmenila prístup IT spoločností k životnému cyklu softvéru. Umožňuje plynulý prechod od technických experimentov, cez vývoj, testovanie, nasadenie až po technickú podporu. V kombinácii s cloudovými službami ako Kubernetes sa kontajnery stali neodmysliteľnou súčasťou continuous integration a continuous delivery postupov, ktoré napomáhajú spoločnostiam vydávať nové verzie softvéru a opravy rýchlejšie.

S rastúcou obľúbenosťou nasadzovania aplikácií na platformy ako Kubernetes prirodzene vznikla potreba behu databázových systémov v kontajneroch. Databázy avšak vyžadujú väčšiu opatrnosť pri nasadzovaní s orchestračnými nástrojmi ako Kubernetes a môžu naraziť na ich limity. Oproti typickým webovým aplikáciám vyžadujú uchovanie stavu, čo ide proti základným princípom Docker kontajnerov. Nasadenie databázového uzlu vyžaduje koordináciu s ostatnými databázovými uzlami. Majú unikátny stav, takže v prípade havárie nie je možné uzol nahradiť ľubovoľným iným uzlom. Dôležitá je vysoká priepustnosť a nízka latencia siete. Kritická je tiež podpora perzistentného úložiska pre uchovávanie veľkého množstvo dát. Databázy často vyžadujú komplexnú konfiguráciu, ktorá môže byť nad rámec možnosti automatizácie, ktorú ponúkajú orchestračné systémy.

Veľkí hráči na poli databáz, ako je Oracle, SAP, Google, Amazon a Microsoft spolu s menšími, ako je MongoDB, MariaDB naskočili na vlnu kontajnerizačných technológií a pre svoje produkty ponúkajú Docker obrazy. Kubernetes taktiež adresuje tento trend a ponúka nový koncept správy kontajnerov, Sta-

tefulSet, určený pre beh stateful aplikácií.

Cieľom práce je preskúmať súčasné riešenia nasadenia MongoDB, Cassandra a MySQL na platformu Kubernetes na infraštruktúre Google Cloud, AWS a na vlastnom hardware, navrhnúť a implementovať vlastné riešenie využívajúce clustrovanie a dostupnú funkcionalitu Kubernetes pre dosiahnutie vysokej dostupnosti. V prípade MongoDB bol vytvorený vlastný Docker obraz. Pre MySQL bola implementovaná podpora pravidelného automatizovaného zálohovania. Bola tiež navrhnutá a implementovaná webová aplikácia v jazyku Python. Aplikácia bola zabalená do Docker obrazu a nasadená ako kontajner na platforme Kubernetes. Pomocou tejto webovej aplikácie a nástroja Tsung bola otestovaná funkčnosť nasadených databázových systémov.

Kapitola 1

Analýza

1.1 Kontajnery

Aplikačné kontajnery predstavujú spôsob zabalenia, distribúcie a prevádzkovania softvéru. Kontajner obsahuje všetok kód aplikácie, knižnice, závislosti, systémové nástroje a nastavenia, ktoré aplikácia potrebuje k svojmu behu. Oproti virtuálnym strojom (VM) sú kontajnery menej náročné na systémové prostriedky, obraz vyžaduje menej diskového miesta a spustenie nového kontajneru môže byť rýchlejšie než spustenie VM. Kontajnery zdieľajú kernel hostiteľského operačného systému. Oddeľujú aplikáciu od infraštruktúry potrebnej k jej behu. Pre izoláciu jednotlivých kontajnerov sú využité linuxové technológie ako cgroups a namespace. Sú prenositeľné, vďaka tomu môže vývojár vyvíjať aplikáciu na svojom hardware a minimalizuje problémy s spojené s nekompatibilitou prostredí pri produkčnom nasadení. [4]

1.2 Docker

Docker[1] je nástroj, ktorý uľahčuje proces vytvárania, distribuovania a behu kontajnerizovaných aplikácií. Docker využíva existujúce linuxové technológie (copy-on-write filesystem AUFS[2], cgroups, namespaces) a prináša novú funkcionalitu zameranú na vývoj aplikácií ako sú prenositeľné obrazy kontajnerov, automatické zostavenie kontajneru, verzovanie alebo službu Docker Hub, kde vývojari môže nahrávať nimi vytvorené obrazy. [5]

Obrazy sú nahravané do Docker registra. Docker Hub je verejný register, ktoré môže využiť každý vývojár. Nástroj Docker je predkonfigurovaný na hľadanie obrazu práve v tomto registry.

Docker beží natívne na operačných systémoch Linux (s kernelom 3.10+), Windows a v cloude na platforme Amazon EC2, Google Compute Engine,

[1] `https://www.Docker.com/`

[2] `http://aufs.sourceforge.net/`

Microsoft Azure a Rackspace. [6]

Docker Engine je aplikácia typu client-server zložená z 3 hlavných komponent.

- Server bežiaci ako daemon proces (`Dockerd`)
- REST API špecifikujúce rozhranie pre komunikáciu s daemonom.
- Klient ako CLI nástroj (`Docker`)

O zostavenie obrazu, beh a distribúciu Docker kontajneru sa stará Docker deamon.

1.2.1 Zostavenie obrazu

Dockerfile je textový súbor, ktorý pomocou inštrukcií popisuje ako má vyzerať výsledné prostredie kontajneru. Obsahuje systémové a pre Docker špecifické príkazy:

- FROM - Základný obraz z ktorého sa vychádza a vytvára nový obraz. Prvý príkaz v Dockerfile.
- RUN - Vykoná príkaz počas build procesu Docker obrazu.
- ADD - Kopíruje súbor z hosťovského stroja do Docker obrazu.
- ENV - Definovanie premenných prostredia.
- CMD - Príkaz, ktorý sa vykoná po štarte z obrazu
- ENTRYPOINT - Umožňuje konfigurovať spustiteľný kontajner
- EXPOSE - Informuje o porte na ktorom proces čaká na spojenie
- WORKDIR - Nastaví pracovnú zložku pre príkazy RUN, CMD, ENTRYPOINT, COPY a ADD.
- USER - Nastavenie UID pre beh kontajneru a príkazy RUN, CMD a ENTRYPOINT.
- VOLUME - Vytvára prístup k k zložke medzi kontajnerom a hosťovským systémom.
- HEALTHCHECK - Definuje príkaz na zistenie (na základe navratového kódu), či kontajner beží korektne.

Obraz je tvorený vrstvami. Každý príkaz v Dockerfile vytvára novú vrstvu. Každá vrstva je množina zmien oproti predošlej vrstve. To umožňuje rýchle zostavenie a nahrávanie do registra odosielaním len zmenených vrstiev a taktiež sťahovanie len potrebných zmien namiesto celého obrazu. Všetky zmeny vykonané v kontajneri (zápis, uprava, mazanie súborov) sú zapísané do špeciálnej read-write vrstvy, unikátnej pre každý kontajner. Pri zmazaní kontajneru je zmazaná aj táto vrstva. [7]

Na zostavenie (build) Docker obrazu, so špecifikovaním repozitára a názvu pre jeho uloženie, slúži príkaz

```
docker build -t <repozitar>/<nazov> -f cesta_k_Dockerfile .
```

Nahranie vyhotoveného obrazu do registru sa vykoná príkazom

```
docker push <repozitar>/<nazov>
```

1.3 Kubernetes

Kubernetes[3] je open-source platforma typu kontajner ako služba (CaaS) navrhnutá pre automatizované nasadenie, škálovanie a správu kontajnerizovaných aplikácií. Cieľom Kubernetes je umožniť používateľom rýchle a jednoduché nasadenie aplikácií do clustru tvoreného fyzickými alebo virtuálnymi uzlami. Vďaka self-healing funkcionalite umožňuje ich kontrolovaný beh, plynulé aktualizácie, horizontálne škálovanie so zachovaním dostupnosti a iné.

Kubernetes bol pôvodne vyvinutý spoločnosťou Google, na základe ich dlhoročných skúseností s prevádzkovaním kontajnerizačného riešenia Borg. [8] V súčasnosti je projekt pod záštitou Cloud Native Computing Foundation.

Kubernetes poskytuje výhody platformy ako služby (PaaS) s flexibilitou infraštruktúry ako služby (IaaS) so zachovaním portability medzi rôznymi poskytovateľmi IaaS, vďaka čomu je možné Kubernetes prevádzkovať napríklad v Google Cloude, Amazon Web Services alebo na vlastnom hardware.

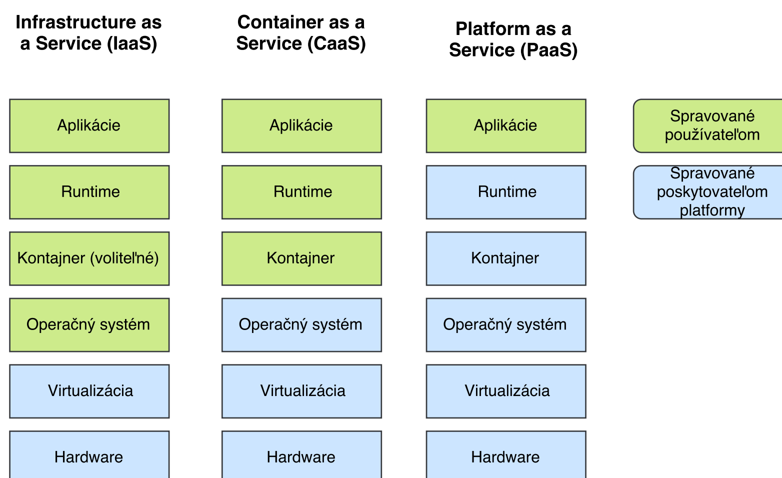
Jedným zo spoločných prvkov PaaS a CaaS je separácia používateľa od nižších vrstiev IT infraštruktúry, ako je znázornené na obrázku 1.3. Používateľ sa tak môže sústrediť na vývoj vlastnej aplikácie a starosť o servery, uložiská, virtualizačné nástroje, zabezpečenie konektivity a celkovú plynulú prevádzku ponechať na poskytovateľa platformy.

Hlavným rozdielom je prístup k nasadzovaniu aplikácií. Základnou jednotkou pre nasadenie aplikácie do PaaS je samotný zdrojový kód aplikácie. PaaS, akým je napríklad Cloud Foundry, automaticky nakonfiguruje potrebné behové prostredie a aplikáciu zabalí do kontajnera, ktorý následne spustí. S tým súvisí obmedzenie, ktoré z danej funkcionality vyplýva. Aby bol PaaS schopný automaticky postaviť kontajner s behovým prostredím, potrebuje mať podporu pre použité technológie ako je programovací jazyk alebo framework.

Na druhej strane, používateľ CaaS má pod plnou kontrolou proces, ktorým sa z aplikácie stane kontajner a na platformu nahráva svoj predpripravený

[3] https://kubernetes.io/

kontajner s aplikáciou. Táto odlišnosť umožňuje beh komplexnejších aplikácií, prakticky bez obmedzenia na použité technológie, ale taktiež predstavuje nutnosť vynaložiť viac úsilia pre nasadenie aplikácie.



Obr. 1.1: Znázornenie charakteristík IaaS, CaaS a PaaS.

1.3.1 Pod

Pod je základný stavebný blok. Je to najmenšia a najjednoduchšia jednotka v objektovom modeli Kubernetes. Reprezentuje bežiaci proces v clustri. Pod zastrešuje aplikačný kontajner (v niektorých prípadoch viacero kontajnerov), zdroje úložiska, unikátnu IP adresu a konfiguráciu, ktoré majú vplyv na beh kontajneru. Najpoužívanejší kontajnerizačný runtime pre Pody je Docker. [9]

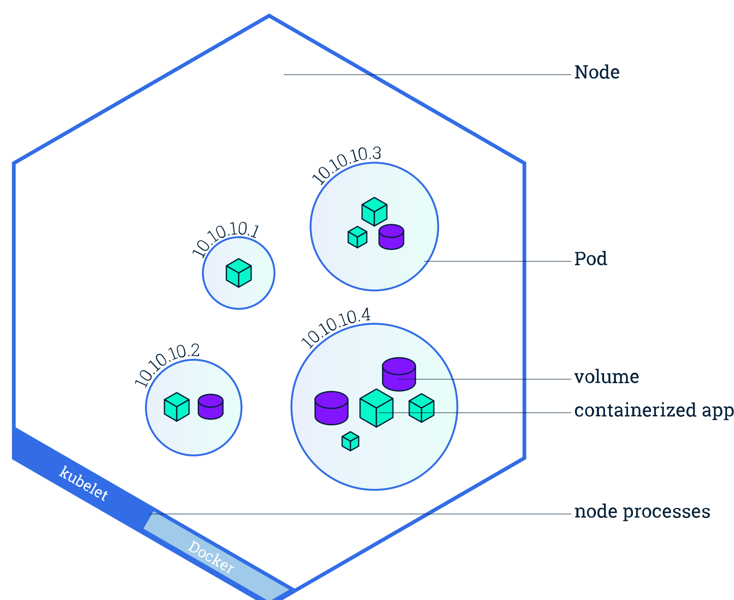
Pody v Kubernetes môžu byť použité dvomi hlavným spôsobmi:

- Pody v ktorých beží len jeden kontajner - Jedná sa o najčastejší use-case. Pod sa dá chápať ako wrapper okolo kontajneru. Kubernetes nespravuje priamo samotný kontajner, ale práve Pod.

- Pody v ktorých beží viacero kontajnerov - Tieto kontajnery najčastejšie spolu úzko spolupracujú a potrebujú zdielať zdroje. Príkladom môže byť kontajner, ktorý predstavuje web server pre prístup k súborom na zdieľanej diskovej jednotke. Ďalší, samostatný kontajner sa môže starať o aktualizáciu týchto súborov zo vzdialených zdrojov. Cieľom je umožniť beh pomocného kontajneru vedľa aplikačného.

Každý pod má na starosti jednu inštanciu danej aplikácie. Pre horizontálne škálovanie aplikácie sa využíva viacero Podov. Tento proces sa v Kubernetes všeobecne nazýva replikácia. Replikované Pody sú zvyčajne vytvárané

a spravované ako skupina prostredníctvom vyššej vrstvy abstrakcie nazývanej kontrolér (Controller).



Obr. 1.2: Znázornenie architektúry behu Podov na Kubernetes uzloch (zdroj: [1])

1.3.1.1 Priradenie Podov na uzly

Pre dosiahnutie vysokej dostupnosti aplikácie je počas škálovania dôležité, aby sa inštancie aplikácie (repliky) spúšťali na rôznych uzloch. Ak by všetky repliky bežali na jednom uzle a uzol havaruje, aplikácia sa stane nedostupnou. Kubernetes štandardne rozmiestňuje Pody na uzly s dostatkom systémových prostriedkov rovnomerne. [10]

Proces alokácie Podov na uzly je možné ďalej konfigurovať a napríklad zamedziť vybraným Podom beh na rovnakom uzle nastavením anti-afinita pre Pody a špecifikovaním labelu, ktorým sú značené Pody, ktoré sa nemajú spúšťať na rovnakom uzle. Naopak, v prípade komunikácie aplikácie s in-memory cache je žiadané, aby Pod s aplikáciou a Pod s cache bežali na rovnakom uzle pre čo najnižšiu latenciu.

Podobný mechanizmus existuje aj pre uzly. Vďaka nemu je možné špecifikovať na ktorých uzloch môže, respektíve nemôže, daný Pod bežať a to na základe labelu uzlu. V Kubernetes existujú 2 typy afinity, respektíve anti-afinity.

- requiredDuringSchedulingIgnoredDuringExecution - Pravidlo, ktoré musí byť splnené
- preferredDuringSchedulingIgnoredDuringExecution - Pravidlo, ktoré vyjadruje preferenciu. Scheduler sa ho pokúsi dodržať, ale negarantuje to. Anti-afinita v tomto prípade môže prestavovať napríklad rozloženie Podov do rôznych zón dostupnosti.

1.3.1.2 Health checking

Pri nasadzovaní Podu Kubernetes ponúka možnosť definovať 2 typy sond, liveness a readiness. Jedná sa o príkazy, spustiteľné scripty, HTTP/TCP požiadavky ktoré svojim návratovým kódom (v prípade TCP ide o kontrolu otvoreného portu) indikujú funkčnosť nasadenej kontajnerizovanej aplikácie. V prípade vrátenia chybového návratového kódu môže Kubernetes uskutočniť akcie, ktoré napomôžu k návratu aplikácie do zdravého stavu. Pri sondách je možné nastaviť niekoľko parametrov

- Interval spúšťania je možné špecifikovať nastavením poľa `periodSeconds`, respektíve `periodSeconds` v `spec.readinessProbe`, štandardná hodnota je 10.
- Prvú kontrolu je možné zdržať špecifikovaním `initialDelaySeconds`.
- `successThreshold` (štandardne 1), respektíve `failureThreshold` (štandardne 3) určujú počet úspešných, respektíve neúspešných kontrol v rade potrebných na dosiahnutie celkového úspechu alebo neúspechu behu.

Readiness sonda slúži pre prípad, keď aplikácia beží korektne, ale ešte nie je pripravená príjimať požiadavky (traffic). Podu, ktorý pomocou readiness sondy indikuje nepripravenosť, nie sú prostredníctvom `Service` smerované žiadne požiadavky. Ak kontajner neposkytuje túto sondu, Pod sa považuje za pripravený.

Liveness sonda indikuje korektnosť behu kontajneru. Ak sonda vráti chybový návratový kód, Kubernetes ukončí kontajner a ten je následne reštartovaný. Kubernetes automaticky reštartuje kontajner v ktorom sa proces aplikácie neukončí korektne. Liveness sonda umožňuje zachytiť tie prípady, pri ktorých proces aplikácie stále beží, ale samotná aplikácia sa nachádza v nežiadúcom stave a je potrebný jej reštart. [11]

1.3.1.3 Init kontajner

Pod môže zastrešovať viacero kontajnerov v ktorých bežia aplikácie a taktiež môže mať jeden alebo viacero Init kontajnerov, ktoré bežia pred aplikačnými kontajnermi. Init kontajnery sú rovnaké ako obyčajné kontajnery s tým rozdielom, že musia stále dobehnúť a nasledujúci kontajner nie je spustený pred

úspešným dobehnutím predošlého. V prípade neúspešného behu Init kontajneru je Pod reštartovaný až dokiaľ nie je beh úspešný. Init kontajnery sú využívané napríklad na predprípravu prostredia pre beh aplikácie, klonovanie git repozitára, pozdržanie spustenia hlavného aplikačného kontajneru, príprava konfiguračných súborov a iné.

1.3.2 Kontroléry

Samotný Pod je neperzistentný objekt a v prípade pádu uzlu na ktorom beží sa sám neobnoví. Kontrolér sa stará o vytvorenie a správu podov, tiež o ich replikáciu a self-healing pre zachovanie vysokej dostupnosti. Napríklad, ak sa vyskytne problém na uzle, kontrolér môže automaticky nahradiť Pod spustením jeho náhrady na inom uzle. Myšlienkou Controlleru je definovanie požadovaného stavu kontajnerov, ktorý sa kontrolér následne snaží udržať. Pre kontroléry ako je Pod, StatefulSet alebo Deployment je možné špecifikovať príkazy, ktoré sa vykonajú po štarte (post-start hook) alebo pred ukončením kontajneru (pre-stop hook). [12]

1.3.2.1 Deployment

Kontrolér vhodný pre nasadenie stateless aplikácie. Využitie Deployment kontroléru je odporúčaný spôsob pre vytvorenie `ReplicaSet`, ktorý následne spravuje Pody. Deployment ponúka deklaratívny spôsob kontroly procesu aktualizovania aplikácie. Je možné špecifikovať maximálny počet nedostupných Podov, rôzne stratégie, kde sa pri nasadení novej verzie kontroluje jej funkčnosť a v prípade problému sa proces aktualizácie zastaví. Tiež je možné nasadiť rôzne revízie deploymentu, čo umožňuje blue/green nasadenie alebo aj canary release. Navyše Deployment uchováva históriu revízií, ktoré je možné využiť pre jednoduchý rollback na staršiu verziu. [13]

1.3.2.2 StatefulSet

Ide o kontrolér určený, ktorý adresuje potrebu správy stateful aplikácií. Narozdiel od Deploymentu, StatefulSet udržuje informáciu o identite jednotlivých Podov a zaručuje konzistentné poradie ich vytvárania a zmazania. Každá replika má pridelený číselný index začínajúci nulou a končiaci počtom replík. Je užitočný pre aplikácie vyžadujúce stabilný (naprieč (re)schedulingu Podov) unikátny sieťový identifikátor - hostname je v tvare $(statefulsetname)-$(index), perzistentné úložisko, nasadenie a škálovanie Podov (či už smerom nahor alebo nadol) v konzistentnom poradí.

Pre priradenie perzistentného úložiska je možné využiť koncept `VolumeClaimTemplate`. Pre každý Pod zo StatefulSetu sa vytvorí perzistentné úložisko podľa tejto špecifikácie. Je možné vyžiadať použitie špecifického typu úložiska (vlasnosť `storageClassName`), prístupový mód, požadovanú veľkosť úložiska a iné. [14]

1.3.2.3 DaemonSet

DaemonSet zabezpečí, aby na každom uzle bežala kópia Podu. Pri pridávaní uzlov do clustru sú na nich spustené dané Pody. Typickým príkladom použitia je:

- prevádzka agentov pre clustrové úložiská (napr. ceph[4])
- zber logov z uzlov (napr. logstash[5], fluentd[6])
- monitoring uzlov (napr. collectd[7])

1.3.2.4 Job a CronJob

Job vytvorí jeden alebo viacero Podov a zabezpečí úspešný beh a ukončenie špecifikovaného počtu. Udržuje prehľad o počte úspešných behov. Po dosiahnutí tohto stavu sú vytvorené Pody ukončené. Využitie spočíva pri kontajneroch, ktoré nemusia bežať kontinuálne, ale len jednorázovo. Tento koncept je tiež možné využiť na beh viacerých jednorázových Podov paralelne. CronJob je rozšírenie tohto konceptu o plánované spúšťanie v špecifikovanom intervale zadanom v štandardnom cron formáte. Takto bežiace úlohy by mali byť idempotentné kvôli problémom s nechceným viacnásobným spustením. [15] Medzi najčastejšie spôsoby využitia CronJob patria napríklad pravidelné zálohovanie databáz alebo odosielanie emailov.

1.3.3 Volumes

Súbory na lokálnom disku v kontajneri nie sú uložené perzistentne. Pri páde kontajnera dôjde k jeho reštartu a takto uložené dáta budú stratené. Ďalším problémom, ktorý Volumes v Kubernetes riešia je zdieľanie dát medzi kontajnermi bežiacimi na rovnakom Pode. [16]

Kubernetes podporuje množstvo typov diskových uložísk. Volume pluginy predstavujú spôsob, akým Kubernetes komunikuje s externými úložiskami. Vďaka týmto pluginom podporuje aj uložiská špecifické pre konkrétnych poskytovateľov Kubernetes platformy ako sú napríklad Amazon Elastic Block Store (EBS), Microsoft Azure Data Disk, Google Compute Engine Persistent Disk . Možné je tiež využiť vlastné úložisko postavené na Ceph alebo NFS. Aby bola požiadavka na vytvorenie perzistentného úložiska nezávisla na platforme, je možné využiť objekt `persistentVolumeClaim`.

[4] `https://ceph.com/`
[5] `https://www.elastic.co/products/logstash`
[6] `https://www.fluentd.org/`
[7] `https://collectd.org/`

PersistentVolume Tento subsystém poskytuje API, ktoré pridáva úroveň abstrakcie nad špecifikami jednotlivých typov diskových úložísk. PersistentVolume (PV) je úložisko, ktoré bolo do Kubernetes sprístupnené administrátorom. Výhodou je, že životný cyklus je nezávislý od podov, ktoré PV používajú.

PersistentVolumeClaim (PVC) Je požiadavka od používateľa na získanie úložiska. Princíp je podobný ako pri Pode. Pod konzumuje zdroje typu node, PersistentVolumeClaim zasa konzumuje zdroje typu PersistentVolume. Takáto požiadavka môže špecifikovať požadovanú veľkosť úložiska a prístupový mód.

Alokácia diskovej jednotky prebieha buď staticky alebo, od Kubernetes 1.6, aj dynamicky. Pri statickej alokácií je nutné, aby administrátor dopredu pripravil niekoľko PV, ktoré nesú detaily o reálnom úložisku, ktoré je k dispozícií pre používateľov clustru. V prípade dynamického provisioningu, sa vďaka konceptu `StorageClass`, cluster môže pokúsiť o dynamické vytvorenie diskovej jednotky podľa požiadavkov používateľa.

- V prípade nešpecifikovania poľa `storageClassName` v `PersistentVolumeClaim` bude na vytvorenie diskových jednotiek použitá štandardná StorageClass nastavená administrátorom Kubernetes platformy.
- Ak je `storageClassName` nastavená na prázdny string, nebude použitá žiadna `StorageClass`, dynamický provisioning je týmto vypnutý. Na uspokojenie PVC sa využijú už existujúce PVs (ktoré nemajú špecifikovanú vlasnosť `storageClassName`).
- Ak je `storageClassName` nastavená na špecifickú hodnotu, na provisioning diskovej jednotky sa použije odpovedajúca `StorageClass`.

Kedže cieľom dynamického provisioningu je automatizovať životný cyklus diskových zdrojov, štandardne sú dynamicky vytvorené PV zmazané spolu s Podom. Znamená to, že pri uvoľnení PVC je dynamicky zriadený disk odstránený poskytovateľom storagu. Túto vlastnosť je možné konfigurovať zmenou `persistentVolumeReclaimPolicy` na `retain` v konfigurácií PV.

Amazon Elastic Block Store (EBS) [8] Jedná sa o diskové úložisko špecifické pre cloud Amazon Web Services (AWS). Pri zmazaní podu dáta zostavajú zachované, disková jednotka sa len odpojí (unmount). Pred použitím diskovej jednotky Kubernetes je najprv nutné ju v prostredí AWS vytvoriť. Na to je možné použiť konzolový nástroj `aws` alebo využiť AWS API.

Špecifikovaním typu jednotky je možné zvoliť SSD a HDD. SSD disky sa delia na 2 skupiny, disky pre všeobecné využitie (typ gp2) a vysokovýkonné disky s nízkou latenciou a vysokou priepustnosťou (typ io1), ktoré su odporučané pre databázové systémy ako je MongoDB alebo Cassandra. [17]

Pri použití je nutné myslieť na niektoré obmedzenia:

[8] `https://aws.amazon.com/ebs/`

- Uzly ku ktorým je disková jednotka pripojená musia byť inštancie typu AWS EC2
- Tieto inštancie musia byť v rovnakom regióne a zóne dostupnosti ako EBS jednotky
- Jedna disková jednotka môže byť naraz pripojená len k jednej EC2 inštancií

Google Cloud Engine Persistent Disk [9] Diskové úložisko špecifické pre Google Cloud. Pri zmazaní Podu, rovnako ako pri AWS, dáta ostavajú zachované a disková jednotka je len odpojená. Diskovú jednotku je taktiež treba pred použitím v Kubernetes vytvoriť manuálne a to buď využitím CLI nástroja `gcloud` alebo GCE API.

Pri použití je nutné myslieť na niektoré obmedzenia:

- Uzly ku ktorým je disková jednotka pripojená musia byť GCE VM
- Tieto inštancie musia byť v rovnakom GCE projekte a zóne ako perzistentné disky.

Výhoda oproti Amazon EBS je v možnosti pripojiť rovnaký disk k viacerým uzlom v read-only režime. Pre read-write režim platí rovnaké obmedzenie ako pre Amazon EBS.

NFS Network File System je protokol pre zdieľanie súborov prostredníctvom počítačovej siete. Pred použitím je nutné mať k dispozicií alebo vybudovať vlastný NFS server. NFS Volume v Kubernetes umožňuje existujúcim NFS bodom ich pripojenie ako diskovej jednotky do Podu. Pri zmazaní Podu je obsah takto pripojeného NFS úložiska zachovaný. Umožňuje pripojenie k viacerým Podom súčasne.

HostPath HostPath volume pripojí súbor alebo zložku z hosťovského systému Kubernetes uzlu do daného Podu. Najčastejšie využitie je pre prístup k Docker konfigurácií na hosťovskom systéme. Je možné špecifikovať konkrétny typ jednotky (zložka, súbor, socket, blokové zariadenie a iné).

Pri behu Podu na rôznych uzloch môžu nastať komplikácie pretože obsah pripojenej zložky sa nemusí zhodovať. Vytvorené zložky a súbory na hosťovskom systéme sú zapisovateľné len rootom. Pre zápis do hostPath diskovej jednotky je nutné proces v kontajneri spustiť pod užívateľom root v privilegovanom kontajneri (proces v kontajneri dostane takmer také práva ako proces na hosťovskom systéme [9]) alebo zmeniť práva na hosťovskom systéme.

[9] `https://cloud.google.com/persistent-disk/`

Local Lokálna disková jednotka reprezentuje pripojenie lokálneho zariadenia akým je disk, partícia alebo zložka, ktorý je viazaný na uzol. V súčastnosti (Kubernetes 1.9) ovládač pre lokálne diskové jednotky nepodporuje dynamický provisioning. Je dôležité brať na vedomie, že pri výpadku uzlu je nedostupná aj lokálna disková jednotka, ktorú Pod využíval. Dôsledkom je že Pod nemôže byť spustený na inom uzle bez straty dát.

Na správne priradenie Podov, ktoré využívajú takúto lokálnu diskovú jednotku, na uzly je nutné pri definícii `PersistentVolume` konfigurovať vlastnosť `nodeAffinity`. Pri zmazaní Podu nedochádza k zmazaniu diskovej jednotky (zložky) z uzlu, na ktorom Pod bežal. [16]

Secret Jedná sa o špeciálny typ diskovej jednotky, určenej na predávanie citlivých údajov, akými sú napríklad heslá alebo tokeny do aplikácií bežiacich v Podoch. Tieto citlivé dáta je možné uložiť do Kubernetes API a pripojiť ich ako súbory do Podov bez priamej závislosti na Kubernetes API.

Diskové jednotky tohto typu su založené na tmpfs (dáta uložené v RAM). Pre vývojára tak nie je nutné poznať Kubernetes API, kedže k týmto dátam môže pristupovať z aplikácie ako ku klasickým súborom.

Možné je tiež manuálne vytvorenie cez konfiguračný súbor formátu YAML. Kde sa zapíšu hodnoty username a password zakodované v base64.

```
apiVersion: v1
kind: Secret
metadata:
  name: mysecret
type: Opaque
data:
  username: YWRtaW4=
  password: MWYyZDFlMmU2N2Rm
```

Ukážka 1.1: Obsah konfiguračného súboru secret.yaml

Nakoniec, pre pripojenie diskovej jednotky do podu je potrebné v konfiguračnom súbore podu vykonať nasledovné:

1. Pridanie diskovej jednotky do `spec.volumes[]`. `spec.volumes[].secret.secretName` musí odpovedať pomenovanie Secret objektu (db-user-pass). Názov jednotky je možné zvoliť ľubovoľne.

2. Pridanie `spec.containers[].volumeMounts[]` do každého kontajneru, kde plánujeme citlivé dáta využívať

3. Každý kľúč z data mapy (username a password) sa stane súborom umiestneným pod `mountPath`

1.3.4 Service

Každému Podu je pridelená jeho vlastná IP adresa vrámci clustru, avšak nie je garantovaná jej statickosť a taktiež nie je pristupná mimo clustru. Service v Kubernetes je prvok, ktorý definuje logickú skupinu Podov a politiku akou k nim pristupovať. Pre výber Podov sa využíva label selektor. Môže slúžiť na load balancing, service discovery alebo ako premostenie medzi vonkajším svetom a Kubernetes clustrom.

Typickým použitím je situácia, kde existujú 2 skupiny uzlov. Jedna skupina sa stará o frontend a druhá o backend služby. Uzly z frontend skupiny potrebujú komunikovať s uzlami z backend skupiny. Kedže sa Pody, ktoré zabezpečujú fungovanie backendu môžu meniť, frontend uzly by museli držať zoznam aktivných backend uzlov svojpomocne.

Na každom uzle v Kubernetes clustri beží komponenta `kube-proxy`, ktorá je zodpovedná za implementáciu virtuálnej IP pre Service každého typu (okrem `ExternalName`). Nájdenie a kontaktovanie Service je možné prostredníctvom premenných prostredia alebo cez DNS. Druhá možnosť je cluster addon, ktorý však väčšinou býva nainštalovaný na Kubernetes platforme. V prípade použitia iného spôsobu service discovery, nepotreby load balancingu a jednej IP adresy je možné využiť tzv. headless service. [18]

Service môže byť sprístupnená niekoľkými spôsobmi:

- Cluster IP - Sprístupní Service na internej IP adrese v clustri. Je prístupná len z vnútra clustra.
- NodePort - Sprístupní Service na rovnakom statickom porte (defaultne z rozsahu 30000-32767) na každom z vybraných uzlov a smeruje traffic na Cluster IP Service, ktorá je automaticky vytvorená. Kontaktovanie Service bude možné cez <nodeIP>:<NodePort>
- LoadBalancer - Sprístupní Service externe s využitím load balancingu od poskytovateľa cloudu. Pri tomto type sa automaticky vytvára ClusterIP a NodePort Service.
- ExternalName - Namapuje službu na nakonfigurovanú adresu vratením CNAME záznamu

1.3.4.1 Headless Service

Headless Service sa definuje použitím hodnoty `None` pre `spec.clusterIP` v konfigurácií Service. Pri tejto konfigurácií nie je alokovaná cluster IP, kube-proxy sa o túto Service nestará, samotnou platformou nie je poskytnutý žiadny load balancing alebo proxy. Konfigurácia DNS závisí od využitia selektorov v konfigurácií Service. S definovaným selektorom sa vytvorí Endpoint objekt v API a do DNS sa pridá A záznam, ktorý ukazuje na Pody spravované danou Service. Bez využitia selektoru:

- Nevytvorí sa Endpoint objekt v API
- Pre ExternalName Service sa pridá CNAME záznam
- Pre ostatné typy Service sa pridá A záznam pre každý Endpoint, ktorý zdieľa názov so Service

Headless Service je možné využiť pre správu domény pre Pody zo StatefulSetu. Takto spravovaná doména bude v tvare $(názov service).$(namespace).svc.-cluster.local, kde cluster.local je štandardná doména clustru, ktorú je možné konfigurovať. Každý Pod dostane záznam v tvare $(názov podu).$(správcovská doména), kde správcovská doména je definovaná poľom `serviceName` na StatefulSete.

1.4 Škálovanie

Škálovanie sa vo všeobecnosti realizuje dvoma spôsobmi. Horizontálnym a vertikálnym škálovaním. Vertikálne škálovanie sa realizuje zmenou množstva pridelených systémových prostriedkov. Tento spôsob je jednoduchší a väčšinou nevyžaduje žiadne úpravy softwaru. Možnosť škálovania tohto typu je značne obmedzená a od určitého momentu aj finančne nevýhodná. Horizontálne škálovanie predstavuje zmenu počtu inštancií aplikácie, služby alebo dokonca celých serverov.

V dnešnej dobe viac procesorových systémov so značným množstvom operačnej pamäte často nastáva situácia, keď výkon jedného servera poskytuje dostatok priestoru pre vertikálne škálovanie. Je však potrebné zvážiť ďalšie faktory, ktoré hovoria v prospech horizontálneho škálovania. [19]

Tým je napríklad vysoká dostupnosť a redundancia. V prípade výpadku jednej inštancie aplikácie alebo servera systém zostáva stále funkčný. Týmto sa vyhneme tzv. single point of failure, čo je stav pri ktorom funkčnosť celého systému závisí na jednom prvku, bode.

Vertikálne škálovanie je obmedzené aktuálne dostupným hardwarom a taktiež jeho cenou, ktorá stúpa exponenciálne v závislosti od výkonu komponenty a získaný výkon jej nemusí zodpovedať. [20] Horizontálne škálovanie naopak poskytuje priestor na optimalizáciu ceny a výkonu hardwaru. Do prevádzky sa môžu zapojiť menej výkonné, ale o to lacnejšie servery, ktoré pri vyššom počte poskytnú porovnateľný, alebo aj vyšší výkon za nižšiu cenu oproti jedinému serveru s výkonným a drahým hardwarom. [19] Pre horizontálne škálovanie sa v dnešnej dobe často využíva, namiesto pridávania celých fyzických serverov, pridávanie virtuálnych serverov. Pri tejto metóde je treba zabezpečiť aby virtuálne servery boli rozložené na viacero fyzických serverov alebo zón dostupnosti pre zamedzenie úplnej havárie systému v prípade zlyhania jediného fyzického serveru.

V databázových systémoch sa pre horizontálne škálovanie využívajú metódy ako je master-slave, multi-master replikácia alebo sharding.

1.4.1 Master-slave replikácia

Jedná sa o horizontálne škálovanie. Mnoho vývojárov využíva master-slave replikáciu na riešenie hneď niekoľkých problémov. Ide o problémy s výkonom, podporu zálohovania databázy alebo systémové havárie. Táto technika je typická pre relačné databázy (MySQL). [21]

Master-slave replikácia umožňuje replikovanie dát z jedného serveru (master) na jeden alebo viacero ďalších databázových serverov (slaves). Master server sa stará o spracovanie požiadaviek na zápis alebo čítanie. Slave servery neobsluhujú požiadavky na zápis, ale len na čítanie. Tým pádom je zvyšovaním počtu slave serverov škálovať priepustnosť operácií na čítanie.

Slave servery sú často využívané na beh dlho trvajúcich požiadaviek pre vytváranie BI (Business Intelligence) report. Výhodou je tiež zálohovanie databázy zo slave serveru bez ovplyvnenia výpadku master serveru. [22]

Táto architektúra si so sebou nesie svoje nevýhody. V prípade výpadku master serveru musí byť master nahradený slave serverom, čo sa nezaobíde bez výpadku, prípadne straty dát. Aplikácia využivajúca databázu musí byť prispôsobená na odosielanie operácií k zápisu na master server a čítania slave serverom pre rozloženie záťaže. Každé pridanie slave serveru zvyšuje záťaž na master servery kvôli kopírovaniu dát na každý slave. [23]

1.4.2 Multi-master replikácia

Taktiež prípad horizontálneho škálovania. Podobne ako pri master-slave, dáta su replikované na ďalšie uzly. Rozdielom je, že operácie na zápis môže spracovávať viacero, prípadne ktorýkoľvek, uzol. Čím sa záťaž rozloží medzi uzly. V prípade výpadku uzlu je jeho náhrada jednoduchá, keďže požiadavku môže spracovať ktorýkoľvek uzol. Vďaka možnosti zápisu na ktorýkoľvek uzol nie sú potrebné veľké úpravy aplikácií využívajúcich databázový systém. [24]

Pre dosiahnutie vysokej dostupnosti v prostredí cloudu môže byť multi-master replikácia žiadanejšia než master-slave replikácia. Zlyhanie master uzlu pri master-slave replikácií je spojené s výpadkom, je potrebná detekcia zlyhania master uzlu a jeho nahradenie.

1.4.3 Sharding

Sharding je metóda, pre rozdelenie veľkých dat na viaceré stroje. (shards). [2] Sharding umožňuje nasadenie veľkých datasetov a zachovanie vysokej priepustnosti operácií. Jedná sa o prípady s ktorými by jeden server mohol mať výkonnostný problém, či už kvôli obmedzeniu na strane výkonu procesora, kapacity disku alebo operačnej pamäte. Každý server funguje ako samostatná databáza a spolu tvoria jednu logickú databázu. Využitím shardingu je možné rozložiť spracovanie operácií medzi jednotlivé stroje čím sa zvýši celková priepustnosť požiadaviek na čítanie alebo zápis. Vďaka využitiu viacerých serverov sa tiež znižujú nároky na diskový priestor jednotlivých serverov (databáza

o veľkosti 1TB môže byť rozdelená medzi 4 stroje, kde každý stroj uchováva 256GB).

1.5 MongoDB

MongoDB[10] je NoSQL (Not only SQL) databázový systém, ktorý sa vyznačuje vysokou výkonnosťou, dostupnosťou a automatickým škálovaním. Záznam v MongoDB sa nazýva dokument. Samotné dokumenty sú uložené v kolekciách (collections). Kolekcie sú analógiou k tabuľkám v relačných databázových systémoch a sú ukladané v databázach. [25]

Na automatický failover a redundanciu dát je v MongoDB možné využiť koncept nazývaný replica set. Jedná sa o skupinu MongoDB serverov, ktoré spravujú rovnaké dáta a poskytujú tak redundanciu a zvýšenú dostupnosť.

MongoDB ponúka podporu pre horizontálne škálovanie ako jednu zo základných funkcií. Na rozloženie dát naprieč clustrom sa využíva sharding. Vďaka čomu je možné uchovávať veľké datasety a dosiahnuť vysokú priepustnosť operácií.

1.5.1 Replikácia dát

MongoDB replikácia dát, od verzie MongoDB 3.6, nahrádza master-slave metódu replikácie. Prináša vyššiu robustnosť, viac redundancie a automatický failover (automatická náhrada havarovaného uzlu). [26] Replikácia poskytuje redundanciu dát a zvyšuje dostupnosť. [27] S viacerými kópiami dát na rôznych databázových serveroch replikácia poskytuje poistku proti strate jedného alebo viacerých databázových serverov. V niektorých prípadoch môže zvýšiť priepustnosť čítania vďaka tomu, že klienti odosielajú požiadavky na čítanie na rôzne servery. Uchovávanie dát v rôznych datacentrách tiež môže prispieť k lepšej prístupnosti k dátam a zvýšenej dostupnosti aplikácie, ktorá je distribuovaná (beží na viacerých strojoch). Viacero kópií dát tiež môže slúžiť ako záloha.

Replica set je skupina inštancií (`mongod` procesov), ktoré uchovávajú rovnaké dáta. Jedna takáto skupina obsahuje niekoľko uzlov s dátami a voliteľným arbiter uzlom (uzol, ktorý nespravuje žiadne dáta, ale hlasuje pri výbere primárneho uzlu). Jeden z dátových uzlov je určený ako primárny. K nemu smerujú všetky operácie k zápisu a zmeny sa zapisujú do operačného logu (oplog). Ostatné uzly sú sekundárne a asynchrónne replikujú zmeny v primárnom uzly. Ak je primárny uzol nedostupný, jeden zo sekundárnych sa zvolí ako primárny.

Štandardne klient číta z primárneho uzla. Existuje však niekoľko možností:

- primary - Čítanie z primárneho uzla

[10] `https://www.mongodb.com/`

- primaryPreferred - Preferované čítanie z primárneho uzla. V prípade jeho nedostupnosti čítanie prebieha zo sekundárneho uzla
- secondary - Cítanie zo sekundárnych uzlov.
- secondaryPreferred - Podobne ako primaryPreferred, preferované čítanie zo sekundárnych uzlov.
- nearest - Čitanie z uzla z najnižšou sieťovou latenciou.

Kvôli asynchrónnej replikácii dáta na sekundárnom uzle nemusia odrážať aktuálny stav primárneho uzlu.

1.5.2 Voľba primárneho uzlu

Voľba primárneho uzlu priebieha pri vytvorení replica setu a v prípade nedostupnosti doposiaľ primárneho uzlu. Kedže primárny uzol je jediný uzol, ktorý prijíma operácie k zápisu, automatická voľba nového primárneho uzlu umožňuje zachovať dostupnosť clustru bez väčšieho zásahu. Túto voľbu iniciuje jeden zo sekundárnych uzlov. Taktiež ak sa primárnemu uzlu javí väčšina uzlov v replica sete ako nedostupná, uzol zmení svoj typ na sekundárny.

Počas voľby môže nastať obdobie, kedy cluster nemá primárny uzol a teda nie je možné do databázy zapisovať, avšak čítanie je stále umožnené zo sekundárnych uzlov.

Uzly v replica sete medzi sebou zasielajú heartbeat (pingy) každé 2 sekundy. Ak sa heartbeat nevráti do 10 sekúnd, ostatní členovania považujú uzol za nedostupný.

1.5.3 Sharding

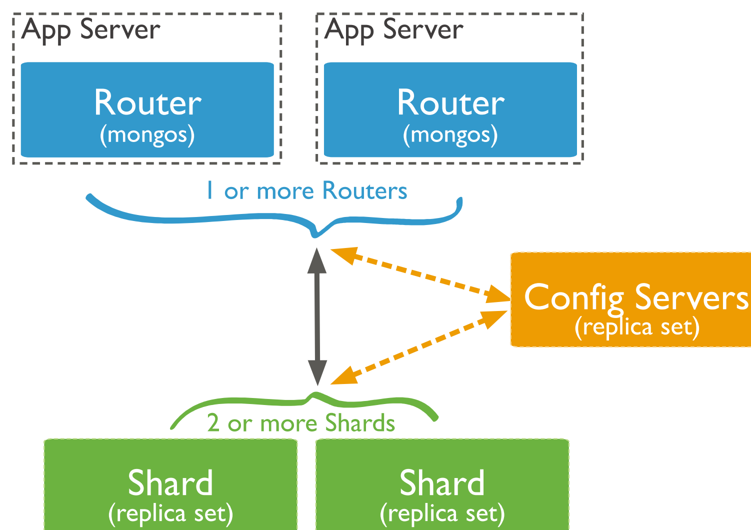
MongoDB má podporu pre horizontálne škálovanie využitím metódy rozdelenia dát, sharding. Pre dosiahnutie vysokej dostupnosti je vhodné aby každý shard bol `replica set`.

Rozdelenie dát do shardov sa realizuje na základne tzv. shard kľúča. Jedná sa o nemenné pole alebo polia, ktoré sa nachádzajú v každom dokumente s shardovanej kolekcii. Výber vhodného kľúča ovplyvňuje výkonnosť, efektivitu a škálovateľnosť shardovaného clustra. [2]

Mongo shard cluster pozostáva z nasledujúcich komponent:

- shard - Každý shard obsahuje časť shardovaných dát. Musí byť nasadený ako replica set.
- mongos - Router, ktorý smeruje požiadavky do clustru. Jedná sa o interface medzi klientskými aplikáciami a shard clustrom. Udržuje prehľad o tom, aké dáta sa nachádzajú na akom uzle a to využíva pri smerovaní požiadaviek na mongod inštancie. Nemá perzistentné stav a nie je náročný na systémové prostriedky. [28]

- config servery - Uchovávajú metadata a konfiguráciu potrebnú pre beh clustru. Inštancie mongos tieto metadáta cachujú a využívajú pre smerovanie požiadaviek na čítanie a zápis na správny shard. Config servery musia byť nasadené ako replica set a každý shard cluster potrebuje vlastný config server. Dostupnosť konfiguračných serverov je kritická pre správne fungovanie shard clustru. [29]



Obr. 1.3: Diagram znazorňujúci komunikáciu vo vnútri shardovaného clustra. (zdroj: [2])

1.5.4 Nasadenie a konfigurácia

1. V prvom kroku je nutné vytvoriť replica set pre config servery
   1.1. Naštartovanie každého mongod procesu s prepínačmi `-configsvr`, `--replSet`, `--bind_ip` alebo s využitím konfiguračného súboru a prepínača `--config <cesta_k_suboru>`
   1.2. Pripojenie na jeden z config serverov pomocou `mongo --host <hostname> --port <port>`
   1.3. Inicializácia replica setu. Z mongo shellu príkazom `rs.initiate()`, ktorá ako voliteľný parameter berie konfiguráciu v json formáte, kde je možné špecifikovať názov replica setu (pole `_id`), zoznam členov (pole `members`). V prípade konfigurácie config serveru je potrebné nastaviť boolean hodnotu poľa `configsvr` na true.
2. Vytvorenie shard clustru

   2.1. Naštartovanie mongod procesu na každom uzle s prepínačmi --shardsvr, --replSet, --bind_ip alebo s využitím konfiguračného súboru

   2.2. Pripojenie na jeden z config serverov pomocou mongo --host <hostname> --port <port>

   2.3. Inicializácia replica setu, podobne ako pri konfigurácií config serverov.

   2.4. Pripojenie mongos do clustru spustením procesu s prepínačmi --configdb a --bind_ip alebo konfiguračným súborom. Pripojenie na mongos inštanciu a použitie príkazu sh.addShard() na pridanie každého shard uzlu do clustru. Ak je shard replica set špecifikuje sa názov replica setu.
   sh.addShard(«replSetName>/s1-mongo1.example.net:27017")

3. Zapnutie shardingu pre databázu sh.enableSharding(«database>")

1.6 Cassandra

Apache Cassandra[11] je vysoko škálovateľná distribuovaná NoSQL databáza založená na ColumnFamily (stĺpcovo orientovanom) dátovom modely. Je navrhnutá pre beh na veľkom počte uzlov aj viacerými naprieč datacentrami.

Každý uzol vymieňa informácie o sebe a ostatných uzloch v clustri pomocou peer-to-peer komunikačného protokolu, gossip. Táto procedúra je spúšťaná každú sekundu a každý uzol komunikuje s najviac 3 ďalšími. Seed uzly slúžia na prvotné naštartovanie komunikačného procesu medzi novým uzlom a ostatnými uzlami v clustri. Vzťahuje sa k nim pár odporučaní [30]:

- Každý uzol by mal mať rovnaký zoznam seed uzlov.
- V prípade prevádzky viacerých datacentier je pre robustnosť systému tiež vhodné definovať viac než 1 seed uzol na datacentrum. Inak by v prípade nedostupnosti seed uzlu z lokálneho datacentra musel uzol komunikovať so seed uzlom z druhého datacentra.
- Používať malý počet seed uzlov, kvôli réžii spojenej s gossip protokolom.

1.6.1 Sharding

Dáta sú automaticky rozdistribuované naprieč clustrom. O rozdelenie dát sa stará partitioner. Ide o funkciu, ktorá vytvorí token reprezentujúci riadok na základe jeho partition kľúča, najčastejšie použitím hashovacej funkcie. [31] Každý node v clustri má rovnakú rolu. Požiadavky na čítanie alebo zápis

[11] http://cassandra.apache.org/

môžu byť smerované na ktorýkoľvek uzol. Úroveň konzistencie zápisu špecifikuje koľko uzlov musí potvrdiť zápis pre úspešné spracovanie požiadavky. Uzol, ktorý obsluhuje požiadavku (na čítanie alebo zápis) od klienta sa nazýva koordinátor. Má funkciu sprostredkovateľa medzi klientskou aplikáciou a uzlami na ktorých sa nachádzajú relevantné dáta pre danú požiadavku. Koordinátor zisťuje, na ktoré uzly má ďalej predať požiadavku. [32]

1.6.2 Replikácia

Dáta sú automatický replikované na viacero uzlov, ich množstvo špecifikuje replikačný faktor. Havarované uzly môžu byť rýchlo nahradené, bez výpadku. Konzistencia operácií je konfigurovateľná. [33] Ak máme napríklad cluster so 4 uzlami, replikačným faktorom 3 (dáta sa replikujú na 3 uzly), a levelom konzistencie quorum (operáciu musí potvrdiť väčšina uzlov), tak pri výpadku jedného uzlu sa dostupnosť clustru nenaruší (je stále dostupná väčšina uzlov) a klient na svoju požiadavku dostane odpoveď.

Replikačná stratégia určuje uzly na ktoré sú umiestnené repliky. [34]

- SimpleStrategy - Používaná len pre jeden rack a datacentrum. Nezohľadnuje sa topológia.
- NetworkTopologyStrategy - Špecifikuje počet replík v jednom datacentre. Snaží sa o rozmiestnenie replík v rovnakom datacentre na rôzne racky.

Pre multi datacentrové nasadenie sa v praxi odporúča používať len jeden rack a to z viacerých dôvodov. [35]

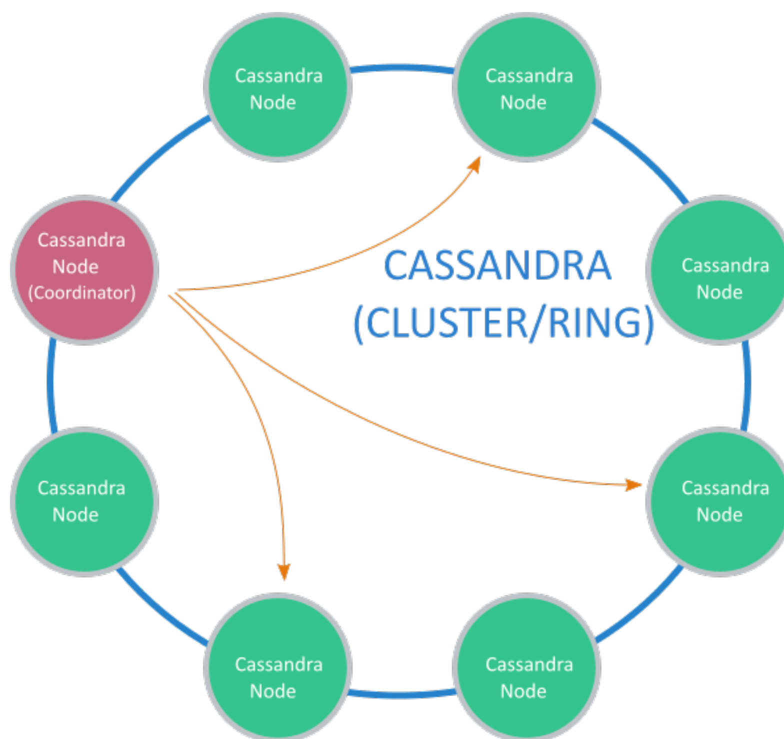
- Stáva sa, že užívatelia nasadenie viacerých rackov nevyužívajú efektívne a pre každý fyzický uzol vytvárajú samostatný rack
- Pre korektné použitie by každý rack mal mať rovnaký počet uzlov
- Naročná expanzia clustru

1.6.3 Snitching

Snitch má v Cassandre 2 úlohy. Získavanie informácií o topológii siete pre efektívne smerovanie požiadaviek a umožňuje rozloženie replík naprieč clustrom pre zvýšenie dostupnosti pri lokálnych výpadkoch. Umiestňuje uzly do skupín ako je datacentrum a rack. Snahou je nemať v rovnakom racku viac než jednu repliku. [36]

Dynamický snitching monitoruje latenciu čítania pre zamedzenie čítania z pomalých uzlov.

Na rozhodovanie či sú uzly v rovnakom datacentre alebo na rovnakom racku Cassandra ponúka niekoľko implementacií snitchu:



Obr. 1.4: Diagram znazorňujúci Casandra cluster. Coordinator je uzol, ktorý obsluhuje požiadavku (zdroj [3])

- SimpleSnitch - Prakticky neposkytuje snitching. Všetky uzly sú lokalizované v rovnakom racku a datacentre. Vhodné len pre single-datacenter nasadenie alebo pre nasadenie kde informácia o datacentre nie je dostupná.
- PropertyFileSnitch - Spôsob akým explicitne uviesť uzly pod dané datacentrum a rack ich zápisom do súboru `cassandra-topology.properties`.
- GossipingPropertyFileSnitch - Princíp je podobný ako pri PropertyFileSnitch. Rozdielom je, že nie je potrebné uvádzať zoznam uzlov v jednotlivých datacentrách a rackoch. Rack a datacentrum pre lokálny uzol sú definované v `cassandra-rackdc.properties` a propagované ostatným uzlom. Odporúča sa použitie tohto snitchu.
- Ec2Snitch - Vhodné pre nasadenie v na Amazon EC2 inštanciách v jednom regióne. Načíta o informácie regióne (pokladá sa za datacentrum) a zóne dostupnosti (rack) z EC2 API.
- Ec2MultiRegionSnitch - Podobné ako Ec2Snitch.Miesto privátnych využíva public IP adresy a umožňuje tak komunikáciu naprieč regionmi.

1.7 MySQL

MySQL[12] je jedným z najpoužívanejších relačných databázových systémov. Manipulácia s dátami prebieha za pomoci štruktúrovaného dotazovacieho jazyka SQL. MySQL server umožňuje použitie rôznych storage engine. Ide o komponentu ktorá ma na starosti obsluhu SQL operácií. Štandardne je použitý engine InnoDB. Podporuje replikáciu (master-slave) a od konca roku 2016 aj clustering v podobe MySQL Group Replication. Obľubené je aj rozšírenie od tretej strany, spoločnosti Codership, s názvom Galera, ktoré implementuje vlastné riešenie replikácie.

1.7.1 MySQL Replication

MySQL replikácia je súčasťou štandardnej MySQL databázy. Umožňuje vytvorenie kópie dát z jedného servera (master) na jeden alebo viacero ďalších databázových serverov (slaves). Master uzol prijíma všetky požiadavky na zápis. Požiadavky spracuje a odošle na slave uzly pre zachovanie konzistencie dát naprieč zúčastnenými uzlami. Je asynchrónna, čo znamená, že nie je garantované že slave uzol má rovnaké dáta v okamihu, keď master vykoná zmeny. Výhod tohto modelu je niekoľko. Viacero uzlov poskytuje robustnosť proti úplnému výpadku. Využitie slave uzlov znamená zvýšenie výkonnosti. Zápisy síce musia smerovať na master uzol, ale čítanie môže prebehnúť z ľubovoľného slave uzla. Taktiež pravidelné zálohovanie databázy môže prebiehať zo slave uzlu, s využitím pozastavenia replikácie, čím sa zamedzí riziko poškodenia dát. [37]

Nevýhodou je spomínané chýbajúce škálovanie počtu zápisov, ktoré je limitované výkonom jediného master uzla.

1.7.2 Galera

Galera od Codership [13] je plugin pre MySQL InnoDB engine, ktorý umožňuje synchronizovanú multi-master replikáciu. Odlišuje sa od štandardnej MySQL replikácie a rieši niektoré problémy ako napríklad konflikt pri zápise na viacero master uzlov, latenciu pri replikácií a nezosynchronizovanosť master a slave uzlov. Užívateľ nepotrebuje mať informáciu o tom či je uzol master alebo slave. Zápis je možný na ľubovoľnom uzle v Galera clustri. Následne je transakcia aplikované na všetky uzly.

Minimálny Galera cluster je zostavený z 3 uzlov. Je odporučané prevádzkovať nepárny počet uzlov, ak sa naskytne problém pri aplikovaní transakcie na jednom uzle, zvyšné dva zdravé uzly predstavujú quorum (väčšinu) a transakcia je potvrdená. Po zotavení havarovaného uzlu sa automaticky synchronizuje

[12] `https://www.mysql.com/`

[13] `https://github.com/codership/galera`

s ostatnými uzlami pomocou State Snapshot Transfer (SST) alebo Incremental State Transfer (IST) v závislosti na poslednom známom stave. Pridávanie nového Galera uzlu nasleduje rovnakú procedúru. Predpoklad je, že nový uzol neobsahuje žiadne dáta, takže Galera vykoná plný state snapshot transfer (SST).

Kedže všetky uzly v Galera clustri sú si rovné, tj. spravujú rovnaké dáta a môžu obsluhovať operácie čítania a zápisu zároveň, môže byť na smerovanie (load balancing) požiadaviek použitá reverzná proxy ako napríklad HAProxy alebo MySQL proxy ProxySQL[14], ktorú využíva aj riešenie Percona XtraDB Cluster zalžené na Galera plugine. Prijíma požiadavky od klientov a smeruje ich k MySQL backend serveru. Podporuje rôzne replikačné topológie ako aj multi-master Galera cluster s funkciami ako je query routing, sharding, query rewrite, query caching, connection pooling a iné.

Galera Cluster funkcionalitou aktívne zamedzuje prípadu keď sa jeden uzol spomalí natoľko, že jeho dáta nie sú aktuálne (flow control). Po dosiahnutí limitu uzol zablokuje všetky zápisy do clustra. Funkčnosť clustru sa obnoví až v momente keď uzol, ktorý mal neaktuálne dáta aplikuje všetky zmeny nariadené replikáciou.

Galera podporuje nasadenie vo WAN sieti. Je možné nasadiť arbitrator uzol, ktorý neuchováva dáta, ale prispieva k zdravému stavu clustra svojou prítomnosťou (napríklad pri hľadaní primárneho uzla).

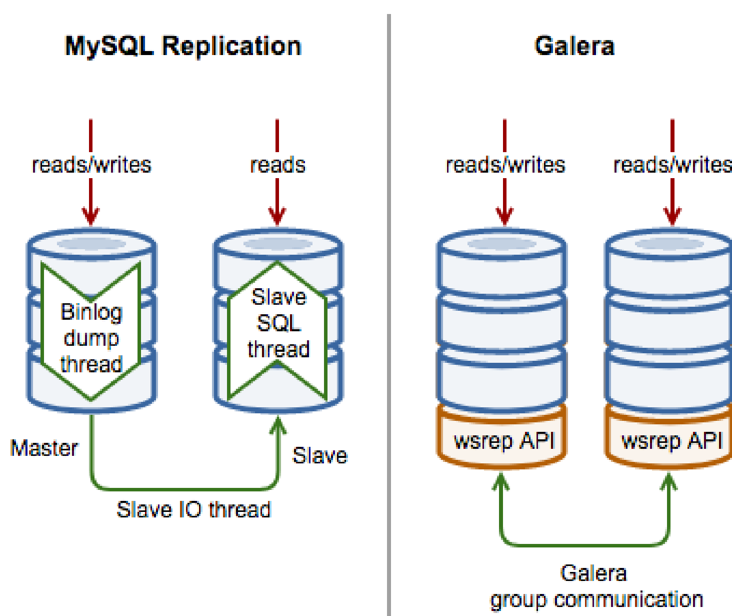
Medzi distrubúcie MySQL, ktoré využívajú Galera plugin patrí Galera Cluster (podpora pre Linux a FreeBSD), Percona XtraDB Cluster (podpora len pre Linux) [15] a MariaDB [16] Galera Cluster (od verzie MariaDB 10.1 je Galera zabudovaná a nie je nutné sťahovať samostatnú distribúciu).

Percona XtraDB cluster má navyše funkciu striktný mód, ktorý zabraňuje využitiu experimentálnych alebo nepodporovaných funkcií ako je napríklad použitie iných operácií ako InnoDB, vyžaduje definovaný primárny kľúč na tabuľkách. Tiež vypína nepodporované funkcie ako je napríklad level izolácie transakcií serializable. Percona XtraDB cluster tiež ponúka viac možností konfigurácie vzťahujúcej sa k prevádzke clustru a opravy chýb spojených so škálovaním. Spoločnosť Percona tiež vyvíja vlastné riešenie pre zálohovanie InnoDB a XtraDB databázy Percona XtraBackup[17].

1.7.3 Group Replication

Group Replication[18] je pllugin vyvinutý priamo spoločnosťou Oracle a súčasť MySQL od verzie 5.7.17 (december 2016). Jedná sa tak o pomerne mladé riešenie. Poskytuje distribuovanú replikáciu s dôrazom na automatickú koordináciu

[14] `http://www.proxysql.com/`
[15] `https://www.percona.com/software/mysql-database/percona-xtradb-cluster`
[16] `https://mariadb.org/`
[17] `https://www.percona.com/software/mysql-database/percona-xtrabackup`
[18] `https://dev.mysql.com/doc/en/group-replication.html`



Obr. 1.5: Znázornenie high-level rozdielov medzi MySQL replikáciou a Galera clustrom.

medzi uzlami. Štandardne zápisy obsluhuje jeden uzol (master-slave) pri ktorého havárií jeho úlohu prevezme novo zvolený uzol. Je možná multi-master konfigurácia avšak nie je doporučovaná. Rovnako ako pri Galera, všetky uzly disponujú kompletným datasetom. Konflikty sú detekované a automaticky riešené. Pri Galera clustri musia transakciu potvrdiť všetky uzly, pri Group Replication je vyžadovaná len väčšina. [38]

Najväčším nedostatkom je chýbajúci automatický provisioning (State Snapshot Transfer v Galera). Pre pridanie nového uzlu je nutné vytvoriť zálohu existujucého uzlu a následne vykonať obnovu z tejto zálohy na novom uzle. Uzol ktorý stratí spojenie s ostatnými členmi clustru stále prima operácie na čítanie, čo môže viesť k neželaným výsledkom. Havarovaný uzol musí byť manuálne znova pridaný do clustru, čo pri nekvalitnom sieťovom spojení môže mať za následok rýchly rozpad clustru. Flow control zahrňa mechanizmus pri ktorom každý uzol má k dispozícií štatistiky pre ostatné uzly. Každý uzol sa rozhodne že spomalí vlastné zápisy v prípade ak nejaký z iných uzlov nestíha.

Group Replication je spolu s ďalšími 2 komponentami súčasťou balíka InnoDB Cluster. Ďalšou komponentou je nástroj MySQL Shell, ktorý používateľ môže použiť na konfiguráciu, pridanie alebo odobranie uzlov a iné. Patrí tu aj MySQL Router, ktorý slúži ako load balancer a smeruje požiadavky. [39]

Group Replication podporuje nasadenie na operačný systémy Linux, Windows, Solaris, OSX,FreeBSD.

Kapitola **2**

Prieskum sučasných riešení

Súčasné riešenia automatizovaného nasadenia databázových systémov Cassandra, MongoDB a MySQL do prostredia Kubernetes na infraštruktúru Google Cloud, AWS a na vlastný hardware sú často neaktualizované, pripravené pre staršiu verziu Kubernetes, čo kvôli rapídnemu vývoju Kubernetes vyústi do nekompatibility so súčasnou verziou. Často slúžia len ako základný stavebný blok, nevyužívajú dostupnú funkcionalitu Kubernetes, ktorá zvýši robustnosť a spoľahlivosť systému. Pre nasadenie na rôzne infraštruktúry sú v niektorých prípadoch potrebné manuálne úpravy a dodatočná konfigurácia. Vo svojom stave tak nie sú vhodné pre reálne využitie.

2.1 Cassandra

Spoločnosť IBM zverejnila na GitHube[19] projekt demonštrujúci nasadenie Cassandra clustra na Kubernetes.

Kontajner je vytváraný z oficiálneho obrazu Cassandra pre Docker[20] spravovaného Docker komunitou. Cassandra uzly sú spravované prostredníctvom StatefulSetu.

Diskové jednotky sú vytvárané ručne, nie je využitý dynamický provisioning. Priložené konfiguračné súbory pre vytváranie diskových jednotiek sú typu `hostPath`, ktoré pre svoje vlastnosti (popísané v 1.3.3) nie sú vhodné na produkčné nasadenie. Pri zmene uzlu na ktorom daný Pod beží môže dôjsť k strate dát.

Pre sformovanie clustra a discovery seed uzlov je využitá headless `Service`. Tá je použitá tiež pre pripojenie aplikácií k databáze. Ako seed uzol je napevno nastavený prvý nasadený uzol. Riešenie využíva `GossipingPropertyFileSnitch`, všetky uzly sú tak umiestnené do jediného datacentra a racku, čo neumožňuje Cassandre riadiť uchovávanie replík na rôznych fyzických uzloch.

[19]https://github.com/IBM/Scalable-Cassandra-deployment-on-Kubernetes
[20]https://github.com/Docker-library/cassandra

Prístup k databáze nie je chránený žiadnym autentifikačným mechanizmom.

V konfigurácií taktiež nie je špecifikované priraďovanie Podov na Kubernes uzly prostredníctvom afinity. Riešenie nevyužíva žiadne post-start a pre-stop hooky. Implementovaná nie je ani indikácia pripravenosti uzla spracovávať požiadavky pomocou readiness sondy. Headless `Service` tak dovolí smerovať požiadavky aj na uzol, ktorý nie je reálne pripravený.

Projekt obsahuje scripty a konfiguráciu pre nasadenie do IBM Bluemix cloudu.

2.2 MongoDB

2.2.1 MongoDB od Sandeep Dinesh

Zamestnanec spoločnosti Google, Sandeep Dinesh, pri príležitosti sprístupnenia objektu `StatefulSet` napísal príspevok[21] o nasadení MongoDB replica setu na Kubernetes na infraštruktúru Google Cloud. Projekt je taktiež umiestnený na GitHube[22]. Projekt sa zaoberá výlučne replica setom a vynecháva konfiguráciu sharded clustra.

Navrhnuté riešenie na správu Podov využíva StatefulSet. Pre priradenie DNS adries jednotlivým Podom replica setu je použitá headless `Service`. Kontajner s MongoDB je vytváraný z oficiálneho obrazu[23] spravovaného Docker komunitou. Proces mongod je spustený s prepínačmi `--replset` nastavujúcim názov replica setu. `--smallfiles`, ktorý zmenšuje počiatočnú veľkosť dátových súborov a limituje ju na 512 MB. Ide o nastavenie vhodné pre veľký počet databáz, ktoré obsahujú málo dát. [40]. Posledným prepínačom je `noprealloc`, ktorý vypína alokáciu dátových súborov. Riešenie nepodporuje internú autentifikáciu uzlov ani autentifikáciu užívateľa.

Do MongoDB kontajnera je pripájané úložisko namapované do zložky `/data/db`, čo je lokácia kde MongoDB ukladá dáta. Pre vytvorenie diskovej jednotky slúži `volumeClaimTemplates`. Vďaka vlastnej `StorageClass` sú využité sú SSD disky na infraštruktúre Google Cloud. Kvôli pevnému špecifikovaniu diskového úložiska dostupného len na Google Cloud, nie je možné okamžité nasadenie na iné infraštruktúry.

2.2.1.1 Sidecar

Na Pode beží ešte druhý kontajner, tzv. sidecar[24]. Ten sa stará o konfiguráciu replica setu a jeho škálovanie pridávaním alebo odoberaním uzlov prostred-

[21] `https://kubernetes.io/blog/2017/01/running-mongodb-on-kubernetes-with-statefulsets`

[22] `https://github.com/thesandlord/mongo-k8s-sidecar`

[23] `https://hub.Docker.com/_/mongo/`

[24] `https://hub.Docker.com/r/cvallance/mongo-k8s-sidecar/`

níctvom zmeny počtu replík `StatefulSetu`. Každých 5 sekúnd skontroluje na ktorých Podoch beží MongoDB a následne rekonfiguruje replica set. Do replica setu je pridaný každý Pod (uzol), ktorý nájde a odstránený každý uzol, ktorý sa už nejaví ako dostupný.

Problémov s týmto spôsobom spravovania replica setu je niekoľko: Nastáva riziko split-brainu pri problémoch so sieťovou komunikáciou medzi jednotlivými uzlami. Jedná sa o prípad keď sa v clustri o 3 uzloch oddelí primárny uzol od sekundárnych. Primárny uzol sa vzdá svojho postu, keďže nedokáže komunikovať s väčšinou uzlov. Zvyšné dva sekundárne uzly, ktoré spolu môžu komunikovať vytvoria quorum (väčšinu) a jeden z nich sa stane primárnym uzlom.

Počas oddelenia primárneho uzlu od sekundárnych, sidecar na primárnom uzle sa vyhodnotí, že sekundárne uzly boli z replica setu odstránené a s touto informáciou rekonfiguruje replica set. Novo vytvorený replica set tak má len jedného člena. Sidecar na zvyšných 2 uzloch taktiež rekonfiguruje replica set a odstráni z neho bývalý primárny uzol s ktorým nevie komunikovať. Týmto vzniknú 2 clustre s rovnakým názvom replica setu.

Ďalším problémom je, že aplikácie, ktoré využívajú databázu môžu mať nastavený `WriteConcerns` (počet uzlov, ktoré musia potvrdiť zápis) na väčšinu alebo všetky uzly. V prípade takejto rekonfigurácie replica setu a dynamického menenia jeho členov táto vlastnosť stráca na význame, keďže replica set bude stále obsahovať len aktívne a dostupné uzly. Zápis s `WriteConcerns` nastaveným na všetky uzly by tak prebehol úspešne bez ohľadu na to či sú reálne dostupné všetky 3 uzly alebo len 1.

2.2.2 MongoDB od Paul Done

Riešenie[25] od Paul Done, pripravené pre staršiu verziu Kubernetes, taktiež využíva `StatefulSet` pre správu Podov, headless `Service` pre priradenie DNS záznamu a umožnenie komunikácie medzi podmi. Kontajner s MongoDB je vytváraný z oficiálneho obrazu[26] spravovaného Docker komunitou.

2.2.2.1 Replica set

Riešenie využíva nastavenie afinity pre beh jednotlivých MongoDB uzlov z replica setu na rôznych Kubernetes uzloch. Proces `mongod` je spúšťaný so zapnutou autentifikáciou užívateľa, obmedzením veľkosti cache pre WiredTiger engine, internou autentifikáciou uzlov pomocou súboru s kľúčom. Kľúč je umiestnený v objekte typu `Secret` a namapovaný ako súbor do do umiestnenia `/etc/secrets-volume/internal-auth-mongodb-keyfile`. Samotný proces je spustený nástrojom `numactl`, čo je odporučané opatrenie pri prevádzkovaní MongoDB na NUMA (Non-Uniform Access Memory) hardware. [41].

[25] `http://k8smongodb.net/`
[26] `https://hub.Docker.com/_/mongo/`

Pred samotným kontajnerom s MongoDB je Kubernetes objektom `DaemonSet` zabezpečené vypnutie THP (Transparent Huge Pages) pre vyšší výkon. [42]

Riešenie je prispôsobené behu na infraštruktúre Google Cloud. Využíva SSD disky špecifikovaním vlastnej `StorageClass`, avšak nie je využitý dynamický provisioning a disky s filesystémom XFS sú vytvárané manuálne pomocou generovaných `PersistentVolume` objektov. Na infraštruktúre Microsoft Azure sú použité LRS (locally redundant storage) disky. Na zvyšných podporovaných infraštruktúrach je využitý dynamický provisioning s využitím štandardnej `StorageClass` nastavenej prevádzkovateľom platformy.

Riešenie obsahuje podporu pre nasadenie na Google Cloud, OpenShift, Microsoft Azure Container Service a lokálne Minikube prostredie.

Samotná inicializácia replica setu a nastavenie admin užívateľa využíva externý shell script spustený mimo prostredia Kubernetes. Využíva konzolový nástroj `kubectl` pre pripojenie na prvý uzol zo StatefulSetu a zavolanie metódy `rs.initiate(...)` s pevne nakonfigurovanými adresami troch uzlov. Riešenie tak neumožňuje pohodlné škálovanie bez nutnosti dodatočnej konfigurácie MongoDB replica setu.

2.2.2.2 Sharding

Pre infraštruktúru Google Cloud je pripravené aj riešenie nasadenia MongoDB s podporou shardingu. Jednotlivé shardy sú vytvorené z replica setu o 3 uzloch. Pre replica sety platí všetko, čo bolo spomenuté vyššie. Proces `mongod` je v kontajneri spustený s pridaným prepínačom `--shardsvr`.

Sharding vyžaduje beh konfiguračných serverov, tie sú nasadené tiež ako replica sety. Majú priradenú vlastnú headless service. Od obyčajného replica setu sa spustenie `mongod` procesu vo vnútri kontajnera líši použitím prepínaču `--configsvr`. Config servery majú napamované diskové úložisko do zložky `/data/db`

Mongos router je nasadený ako `Deployment`. Nemá priradené perzistentné diskové úložisko. V kontajneri je spustený proces `mongos` s podobnými prepínačmi ako štandardný `mongod` proces v replica sete. Prepínačom `--configdb` sú nastavené adresy jednotlivých konfiguračných uzlov z replica setu. Pre zabezpečenú internú komunikáciu mongos routera s konfiguračným serverom je súbor s kľúčom namapovaný do zložky `/etc/secrets-volume` a špecifikovaný v prepínači `--keyFile`.

Inicializácia sharded clustra prebieha podobne ako v prípade replica setu, externým shell scriptom spúšťaným mimo prostredia Kubernetes. Manuálne vytvára diskové jednotky, generuje konfiguračné súbory so správnou veľkosťou diskových jednotiek, vytvára súbor s kľúčom, nasadzuje konfiguračný server, 3 shard replica sety použitím nástroja `kubectl`. Následne je nasadený mongos router. Po počkaní na inicializáciu všetkých Podov sa pomocou `kubectl` pripája na prvý Pod konfiguračného serveru a inicializuje replica set. Rovnaký postup sa opakuje na každom shard replica sete. Po inicializácií mongos

routra sa z prostredia kontajneru v Mongo shelli zavolá príkaz `sh.addShard`, ktorý inicializuje všetky 3 shardy. Nakoniec sa ďalším pripojením na mongos vytvorí admin užívateľ s prihlasovacími údajmi špecifikovanými v premenných na začiatku scriptu.

Riešenie tak neumožňuje škálovanie jednotlivých replica setov, taktiež ani pridanie ďalšieho shardu do clustra bez manuálneho zásahu a rekonfigurácie.

2.3 MySQL

Keďže master-slave replikácia neškáluje operácie na zápis, práca sa zamierava na prieskum riešení, ktoré využívajú Galera plugin a lepšie zapadnú do kontextu Cassandry a MongoDB shardingu.

2.3.0.1 Mysql-galera od eBay

Projekt demonštruje jednoduché nasadenie MySQL s replikáciou pomocou Galera pluginu, konkrétne implementácie Percona XtraDB cluster. Projekt je dostupný na GitHube [27].

Riešenie využíva vlastný Docker obraz postavený na operačnom systéme Ubuntu. Pri zostavení obrazu sa z repozitára Percony inštaluje Percona XtraDB cluster, skopírujú konfiguračné súbory s nastavením replikácie (cluster.cnf a my.cnf). Konfiguračný súbor obsahuje napevno zvolené meno a heslo užívateľa použitého pre replikáciu, čo znamená že zmena týchto údajov vyžaduje znovu zostavenie obrazu.

Nasadzovaný je fixný počet uzlov a to tri. Každý uzol je nasadený ako samostatný Pod použitím dnes už zastaralého kontroléru `ReplicationController` [43]. Riešenie zostavuje adresy uzlov na pripojenie dynamicky v scripte. Pri nasadzovaní kontroléru je nutné dodržať správne poradie, inak nastáva situácia, keď sa pri bootstrape clustra uzol pokúša pripájať na adresu, ktorá obsahuje uzol ktorý ešte nebol vytvorený. Každému uzlu je pridelená `Service`. Riešenie nepodporuje natívne škálovanie v prostredí Kubernetes. Pri pridaní ďalšieho uzlu je potrebné vygenerovanie nového konfiguračného súboru a nastavenie premennej prostredia `NUM_NODES`.

Riešenie nepoužíva perzistentné úložisko, afinitu, sondy na zisťovanie stavu uzlu a je celkovo zastaralé a nevhodné pre použitie.

2.3.0.2 k8s-percona-pxc od Paul Czarkowski

Demo nasadenia Percona XtraDB Cluster. Projekt je umiestnený na GitHube[28]. Využíva vlastný Docker obraz vychádzajúci z oficiálneho obrazu Percona XtraDB Clustru od spoločnosti Percona[29].

[27] `https://github.com/eBay/Kubernetes/tree/master/examples/mysql-galera`
[28] `https://github.com/paulczar/k8s-percona-xtradb-cluster`
[29] `https://hub.Docker.com/r/percona/percona-xtradb-cluster/`

Percona obraz podporuje špecifikovanie uzlov na pripojenie alebo využitie etcd[30] na discovery uzlov. Použitie etcd avšak nie je nevyhnutné, riešenie od Paula Czarkowski pre discovery využíva kombináciu `StatefulSetu` a headless `Service`. Headless `Service` sprístupňuje niekoľko portov potrebných pre úspešnú replikáciu. Pre prístup klientských aplikácií k databáze je zriadená `Sevice` typu NodePort.

Prostredníctvom premenných prostredia je možné konfigurovať názov clustra a heslo admin užívateľa.

Chýbajú mechanizmy na zabezpečenie vysokej dostupnosti clustra. Riešenie nekonfiguruje afinitu jednotlivých Podov v `StatefulSete`, vďaka čomu je umožnený beh Podov na rovnakom Kubernetes uzle a tým pádom hrozí riziko pádu celého clustra haváriou jediného Kubernetes uzla.

Na zisťovanie stavu uzlu sú definované readiness a liveness sondy. Liveness sonda používa príkaz `mysqladmin ping`, ktorý má navratovú hodnotu 0 v prípade že server beží, čo ale nemusí znamenať, že je pripravený prijať spojenie [44]. Na kontrolu prijatia spojenia readiness sonda za pomoci `mysql` klienta uskutoční jednoduchý SELECT.

Kedže ide o demo nie je v ňom špecifikované použitie perzistentného úložiska. Pri reštarte Podu tak dochádza k strate dát.

[30]`https://coreos.com/etcd/`

Kapitola 3

Príprava vlastného riešenia

3.1 Cassandra

Vlastné riešenie využíva oficiálny Cassandra Docker obraz[31]. Stavia na konfigurácií pripravenej spoločnosťou IBM, spomenutej v 2.1.

Riešenie ponecháva použitie `GossipingPropertyFileSnitch` snitchu a na základe odporúčania [35] umiestňuje všetky Pody do rovnakého racku. zachovaná je tiež headless `Service` pre discovery uzlov.

Riešenie pridáva pre-stop hook pre bezpečne odobratie uzlu pri škálovaní, afinitu vďaka čomu prispieva k robustnosti systému plánovaním Podov na rozličné uzly. Podpora ďalšieho datacentra je zabezpečená jednoduchým nasadením ďalšieho `StatefulSetu` s niekoľkými zmenami definovaných premenných prostredia. Implementovaná bola tiež podpora pre autentifikáciu užívateľa.

3.1.1 Príprava Docker obrazu

Docker obraz je postavený na základnom obraze Cassandra a je umiestnený na Docker Hube[32]. Obraz je zostavovaný automaticky z repozitára na GitHube `slowbackspace/k8s-cassandra`[33]. Základný obraz bol rozšírený o ďalšie súbory:

entrypoint.sh Script, ktorým sa spúšťa cassandra proces prevzatý z riešenia od 2.1. Pridaná bola podpora aktivácie autentifikácie úpravou konfiguračného súboru cassandra.yaml

post-start.sh Script spušťaný po štarte.Stará sa o vytvorenie nového užívateľa a deaktiváciu štandardného užívateľa cassandra.

[31] `https://hub.Docker.com/_/cassandra/`
[32] `https://hub.docker.com/r/spakmaro/k8s-cassandra/`
[33] `https://github.com/slowbackspace/k8s-cassandra`

pre-stop.sh Script spušťaný pred zastavením Podu. Stará sa o bezpečné odobratie uzla z clustru a pripraví perzistentné úložisko na ďalší štart Podu premazaním dátových zložiek.

ready-probe.sh Sonda, ktorá sa stará o kontrolu stavu uzlu v clustri. Využíva nástroj `nodetool status`.

3.1.2 Diskové jednotky

Na pripájanie diskových jednotiek k jednotlivým Podom zo `StatefulSetu` je využitý dynamický provisioning prostredníctvom špecifikovania šablóny `volumeClaimTemplates`. Bez špecifikovaného názvu `StorageClass` sa využije štandardná `StorageClass` pre každú platformu. Tým je zaručené univerzálne, na platforme nezávislé, riešenie sprostredkovania perzistentného úložiska. Disková jednotka je namapovaná do kontajneru do zložky `/var/lib/cassandra/data`.

```
volumeMounts:
   - name: cassandra-data
     mountPath: /var/lib/cassandra/data
```

Ukážka 3.1: Definícia pripojených diskových jednotiek.

3.1.3 Komunikácia a smerovanie portov

Komunikácia uzlov vrámci clustra je umožnená vďaka nasadeniu headless `Service`, ktorá zabezpečí preklad DNS adries na IP adresy, ktoré Cassandra využíva. Umožňuje pripojenie klientských aplikácií prostredníctvom portu 9042 určeného pre CQL transport. Seed uzol je nastaviteľný premennou prostredia `CASSANDRA_SEEDS`, prednastavené je použitie adresy prvého Podu zo `StatefulSetu`. Obsah tejto premennej sa po spustení kontajneru vloží do konfiguračného súboru `cassandra.yaml`.

Premenné prostredia `CASSANDRA_DC` a `CASSANDRA_RACK` špecifikujú názov racku a datacentra do ktorého uzol patrí. Hodnoty sú po spustení kontajneru vložené do súboru `cassandra-rackdc.properties`.

```
env:
- name: CASSANDRA_SEEDS
  value: cassandra-0.cassandra.default.svc.cluster.local
- name: CASSANDRA_CLUSTER_NAME
  value: "Cassandra"
- name: CASSANDRA_DC
  value: "DC1"
- name: CASSANDRA_RACK
  value: "Rack1"
```

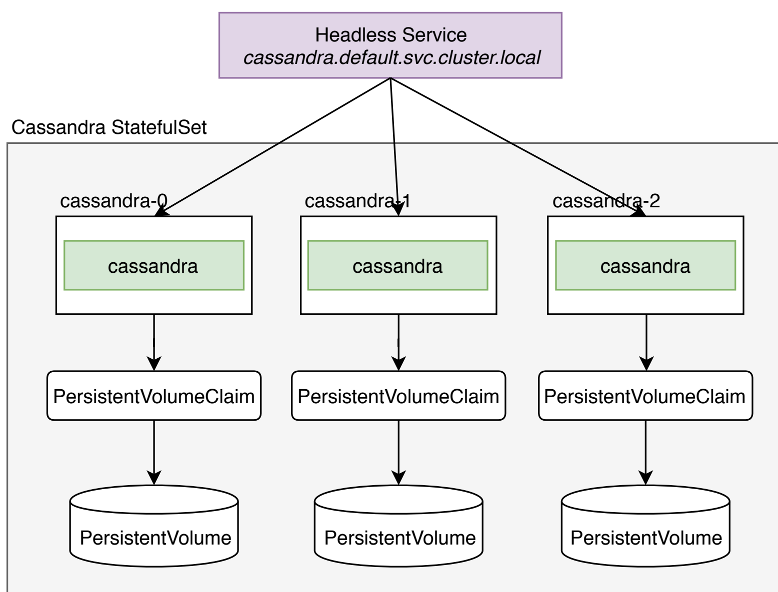
Ukážka 3.2: Prednastavené hodnoty premenných prostredia

3.1.4 Autentifikácia užívateľa

Bol upravený štartovací script `entrypoint.sh`, ktorý nahrádza hodnotu poľa `authenticator` a umožňuje zapnutie autentifikácie užívateľa. Po štarte je vytvorený užívateľ s menom z premennej prostredia `CASSANDRA_USERNAME` a a heslom `CASSANDRA_PASSWORD` príkazom:

```
CREATE ROLE IF NOT EXISTS $CASSANDRA_USERNAME WITH PASSWORD =
    '$CASSANDRA_PASSWORD' AND SUPERUSER = true AND LOGIN =
    true;
```

Prístup k štandardnému užívateľovi (cassandra) je znemožnený zmenením jeho hesla na vygenerovanú náhodnú hodnotu.



Obr. 3.1: Pohľad na architektúru nasadenia a využitie headless Service pri smerovaní požiadaviek k jednotlivým Cassandra uzlom.

3.1.5 Priraďovanie Podov na uzly

Každá Cassandra Pod je označený labelom s názvom app nastaveným na hodnotu cassandra. Pre dosiahnutie vysokej dostupnosti je preferované spúšťanie Podov na uzloch, ktoré ešte neobsahujú žiaden Cassandra Pod. To je dosiahnuté použitím anti-afinity a špecifikovaním `labelSelector` s pravidlom kontrolujúcim obsah labelu s kľúčom app.

Pre podporu behu vo viacerých datacentrách je tiež špecifikovaná afinita pre Kubernetes uzly. Plánovanie podu je povolené len na uzle, ktorý má label s kľúčom dc nastaveným na hodnotu dc1. Pri nasadení ďalšieho datacentra, tak stačí Kubernetes uzly označiť labelom dc2 a zmeniť pravidlo.

```
affinity:
  podAntiAffinity:
    preferredDuringSchedulingIgnoredDuringExecution:
    - weight: 100
      podAffinityTerm:
        labelSelector:
          matchExpressions:
          - key: app
            operator: In
            values:
            - cassandra
        topologyKey: kubernetes.io/hostname
  nodeAffinity:
    requiredDuringSchedulingIgnoredDuringExecution:
      nodeSelectorTerms:
      - matchExpressions:
        - key: dc
          operator: In
          values:
          - dc1
```

Ukážka 3.3: Špecifikácia anti-afiinity pre body a afinity pre uzly

3.1.6 Kontrola zdravia uzlov

Na kontrolu schopnosti spracovávať požiadavky je použitá readiness sonda. Využíva nástroj `nodetool` pomocou ktorého overuje status a stav uzlu.

```
if [[ $(nodetool status | grep $POD_IP) == *"UN"* ]]; then
  echo "Node is up!";
  exit 0;
else
  echo "Node is not up!";
  exit 1;
fi
```

Ukážka 3.4: Readiness sonda kontrolujúca stav pomocou nástroja `nodetool`

3.1.7 Hooky

Pred odstránením Podu je pomocou pre-stop hooku zavolaný príkaz `nodetool decommission`, ktorý zabezpečí odovzdanie dát odstraňovaného uzla iným uzlom, tým sa predíde potencionálnej strate dát pri škálovani clustra na nižší počet uzlov.

Po štarte Podu sa post-start script postará o vytvorenie nového užívateľa s prihlasovacími údajmi, ktoré sú nastavené v premenných prostredia. Štandardný užívateľ cassandra je deaktivovaný a je zvýšený počet replík keyspace `system_auth` pre zabezpečenie funkčnosti prihlasovania aj pri strate uzlu.

3.2 MongoDB

MongoDB databázové uzly sú stateful. Pripravené sú konfiguračné súbory pre nasadenie jedného uzlu, replica setu, a sharded clustru, kde každý shard tvorí replica set. Všetky tieto metódy nasadenia využívajú vlastný Docker obraz. Riešenie adresuje tieto body:

- autentifikácia užívateľa
- interná autentifikácia vrámci clustra
- inicializácia replica setu
- možnosť škálovania replica setu natívne bez použitia ďalších scriptov
- pridávanie nových shardov
- využitie sond na kontrolu zdravia uzlu
- využitie SSD diskov na infraštruktúre Google Cloud a AWS
- využívanie `StatefulSet` a `PersistentVolumeClaims` pre uchovanie dát
- anti-afinita pre zaručenie vysokej dostupnosti clustra

3.2.1 Príprava Docker obrazu

Docker komunita poskytuje oficiálny image MongoDB, ktorý je možné stiahnuť z Docker Hub[34]. Obraz okrem iného obsahuje aj konzolový nástroj `mongo`, určený pre správu, dotazovanie alebo update dát v databáze. Tento obraz bol rozšírený o ďalšie súbory zabezpečujúce vytvorenie replica setu, pridanie uzlu do replica setu alebo jeho zmazanie a taktiež podporu shardingu.

[34] `https://hub.Docker.com/_/mongo/`

repSetinit.js Javacript script určený pre beh v prostredí MongoDB shellu. Slúži na inicializáciu replica setu. Zavolá metódu `rs.initiate()` so špecifikovanou plnou adresou uzlu vrámci naviazanej Kubernetes `Service`.

RepSetAdd.js Javascript script určený pre beh v prostredí MongoDB shellu. Slúži na pridanie uzlu do replica setu. Zavolá metódu `rs.add()` so špecifikovanou plnou adresou uzlu vrámci pripojenej Kubernetes `Service`.

post-start.sh Shell script určený na spustenie po štarte Podu. Stará sa o pripojenie k MongoDB shellu pre inicializáciu replica setu a pridanie ďalších uzlov, vytvorenie admin užívateľa a pridanie shardu.

pre-stop.sh Shell script určený na spustenie pred ukončením Podu. Stará sa o plynulý odchod uzlu z replica setu zavolaním metódy `rs.remove()` v MongoDB shelli.

functions.sh Shell script obsahujúci funkciu na vytváranie URL na pripojenie k MongoDB replica setu, ktoré využívajú ďalšie scripty. Stará sa o vloženie premenných prostredia `K8S_SERVICE_URL`, `REPSET_NAME`, `ROLE` do odpovedajúcich premenných v javascript scriptoch použitých v MongoDB Shelli.

3.2.1.1 Zostavenie Docker obrazu

Obraz je automaticky zostavovaný z GitHub repozitára `slowbackspace/k8s-mongodb`[35].

```
FROM mongo
COPY . /helpers
ENTRYPOINT ["/helpers/entrypoint.sh"]
```

Ukážka 3.5: Dockerfile pre MongoDB

Vytvorený obraz je umiestnený na Docker Hube vo verejnom repozitári spakmaro/k8s-mongo[36]. Odtiaľ sa bude automaticky sťahovať pri nasadzovaní na platformu Kubernetes.

3.2.2 Príprava replica setu

Konfigurácia pre nasadenie replica setu využíva koncept `StatefulSets`. Špecifikuje niekoľko premenných prostredia

- `ADMIN_USERNAME` - Meno pre vytváraný administrátorský účet
- `ADMIN_PASSWORD` - Heslo pre vytváraný administrátorský účet

[35]`https://github.com/slowbackspace/k8s-mongodb`
[36]`https://hub.Docker.com/r/spakmaro/k8s-mongo/`

- `REPSET_NAME` - Názov replica setu
- `K8S_SERVICE_URL` - URL adresa pripojenej `Service`

Každý Pod je spustený s príkazom `mongod` s parametrami:

- --wiredTigerCacheSizeGB 0.25 - Veľkosť internej cache pamäte. Bez explicitného špecifikovania MongoDB štandardne použije väčšiu z hodnôt 0.5*(RAM-1GB) a 256MB.
- `--bind_ip 0.0.0.0` - Proces čaká na spojenie na všetkých IPv4 adresách. Potrebné pre komunikáciu medzi uzlami naprieč clustrom.
- `--replSet $(REPSET_NAME)` - Špecifikovanie názvu replica setu.
- `--auth` - Zapnutie autentifikácie pre užívateľov
- `--clusterAuthMode keyFile` - Zapnutie internej autentifikácie uzlov pre vzájomnú komunikáciu s autentifikovaním prostredníctvom súboru s kľúčom.
- `--keyFile /etc/secrets-volume/internal-auth-mongodb-keyfile` - Cesta k súboru s kľúčom
- `--setParameter authenticationMechanisms=SCRAM-SHA-1` - Špecifikovanie spôsobu autentifikácie

Po naštartovaní kontajneru sa vďaka post-start hooku spustí script `post-start.sh`. Pody v StatefulSete majú jednotný hostname líšiaci sa číselným indexom Podu. Prvý pod, s indexom 0, spustí inicializáciu replica setu a čaká na jej dokončenie. Ak sú nastavené premenné `ADMIN_USERNAME` a `ADMIN_PASSWORD` sa po dokončení inicializácie replica setu vytvorí admin užívateľ.

```
rs.add({host: getHostName() + "." + K8S_SERVICE_URL + ":27017",
    priority: 0, votes: 0});

var cfg = rs.conf();
for(var i=0;i<cfg.members.length;i++){
    if (cfg.members[i].priority != 1) {
        cfg.members[i].priority = 1;

        if (i < 7) { cfg.members[i].votes = 1; }

        rs.reconfig(cfg);
    }
}
```

Ukážka 3.6: Script repSetInit.js pre inicializáciu replica setu.

Pre ďalšie Pody sa spustí script pre pridanie uzlu do replica setu. Uzly sú pridávané s vypnutou účasťou na voľbe nového primárneho uzlu. Po pridaní uzlu je táto vlastnosť metódou `rs.reconfig` opäť zapnutá. Zúčastňovať sa voľby primárneho uzlu môže len maximálne 7 uzlov.

```
isConfigSvr = (ROLE == "configdb")? true: false;

if ((rs.status()["codeName"] == "NotYetInitialized")) {
    rs.initiate({"_id": REPSET_NAME, configsvr: isConfigSvr,
        version: 1, members: [{_id: 0, host: getHostName() + "."
        + K8S_SERVICE_URL + ":27017"}]});

    while ((rs.status().hasOwnProperty("myState") &&
        rs.status().myState != 1)) {
        sleep(1000);
    }
}
```

Ukážka 3.7: Script repSetInit.js pre inicializáciu replica setu.

Pri odobraní uzlu z replica setu je vhodné zavolať metódu `rs.remove`. To sa realizuje scriptom spúšťaným pri ukončení behu Podu.

```
#!/bin/bash
. /helpers/functions.sh
mongo admin --host $(buildURI) -u ${ADMIN_USERNAME} -p
    ${ADMIN_PASSWORD} --eval "rs.remove(getHostName()␣+␣
    '.${K8S_SERVICE_URL}:27017')"
```

Ukážka 3.8: Script pre-stop.sh

3.2.2.1 Diskové jednotky

V prípade havárie kontajneru je nevyhnutné aby dáta zostali zachované. Na to je možné využiť `PersistentVolumes`, ktoré namapujú zložku s databázami v kontajneri do perzistentného úložiska, ktoré existuje nezávisle na Pode a teda dáta sú uchované aj počas havárie kontajneru alebo presunu Podu na iný Kubernetes uzol.

V prostredí Google Cloud riešenie využíva rýchle SSD disky vytvorením `StorageClass` s názvom fast, nastavenou vlastnosť provisioner na `kubernetes.io/gce-pd` a parametrom `type: pd-ssd`.

```
kind: StorageClass
metadata:
 name: fast
provisioner: kubernetes.io/gce-pd
parameters:
 type: pd-ssd
```

Ukážka 3.9: Špecifikácia `StorageClass` s názvom fast využivajúca SSD disky na Google Cloud.

Vyžiadanie vytvorenia disku sa realizuje pomocou šablóny `volumeClaim-Templates`. Táto šablóna zabezpečí vytvorenie `PersistentVolumeClaim` pre každý pod z replica setu. Nastavenie hodnoty `storageClassName` na fast ma za následok využitie dopredu definovanej `StorageClass`.

```
volumeClaimTemplates:
  - metadata:
      name: mongodb-persistent-storage-claim
    spec:
      storageClassName: fast
      accessModes: [ "ReadWriteOnce" ]
      resources:
        requests:
          storage: 1Gi
```

Ukážka 3.10: Definícia šablóny pre vytvorenie `PersistentVolumeClaim`.

MongoDB zapisuje dáta do zložky `/data/db`. Vo `volumeMounts` nastavíme pole name s názvom vytvoreného `PersistentVolumeClaim` (mongodb-persistent-storage-claim) a mountPath na `/data/db`.

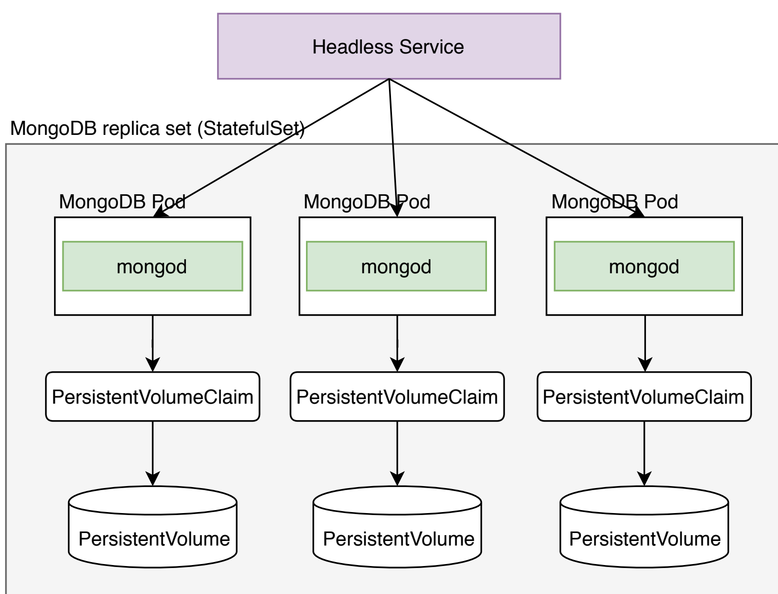
```
...
volumeMounts:
  - name: secrets-volume
    readOnly: true
    mountPath: /etc/secrets-volume
  - name: mongodb-persistent-storage-claim
    mountPath: /data/db
```

Ukážka 3.11: Definícia pripojených diskových jednotiek.

3.2.2.2 Komunikácia a smerovanie portov

MongoDB databázové uzlom vrámci replica setu musí byť umožnené navzájom komunikovať. Každý MongoDB uzol z replica setu potrebuje vedieť adresu

ostatných uzlov. Pri zmene Kubernetes uzlu na ktorom Pod beží je ale pravdepodobné, že nová inštancia Podu dostane pridelenú novú IP adresu, odlišnú od predchádzajúcej. Riešením je k Podom pripojiť `Service`, ktorá využíva DNS systém a každému Podu pridelí dns záznam, ktorý sa nemení ani pri zmene uzlu na ktorom Pod beží.



Obr. 3.2: Pohľad na Využitie headless `Service` pri smerovaní požiadaviek k jednotlivým uzlom replica setu.

Štandardný port na ktorom uzol čaká na spojenie je 27017. Komunikácia je zabezpečená prostredníctvom headless `Service` sprostredkujúcej port 27017. `Service` je naviazaná na Pody použitím selectoru využívajúceho label Podu.

```
apiVersion: v1
kind: Service
metadata:
  name: mongodb-rs-service
  labels:
    app: mongod-rs
spec:
  ports:
  - port: 27017
    targetPort: 27017
  clusterIP: None
  selector:
    app: mongod-rs
```

Ukážka 3.12: Špecifikácia headless Service pre MongoDB replica set.

3.2.2.3 Priraďovanie Podov na uzly

Pre dosiahnutie vysokej dostupnosti samotný beh viacerých MongoDB uzlov nestačí. Je nutné zabezpečiť aby neboli spúšťané na rovnakom fyzickom Kubernetes uzle. Na to je využitá možnosť špecifikovania vlastnosti `anti-affinity` pre Pody. Každý Pod má label s kľúčom replicaset pod ktorým sa nachádza názov replica setu. S využitím anti-afinity a špecifikovaním `labelSelector` je dosiahnuté plánovanie Podu na uzle na ktorom ešte nebeží žiadny ďalší Pod z daného replica setu. V prípade ak takýto uzol nie je dostupný, je vďaka metóde `preferredDuringSchedulingIgnoredDuringExecution` obmedzenie ignorované a Pod je spustený na ktoromkoľvek uzle.

```
affinity:
  podAntiAffinity:
    preferredDuringSchedulingIgnoredDuringExecution:
    - weight: 100
      podAffinityTerm:
        labelSelector:
          matchExpressions:
          - key: replicaset
            operator: In
            values:
            - MainRepSet
        topologyKey: kubernetes.io/hostname
```

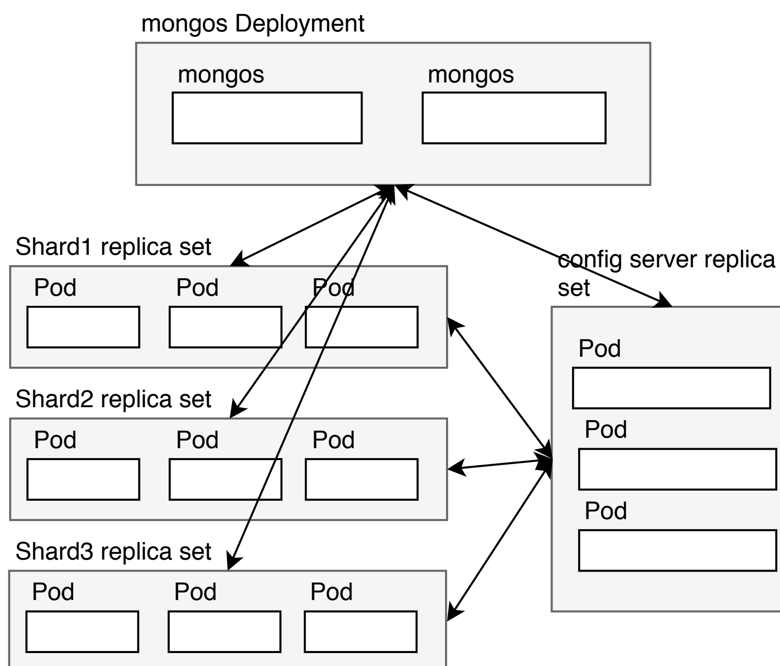
Ukážka 3.13: Nastavenie affinity pre replica set.

3.2.2.4 Kontrola zdravia uzlov

Je využitá sonda typu readiness, ktorá má na starosti kontrolu schopnosti MongoDB uzla spracovávať požiadavky. Tvorí ju príkaz, ktorý využije MongoDB shell a zavolá funkciu `db.getMongo()`, ktorá otestuje schopnosť shellu pripojiť sa na databázovú inštanciu.

3.2.3 Sharding

Pre horizontálne škálovanie sa využije metóda distribuovania dát naprieč viacerými uzlami (shards), sharding. Takýto cluster vyžaduje nasadenie 3 komponent - konfiguračného serveru, shardu a mongos routerov.



Obr. 3.3: Pohľad na celkovú architektúru nasadenia MongoDB s podporou shardingu

3.2.3.1 Konfiguračný server

Konfiguračný server uchováva metadata a konfiguračné nastavenia clustru. Jedná sa o stateful aplikáciu. Navyše config server musí byť od verzie MongoDB 3.4 nasadený ako replica set, čiže správu Podov bude obsluhovať objekt `StatefulSet`. Inicializácia replica setu konfiguračného serveru sa od klasickej odlišuje spustením mongod procesu s prepínačom `--configsvr`. Štandardný port na ktorom inštancia config servera čaká na spojenie je 27019, pre zachovanie konzistencie pri definicií `Service`, je tento port zmenený na 27017.

Každý Pod, ktorý ma rolu shardu ma nastavenú premennú prostredia `ROLE` na configdb. Premenná `K8S_SERVICE_URL` nastavuje adresu headless Service, vďaka ktorej sú počas behu prekladané doménové adresy Podov na ich IP adresy. Adresa je nastavená na `mongodb-configdb-service.default.svc.cluster.local`. Prvý Pod zo StatefulSetu tak bude mať adresu `mongod-configdb-0.<adresa_service>`.

3.2.3.2 Shard

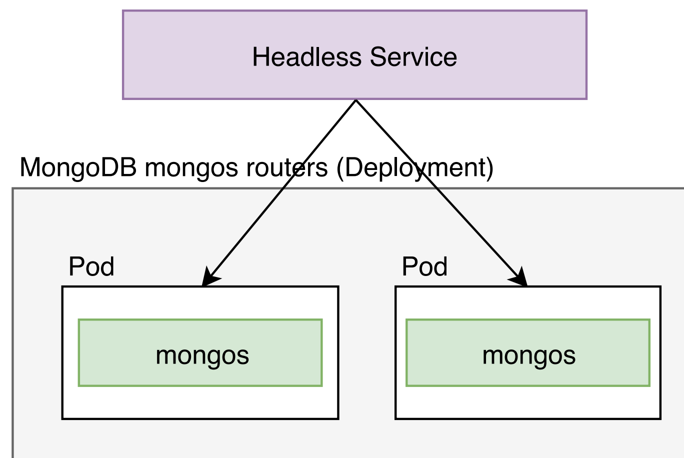
Shard uchováva subset shardovaných dát. Pre zvýšenie robustnosti a zabezpečenie vysokej dostupnosti je shard nakonfigurovaný ako replica set, skladajúci sa z viacerých uzlov. Pre takýto replica set vrámci shardu platí všetko, čo aj

pre samostatný replica set. Rozdiel je v spustení `mongod` procesu. Pridaný je prepínač `--shardsvr`. Port na ktorom proces čaká na spojenie je opäť nakonfigurovaný na 27017.

Každý Pod, ktorý ma rolu shardu ma nastavenú premennú prostredia `ROLE` na shard. Vďaka tomu sa v post-start scripte spustí pridanie shardu do clustru. Opäť je definovaná premenná `K8S_SERVICE_URL` v ktorej je uložená adresa `mongodb-shardX-service.default.svc.cluster.local`, kde X je index shardu. Premenná `K8S_MONGOS_SERVICE_URL` obsahuje adresu mongos routera, ktorá sa použije na pripojenie k routeru a pridanie shardu do konfiguračného serveru.

3.2.3.3 Mongos router

Mongos routre nevyžadujú pripojenie perzistentného úložiska, sú stateles. Nasadenie je typu `Deployment`. K Deploymentu je priradená je headless `Service`, ktorá umožňuje prístup aplikáciam k MongoDB clustru. Pre vysokú dostupnosť routerov je pomocou afinity zabezpečené ich nasadenie na rôzne Kubernetes uzly. Proces `mongos` je spustený aj s prepínačom `--configdb` špecifikujúcim adresu replica setu konfiguračného serveru. Tá je nastavená na `ConfigDBRepSet/mongod-configdb-0.mongodb-configdb-service.default.svc.cluster.local:27017`



Obr. 3.4: Pohľad na architektúru nasadenia mongos routerov.

3.3 MySQL

Riešenie imeplemntuje nasadenie PerconaXtraDB Cluster. Je postavené na existujúcom projekte od Paula Czarkowski (2.3.0.2). Kontajner s databázou, ale používa oficiálny Docker obraz PerconaXtraDB Cluster[37], do ktorého sú pripojené doplňujúce 2 súbory prostredníctvom `ConfigMap`.

Nasadený je tiež init kontajner s obrazom busybox[38], ktorý rieši problém[39] pri pridávaní uzlu do clustra. Pridaná je podpora pre perzistentné úložisko, zvýšenie robustnosti definovaním pravidiel pre anti-afinitu. Implementová je tiež podpora pre zálohovanie a obnovu databáze využitím Kubernetes objektov `CronJob` a `Job`.

3.3.1 Diskové jednotky

Podobne ako v prípade MongoDB je využitý dynamický provisioning diskových jednotiek. Šablóna `volumeClaimTemplates` vytvorí `PersistentVolumeClaim` objekty vďaka ktorým sú vytvorené odpovedajúce `PersistentVolumes` pre každý Pod zo `StatefulSetu`. Úložisko je namapované do zložky `/var/lib/mysql`.

3.3.2 Komunikácia a smerovanie portov

Na zistenie adries všetkých uzlov je využitá headless `Service` v kombinácií s nástrojom `resolveip`, ktorý je súčasťou inštalácie Percona XtraDB Cluster.

```
resolveip -s "${K8S_SERVICE_NAME}"
```

Príkaz vráti zoznam všetkých IP adries Podov využivajúcich danú `Service`. Ich zoznam je predaný procesu `mysqld` prepínačom `--wsrep_cluster_address`.

Nasadené sú dve headless `Service`. Jedna slúži na smerovanie internej komunikácie medzi uzlami clustra. Sprístupňuje niekoľko portov slúžiac pre replikáciu (4567), Incremental State Transfer (4568) a State Snapshot Transfer (4444). Druhá `Service` sprístupňuje port 3306 a je určená pre klientské požiadavky na MySQL databázu.

```
apiVersion: v1
kind: Service
metadata:
  labels:
    app: percona-galera
  name: percona-galera-xtradb
spec:
```

[37] https://hub.Docker.com/r/percona/percona-xtradb-cluster/

[38] https://hub.Docker.com/_/busybox/

[39] https://www.percona.com/forums/questions-discussions/percona-xtradb-cluster/34247-second-node-won-t-join-cluster-sst-fails

```
  clusterIP: None
  ports:
  - name: galera-replication
    port: 4567
  - name: state-transfer
    port: 4568
  - name: state-snapshot
    port: 4444
  selector:
    app: percona-galera
```

Ukážka 3.14: Service pre internú komunikáciu vrámci clustra

3.3.3 Priraďovanie Podov na uzly

Každý Pod ma pridelený label s kľúčom app nastaveným na hodnotu percona-galera. Pre vyššiu robustnosť systému je pomocou anti-afinity nastavené plánovanie Podov prednostne na uzly, na ktorých ešte nebeží žiadny Pod zo `StatefulSetu`.

```
affinity:
  podAntiAffinity:
    preferredDuringSchedulingIgnoredDuringExecution:
    - weight: 100
      podAffinityTerm:
        labelSelector:
          matchExpressions:
          - key: app
            operator: In
            values:
            - percona-galera
        topologyKey: kubernetes.io/hostname
```

Ukážka 3.15: Nastavenie afinity pre Pody.

3.3.4 Kontrola zdravia uzlov

Readiness aj liveness sondy sú zachované z pôvodného projektu. Liveness sonda, overujúca beh systému využíva príkaz

```
mysqladmin -p$(MYSQL_ROOT_PASSWORD) ping
```

Readiness sonda vykonáva jednoduchý SELECT.

```
mysql -h 127.0.0.1 -p$(MYSQL_ROOT_PASSWORD) -e SELECT 1
```

3.3.5 Zálohovanie

Bolo implementované pravidelné zálohovanie clustra. Riešenie využíva vlastný Docker obraz umiestnený na Docker Hube[40]. Podporuje plnú a inkrementálnu zálohu prostredníctvom Kubernetes `CronJob`. Na obnovu zo zálohy sa využíva `Job` objekt.

Na samotné zálohovanie a obnovu sa využíva nástroj `innobackupex` z balíka Percona XtraBackup[41].

Pri procese zálohovania sa vďaka headless `Service`, ktorá obsluhuje Pody z PerconaXtraDB clustra získa IP adresy prvého uzlu v clustri. Následne sa nástrojom `mysql` pripojí k databázovému stroju a získajú sa informácie o zložkách, kde sú uchovávané dáta. Zo získaných údajov sa vytvorí konfiguračný súbor, ktorý využíva nástroj `innobackupex` pre pripojenie k DB serveru a vytvoreniu záloh.

Zálohy sú komprimované a šifrované s algoritmom AES256 s kľúčom zo súboru `.mykeyfile`. Záloha je streamovaná na štandardný výstup v špeciálnom xbstream formáte. História zálohovania je ukladaná do databáze `PERCONA_SCHEMA` prepínačom `--history` s dátumom poslednej zálohy. Plná záloha sa vytvorí príkazom

```
innobackupex
    --defaults-file=/xtrabackup/backup-my.cnf \
    --host=$host \
    --no-timestamp \
    --stream=xbstream \
    --parallel=4 \
    --encrypt=AES256 \
    --encrypt-key-file=/xtrabackup-configs/.mykeyfile \
    --encrypt-threads=4 \
    --compress \
    --compress-threads=4 \
    --history=$(date +%d-%m-%Y) ./ > \
    mysqlbackup-$(date +%d-%m-%Y).qp.xbc.xbs
```

kde v premennej `$host` je získaná adresa Podu z clustra. Záloha je zapísaná do súboru a umiestnená do zložky `/backups`.

Pri obnove sa nakopirujú požadované archívy (dátum obnovy je nastavený premennou prostredia `$RESTORE_DATE`) zo zložky `/backups` rozbalia a odšifrujú nástrojom `xbcrypt`.

Pred samotnou obnovou sa na zálohu aplikujú necommitnuté transakcie príkazom

```
innobackupex --use-memory=1G --apply-log /backups/restore/full
```

[40] https://hub.Docker.com/r/spakmaro/xtrabackup/

[41] https://www.percona.com/software/mysql-database/percona-xtrabackup

Následne sa sa zavolá príkaz
`innobackupex -copy-back /backups/restore/full`, ktorý pripravenú zá-loho obnoví na servery.

3.3.5.1 Príprava Docker obrazu

Ako základný obraz je použitý obraz Ubuntu vo verzii 16.04. Príprava obrazu zahŕňa pridanie repozitára Percona, inštaláciu balíka Percona XtraBackup, MySQL klienta a ďalších dodatočných nástrojov pomocou príkazu `RUN`. Príkazom `COPY` sú následne skopírované potrebné súbory do obrazu.

```
FROM ubuntu:16.04

RUN apt-get update && apt-get install -y wget \
    && wget --quiet
       repo.percona.com/apt/percona-release_0.1-5.xenial_all.deb
       \
    && dpkg -i percona-release_0.1-5.xenial_all.deb && apt-get
       update \
    && apt-get install -y percona-xtrabackup-24 libgcrypt20
       openssl \
    && rm -f percona-release_0.1-5.xenial_all.deb \
    && wget http://www.quicklz.com/qpress-11-linux-x64.tar &&
       tar -xf qpress-11-linux-x64.tar -C /usr/bin/
RUN apt-get install -y nano mysql-client && rm -rf
    /var/lib/apt/lists/*
COPY . .
ENTRYPOINT ["/entrypoint.sh"]
```

Ukážka 3.16: Dockerfile pre zostavenie obrazu pre zálohovanie.

Do koreňovej zložky kontajneru sú skopírované tieto súbory:

- backup-assist.sh - Shell script, ktorý pripraví konfiguračný súbor pre nástroj `innobackupex`. Vloží doň prihlasovacie údaje MySQL užívateľa určeného pre zálohu, cestu k dátovej zložke MySQL, InnoDB log group a InnoDB data zložke, ktoré získava priamo pripojením k zálohovanému MySQL serveru.

- inc-backup.sh - Shell script vykonávajúci inkrementálne zálohovanie od poslednej plnej zálohy nástrojom `innobackupex`.

- full-backup.sh - Shell script vykonávajúci plnú zálohu MySQL clustra.

- restore-assist.sh - Script, ktorý nájde požadované zálohy, odšifruje ich a rozbalí archívy.

- restore.sh - Script, ktorý presunie pripravené súbory na obnovu do MySQL.
- entrypoint.sh - Script spúšťaný po štarte kontajnera. Na základe premennej prostredia `BACKUP_TYPE` spusti zálohovanie (inkrementálne alebo plné) alebo obnovu zo zálohy.

3.3.5.2 Implementácia v Kubernetes

Plná a inkrementálna záloha je spušťaná v pravidelných intervaloch vďaka definovaniu Kubernetes `CronJob` kontroléru. Pre obnovu zo zálohy je pripravená definícia `Job` objektu. Záloha je uložená na perzistentné úložisko vytvárané dynamicky pomocou `PersistentVolumeClaim`. Na šifrovanie záloh sa používa algoritmus AES256 s kľúčom definovaným v `ConfigMap` objekte.

Interval zálohovanie je špecifikovaný v `spec.schedule` a je zadávaný v štandardnom Cron formáte. Pre denné zálohy o 1 hodine ráno stačí zadať hodnotu `0 1 * * *`. Aby neprišlo k viacnásobnému spusteniu zálohovania je `spec.concurrencyPolicy` nastavená na `Forbid`, čo znamená že v danom okamihu môže bežať len 1 inštancia Podu vytvoreného kontrolérom.

V `spec.jobTemplate.spec.template.spec.name.env` sú definované premenné prostredia

- `K8S_SERVICE_NAME` - Názov `Service` pripojenej k Podom MySQL clustra.
- `XTRABACKUP_PASSWORD` - Heslo pre užívateľa backup, ktorý sa používa na pripojenie k databáze.
- `BACKUP_TYPE` - Špecifikuje či ide o plnú alebo inkrementálnu zálohu.
- `RESTORE_DATE` - Špecifikuje dátum z ktorého má byť záloha obnovená.

Do kontajneru je pripojených niekoľko diskových jednotiek. Disková jednotka pre umiestnenie zálohy je vytvorená pomocou `PersistentVolumeClaim` a namapovaná do kontajneru do zložky `/backups`. Pre vytvorenie zálohy je potrebné pripojenie diskovej jednotky, ktorú využíva Pod z Percona XtraDB clustra. To je docielené vyžiadaním pripojenia diskovej jednotky, ktorá je naviazaná na `persistentVolumeClaim` s názvom `mysql-data-percona-galera-0`. Namapovaná je do tradičnej zložky `/var/lib/mysql` Ďalej je pripojená disková jednotka vytvorená z `ConfigMap`, ktorá obsahuje kľúč použitý k šifrovaniu a odšifrovaniu záloh.

```
volumes:
  - name: backup-dir
    persistentVolumeClaim:
      claimName: backup-dir-pv-claim
  - name: mysql-data
    persistentVolumeClaim:
      claimName: mysql-data-percona-galera-0
  - name: xtrabackup-configs
    configMap:
      name: xtrabackup-configs
```

Ukážka 3.17: Definovanie diskových jednotiek, ktoré sú pripajané k Podu

```
kind: ConfigMap
apiVersion: v1
metadata:
  name: xtrabackup-configs
  namespace: default
data:
  .mykeyfile: "Rn8cFBcCc/ccuw9+4Go/t0rFRPB7P13P"
```

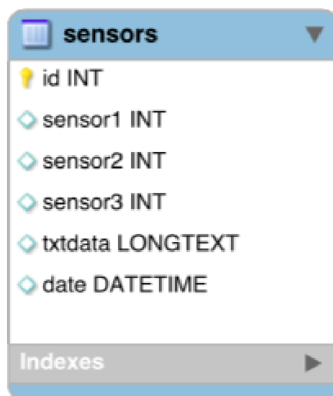
Ukážka 3.18: Definovanie `ConfigMap` s kľúčom pre šifrovanie záloh

3.4 Prototyp web aplikácie

Bola implementovaná web aplikácia demonštrujúca použitie nasadeného MongoDB napísaná v jazyku Python s použitím frameworku pre tvorbu webových aplikácií - Flask[42] a knižnice PyMongo[43], mysqlclient[44] a Python Cassandra Driver[45]. Samotná Flask aplikácia beží na WSGI servery Gunicorn[46]. Aplikácia je umiestnená do Docker kontajneru a nasadená na Kubernetes. Docker obraz je umiestnený na Docker Hube `spakmaro/flask_app`[47].

Aplikácia sa stará o vytvorenie potrebných štruktúr v databáze, vloženie a výpis dát. Pre demonštráciu je vytvorená tabuľka `sensors`, ktorú tvorí niekoľko polí. Dátové typy polí majú na jednotlivých databázových systémoch malé odlišnosti. Pre MySQL tabuľka vyzerá nasledovne:

[42] http://flask.pocoo.org/
[43] https://api.mongodb.com/python/
[44] https://github.com/PyMySQL/mysqlclient-python
[45] https://datastax.github.io/python-driver/
[46] http://gunicorn.org/
[47] https://hub.Docker.com/r/spakmaro/k8s-flask-app/



Obr. 3.5: Štruktúra vytváranej tabuľky pre MySQL

3.4.0.1 Príprava Docker obrazu

Docker obraz je vytvorený zo základného obrazu s nainštalovaným jazykom Python, ktorý je spravovaný Docker komunitou[48]. Potrebné súbory a zložky aplikácie sa nakopírujú do umiestnenia `/usr/src/app`. Súbor `requirements.txt` obsahuje zoznam knižníc, ktoré aplikácia využíva. Závislosti sa nainštalujú spustením príkazu `pip install -r requirements.txt`. Kontajner je spúšťaný scriptom `run_gunicorn.sh`, ktorý spustí príkaz `gunicorn --bind 0.0.0.0:5000 wsgi:app`.

```
FROM python:3
RUN apt-get install libmysqlclient-dev
RUN mkdir -p /usr/src/app
WORKDIR /usr/src/app
COPY requirements.txt ./
RUN pip install --no-cache-dir -r requirements.txt
COPY run_gunicorn.sh ./
COPY source source
EXPOSE 5000
CMD [ "./run_gunicorn.sh" ]
```

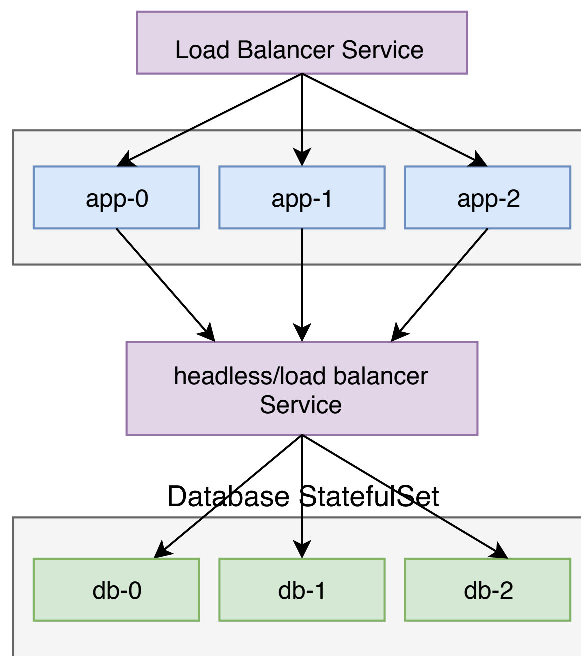
Ukážka 3.19: Dockerfile pre vytvorenie obrazu s implementovaným prototypom aplikácie

Obraz je automaticky zostavený z GitHub repozitára `slowbackspace/k8s-mongodb`[49].

[48] https://hub.Docker.com/_/python/
[49] https://github.com/slowbackspace/k8s-mongodb

3.4.0.2 Kubernetes

Pretože aplikácia nepotrebuje uchovávať stav využíva Kubernetes kontrolér `Deployment` a `Service` typu load balancer, ktorá aplikácií pridelí externú IP adresu umožňujúcu prístup k aplikácií z internetu. Požiadavka je následne smerovaná na jednu z inštancií aplikácie. Pri obsluhe požiadavky sa aplikácia pripája k databázovému systému buď prostredníctvom adries poskytovaných load balancerom (v prípade PerconXtraDB clusteru) alebo headless `Service` (MongoDB, Cassandra). Znázornenie architektúry v prostredí Kubernetes je možné vidieť na obrázku 3.6.



Obr. 3.6: Pohľad na návrh nasadenia webovej aplikácie v kontexte databázových systémov

3.4.0.3 MongoDB

V premenných prostredia v objekte `Deployment` je špecifikovaná premenná `MONGODB_CONNECTION_URI` s hodnotou `mongodb://<meno>:<heslo>@mongos:27017/admin`. hostname mongos sa vďaka `Service` s názvom mongos pripojenej k mongos routeru preloží na IP adresu Podu, na ktorom beží router.

Pre MongoDB je implementovaných niekoľko HTTP GET endpointov.

- `/enable-sharding` - Zapne sharding na ukážkovej databáze `sensors` a kolekcií `dc`

- `/insert-simple/<w>` - Vloží do kolekcie jednoduchý dokument obsahujúci niekoľko polí. Zadaním w=1 (default) PyMongo driver vyžiadá potvrdenie o zápise kolekcie na primárny uzol. zadaním w=0 sa takéto potvrdenie nevyžaduje.
- `/insert-big/<w>` - Vloží do kolekcie niekoľko megabajtový dokument.
- `/view/<read_pref>` - Zobrazí všetky dáta z kolekcie. Ich počet je možné obmedziť GET premennou `limit`. Výber uzlov na čítanie je možný prostredníctvom zadania `read_pref` v URL adrese. Preferencia je predaná PyMongo ovládaču pred čítaním dát z kolekcie. Podporované sú hodnoty primary, secondary, primary_preferred, secondary_preferred a nearest.
- `/status` - Zobrazí informácie o nasadených routeroch, shardoch, replica setoch a zoznam blokov dát, ktoré sa shardujú.

3.4.0.4 MySQL

V objekte `Deployment` je možné definovať niekoľko premenných prostredia, ktoré sa využívajú k pripojeniu k databázovemu stroju:
`MYSQL_USER`, `MYSQL_PASSWORD`, `MYSQL_HOST`, `MYSQL_DATABASE`.

Pre interakciu s MySQL je implementovaných niekoľko HTTP GET endpointov.

- `/create-table` - Vytvára tabuľku `sensors` určenú pre testovanie databáze.
- `/insert-simple` - Vloží do tabuľky jednoduchý záznam.
- `/view` - Slúži na zobrazenie záznamov z databáze. Parametrom `limit` v URL adrese je možné obmedziť počet vratených výsledkov.

3.4.0.5 Cassandra

V objekte `Deployment` je možné definovať premennú prostredia `CASSANDRA_HOST`, ktorú tvorí zoznam adries cassandra uzlov oddelených čiarkov.
`CASSANDRA_KEYSPACE` nastavuje keyspace v ktorom sa bude operovať.

Pre interakciu s Cassandrou je implementovaných niekoľko HTTP GET endpointov.

- `/create-table` - Vytvára keyspace tabuľku `sensors` určenú pre testovanie databáze.
- `/insert-simple` - Vloží do tabuľky jednoduchý záznam.
- `/view` - Slúži na zobrazenie záznamov z databáze. Parametrom `limit` v URL adrese je možné obmedziť počet vratených výsledkov.

Kapitola 4

Nasadenie

4.1 Príprava prostredia Kubernetes

Automatizované nasadenie databázových systémov MongoDB, MySQL a Cassandra prebiehalo v prostredí Kubernetes na platforme Google Cloud, Amazon Web Services a na bare metal. Na správu a nasadzovanie aplikácií do clustra slúži nástroj kubectl.

4.1.1 Google Cloud

Na infraštruktúre Google Cloud bola využitá služba Google Kubernetes Engine[50], ktorá ponúka pred-konfigurovaný Kubernetes, ktorého údržbu má na starosti poskytovateľ platformy, s možnosťou využitia ďalších služieb ako je napríklad loadbalancer alebo diskové jednotky určené pre platformu Google Cloud.Využitý bol free tier program ponúkajúci, okrem iného, zdarma manažment clustru, 5 GB na Google Cloud Storage, jednu f1-micro inštanciu a kredit 300 dolárov na ďalšie využívanie služby.

1. Inštalácia Google Cloud SDK[51], ktoré obsahuje nástroj `gcloud`
2. Inštalácia `kubectl`

   ```
   gcloud components install kubectl
   ```

3. Autorizácia nástroja pre prístup k Google Cloud účtu a špecifikácia projektu a zóny.

   ```
   gcloud init
   gcloud config set project dp-kubernetes
   gcloud config set compute/zone us-central1-a
   ```

[50] https://cloud.google.com/kubernetes-engine/
[51] https://cloud.google.com/sdk/docs/quickstarts

4. Vytvorenie clustra s 6 uzlami typu g1-small s master uzlom v zóne us-central1-a.

```
gcloud container clusters create "cluster1"
    --image-type=COS --machine-type=g1-small --zone
    us-central1-a --additional-zones us-central1-b
    --node-labels=dc=dc1
```

5. Odovzdanie prihlasovacich udajov nástroju kubectl

```
gcloud container clusters get-credentials
```

6. Označenie uzlov v zóne us-central1-a s labelom dc nastaveným na hodnotu dc1 a uzlov v zóne us-central1-b s labelom dc=dc2

```
kubectl label nodes -l
    failure-domain.beta.kubernetes.io/zone=us-central1-a
    dc=dc1
kubectl label nodes -l
    failure-domain.beta.kubernetes.io/zone=us-central1-b
    dc=dc2
```

4.1.2 Amazon Web Services

Na Amazon Web Services, v čase písania práce, nebola spristupnená služba spravovanej (managed) platformy Kubernetes, Amazon Elastic Container Service for Kubernetes[52]. Kubernetes bol manuálne nasadený na EC2[53] inštancie za pomoci nástroja Kubernetes Operations[54] (kops), ktorý slúži na nasadenie produkčného Kubernetes Clustra s podporou infraštruktúry AWS. Free tier účet má k dispozícií zdarma jedinú f1.micro inštanciu(1VCPU, 1GB RAM, 750 hodín behu mesačne) preto možnosti testovania na tejto infraštruktúre boli značne obmedzené.

1. Vytvorenie nového užívateľa v Identity and Access Management[55] (IAM) portáli

2. Inštalácia konzolového nástroja awscli prostredníctvom python package managera pip.

```
pip install awscli
```

3. Inštalácia nástroja kops[56]

[52] https://aws.amazon.com/eks/
[53] https://aws.amazon.com/ec2/
[54] https://github.com/kubernetes/kops
[55] https://aws.amazon.com/iam/
[56] https://github.com/kubernetes/kops#installing

```
kops_url=https://github.com/kubernetes/kops/releases/download
curl -LO ${kops_url}/1.9.0/kops-linux-amd64
chmod +x kops-linux-amd64
sudo mv kops-linux-amd64 /usr/local/bin/kops
```

4. Nakonfigurovanie prihlasovacích údajov užívateľa vytvoreného v prvom kroku v nástroji awscli

   ```
   aws configure
   ```

   Interaktívny sprievodca vyžiada zadanie AWS Access Key ID, AWS Secret Access Key, štandardného regiónu (použitý eu-west-1)

5. Pre použitie nástroja kops je potrebné vytvoriť užívateľa s menom kops, to je opäť možné cez IAM alebo pomocou awscli nástroja. Pre správne fungovanie je nutné užívateľovi priradiť tieto práva:

   - AmazonEC2FullAccess
   - AmazonRoute53FullAccess
   - AmazonS3FullAccess
   - IAMFullAccess
   - AmazonVPCFullAccess

   ```
   # vytvorenie uzivatelskej skupiny
   aws iam create-group --group-name kops

   # nastavenie prav
   aws iam attach-group-policy --policy-arn
       arn:aws:iam::aws:policy/AmazonEC2FullAccess
       --group-name kops
   aws iam attach-group-policy --policy-arn
       arn:aws:iam::aws:policy/AmazonRoute53FullAccess
       --group-name kops
   aws iam attach-group-policy --policy-arn
       arn:aws:iam::aws:policy/AmazonS3FullAccess
       --group-name kops
   aws iam attach-group-policy --policy-arn
       arn:aws:iam::aws:policy/IAMFullAccess --group-name kops
   aws iam attach-group-policy --policy-arn
       arn:aws:iam::aws:policy/AmazonVPCFullAccess
       --group-name kops

   # vytvorenie uzivatela
   aws iam create-user --user-name kops
   ```

```
# pridanie uzivatela do skupiny
aws iam add-user-to-group --user-name kops --group-name
    kops
```

6. Namiesto konfigurácie DNS sa na komunikáciu sa využije gossip protokol. To vyžaduje aby sa názov nasadzovaného clustra končil reťazcom ".k8s.local". [45]

```
export NAME=cluster.k8s.local
```

7. Na uloženie stavu a reprezentácia clustra je potrebné vytvoriť S3 bucket. Názov bucketu musí byť unikátny naprieč celým regiónom.

```
aws s3api create-bucket --bucket bucket-${NAME}-state
    --region eu-west-1 --create-bucket-configuration
    LocationConstraint=eu-west-1
export KOPS_STATE_STORE=s3://bucket-cluster.k8s.local-state
```

8. Nasledujúci príkaz vytvorí definíciu clustra, ďalším príkazom sa cluster nasadí. Je možné špecifikovať, okrem iného, veľkosť master inštancie obsahujúcej komponenty Kubernetes a uzlov určených pre beh nasadzovaných aplikácií.

```
# vytvorenie definicie
kops create cluster --name=${NAME} --zones=eu-west-1a
    --master-size="t2.micro" --node-size="t2.micro"
    --node-count="1" --ssh-public-key="~/.ssh/id_rsa.pub"
# nasadenie clustra
kops update cluster ${NAME} --yes
```

9. Na kontrolu úspešnosti nasadenia je možné použiť príkaz

```
kops validate cluster
```

4.1.3 Bare metal

Príprava prostredia na bare metal zahrňovala inštaláciu software pre virtualizáciu operačného systému, VirtualBox[57], nástroja Docker, ktorý slúžil ako podpora pre beh kontajnerov v Kubernetes, ale tiež aj ako nástroj na build vlastných Docker obrazov. Na samotné nasadenie Kubernetes boli použité 2 nástroje. Minikube[58], určený pre jednoduché nasadenie jedno uzlového Kubernetes clustra vo virtuálnom stroji za účelom testovania a vývoja a nástroj

[57] https://www.virtualbox.org/
[58] https://github.com/kubernetes/minikube/

kubeadm, ktorý cieli na nasadenie best-practise clustra. Inštalácia prebiehala na hosťovskom systéme macOS, kde za pomoci VirtualBox a Vagrant[59] bol spustený virtuálny stroj s Ubuntu 16.04 na ktorom ďalej prebiehala inštalácia pomocou nástroja kubeadm.

4.1.3.1 Minikube

1. Zapnutie VT-x or AMD-v v BIOSe je nevyhnutné pre beh virtualizácie
2. Inštalácia hypervisora VirtualBox, nástroja Docker a minikube.

   ```
   brew cask install minikube
   brew install virtualbox
   brew install Docker
   ```

3. Vytvorenie virtuálneho stroja pre beh single node clustra príkazom

   ```
   minikube start --memory=2048
   ```

   Tento príkaz automaticky nastaví kubectl pre komunikáciu s novo vytvoreným clustrom. Ak sa tak neudeje je nutné nastaviť kontext manuálne použitím

   ```
   kubectl config use-context minikube
   ```

Minikube VM je sprístupnená hostiteľskému systému cez IP adresu, ktorú je možné zistiť príkazom `minikube ip`. Každá Kubernetes Service typu NodePort je sprístupnená na tejto IP adrese.

Minikube podporuje diskové jednotky (PersistentVolumes) typu hostPath. Tie sú namapované na zložku vo vnútri minikube VM.

4.1.3.2 Kubeadm

1. Vytvorenie Vagrantfile súboru pre VM s Ubuntu 16.04.

   ```
   vagrant init ubuntu/xenial64
   ```

2. Nastavenie množstva pridelenej operačnej pamäte pridaním nasledujúceho kódu do vytvoreného Vagrantfile súboru

   ```
   config.vm.provider "virtualbox" do |vb|
       vb.memory = "2048"
   end
   ```

3. Spustenie VM podľa špecifikácie z Vagrantfile súboru

[59] https://www.vagrantup.com/

```
vagrant up
```

4. Inštalácia Dockeru z repozitáre Ubuntu 16.04, ktorý ponúka staršiu verziu 1.13, ktorej funkčnosť s Kubernetes je známa.

```
apt-get update
apt-get install -y Docker.io
```

5. Inštalácia kubeadm, kubelet a kubectl.

```
curl -s
    https://packages.cloud.google.com/apt/doc/apt-key.gpg
    | apt-key add -
cat <<EOF >/etc/apt/sources.list.d/kubernetes.list
deb http://apt.kubernetes.io/ kubernetes-xenial main
EOF
apt-get update
apt-get install -y kubelet kubeadm kubectl
```

6. Inicializácia Kubernetes master uzla.

```
kubeadm init
```

7. Plánovanie Podov na master uzle je štandardne pre bezpečnostné dôvody zakázané, ale pre účely práce je vrámci šetrenia výpočetných prostriedkov vhodné túto možnosť povoliť.

```
kubectl taint nodes --all node-role.kubernetes.io/master-
```

8. Inštalácia siete Weave Net, ktorá umožňuje Podom medzi sebou komunikovať.

```
sysctl net.bridge.bridge-nf-call-iptables=1
export kubever=$(kubectl version | base64 | tr -d '\n')
kubectl apply -f
    "https://cloud.weave.works/k8s/net?k8s-version=$kubever"
```

   Stav je možné skontrolovať príkazom

```
kubectl get pods --all-namespaces
```

   Požadovaný status všetkých Podov je "running"

9. Pre aktiváciu dynamického zriaďovania diskových jednotiek bol nasadený hostPath provisioner, ktorý sprostredkuje zložky z hosťovského systému Kubernetes uzlu do Podu.

```
kubectl apply -f provisioner.yaml
```

```
kind: Pod
apiVersion: v1
metadata:
  name: hostpath-provisioner
spec:
  containers:
    - name: hostpath-provisioner
      image: jaxxstorm/hostpath-provisioner:0.1
      env:
        - name: NODE_NAME
          valueFrom:
            fieldRef:
              fieldPath: spec.nodeName
      volumeMounts:
        - name: pv-volume
          mountPath: /tmp/hostpath-provisioner
  volumes:
    - name: pv-volume
      hostPath:
        path: /tmp/hostpath-provisioner
```

Ukážka 4.1: Definícia Podu v súbore provisioner.yaml, ktorý umožní dynamické zriaďovanie diskových jednotiek typu hostPath

10. Nasadenie novej StorageClass využivajúcej hostPath provisioner, ktorá sa nastaví ako štandardná pre vytváranie PV.

    ```
    kubectl apply -f standard.yaml
    ```

```
apiVersion: storage.k8s.io/v1beta1
kind: StorageClass
metadata:
  namespace: kube-system
  name: standard
  annotations:
    storageclass.beta.kubernetes.io/is-default-class:
        "true"
  labels:
    addonmanager.kubernetes.io/mode: Reconcile
provisioner: kubernetes.io/host-path
```

Ukážka 4.2: Definícia StorageClass standard.yaml

4.2 Nasadenie Cassandra

1. (voliteľné) Nasadenie požadovanej `StorageClass` pre zvýšenie výkonu v infraštruktúre Google Cloud (súbor `google_cloud_ssd.yaml`) alebo AWS (súbor `aws_ssd.yaml`). Nie je prednastavená žiadna `StorageClass` čo umožní využiť štandardné nastavenie danej platformy.

   1.1. Nastavenie `volumeClaimTemplates.metadata.storageClassName` na názov použitej `StorageClass`

   1.2. `kubectl apply -f <pozadovana-storage-class.yaml>`

2. Nasadenie `ConfigMap` objektu, kde sa nachádza script pre readiness sondu, pre-stop hook a Docker entrypoint.

   ```
   kubectl apply -f cassandra-configmap.yaml
   ```

3. Nasadenie headless `Service`

   ```
   kubectl apply -f cassandra-service.yaml
   ```

4. Konfigurácia a nasadenie Cassandra uzlov

   4.1. Voľba požadovanej veľkosti pripojenej diskovej jednotky na ktorú Cassandra ukladá dáta (prednastavené na 1 GB) v `volumeClaimTemplates.spec.resources.requests.storage` v súbore cassandra-statefulset.yaml

   4.2. V časti `spec.template.spec.containers.name.env` nastavenie premenných prostredia v súbore cassandra-statefulset.yaml

   4.3. Nasadenie `StatefulSetu`

   ```
   kubectl apply -f cassandra-statefulset.yaml
   ```

4.3 Nasadenie MongoDB

Riešenie automatizovaného nasadenia MongoDB bolo vyhotovené v 2 variantách. Prvá nasadzuje replica set o zvolenom počte uzlov. Táto metóda zabezpečuje redundanciu a rozloženie záťaže pri operáciach čítania. Druhá varianta nasadzuje shardy, kde každý shard je tvorený replica setom.

1. (voliteľné) Nasadenie požadovanej `StorageClass` pre zvýšenie výkonu v infraštruktúre Google Cloud (súbor `google_cloud_ssd.yaml`) alebo AWS (súbor `aws_ssd.yaml`). Nie je prednastavená žiadna `StorageClass` čo umožní využiť štandardné nastavenie danej platformy.

1.1. Nastavenie `volumeClaimTemplates.metadata.storageClassName` na názov použitej `StorageClass`

1.2. `kubectl apply -f <pozadovana-storage-class.yaml>`

4.3.1 Replica set

1. Príprava konfiguračného súboru `mongo-replicaset.yaml`.

   1.1. Voľba požadovanej veľkosti pripojenej diskovej jednotky na ktorú MongoDB ukladá dáta. (prednastavené na 2GB) v `volumeClaimTemplates.spec.resources.requests.storage`

   1.2. V časti `spec.template.spec.containers.name.env` nastaviť premenné prostredia `ADMIN_USERNAME` (meno admin užívateľa), `ADMIN_PASSWORD` (heslo admin užívateľa). Prípadne aj `REPSET_NAME` (názov replica setu) a `K8S_SERVICE_URL` (cesta k priradenej `Service)`

   1.3. Špecifikovanie počtu nasadených uzlov v `spec.replicas` (prednastavené na 3)

2. Vytvorenie `Secret` s kľúčom pre internú autentifikáciu MongoDB uzlov. Na vytvorenie je možné použiť priložený shell script `deploy-keyfile.sh` alebo vytvoriť kľúč zadaním:

```
kubectl create secret generic shared-bootstrap-data
    --from-file=internal-auth-mongodb-keyfile=$CESTA_K_SUBORU
```

4.3.2 Shards

1. Príprava konfiguračného súboru `mongo-shardrs-template.yaml`.

   1.1. Voľba požadovanej veľkosti pripojenej diskovej jednotky na ktorú MongoDB ukladá dáta (prednastavené na 2GB) v `volumeClaimTemplates.spec.resources.requests.storage`

   1.2. V časti `spec.template.spec.containers.name.env` nastaviť premenné prostredia `ADMIN_USERNAME` (meno admin užívateľa), `ADMIN_PASSWORD` (heslo admin užívateľa). Prípadne aj `REPSET_NAME` (názov replica setu) a `K8S_SERVICE_URL` (URL `Service`, ktorá je priradená jednotlivým shard clustrom) a `K8S_MONGOS_SERVICE_URL` (URL `Service` priradenej k mongos routeru)

   1.3. Špecifikovanie počtu nasadených uzlov v `spec.replicas` (prednastavené na 3)

2. Vytvorenie `Secret` s kľúčom pre internú autentifikáciu MongoDB uzlov. Na vytvorenie je možné použiť priložený shell script `deploy-keyfile.sh` alebo vytvoriť kľúč zadaním:

```
kubectl create secret generic shared-bootstrap-data
    --from-file=internal-auth-mongodb-keyfile=$CESTA_K_SUBORU
```

3. Spustenie shell scriptu `deploy-shard.sh` v ktorom je možné nastaviť administrátorské meno a heslo, počet nasadených shardov. Script sa postará o vygenerovanie konfiguračných yaml súborov a nasadenie všetkých potrebných komponent ako sú shard replica sety, mongos router, konfiguračný server a k nim pridružené `Services`.

4.4 Nasadenie MySQL

1. (voliteľné) Nasadenie požadovanej `StorageClass` pre zvýšenie výkonu v infraštruktúre Google Cloud (súbor `google_cloud_ssd.yaml`) alebo AWS (súbor `aws_ssd.yaml`). Nie je prednastavená žiadna `StorageClass` čo umožní využiť štandardné nastavenie danej platformy.

   1.1. Nastavenie `volumeClaimTemplates.metadata.storageClassName` na názov použitej `StorageClass`

   1.2. `kubectl apply -f <pozadovana-storage-class.yaml>`

2. Nasadenie `ConfigMap` objektu, kde sa nachádza Docker entrypoint.

   ```
   kubectl apply -f percona-configmap.yaml
   ```

3. Nasadenie headless `Service`

   ```
   kubectl apply -f percona-service.yaml
   ```

4. Konfigurácia a nasadenie Percona XtraDB uzlov

   4.1. Voľba požadovanej veľkosti pripojenej diskovej jednotky na ktorú MySQL ukladá dáta (prednastavené na 1 GB) v `volumeClaimTemplates.spec.resources.requests.storage` v súbore percona-statefulset.yaml

   4.2. V časti `spec.template.spec.containers.name.env` nastavenie premenných prostredia v súbore percona-statefulset.yaml.

   4.3. Nasadenie `StatefulSetu`

   ```
   kubectl apply -f percona-statefulset.yaml
   ```

Kapitola 5

Testovanie

Pre otestovanie funkčnosti nasadených databázovych systémov bola implementovaná web aplikácia v jazyku Python, ktorá bola umiestnená do Docker kontajneru a nasadená na platformu Kubernetes. Aplikácia sprístupňuje HTTP endpointy, ktoré slúžia ako ukážka zápisu a čítania dát z nasadených databázových systémov. Aby sa úzkym hrdlom testovania nestala implementovaná webová aplikácia, tak pre účely testovania bolo nasadených 6 inštancií (replík) tejto aplikácie. Požiadavky boli jednotlivým replikám smerované pomocou load balancer `Service`.

Záťažové testovanie sa uskutočnilo nástrojom Tsung[60], ktorý simuloval prichádzajúce HTTP požiadavky. Jeho výhodou je schopnosť simulovať veľké množstvo návštev z jedného stroja. Priebeh testu je popísaný vo formáte XML. Test je rozdelený na fázy, ktoré je možné ďalej konfigurovať.

Testovaná bola taktiež robustnosť nasadených clustrov, ich škálovanie a správanie počas zmazania ľubovoľného uzlu. Po dohode s vedúcim práce, sa pre obmedzené možnosti účtu zdarma na AWS testovanie uskutočnilo len na platforme Google Cloud, kde bol vytvorený cluster so 6 uzlami s 1CPU a 3.75 GB pamäte. Pre dosiahnutie väčšieho efektu pri škálovaní a zjednodušenie testovania bolo obmedzené využitie CPU daným kontajnerom (20 % z 1 CPU), čo simulovalo rýchlejšie narazenie na limit výkonu uzlu.

5.1 MongoDB

Použitím scriptu `deploy-shards.sh` sa vytvorí sharded cluster o 3 shardoch, kde každý shard je tvorený replica setom o 3 uzloch. Jednotlivé Pody v danom shard replica sete majú hostname v tvare mongod-shardX-0, kde X je index shardu.

Príkazom `kubectl get pods` získame zoznam nasadených Podov

[60]`http://tsung.erlang-projects.org/`

```
NAME READY STATUS RESTARTS AGE
mongod-configdb-0 1/1 Running 0 27m
mongod-configdb-1 1/1 Running 0 26m
mongod-configdb-2 1/1 Running 0 25m
mongod-shard1-0 1/1 Running 0 27m
mongod-shard1-1 1/1 Running 0 26m
mongod-shard1-2 1/1 Running 0 25m
mongod-shard2-0 1/1 Running 0 27m
mongod-shard2-1 1/1 Running 0 26m
mongod-shard2-2 1/1 Running 0 25m
mongod-shard3-0 1/1 Running 0 27m
mongod-shard3-1 1/1 Running 0 26m
mongod-shard3-2 1/1 Running 0 25m
mongos-68b55987db-wh9dn 1/1 Running 0 26m
mongos-68b55987db-das8o 1/1 Running 0 26m
```

Ukážka 5.1: Výstup príkazu kubectl get pods

Z výpisu možno vidieť úspešné nasadenie 3 Podov pre replica set konfiguračného serveru, 3 Pody pre každý shard replica set a 2 inštancie mongos routera. Všetky Pody majú status `runnnig`, čo značí, že cluster je pripravený spracovávať požiadavky.

Spustením Mongo shellu z Podu z prvého shardu príkazom `mongo admin -u admin -p <heslo>` a zadaním príkazu `rs.status()`, môžeme vidieť stav uzlov v replica sete.

```
{
"set" : "Shard1RepSet",
"members" : [
{
    "_id" : 0,
    "name" : "mongod-shard1-0.mongodb-shard1-service:27017",
    "stateStr" : "PRIMARY",
    "self" : true,
    ...
},{
    "_id" : 1,
    "name" : "mongod-shard1-1.mongodb-shard1-service:27017",
    "stateStr" : "SECONDARY",
    "syncingTo" :
       "mongod-shard1-0.mongodb-shard1-service:27017",
    ...
},{
    "_id" : 2,
    "name" : "mongod-shard1-2.mongodb-shard1-service:27017",
```

```
    "stateStr" : "SECONDARY",
    "syncingTo" :
       "mongod-shard1-1.mongodb-shard1-service:27017",
    ...
}]
}
```

Ukážka 5.2: Výstup príkazu rs.status()

Z výstupu príkazu možno vidieť, že replica set pre prvý shard bol úspešne inicializovaný. Ako primárny uzol bol zvolený prvý nasadzovaný Pod z replica setu.

Pre overenie inicializácie shardingu sa pripojíme k mongos routeru príkazom `mongo admin --host mongos -u admin -p <heslo>` a spustíme: `sh.status()`.

```
shards:
  { "_id" : "Shard1RepSet", "host" :
      "Shard1RepSet/mongod-shard1-0.mongodb-shard1-service:27017,
      mongod-shard1-1.mongodb-shard1-service:27017,
      mongod-shard1-2.mongodb-shard1-service:27017", "state" :
      1 }
  { "_id" : "Shard2RepSet", "host" :
      "Shard2RepSet/mongod-shard2-0.mongodb-shard2-service:27017,
      mongod-shard2-1.mongodb-shard2-service:27017,
      mongod-shard2-2.mongodb-shard2-service:27017", "state" :
      1 }
  { "_id" : "Shard3RepSet", "host" :
      "Shard3RepSet/mongod-shard3-0.mongodb-shard3-service:27017,
      mongod-shard3-1.mongodb-shard3-service:27017,
      mongod-shard3-2.mongodb-shard3-service:27017", "state" :
      1 }
active mongoses:
  "3.6.3" : 1
autosplit:
  Currently enabled: yes
balancer:
  Currently enabled: yes
```

Ukážka 5.3: Časť výstupu príkazu sh.status()

Z výstupu možno vidieť pridané 3 shardy (Shard1RepSet, Shard2RepSet, Shard3RepSet), kde každý tvoria 3 uzly a jeden mongos router.

5.1.1 Škálovanie replica setu

Podporované je natívne škálovanie replica setu prostredníctvom Kubernetes mechanizmov. Nasledujúci príkaz rozšíri prvý shard replica set na 5 uzlov.

```
kubectl scale statefulsets mongod-shard1 --replicas=5
```

Zadaním príkazu `rs.status()` v mongod-shard1 replica sete môžeme vidieť, že nové uzly sa úspešne pripojili do clustra.

```
"members" : [
...
{
    "_id" : 3,
    "name" : "mongod-shard1-3.mongodb-shard1-service:27017",
    "stateStr" : "SECONDARY",
    "syncingTo" :
        "mongod-shard1-2.mongodb-shard1-service:27017",
    ...
},{
    "_id" : 4,
    "name" : "mongod-shard1-4.mongodb-shard1-service:27017",
    "stateStr" : "SECONDARY",
    "syncingTo" :
        "mongod-shard1-1.mongodb-shard1-service:27017",
    ...
}]
```

Ukážka 5.4: Výstup príkazu rs.status() s pridanými novými uzlami

Škálovanie clustra smerom nadol taktiež funguje. Uzly sú automaticky odstránené zavolaním príkazu rs.remove() v v pre-stop scripte.

5.1.2 Škálovanie počtu shardov

Pridanie shardu je možné vygenerovaním konfigurácie nového `StatefulSetu` alebo spustením scriptu `deploy-shards.sh` s nastavenou premennou `SHARDS` na požadovanú hodnotu. Novo nasadený shard je automatický pridaný do clustru.

```
shards:
...
  { "_id" : "Shard4RepSet", "host" :
      "Shard4RepSet/mongod-shard4-0.mongodb-shard4-service:27017,
      mongod-shard4-1.mongodb-shard4-service:27017,
      mongod-shard4-2.mongodb-shard4-service:27017" }
```

Ukážka 5.5: Časť výstupu príkazu sh.status() zobrazujúca pridaný shard

Škálovanie smerom nadol automatizovane nie je možné, kedže Kubernetes neposkytuje spôsob ako zistiť, či ide o odstránenie Podov za účelom aktualizácie/reštartu, prípadne krátkodobý výpadok alebo trvalé odstránenie StatefulSetu. Taktiež odobranie shardu môže byť pomalá operácia a Kubernetes odstráni Pod po uplynutí `terminationGracePeriodSeconds` sekúnd nezávisle od dokončenia behu pre-stop hooku, ktorý by mohol mať na starosti odobratie uzlu. [12]

5.1.3 Havária uzlu

Havária Kubernetes uzlu bola simulovaná zmazaním Podu. Kubernetes kontrolér Pod znovu vytvorí a ten sa pripojí do replica setu. Špeciálne pri zmazaní primárneho uzlu, MongoDB zvolí nový primárny uzol a obnovený Pod je pripojený ako sekundárny uzol. Funkčnosť clustra nie je nijako narušená.

5.1.4 Sharding

Opakovaným prístupom na HTTP endpoint `/insert-big`, sa zapíšu do databázy `sensors` a kolekcie `dc` testovacie dáta. Štandardne jednotka na prerozdelenie dát medzi shardy (chunk) má veľkosť 64MB. Po prekročení tejto veľkosti sa dáta rozdelia na jednotlivé chunky, ktoré sa rozložia medzi jednotlivé shard replica sety.

Na opakované odoslanie požiadavky sa použil nástroj Apache Benchmark (ab), spustením príkazu `ab -c 2 -n 100 <url>/insert-big`

Na HTTP endpointe /status alebo zadaním príkazu v Mongo shelli `sh.status()` môžeme vidieť prerozdelenie dát do shardov.

Výpis uložených dát je sprístupnený na `/view/<read_pref`>, kde read_pref je preferencia čítania. Požiadavka na `/view/primary` číta dáta z primárneho uzlu, požiadavka na `/view/secondary` zo sekundárneho uzlu.

```
sensors.dc
  shard key: { "_id" : 1 }
  unique: false
  balancing: true
  chunks:
    Shard1RepSet 2
    Shard2RepSet 3
    Shard3RepSet 2
```

Ukážka 5.6: Časť výstupu príkazu sh.status()

Z výstupu možno vidieť vytvorenie 7 blokov dát, ktoré su prerozdelené medzi 3 shardy.

Tabuľka 5.1: Výsledky záťažového testu MongoDB

# uzlov	Highest Rate [req/s]	Mean Rate [req/s]	Mean duration [s]
1	28.5	23.64	22.91
3	59.5	43.88	3.43
5	83.6	62.23	0.80

5.1.5 Záťažové testovanie

Test bol nakonfigurovaný v 2 fázach. V prvej sa odosielalo 50 požiadaviek za sekundu, v 2 fáze 100 požiadaviek za sekundu na HTTP endpoint `/view`. Test prebehol na jednom (čítanie z primárneho uzlu), troch (2 sekundárne) a piatich uzloch (4 sekundárne). Testovanie prebiehalo na MongoDB replica sete.

Z výsledkov (tabuľka 5.1) možno vidieť, že čítaním zo sekundárnych uzlov sa zvýšil výkon aplikácie. Výkon taktiež stúpa so zvyšovaním počtu uzlov. Priemerná priepustnosť stúpla z 23.64 na 62.23 požiadaviek za sekundu. Priemerné trvanie požiadavky kleslo z takmer 23 sekúnd na 0.8 sekundy. Vyťaženie CPU sekundárnych uzlov bolo počas testu približne rovnaké.

5.2 Cassandra

Scriptom `deploy.sh` sa nasadí Cassandra cluster s 3 uzlami. Po pripojení na bežiaci uzol a zadaní príkazu `nodetool status`, môžeme vidieť že všetky 3 uzly sú úspešne pripojené v clustri.

```
Datacenter: DC1
===============
Status=Up/Down
|/ State=Normal/Leaving/Joining/Moving
-- Address Load Tokens Owns (effective) Host ID Rack
UN 10.8.12.44 93.95 KiB 256 65.5% 3e898b3b... Rack1
UN 10.8.11.76 69.84 KiB 256 66.7% 7fcd24b9...  Rack1
UN 10.8.13.47 108.61 KiB 256 67.8% d6ba32f9...  Rack1
```

Ukážka 5.7: Zoznam bežiacich uzlov Cassandra clustru

5.2.1 Škálovanie počtu uzlov

Škálovanie je možné úpravou počtu replík v `StatefulSet objekte` alebo zadaním príkazu

```
kubectl scale statefulsets cassandra --replicas=5
```

Novo vytvorené pody sa postupne pripoja do clustra vďaka komunikácií so seed uzlom. Pri zmene počtu replík pre daný keyspace na 5 možno vidieť, že

repliky sa distribuujú medzi uzly, čiže každý uzol uchováva 100 % dát. Zmena sa realizovala príkazom v cqlsh a spustením opravy uzlov.

```
ALTER KEYSPACE testing WITH REPLICATION = {'class':
    'NetworkTopologyStrategy', 'DC1': '<pocet-replik>'};

for (( i=0;i$pocet_uzlov;i++)); do
  kubectl exec -t cassandra-$i -- nodetool repair -pr testing;
done
```

```
-- Address Load Tokens Owns (effective) Host ID Rack
UN 10.8.12.46 145.77 KiB 256 100% 3e898b3b... Rack1
UN 10.8.11.78 126.66 KiB 256 100% 7fcd24b9... Rack1
UN 10.8.13.47 120.28 KiB 256 100% d6ba32f9... Rack1
UN 10.8.9.48 144.96 KiB 256 100% cc5f608f... Rack1
UN 10.8.8.20 148.64 KiB 256 100% 414d566c... Rack1
```

Ukážka 5.8: Zoznam bežiacich uzlov Cassandra po škálovaní clustru

Pri škálovaní smerom nadol sú dáta z odstráneného uzlu presunuté na iný uzol.

5.2.2 Zaťažové testovanie

Test bol nakofingurovaný v 2 fázach, v prvej sa odosielalo 15 požiadaviek za sekundu, v druhej 20 na cieľový HTTP endpoint `/view`. Počet replík dát bol zvyšovaný vzhľadom k počtu uzlov.

Tabuľka 5.2: Výsledky záťažového testu Cassandra

# uzlov	Highest Rate [req/s]	Mean Rate [req/s]	Mean duration [s]
3	7.4	5.36	44.98
4	17	11.1	2.20
5	20	13.06	4.46

Z výsledkov (tabuľka 5.2) možno vidieť, že priemerná priepustnosť požiadaviek stúpla z 5.38 na 11.1 požiadaviek za sekundu. Priemerné trvanie požiadavky kleslo z približne 45 sekúnd (pri 3 uzloch) na 4.46 (pri 5 uzloch) sekundy. Vyťaženie CPU uzlov počas testu bolo približne rovnaké. Veľké množstvo CPU si vyžiadala readiness sonda, ktorá sa spúšťala každých 60 sekúnd a spôsobovala preťaženie uzla.

5.3 MySQL

Scriptom `deploy.sh` sa nasadí Percona XtraDB Cluster s 3 uzlami. Hostname jednotlivých Podov je v tvare percona-galera-X, kde X je index Podu.

Príkazom `kubectl get pods` opäť získame zoznam nasadených Podov.

```
percona-galera-0 1/1 Running 0 46m
percona-galera-1 1/1 Running 0 45m
percona-galera-2 1/1 Running 0 43m
```

Ukážka 5.9: Zoznam bežiacich uzlov Percona XtraDB clustru

Z výstupu možno vidieť že sú všetky v stave `running`.

K Podom je pripojená `Service` typu load balancer. Pripojením na IP adresu poskytnutú load balancerom prostredníctvom MySQL klienta `mysql -host <ip> -p<heslo>` sa pripojíme na ľubovoľný Pod z clustra.

Zadaním nasledujúceho príkazu môžeme overiť počet uzlov v clustri.

`SHOW GLOBAL STATUS LIKE wsrep_cluster_size`

```
mysql> SHOW GLOBAL STATUS LIKE 'wsrep_cluster_size';
+--------------------+-------+
| Variable_name | Value |
+--------------------+-------+
| wsrep_cluster_size | 3 |
+--------------------+-------+
```

Ukážka 5.10: Výpis premennej `wsrep_cluster_size` obsahujúcej počet uzlov v C

5.3.1 Škálovanie počtu uzlov

Škálovanie je možné úpravou počtu replík v `StatefulSet objekte` alebo zadaním príkazu

`kubectl scale statefulsets percona-galera --replicas=5`

Uzly sa úspešne pripoja do clustra a pomocou SST replikujú dáta z existujúcich uzlov. Status každého uzla je `Primary`. Stav pripojenia uzla do clustra je možné overiť výpisom premennej `wsrep_local_state`.

```
mysql> SHOW GLOBAL STATUS LIKE 'wsrep_local_state_comment';
+---------------------------+--------+
| Variable_name | Value |
+---------------------------+--------+
| wsrep_local_state_comment | Synced |
+---------------------------+--------+
```

Ukážka 5.11: Výpis premennej `wsrep_local_state`

Pri škálovaní smerom nadol príkazom

`kubectl scale statefulsets percona-galera --replicas=3`

sú prebytočné Pody postupne odstránené a funkčnosť clustra je zachovaná.

5.3.2 Havária uzlu

Havária Kubernetes uzlu bola simulovaná zmazaním Podu. Kubernetes kontrolér Pod znovu vytvorí a ten sa pripojí do clustra. Od existujúceho uzlu v clustri vyžiada aktuálne dáta prostredníctvom IST/SST protokolu. Keďže je možné zapisovať na každý uzol a naviazaná `Service`, ktorú vyžíva klientská aplikácia, automaticky prestane smerovať požiadavky na odstránený uzol je činnosť clustra nenarušená.

5.3.3 Záťažové testovanie

Pre spustením testu sa do databázy vložilo približne 250 tisíc záznamov odosielaním požiadaviek na `/insert-simple`.

Test bol nakonfigurovaný v 3 fázach. V prvej sa odosielalo 30 požiadaviek za sekundu, v druhej fáze 40 požiadaviek a v poslednej tretej fáze 50 požiadaviek za sekundu na HTTP endpoint `/view`. Test prebehol na clustri o veľkosti 3,5 a 7 uzlov.

Tabuľka 5.3: Výsledky záťažového testu nasadeného PerconaXtraDB clustru

# uzlov	Highest Rate [req/s]	Mean Rate [req/s]	Mean duration [s]
3	28.1	22.15	18.53
5	37.5	29.54	5.26
7	44.7	36.71	1.27

Z výsledkov (tabuľka 5.3) možno vidieť, že čítaním z viacerých uzlov sa zvýšil výkon aplikácie. Najvyššia priepustnosť požiadaviek stúpla z 28.1 na 44.7 požiadaviek za sekundu. Priemerné trvanie požiadavky kleslo z približne 22 sekúnd na 1.27 sekundy. Vyťaženie CPU uzlov počas testu bolo vďaka load balancingu požiadavok približne rovnaké.

Záver

Cieľom práce bolo zanalyzovať platformu Kubernetes, jej možnosti nasadenia databázových systémov a tiež navrhnúť a implementovať vlastné alebo doplniť existujúce riešenia pre automatizované nasadenie.

Súčasťou práce bola tiež príprava platformy Kubernetes na infraštruktúre Google Kubernetes Engine a inštalácia na AWS a vlastnom hardware. Kubernetes poskytuje dostatočnú abstrakciu nad infraštruktúrou a umožňuje tak prípravu na infraštruktúre nezávislého riešenia nasadenia bez väčších komplikácií. Rozlišovacím prvkom je podpora perzistentného diskového úložiska, ktorého parametre a definícia sa môže na jednotlivých infraštruktúrach líšiť.

Samotný návrh riešenia nasadenia databáz prihliada na potreby vysokej dostupnosti a škálovania. Pre beh na platforme Kubernetes sú jednotlivé riešenia zabalené do Docker kontajneru. Boli využité ponúkané možnosti platformy Kubernetes, ako sú kontroléry pre nasadenie stateful aplikácií, sondy kontrolujúce funkčnosť systémov, nastavenie plánovania kontajnerov na uzly pomocou afinity alebo scripty spúšťané po vytvorení a pred odstránením kontajneru. V prípade MySQL bolo implementované aj pravidelné zálohovanie prostredníctvom Kubernetes konceptu cron úloh.

Testovanie funkčnosti a robustnosti nasadených databázových systémov prebehlo na infraštruktúre Google Cloud. Testovanie prístupu k databáze a spracovanie požiadaviek bolo uskutočnené na implementovanej ukážkovej webovej aplikácii napísanej v jazyku Python. Aplikácia bola taktiež zabalená do Docker kontajneru a bola pre ňu vytvorená konfigurácia pre nasadenie na Kubernetes.

Pre pohodlnejšiu konfiguráciu pri nasadzovaní a pokročilé možnosti spravovania kontajnerov môže byť v budúcnosti implementované rozšírenie Kubernetes API pomocou Operátorov.

Literatúra

[1] The Kubernetes Authors: Kubernetes Documentation. *Viewing Pods and Nodes* [online]. [cit. 2018-02-04]. Dostupné z: https://kubernetes.io/docs/tutorials/kubernetes-basics/explore-intro/.

[2] MongoDB: The MongoDB 3.2 Manual. *Sharding Introduction* [online]. [cit. 2016-03-21]. Dostupné z: https://docs.mongodb.org/manual/core/sharding-introduction/.

[3] Abbas, A.: A quick glance into Cassandra. *NoSQL and Cloud Databases Community* [online], [cit. 2016-03-21]. Dostupné z: http://skillachie.com/2014/07/25/mysql-high-availability-architectures/.

[4] Docker Inc: *What is container* [online]. [cit. 2018-02-02]. Dostupné z: https://www.docker.com/what-container.

[5] Docker Inc: Docker frequently asked questions (FAQ). *What does Docker technology add to just plain LXC?* [online]. [cit. 2018-02-02]. Dostupné z: https://docs.docker.com/engine/faq/#what-does-docker-technology-add-to-just-plain-lxc.

[6] Docker Inc: Docker frequently asked questions (FAQ). *What platforms does Docker run on?* [online]. [cit. 2018-02-02]. Dostupné z: https://docs.docker.com/engine/faq/#what-platforms-does-docker-run-on.

[7] Docker Inc: Docker Documentation. *Docker overview* [online]. [cit. 2018-02-02]. Dostupné z: https://docs.docker.com/engine/docker-overview/#docker-objects.

[8] Verma, A.; Pedrosa, L.; Korupolu, M. R.; aj.: Large-scale cluster management at Google with Borg. In *Proceedings of the European Conference on Computer Systems (EuroSys)*, Bordeaux, France, 2015.

[9] The Kubernetes Authors: Kubernetes Documentation. *Pods* [online]. [cit. 2018-02-04]. Dostupné z: `https://kubernetes.io/docs/concepts/workloads/pods/pod/`.

[10] The Kubernetes Authors: Kubernetes Documentation. *Assigning Pods to Nodes* [online]. [cit. 2018-02-04]. Dostupné z: `https://kubernetes.io/docs/concepts/configuration/assign-pod-node/#affinity-and-anti-affinity`.

[11] The Kubernetes Authors: Kubernetes Documentation. *Volumes* [online]. [cit. 2018-02-04]. Dostupné z: `https://kubernetes.io/docs/concepts/workloads/pods/pod-lifecycle/#when-should-you-use-liveness-or-readiness-probes`.

[12] The Kubernetes Authors: Kubernetes Documentation. *Container Lifecycle Hooks* [online]. [cit. 2018-02-04]. Dostupné z: `https://kubernetes.io/docs/concepts/containers/container-lifecycle-hooks/`.

[13] Abbassi, P.: Understanding Basic Kubernetes Concepts II - Using Deployments To Manage Your Services Declaratively. *Giant Swarm* [online], [cit. 2016-03-21]. Dostupné z: `https://blog.giantswarm.io/understanding-basic-kubernetes-concepts-using-deployments-manage-services-declaratively/`.

[14] The Kubernetes Authors: Kubernetes Documentation. *Volumes* [online]. [cit. 2018-02-04]. Dostupné z: `https://kubernetes.io/docs/concepts/workloads/controllers/statefulset/`.

[15] The Kubernetes Authors: Kubernetes Documentation. *CronJob* [online]. [cit. 2018-02-04]. Dostupné z: `https://kubernetes.io/docs/concepts/workloads/controllers/cron-jobs/`.

[16] The Kubernetes Authors: Kubernetes Documentation. *Volumes* [online]. [cit. 2018-02-04]. Dostupné z: `https://kubernetes.io/docs/concepts/storage/volumes/`.

[17] Amazon Web Services: AWS Documentation. *Amazon EBS Volume Types* [online]. [cit. 2016-03-21]. Dostupné z: `https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/EBSVolumeTypes.html`.

[18] Abbassi, P.: Understanding Basic Kubernetes Concepts III - Services Give You Abstraction. *Giant Swarm* [online], [cit. 2016-03-21]. Dostupné z: `https://blog.giantswarm.io/basic-kubernetes-concepts-iii-services-give-abstraction/`.

[19] Shalom, N.: Scale-out vs Scale-up. *Nati Shalom's Blog* [online], [cit. 2016-03-21]. Dostupné z: http://natishalom.typepad.com/nati_shaloms_blog/2010/09/scale-up-vs-scale-out.html.

[20] Henderson, C.: *Building Scalable Web Sites*. Sebastopol, CA, USA: O'Reilly Media, první vydání, 2008, ISBN 978-0-596-10235-7, 352 s.

[21] Cloud 66: RealScale Architecture from Cloud 66. *Database Server Scaling Strategies* [online]. [cit. 2018-02-04]. Dostupné z: http://realscale.cloud66.com/database-server-scaling-strategies/.

[22] Bartholomew, D.: Database Master-Slave Replication in the Cloud. *MariaDB Resources* [online], [cit. 2016-03-21]. Dostupné z: https://mariadb.com/resources/blog/database-master-slave-replication-cloud.

[23] Campbell, D.: MySQL High Availability Architectures. *Skillachie's Blog* [online], [cit. 2016-03-21]. Dostupné z: http://skillachie.com/2014/07/25/mysql-high-availability-architectures/.

[24] Severalnines AB.: Galera Cluster for MySQL. *Learn the difference between Multi-Master and Multi-Source replication* [online]. [cit. 2018-02-02]. Dostupné z: https://severalnines.com/blog/learn-difference-between-multi-master-and-multi-source-replication.

[25] MongoDB: The MongoDB 3.2 Manual. *Introduction to MongoDB* [online]. [cit. 2018-02-02]. Dostupné z: https://docs.mongodb.org/manual/introduction/.

[26] MongoDB: The MongoDB 3.6 Manual. *Master Slave Replication* [online]. [cit. 2018-02-02]. Dostupné z: https://docs.mongodb.com/manual/core/master-slave/.

[27] MongoDB: The MongoDB 3.6 Manual. *Replication Introduction* [online]. [cit. 2018-02-02]. Dostupné z: https://docs.mongodb.org/manual/core/replication-introduction/.

[28] MongoDB: The MongoDB 3.6 Manual. *Sharded Cluster Query Router* [online]. [cit. 2018-02-02]. Dostupné z: https://docs.mongodb.com/manual/core/sharded-cluster-query-router/.

[29] MongoDB: The MongoDB 3.6 Manual. *Config Servers* [online]. [cit. 2018-02-02]. Dostupné z: https://docs.mongodb.com/manual/core/sharded-cluster-config-servers/.

[30] Apache Software Foundation: Apache Cassandra Documentation. *What are seeds?* [online]. [cit. 2018-02-02]. Dostupné z:

http://cassandra.apache.org/doc/latest/faq/index.html#what-are-seeds.

[31] DataStax: Apache Cassandra™ 3.0 for DSE 5.0. *Partitioners* [online]. [cit. 2018-02-02]. Dostupné z: https://docs.datastax.com/en/cassandra/3.0/cassandra/architecture/archPartitionerAbout.html.

[32] DataStax: Apache Cassandra™ 3.0 for DSE 5.0. *How are write requests accomplished?* [online]. [cit. 2018-02-02]. Dostupné z: https://docs.datastax.com/en/cassandra/3.0/cassandra/dml/dmlClientRequestsWrite.html?hl=requests.

[33] Apache Cassandra [online]. [cit. 2016-02-02]. Dostupné z: http://cassandra.apache.org/.

[34] DataStax: Apache Cassandra™ 3.0 for DSE 5.0. *Data replication* [online]. [cit. 2018-02-02]. Dostupné z: https://docs.datastax.com/en/cassandra/3.0/cassandra/architecture/archDataDistributeReplication.html.

[35] CASARES, J.: Multi-datacenter Replication in Cassandra. *DataStax Blog* [online], [cit. 2018-02-02]. Dostupné z: https://www.datastax.com/dev/blog/multi-datacenter-replication.

[36] Neeraj, N.: *Mastering Apache Cassandra - Second Edition*. Packt Publishing - ebooks Account, 2015, ISBN 1784392618. Dostupné z: https://www.amazon.com/Mastering-Apache-Cassandra-Nishant-Neeraj/dp/1784392618

[37] Oracle Corporation: MySQL 5.7 Reference Manual. *Replication* [online]. [cit. 2018-02-02]. Dostupné z: https://dev.mysql.com/doc/refman/5.7/en/replication.html.

[38] Malkowski, P.: Quest for Better Replication in MySQL: Galera vs. Group Replication. *Percona Database Performance Blog* [online], [cit. 2016-03-21]. Dostupné z: https://www.percona.com/blog/2017/02/24/battle-for-synchronous-replication-in-mysql-galera-vs-group-replication/.

[39] Gryp, K.; Sivaraman, R.: Percona XtraDB Cluster, Galera Cluster, MySQL Group Replication, 2017, percona Live Open Source Database Conference. Dostupné z: https://www.percona.com/live/17/sessions/percona-xtradb-cluster-galera-cluster-mysql-group-replication

[40] MongoDB: The MongoDB 3.6 Manual. *Configuration File Options* [online]. [cit. 2018-02-02]. Dostupné z: `https://docs.mongodb.com/manual/reference/configuration-options/#storage.mmapv1.smallFiles`.

[41] MongoDB: The MongoDB 3.6 Manual. *Production Notes* [online]. [cit. 2018-02-02]. Dostupné z: `https://docs.mongodb.com/manual/administration/production-notes/#mongodb-and-numa-hardware`.

[42] MongoDB: The MongoDB 3.6 Manual. *Transparent Huge Pages* [online]. [cit. 2018-02-02]. Dostupné z: `https://docs.mongodb.com/manual/tutorial/transparent-huge-pages/`.

[43] The Kubernetes Authors: Kubernetes Documentation. *ReplicationController* [online]. [cit. 2018-02-04]. Dostupné z: `https://kubernetes.io/docs/concepts/workloads/controllers/replicationcontroller/`.

[44] Oracle Corporation: MySQL 5.7 Reference Manual. *Client for Administering a MySQL Server* [online]. [cit. 2018-02-02]. Dostupné z: `https://dev.mysql.com/doc/refman/8.0/en/mysqladmin.html`.

[45] kops - Kubernetes Operations. [cit. 2016-03-21]. Dostupné z: `https://github.com/kubernetes/kops/blob/master/docs/aws.md#configure-dns`.

Dodatok A

Zoznam použitých skratiek

API Application Programming Interface

AWS Amazon Web Services

CaaS Containers as a service

CLI Command Line Interface

DNS Domain Name System

GCE Google Cloud Engine

HTTP Hypertext Transfer Protocol

IaaS Infrastructure as a service

JSON JavaScript Object Notation

PaaS Platform as a service

PV Persistent Volume

PVC Persistent Volume Claim

RBAC Role-based access control

SQL Structured Query Language

URL Uniform Resource Locator

VM Virtual Machine

WSGI Web Server Gateway Interface

XML Extensible Markup Language

YAML Ain't Markup Language

Dodatok B

Obsah priloženého CD

```
│ readme.txt ........................ stručný popis obsahu CD
├── src
│   ├── impl ........................ zdrojové kódy implementácie
│   └── thesis ...................... zdrojová forma práce vo formáte LaTeX
├── tests ........................... výsledky záťažových testov
└── text ............................ text práce
    ├── thesis.pdf .................. text práce vo formáte PDF
    └── thesis.ps ................... text práce vo formáte PS
```