VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INFORMAČNÍCH SYSTÉMŮ
DEPARTMENT OF INFORMATION SYSTEMS

IMPLEMENTACE SLUŽBY POSKYTUJÍCÍ FRONTU ZPRÁV V TECHNOLOGII CLOUD COMPUTING

IMPLEMENTATION OF MESSAGE QUEUE AS A SERVICE IN CLOUD COMPUTING

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — **Bc. TOMÁŠ HANUS**
AUTHOR

VEDOUCÍ PRÁCE — **RNDr. MAREK RYCHLÝ, Ph.D.**
SUPERVISOR

BRNO 2018

Zadání diplomové práce/20635/2017/xhanus11

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav informačních systémů Akademický rok 2017/2018

Zadání diplomové práce

Řešitel: **Hanus Tomáš, Bc.**
Obor: Informační systémy
Téma: **Implementace služby poskytující frontu zpráv v technologii cloud computing**
Implementation of Message Queue as a Service in Cloud Computing
Kategorie: Softwarové inženýrství

Pokyny:

1. Seznamte se s přístupy a nástroji pro předávání zpráv v distribuovaných frontách (angl. "message queue", dále jen MQ; např. Apache ActiveMQ, Kafka, RabbitMQ aj.).
2. Prozkoumejte různé modely pro MQ, vlastnosti front a jednotlivých zpráv (např. datové typy), způsob jejich registrace, správy, škálovatelnosti, atd. Popište způsob implementace takových modelů a vlastností v nástrojích z předchozího kroku.
3. Navrhněte službu technologie cloud computing s vlastním programovým rozhraním, která umožňuje aplikacím vytvořit a spravovat fronty MQ různých modelů a vlastností, nezávisle na jejich implementaci.
4. Po konzultaci s vedoucím Implementujte navrženou službu s RESTful rozhraním a vybranou technologií (MQ nástrojem). Umožněte snadnou změnu použité technologie za jinou.
5. Řešení otestujte, zdokumentujte, zhodnoťte a zveřejněte jako open-source.

Literatura:

- Snyder, Bruce, Dejan Bosnanac, Rob Davies. *ActiveMQ in action*. Vol. 47. Greenwich Conn.: Manning, 2011.
- Garg, Nishant. *Apache Kafka*. Packt Publishing Ltd, 2013.
- Boschi, Sigismondo, Gabriele Santomaggio. *RabbitMQ cookbook*. Packt Publishing Ltd, 2013.

Při obhajobě semestrální části projektu je požadováno:

- Body 1 až 3.

Podrobné závazné pokyny pro vypracování diplomové práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva diplomové práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap, které byly vyřešeny v rámci dřívějších projektů (30 až 40% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Rychlý Marek, RNDr., Ph.D.**, UIFS FIT VUT
Datum zadání: 1. listopadu 2017
Datum odevzdání: 23. května 2018

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav informačních systémů
612 66 Brno, Božetěchova 2

L.S.

doc. Dr. Ing. Dušan Kolář
vedoucí ústavu

Abstrakt

Práca hovorí o rôznych spôsoboch komunikácie medzi komponentami distribuovaného systému. Popisuje komunikáciu formou výmeny správ a zároveň pripomína, aké existujú alternatívy. Približuje rôzne modely výmeny správ, rôzne formáty správ a rôzne špecifikácie. Sú detailne predstavené komerčné nástroje ActiveMQ, RabbitMq a Kafka. Dôraz je kladený na predstavenie princípu, akým vymieňajú správy, možnosti ich škálovateľnosti a ďalšie. Na základe popísaných vlastností je navrhnutá webová služba. Jej účelom je správa a monitorovanie nástroja podľa používateľovho výberu a jednoduchá zmena použitého nástroja za iný. Navrhnutá aplikácia je implementovaná v jazyku Kotlin pre zvolený nástroj RabbitMQ. Implementované riešenie umožňuje jednoduchú výmenu správ cez REST api.

Abstract

Thesis discusses about different ways of a communication between components of a distributed system. It describes a communication using a message exchange and at the same time talks about other alternatives. It adds details about various models of a message exchange, various message types and about various specifications as well. Commercial tools ActiveMQ, RabbitMQ and Kafka are presented. Special emphasis is placed on describing the way these tools exchange messages, scalability options and others. The web service is designed according to the described features. Its main purpose is management and monitoring of the tool by user choice and easy replacement of this tool with another one. Designed application is implemented using the Kotlin language for selected tool RabbitMQ. The implemented solution allows a simple exchange of messages through the REST api.

Kľúčové slová

fronta správ, téma, RabbitMQ, ActiveMQ, Kafka, administračný nástroj, monitorovanie

Keywords

message queue, topic, RabbitMQ, ActiveMQ, Kafka, administration tool, monitoring

Citácia

HANUS, Tomáš. *Implementace služby poskytující frontu zpráv v technologii cloud computing*. Brno, 2018. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce RNDr. Marek Rychlý, Ph.D.

Implementace služby poskytující frontu zpráv v technologii cloud computing

Prehlásenie

Prehlasujem, že som túto diplomovú prácu vypracoval samostatne pod vedením pána RNDr. Mareka Rychlého Ph.D. Uviedol som všetky literárne pramene a publikácie, z ktorých som čerpal.

.....
Tomáš Hanus
21. mája 2018

Poďakovanie

Chcel by som poďakovať svojmu vedúcemu RNDr. Marekovi Rychlému Ph.D. za jeho odbornú pomoc a rady, ktoré mi poskytol.

Obsah

Kapitola 1

Úvod

Aplikácia, ktorá beží úplne izolovane je v súčasnej dobe skôr zvláštnosťou ako pravidlom. Častá je potreba získavať a synchronizovať dáta z viacerých zdrojov. Takéto zdroje nemusia byť nutne dostupné na jednom a tom istom zariadení, ale môžu byť rôzne distribuované, či už na firemnej sieti, alebo na serveroch po celom svete. Súčasti takéhoto systému môžu byť napísané v rôznych programovacích jazykoch, využívať rozličné verzie nástrojov či môžu byť určené pre úplne iné platformy. Súčasti distribuovaného systému nemusia výlučne pracovať súčasne v tom istom čase.

Výzvam spomenutým vyššie a mnohým ďalším je možné čeliť identifikáciou kritických častí systému a voľbou správneho návrhu pred ich samotnou implementáciou. Jednou z možností, ktorá prichádza s riešením spomenutých problémov je využívanie správ, alebo inak front (`message queues`).

V prvej kapitole budú predstavené zaužívané modely pre vymieňanie správ. Ďalej budú predstavené alternatívy k tomuto modelu, prípadne jeho predchodcovia. Detailne budú identifikované výhody, ktoré tento model prináša.

Ďalšie tri kapitoly budú venované predstaveniu troch nástrojov pre výmenu správ: Apache ActiveMQ, RabbitMQ a na záver Apache Kafka.

Šiesta kapitola popisuje analýzu požiadaviek, predstavuje niektoré existujúce riešenia a dôraz kladie aj na popis aplikácií, ktoré bežia formou webovej služby. V závere popisuje návrh výslednej aplikácie použitý pri implementácii.

Implementácia navrhnutej aplikácie je popísaná v siedmej kapitole. Zároveň budú predstavené použité nástroje, zaujímavosti a úskalia, na ktoré sa v priebehu implementácie nástroja narazilo.

Posledné dve kapitoly sa venujú funkčnému a výkonnostnému testovaniu výslednej aplikácie a v samotnom závere zhrnutiu celej práce.

Kapitola 2

Fronty správ

Dalo by sa povedať, že podnetom pre zaoberanie sa rôznymi spôsobmi komunikácie medzi systémami bol rozmach distribuovaných systémov. Distribuovaný systém sa dá chápať ako funkčný systém zložený z viacerých komponentov prepojených najčastejšie počítačovou sieťou.[9] Jednotlivé komponenty by mali medzi sebou komunikovať za účelom dosiahnutia nejakého spoločného cieľa. Spôsobov komunikácie môže byť viacero no v základe ide o zasielanie správ s pridanou hodnotou pre dané komponenty. Na pojem distribuovaný sa môžeme z hľadiska zasielania správ pozerať z viacerých strán. Môže predstavovať zložitý systém skladajúci sa z komponentov umiestnených vo firemnej sieti roztrúsených po celom svete, ale aj systém skladajúci sa iba z komponentov nachádzajúcich sa na jednom zariadení. Takéto komponenty môžu byť heterogénne, implementačne a technologicky nezávislé a rovnako ako komponenty vzdialené od seba tisícky kilometrov si medzi sebou potrebujú vymieňať správy.

2.1 Alternatívy

Pred tým ako sa dostaneme k samotným frontám správ, ktoré sú predmetom tejto práce, budú v skratke predstavené niektoré alternatívne prístupy ku komunikácii medzi systémami. Fungujú na inom princípe a preto sa budem snažiť porovnať ich na vlastnostiach, ktoré nás zaujímajú pri frontách správ.[7]

Za najprimitívnejší z nich môže byť označený spôsob, kedy jednotlivé komponenty medzi sebou komunikujú na základe prenosu súborov. Jeden komponent súbor upraví a odošle ho ľubovoľným spôsobom druhému. Ten môže vykonať nad súborom opäť dopredu dohodnuté úpravy. Musí byť stanovené, kto a kedy do tohoto súboru môže zapisovať. Pri zmene formátu súboru alebo jednotlivých komponent je potrebný často väčší zásah programátora. Takýto prístup môže byť efektívny pri systémoch, ktoré sú pravým opakom `real-time` systémov. Teda je potrebné maximálne raz za pár mesiacov predhodiť súbor na úpravu inej časti systému.

Zasielanie jednoduchých správ môže byť považované za predchodcu súčasnej komunikácie. Dnes už teda nie je veľmi používané pri distribuovaných systémoch. Odosielateľ sa pripojí na príjemcu a ľubovoľne mu zasiela správy. Príjemca spravidla synchrónne tieto správy spracuje. Nevýhoda takéhoto prístupu spočíva v časovej a priestorovej väzbe. Teda odosielateľ aj príjemca musia byť dostupní v rovnakom okamihu a odosielateľ navyše musí poznať adresu príjemcu.

Medzi najpoužívanejšie prístupy patrí vzdialené volanie procedúr – `RPC`. Komponent sprístupňuje svoju funkcionalitu tak, aby dané funkcie mohli byť zavolané vzdialene inými aplikáciami. Komunikácia prebieha v reálnom čase a je celkom náchylná na chyby prenosu, s ktorými je nutné pri implementácii počítať. Opäť je tu prítomná časová aj priestorová väzba. Odosielateľ musí vlastniť vzdialenú referenciu na volanú aplikáciu. Navyše kvôli synchrónnemu správaniu konzumenta, teda aplikácia, ktorej procedúry sú volané vzdialene, je tu aj synchronizačná väzba[1].

Problém s nútenou synchronizáciou pri RPC je možné vyriešiť jednoducho pomocou viacerých asynchrónnych volaní. Odosielateľ odošle volanie vzdialenej procedúry spolu s jej parametrami a k nim pridá ešte `callback`[2]. Príjemca po ukončení svojej činnosti zavolá tento `callback` a informuje odosielateľa o dokončení operácie. Tento prístup teda ponúka odstránenie synchronizačnej väzby, ale stále je prítomná časová a priestorová závislosť.

Posledná alternatíva je zdieľané úložisko dát, alebo zdieľaná databáza. Rozšírenie relačných databáz pomohlo tomuto prístupu, pretože napríklad pre jazyk `SQL` existujú knižnice takmer pre každý programovací jazyk, pre ktorý to dáva zmysel. Zdieľané úložisko odstraňuje časovú a priestorovú závislosť. Odosielateľ nemusí vedieť komu je ním vykonaná zmena dát určená. Rovnako ani aplikácia závislá na tejto zmene nemusí práve bežať, pretože dáta sú v spoločnom úložisku a na túto aplikáciu si počkajú. Jediným problémom je synchronizácia. Síce nemusí byť odosielateľ a príjemca v jednom okamihu k dispozícii, musí byť programovo zaistené, aby sa príjemca nesnažil prečítať dáta skôr ako ich odosielateľ do úložiska zapísal. Tento prístup prináša oveľa viac úskalí, pre ktoré je vo veľa prípadoch označovaný skôr za návrhový antivzor[3].

2.2 Fronty správ

Princíp fungovania front správ sa dá prirovnať k telefónu. Obyčajný hovor medzi účastníkmi je synchrónny. Obaja účastníci musia byť dostupní v rovnakom čase. Avšak, ak je potrebné nechať druhému účastníkovi správu za jeho neprítomnosti, je možné použiť hlasový záznamník. Ten dovoľuje asynchrónnu komunikáciu.

Na obrázku 2.1 sú zobrazené entity, ktoré sa zúčastňujú komunikácie pri využití fronty správ. Producent vytvára správy a zasiela ich do fronty. Fronta zabezpečuje akési dočasné úložisko, ktoré ma za úlohu perzistovať správy až do doby, kedy budú spracované. Konzument číta správy z fronty v poradí, v akom sú vo fronte uložené. V prípade, že konzument potrebuje odoslať správu producentovi, tak budú potrebovať inú frontu správ a ich role sa pre tento prípad vymenia.

Do fronty môže zapisovať ľubovoľné množstvo producentov, rovnako ako z nej môže čítať ľubovoľné množstvo konzumentov. Každá správa je prečítaná len jedným z nich a po prečítaní je z fronty odstránená.

Samotná správa môže byť reprezentovaná či už reťazcom, poľom bytov alebo serializovaným objektom. Obvykle sa skladá z dvoch častí: z hlavičky a z tela správy. Hlavička

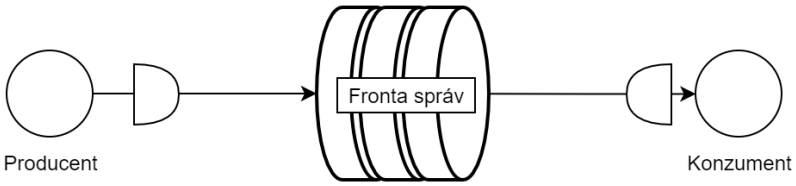
[1] Synchronizačná väzba predstavuje závislosť na synchronizácii. Teda časť aplikácie je blokovaná volaním procedúry druhej aplikácie až kým ju tá aplikácia nedokončí.

[2] Callback je spustiteľný kód, ktorý môže prísť ako argument spolu s volaním funkcie. Tento spustiteľný kód je potom po dokončení operácie spustený a môže slúžiť na upovedomenie iniciátora pôvodnej operácie o jej dokončení.

[3] Alebo aj Anti-pattern je označenie pre obecne známy spôsob riešenia problému, ktorý je pre svoje úskalia neefektívny a kontraproduktívny.

obsahuje meta-dáta o správe samotnej, informácie o odosielateľovi, prípadne iné informácie v závislosti na implementácii.

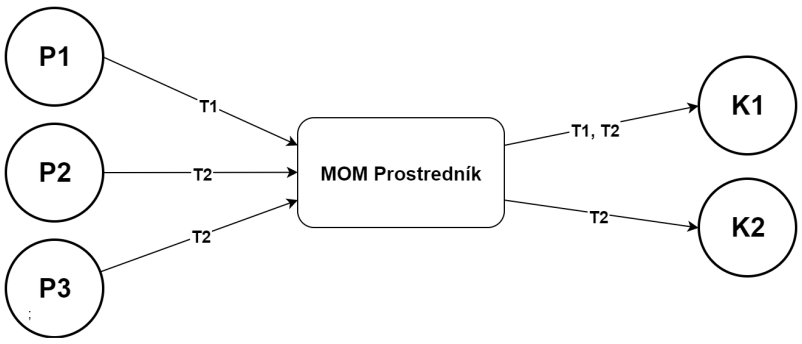
Časová nezávislosť vyplýva už z informácií spomenutých vyššie. Aby bolo skutočne možné dosiahnuť aj priestorovú nezávislosť je potrebné zbaviť sa nutnosti aby producent poznal telefónne číslo konzumenta kedykoľvek ho chce kontaktovať. Za toto je zodpovedný prostredník – agent `MOM` (Message Oriented Middleware popísaný v sekcii 2.6). Potom stačí, aby každý z účastníkov disponoval priamym prístupom iba k tomuto prostredníkovi.



Obr. 2.1: Fronta správ

2.3 Publish/Subscribe

Princíp `publish-subscribe` (`pub-sub`) sa dá označiť za súrodenca fronty správ. Oproti obyčajnej fronte sa líši v niektorých aspektoch. Dalo by sa povedať, že umožňuje pomenovať jednotlivé fronty z ktorých sa potom stávajú témy. Do jednej témy môže prispievať viacero producentov. Odoberať z nej môže tak isto kľudne väčšie množstvo konzumentov. Jedným z rodielov oproti fronte je napríklad to, že správa sa z danej témy nevymaže hneď ako k nej pristúpi prvý konzument. Naopak je k dispozícii každému jednému, kto danú tému odoberá. Naopak v prípade, že príjemca nie je k dispozícii v čase keď je správa vyprodukovaná stráca možnosť prijať túto správu. Špecifikácia JMS, ktorá bude spomenutá neskôr hovorí o tom, že podporuje aj trvácnych odberateľov. Umožňujú perzistovanie správy pre príjemcu, ktorý nie je dostupný v čase odoslania správy.



Obr. 2.2: Publish-Subscribe

Na obrázku 2.2 sú zobrazené prípady, kedy jedna téma (T2) má viac ako jedného producenta (P2 & P3) a správy jednej témy (T2) sú odoberané viacerými konzumentmi (K1 & K2).

2.4 Výhody

Niektoré z nasledujúcich výhod vyplývajú z vecí spomenutých skôr v tejto kapitole. Kvôli názornosti ich spomeniem v skratke a menovite.

Vzdialená komunikácia

Výmena dát medzi objektami vrámci jedného procesu je jednoduchá a úplne si vystačí napríklad so zdieľanou pamäťou. V prípade spájaní nezávislých aplikácií je potrebná vzdialená komunikácia, ktorú zabezpečujú samotné fronty, prípadne `pub-sub`.

Asynchrónna komunikácia

Synchrónna komunikácia bola jednou z nevýhod pri vzdialenom volaní procedúr RPC. Fronty správ prichádzajú s asynchrónnou formou komunikácie pomocou jednoduchého princípu, kedy program odošle správu a ďalej sa nestará. Nemôžeme povedať, že sa nestará úplne. Nemusí sa starať o to kedy správa príde príjemcovi. Program zaujíma akurát to, či správa bola úspešne odoslaná MOM agentovi a úspešne uložená do fronty. Príjemca správy môže chcieť odoslať potvrdenie o úspešnom prijatí správy. To môže spraviť pomocou novej správy od príjemcu k odosielateľovi.

Odstránenie väzby

Aplikácie so slabou väzbou umožňujú jednotlivým častiam komunikovať podľa vhodne navrhnutého rozhrania, namiesto toho aby úzko záviseli na špecifických detailoch jeden druhého. Pri takejto architektúre môžu byť jednotlivé časti systému vyvíjané oddelene. Výrazne to zjednodušuje testovanie jednotlivých komponentov. V tomto prípade to ale výrazne znižuje starosti spojené so syncrhonizáciou jednotlivých účastníkov komunikácie. Toto sú väzby o ktorých je reč pri zasielaní správ: priestorová väzba, časová väzba a synchronizačná väzba.

Podstatou odstránenia priestorovej väzby je to, aby sa jednotliví účastníci komunikácie nemuseli poznať. Vytvorená správa je odoslaná do známej fronty, odtiaľ je doručená na miesto kde sa o samotnú frontu zaujímajú (presný princíp záleží od použitia obyčajnej fronty správ, prípadne metódy `pub-sub`). Odosielateľ v sebe nepotrebuje držať zoznam referencií na všetkých príjemcov, rovnako ani príjemca zoznam odosielateľov. Spoľahlivé doručenie správy zaručuje agent `MOM`.

Odstránenie časovej závislosti umožňuje odosielať správy príjemcom, ktorí aktuálne nie sú pripojení (trvácny príjemca). Na to je potrebné nejaké perzistentné úložisko zasielaných správ (viac v časti 2.6). Tak isto príjemca môže po pripojení dostať aj správy od odosielateľov, ktorí už nie sú dostupní.

Zbavenie sa synchronizačnej väzby znamená, že odosielateľ nie je blokovaný tým, že produkuje správu. Odberateľ určitej témy môže byť o správe upozornený asynchrónne pomocou `callbacku`. Hlavný beh programu, či už odosielateľa alebo príjemcu beží spravidla oddelene od operácií spojenými so samotným odosielaním/prijímaním správ.

Variabilné časovanie

Použitie asynchrónnej komunikácie rieši aj prípadný problém rozdielnej výkonnosti zariadení, alebo operácií spojených s vytvorením a spracovaním správ. Ak jedno zariadenie vytvára správy rýchlejšie ako je príjemca schopný ich spracovávať, tak ho tieto správy počkajú vo fronte. Každý z účastníkov si môže vykonávať operácie svojou rýchlosťou.

Výkon pod kontrolou

Jednou z nevýhod synchrónnej komunikácie (RPC) je riziko preťaženia servera pri zasielaní príliš vysokého množstva požiadaviek. Fronty umožňujú príjemcovi regulovať rýchlosť akou dané požiadavky (správy) spracováva aby ho nebolo možné preťažiť simultánnymi dotazmi.

Škálovateľnosť

Zo slabej väzby pri komponentoch vyplýva dobrá škálovateľnosť systému. V prípade zmenených nárokov je možné upravovať jednotlivé komponenty, nahrádzať ich inými výkonnejšími prípadne menej výkonnými.

Perzistencia

Pri zasielaní správ sa ukladajú do fronty. Z hľadiska základnej funkcionality stačí, aby sa tie správy po dobu ich doručenia ukladali len v pamäti. MQ systémy (`message queue`) musia byť pre garanciu doručenia vyzbrojené aj niečím sofistikovanejším. Ak agent skončí kvôli nejakej chybe, musí vedieť zaručiť doručenie správ po opätovnom zapnutí. To je možné vďaka perzistentnému úložisku dát, kedy sa správy neukladajú len do pamäti ale aj do databázy, súboru, atď. Je to síce spojené s extra nárokmi na systémové zdroje, ale je to cena, ktorú je nutné zaplatiť, ak má byť doručenie správ garantované.

Spoľahlivá komunikácia

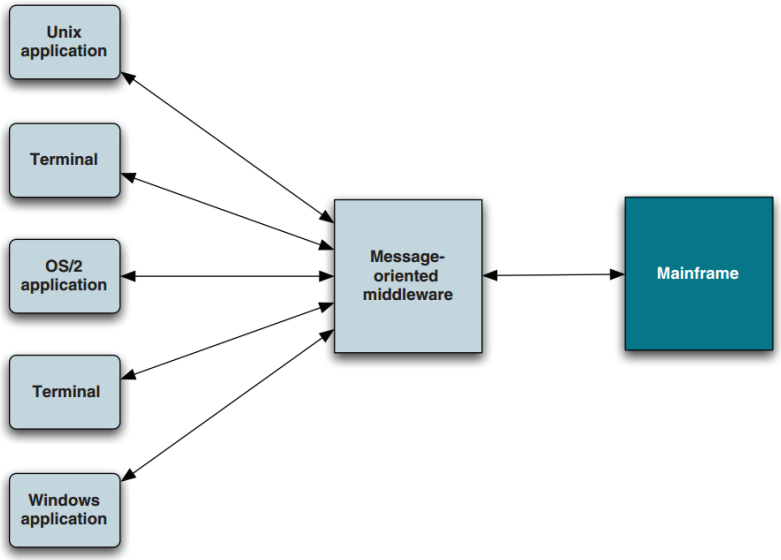
Pretože komunikácia medzi komponentmi je nepriama a asynchrónna, musí mať aplikácia istú dôveru v systém, ktorý sa stará o doručovanie správ. Ten musí garantovať, že správa bude po úspešnom prijatí agentom od odosielateľa vždy doručená príjemcovi. Táto vlastnosť vyplýva z perzistovania dát spomenutého vyššie. Dáta sa teda po páde agenta nestratia ale po jeho opätovnom zapnutí budú doručené. Stále však vznikne opozdenie doručenia tejto správy. Systém môže mať zvýšený nárok na zasielanie správ v reálnom čase a taká omeškaná správa v čase doručenia už vôbec nemusí dávať zmysel. Pri takýchto systémoch túto výzvu možné riešiť pomocou viacerých distribuovaných agentov. Agenti používajú spoľahlivé spôsoby komunikácie medzi sebou. Rozposielajú jednotlivé správy či už všetkým agentom, alebo len nejakej podmnožine. Záleží na presnej implementácii MOM systému.

2.5 Nevýhody

Využitie MOM vo svojej architektúre na komunikáciu napriek všetkým výhodám spomenutým doteraz prináša aj isté nevýhody, ktoré treba zvážiť. Nevýhodou je teda nutnosť nadbytočného komponentu – MOM midlvéru, ktorý sa stará o doručovanie správ. V prípade, že takýto komponent beží na rovnakom zariadení ako systém, ktorý ho využíva, môže priniesť zníženie výkonu celého systému a nárast nákladov potrebných na jeho údržbu. Asynchrónnu komunikáciu, na ktorú sme sa doteraz pozerali ako výhodu môže byť vnímaná aj ako nevýhoda. Samotný typ komunikácie, kedy odošleme požiadavok a nečakáme synchrónne na odpoveď, vyžaduje prácu navyše. Je dôležité aby technológie použité v systéme boli na mieru vybraté.

2.6 MOM

Uprostred komunikácie či už pomocou fronty správ, alebo princípu `pub-sub` stojí `MOM` prostredník. Jeho úlohou je sprostredkovať jednoduché rozhranie pre výmenu správ medzi distribuovanými systémami. Vytvára nad komunikáciou abstrakciu, kedy odstraňuje závislosti na jednotlivej platforme a iných presných technických špecifikáciách danej časti systému (obr. 2.3).



Obr. 2.3: Message Oriented Middleware [13]

Prostredník prináša žiadané vlastnosti systému spomínané skôr v tejto práci: nízku závislosť medzi komponentami, škálovateľnosť, asynchrónnosť a bezpečnosť. Umožňuje vysporiadať sa s nespoľahlivým a pomalým spojením medzi jednotlivými komponentami. Ak príjemca správy pri jej spracovaní zlyhá, odosielateľ sa tým vôbec nemusí zaťažovať. Zaujímavých vlastností je ešte oveľa viac a budú spomenuté či už pri špecifikácii `JMS`, alebo pri jednotlivých nástrojoch.

Kapitola 3

Apache ActiveMQ

Apache ActiveMQ je prvý spomenutý `broker` správ. Jeho hlavným účelom je zjednodušenie vzdialenej komunikácie.[13] Zaväzuje sa k podpore funkcií podľa špecifikácie `JMS` vo verzii 1.1. Okrem tejto základnej funkcionality prináša mnohé ďalšie funkcie, či už na strane ActiveMQ klienta, alebo ActiveMQ brokera, ktoré budú popísané ďalej v tejto kapitole. Nástroj je napísaný v jazyku Java, no poskytuje riešenia pre mnohé ďalšie programovacie jazyky. Je to OpenSource nástroj z dielne spoločnosti *Apache Software Foundation* vydaný pod voľnou licenciou `Apache 2.0`, ktorá veľmi napomohla jeho rozšíreniu. V čase písania tejto práce bola aktuálna verzia nástroja 5.15.2.[1]

3.1 Špecifikácia JMS

Java Message Service – JMS (z roku 1998) je štandard pre zasielanie správ, s ktorým prišla pôvodne spoločnosť *Sun*[1] v spojení s rôznymi spoločnosťami, ktoré v minulosti čelili podobným výzvam. `JMS` samé o sebe nepredstavuje `MOM` prostredníka.[3] Je to skôr štandardizované API, určené na odosielanie a prijímanie správ s využitím jazyka Java, bez ohľadu na použitý `MOM`. Umožňuje využívať správy na komunikáciu v aplikácii bez nutnosti hlbšej znalosti vecí potrebných k spoľahlivému prenosu takýchto správ a rovnako aj určitú mieru prenositeľnosti medzi rozličnými `MOM` podporujúcimi túto špecifikáciu.

3.1.1 JMS správa

Za zmienku určite stojí popísať, ako vyzerá správa podľa špecifikácie – je to predsa najzákladnejšia časť komunikácie. Skladá sa z troch častí: hlavička, vlastnosti a samotné telo. Jediná povinná je hlavička.[5]

Hlavičky

Hlavičky sa dajú rozdeliť do troch kategórií podľa toho, ktorý účastník konverzácie ich nastavuje.

Nastavené `send()` alebo `publish` metódou:

- `JMSDestination` – obsahuje destináciu, kam je správa odosielaná.

[1] Sun Microsystems bola založená v roku 1982 a neskôr v roku 2010 bola odkúpená spoločnosťou Oracle.

- `JMSDeliveryMode` – môže byť perzistentný alebo neperzistentný. Prvý menovaný zaručuje doručenie správay práve jeden krát. Druhý menovaný maximálne 1 krát, takže správa sa pri zlyhaní môže stratiť.
- `JMSExpiration` – správa nebude doručená, ak ubehla doba expirácie.
- `JMSPriority` – zoradovanie správ podľa priority 0 (najnižšia) až 9.
- `JMSMessageID` – z historických dôvodov bolo možné nastaviť ID správy na strane odosielateľa, ktoré sa pripájalo k ID správy nastaveného MOM. Dnes sa veľmi nevyužíva.
- `JMSTimestamp` – obsahuje čas, kedy bola správa odoslaná od jej producenta.

Nastavené nepovinne klientom:

- `JMSCorrelationID` – umožňuje logicky spojiť správu s inou správou ako odpoveď na ňu.
- `JMSReplyTo` – informuje o tom, kde má byť zaslaná odpoveď a o tom, že nejaká odpoveď je očakávaná.
- `JMSType` – označuje typ správy. Označenie je čisto informatívne, nemá nič spoločné so samotným obsahom tela správy.

Posledná môže byť nastavená `MOM` sprostredkovateľom. `JMSRedelivered` určuje, či správa bola doručená, ale z nejakého dôvodu nebola potvrdená.

Telo správy

Špecifikácia prináša päť rozdielnych typov dát v tele správy plus jednu základnú kde telo ostáva prázdne. Sú to tieto:

- `TextMessage` – telo obsahuje obyčajný text. Určené pre `plain-text` a správy formátu `XML`.
- `MapMessage` – zoznam párov kľúč – hodnota, kde kľúč je reťazec a hodnota môže byť nejaký z primitívnych typov jazyka Java.
- `BytesMessage` – telo obsahuje reťazec neinterpretovaných bytov.
- `StreamMessage` – telo obsahuje prúd hodnôt opäť primitívnych typov jazyka Java, ktoré sú čítané sekvenčne.
- `ObjectMessage` – umožňuje posielať serializovaný objekt jazyka Java. Podporuje aj kolekcie.

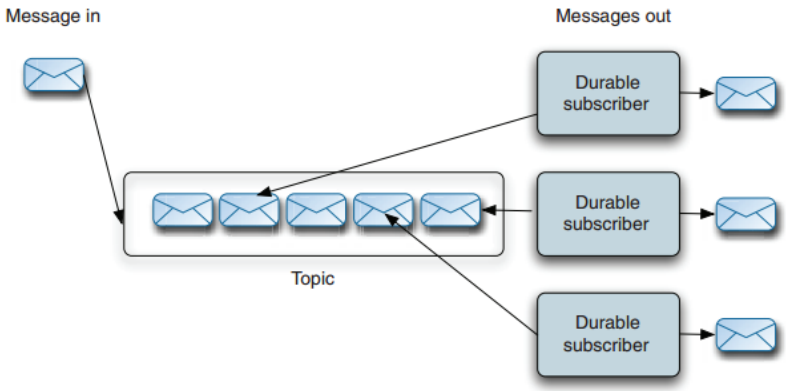
3.2 Úložisko správ

Špecifikácia JMS podporuje dva typy doručovania správ. Správa s perzistentným doručením musí byť uložená v nejakom perzistentnom úložisku minimálne do doby, kedy je úspešne doručená. Pri neperzistentnom doručení sa broker snaží o doručenie, ale správu neukladá. Tento typ sa používa skôr na zasielanie nejakých notifikácií a v situáciách kedy sa viac

dbá na výkon ako na spoľahlivosť. Apache ActiveMQ podporuje oba typy zasielania správ. Okrem toho podporuje aj obnovu správ, kedy sú správy udržiavané v pamäti `cache`. Celkovo podporuje tri spôsoby perzistencie správ: v pamäti, v súbore alebo pomocou relačnej databázy. Základným je KahaDB, ktorá je dodávaná spolu s ActiveMQ. Avšak umožňuje zásuvné úložiská správ a ďalšie možnosti sú tieto: AMQ úložisko, JDBC úložisko a pamäťové úložisko.

Ukladanie správ pre fronty sa líši od ukladania správ pre `pub-sub`. Dôvodom sú optimalizácie, ktoré je možné vykonať pri `pub-sub`, ale nemajú zmysel pri obyčajných frontách. Pri frontách sú správy uložené rovnomenne do fronty – konkrétne FIFO[2] fronta. Správu obdrží vždy len jediný príjemca a až keď potvrdí prijatie, môže byť z úložiska vymazaná.

Pokiaľ ide o doručovanie správ podľa témy, tak každá správa môže mať viac príjemcov. V úložisku dát je táto správa uložená iba raz z dôvodu šetrenia pamäte. Na obrázku 3.1 je znázornený spôsob, akým sa správy dostávajú k svojim príjemcom. Každý príjemca danej témy má na brokerovi svojho konzumenta. Ten v sebe drží ukazovateľ na ďalšiu správu v poradí. Toto je nutné z dôvodu, že každý reálny odberateľ môže prijímať správy rôznou rýchlosťou a je potrebné niekde držať informáciu o tom, aká správa náleží každému z konzumentov. Správa je vymazaná ak je úspešne doručená každému jednému odberateľovi.



Obr. 3.1: Doručovanie správ podľa témy [13]

KahaDB

Ako bolo spomenuté skôr, KahaDB je odporúčaným úložiskom dát a to od verzie 5.3. Ide o súborové úložisko, ktoré disponuje aj denníkom tranzakcií pre spoľahlivosť a možnosť obnovy dát po chybe. KahaDB vyniká svojím vyladením pre rýchlosť ukladania správ. Tá spočíva v kombinácii denníku tranzakcií obsahujúceho datové logy (správy a rôzne príkazy, napr. pre zmazanie správ), optimalizovaného indexovania identifikátorov správ pomocou B stromov[3] a v neposlednom rade dočasného ukladania správ v pamäti. Dočasne sa ukladajú tie správy, ktoré nie je hneď potrebné perzistovať, pretože príjemca je napríklad aktívny v danom momente a správa je mu namiesto ukladania rovno odoslaná.

Ďalšou výhodou je jednoduchosť použitia. Vďaka súborovému systému nie je nutné inštalovať nijakú databázu a celý `MOM` midlvér je možné spojazdniť behom niekoľkých minút.

[2] FIFO (First in First out) – je typ fronty kedy sú požiadavky obslúžené v poradí v akom do fronty prišli.

[3] B-strom je taký strom, ktorý je vyvážený; každá operácia pridania, odstránenia aj vyhľadávania prebehne v logaritmickom čase.

Používa jeden jediný index pre všetky destinácie. Nemá problém s veľkým množstvom aktívnych spojení (kľudne aj 10 000), kde každé jedno má svoju frontu správ, ani s ukladaním veľkého množstvá správ v súborovom systéme.

AMQ úložisko

AMQ bolo navrhnuté a optimalizované na vysokú rýchlosť. Podobne ako KahaDB kombinuje denník tranzakcií pre zotavenie sa z chýb a indexy. Použitie AMQ je potrebné dobre zvážiť, pretože oproti KahaDB používa dva oddelené súbory pre každý index a je tu index pre každého príjemcu. Nemal by byť prvou voľbou, ak existuje požiadavka, že jeden broker bude musieť spracovávať obrovské množstvo front. V prípade zlyhania môže byť zotavenie pomalé. Dôvodom je nutnosť znovu vytvoriť všetky indexy.

JDBC úložisko

Relačné databázy sú najrozšírenejším spôsobom, akým spoločnosti ukladajú svoje dáta. Práve to môže byť dôvod, prečo niekto má záujem využiť už vybudovanú infraštruktúru databáz na perzistovanie správ. Treba však počítať s tým, že súborov KahaDB a AMQ boli vytvorené presne na mieru podľa požiadaviek na zasielanie správ. Relačné databázy na to nie sú stavané, a preto tento spôsob nebude z hľadiska výkonu najoptimálnejší. JDBC na ukladanie správ využíva tri tabuľky. Prvá obsahuje jednotlivé správy, ktoré môžu byť rozdelené na menšie, aby sa zmestili do tabuľky. Druhá obsahuje informácie o príjemcoch v prípade pub-sub a identifikátor posledne prijatej správy. Posledná tabuľka slúži uloženie informácií potrebných pre výlučný prístup len jedného brokera do zvyšných dvoch tabuliek súčasne.

Pamäť

V tomto prípade sú všetky perzistentné správy uložené v pamäti. To môže byť dvojsečná zbraň pre vašu `JVM` pri veľkom množstve správ. Vhodným prípadom pre použitie je prípad, kedy poznáme maximálny počet správ, ktorý bude uložený na brokerovi. Môže to byť napríklad prvotné odladenie systému, kde je jednoduchšie systém odladiť ak sú správy uložené priamo v pamäti ako sa zapodievať ukladaním správ do úložiska a následného čistenia úložiska.

Nastavenie ukladania správ do pamäte je možné dosiahnuť jednoduchým nastavením vlastnosti `persistent` u brokera na `false`.

Cachovanie

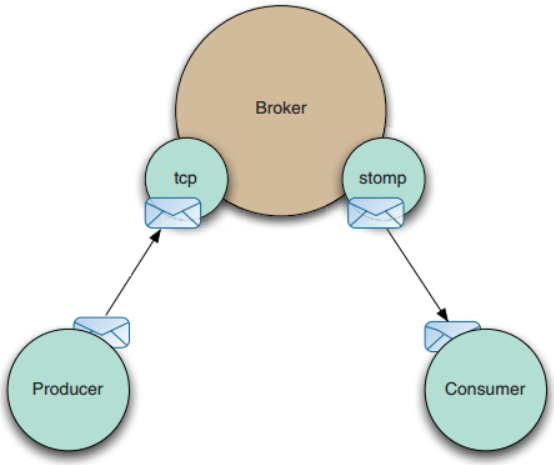
Ako už bolo spomenuté skôr, nie vždy je potrebné aby správa bola uložená v perzistentnom úložisku dát. Cachovanie je teda optimalizácia a nie ďalšia varianta úložiska dát. Pri niektorých typoch aplikácií, kde vieme, že príjemca bude bežať v rovnakom čase ako odosielateľ, vieme povedať, že prenos dát bude takmer v reálnom čase (počíta sa s výpadkami siete a preťažením zariadení). V takomto prípade nemá zmysel dáta perzistovať, ale stačí akési cachovanie v pamäti. ActiveMQ umožňuje toto cachovanie nastaviť voľbou jednej z politík, ktorými sa dá nastaviť logika cachovania: podľa maximmálneho počtu správ, podľa pamäte, ktorú využívajú, podľa dĺžky zotavenia a ďalšie. Do cachovania spadajú správy zaslané pod nejakou témou (s výnimkou dočasných tém a pomocných ActiveMQ tém). Obyčajné fronty nie sú dočasne uložené, pretože pri frontách majú byť perzistované všetky správy.

3.2.1 Široká podpora

Ako bolo spomenuté vyššie ActiveMQ má široku podporu programovacích jazykov. Preto ho nemožno označovať len za `JMS` brokera. Rozširuje použiteľnosť mimo Java sveta. Skriptovacie jazyky Ruby, Python, PHP a Perl môžu pracovať s ActiveMQ pomocou protokolu STOMP[4]. Je to jednoduchý textový protokol. Umožňuje mať v distribuovanom systéme komponenty, kde jeden komunikuje klasickým spôsobom v jazyku Java a druhý napísaný v ľubovoľnom zo skriptovacích jazykov spomenutých vyššie (viď. 3.2).

Kompilované jazyky ako Java, C# či C++ využívajú na komunikáciu sofistikovanejší binárny protokol OpenWire (C++ môže tak isto použiť STOMP). Je oproti STOMP zložitejší na používanie, ale za to je oveľa viac optimalizovaný na šetrenie lepšie využívanie šírky pásma a výkon celkovo. Keďže `JMS` je špecifikácia pre Javu, tak C# a C++ potrebujú nejakú alternatívu. Vznikli dva protokoly, ktoré sú veľmi podobné originálnemu JMS. To umožňuje ich využitie spolu s ActiveMQ. C# využíva NMS (.NET Message Service) a C++ využíva CMS (C++ Message Service).

Vymenovanými jazykmi podpora ActiveMQ nekončí. Pre webové aplikácie je dostupná REST API, prípadne Ajax API čisto pomocou knižnice v jazyku Javascript.



Obr. 3.2: Kompatibilita komponentov v systéme [13]

3.3 Výkon a spoľahlivosť

3.3.1 Vysoká dostupnosť

Pri použití aplikácie v produkčnom prostredí je potrebné počítať so všetkými možnými zlyhaniami. Je nutné aby sa systém dokázal s takýmito situáciami vysporiadať bez akéhokoľvek sekundárneho efektu pre užívateľa, prípadne sekundárneho efektu – záleží na požiadavkach na aplikáciu. Vysokú dostupnosť je možné docieliť pomocou zapojenia viacerých brokerov štýlom `master/slave`. V takomto zapojení je jeden master broker, ktorý sa stará o komunikáciu a jeden alebo viac slave brokerov. V prípade výpadku master brokera sa zo slave brokera stáva nový master a preberá zodpovednosť za komunikáciu aby sa zabránilo výpadku spojenia a strate správ. Existujú dva typy master/slave spojenia a líšia sa v tom, že

[4]STOMP – Streaming Text Oriented Messaging Protocol – http://activemq.apache.org/stomp.html

v prvom prípade má každý svoje vlastné úložisko dát (`shared nothing`). V druhom prípade je jedno zdieľané úložisko dát (`shared storage`).

`Shared nothing` je jednoduchšie na konfiguráciu. Je určený skôr do prostredia, kde je dovolená určitá miera zlyhania. `Master` by mal byť vytvorený ako prvý a slave sa potom pripojí na master. Všetky správy a príkazy z master brokera sú preposielané na slave brokera. Ak producent odošle správu na master broker, ten pred odoslaním potvrdenia počká na potvrdenie od slave brokera o úspešnom spracovaní a uložení správy/požiadavku do svojej databázy a až potom sám odošle potvrdenie odosielateľovi o úspešnom prijatí správy. Takýto typ spojenia vyžaduje vyššie nároky na pamäť kvôli duplikovaniu všetkých správ, a tak isto navýšenie réžie potrebnej k doručeniu správ. Ak master broker zlyhá, slave broker má dve možnosti čo môže spraviť. Môže sa vypnúť a vtedy je na rade administrátor, ktorý systém uvedie znovu do behu. V tomto prípade je funkcia slave brokera skôr uchovať stav master brokera spred vyradenia z prevádzky. Druhým prípadom je, automatická zmena zo slave na master. Aj v tomto prípade je však potrebná činnosť administrátora na obnovenie pôvodného mastera a jeho prípadné nasadenie do role nového slave brokera. V prípade nastavenia hodnoty parametru `waitForSlave` na `true` (master nezačne žiadnu komunikáciu skôr ako je k dispozícii slave) bude systém obmedzený len na jedného aktívneho slave brokera.

`Shared storage` dovoľuje aby master a slave broker fungovali nezávisle na sebe. Aktívny však môže byť len jeden broker súčasne. To je dosiahnuté tak, že aktívny broker je ten, ktorý vlastní zámok k úložisku dát. V prípade výpadku jedného z brokerov nie je potrebný v podstate (ak je dostupný nejaký slave broker) žiadny zásah aby bola zachovaná integrita. Rovnako neexistuje obmedzenie na počet slave brokerov. Jediným obmedzením tohoto systému je nutnosť použitia takého prostredia, ktoré umožňuje distribuované zamykanie zdieľaného úložiska.

Sieť brokerov

Vo veľa prípadoch si systém vystačí s jedným brokerom, prípadne master/slave brokermi. V niektorých prípadoch je požadovaná lepšia dostupnosť a škálovateľnosť služieb. Sieť brokerov (`Network of brokers`) je prostriedkom ako to dosiahnuť. Ide o skupinu brokerov, ktorí sú medzi sebou prepojení. Zmyslom môže byť zvýšenie počtu spojených klientov či prekonanie geografických prekážok pomocou použitia vhodnej topológie prepojenia brokerov. Spojenie medzi brokermi je štandardne jednosmerné, ale je možné zvoliť duplexné spojenie (môže sa hodiť ak je broker ukrytý za firewallom). Brokeri sa spájajú pomocou sieťových konektorov špecifikovaných v XML konfiguračnom súbore, ktorý obsahuje každý broker. Administrátor môže zvoliť medzi statickým (viď. nasledujúca ukážka) a dynamickým konektorom. Dynamický konektor vyhľadáva ostatných brokerov pomocou multikastu – je dobré zvoliť unikátne meno skupiny, aby nenašiel brokerov inej aplikácie.

```
<networkConnectors>
   <networkConnector uri="static:(tcp://host1:616,tcp://host2:617,..)"/>
</networkConnectors>
```

3.3.2 Škálovateľnosť

ActiveMQ prináša tri techniky škálovateľnosti. Prvá z nich (vertikálna) je pre prípad, keď jediný broker je použitý pre tisíce spojení a front. Druhá (horizontálna) kedy sú v hre

desiatky tisíc spojení pomocou sieti brokerov. Posledná spája prvé dva princípy (delenie prenosu – `traffic partitioning`).

Prvou z možností vertikálneho zvyšovania výkonu je náhrada obyčajného `TCP` transportného protokolu protokolom `NIO` [5]. `NIO` je vylepšením klasického `TCP`, kde pre každé spojenie bolo vytvorené nové vlákno. V blokujúcej implementácii existuje jedno vlákno pre každé spojenie. `NIO` využíva na serverovej strane pool vlákien. Počet vlákien nezávisí od počtu aktívnych spojení. Rovnako je možné zamedziť, aby bolo vytvorené nové vlákno pre každú jednu frontu (viď nasledujúca ukážka). Ďalšou možnosťou je vypnúť úsporné kódovanie protokolu `OpenWire` čím sa zníži nárok na CPU (zvýši sa nárok na šírku pásma – obecne sa ale výkon zlepší). Dobrým prístupom je nastaviť limity podľa konfigurácie brokera či už na využitie pamäte alebo využitie úložiska. V neposlednom rade pomáha voľba správneho perzistentného úložiska (štandardné nastavenie KahaDB je najlepšia voľba).

```
ACTIVEMQ_OPTS="-Dorg.apache.activemq.UseDedicatedTaskRunner=false"
```

Doteraz spomenuté zlepšenia boli zlepšenia jedného brokera. Horizontálna škálovateľnosť sa dá dosiahnuť práve vďaka sieti brokerov spomenutej skôr v tejto kapitole. Hoci prináša vyššiu latenciu z dôvodu komunikácie medzi samotnými brokermi, umožňuje spojenie väčšieho počtu klientov.

Posledný princíp spája výhody oboch. V systéme existuje viacero brokerov ale tí nie sú medzi sebou prepojení. To vyžaduje vyššiu komplexnosť klientov. Tí musia udržovať väčšie množstvo `JMS` spojení s brokermi a tak isto sa musia rozhodnúť, ktorému brokerovi odošlú svoju správu.

3.4 Ďalšie vlastnosti

3.4.1 Špeciálne vlastnosti brokera

Wildcards

`Wildcards`[6] alebo žolíkové znaky sa v ActiveMQ požívajú na odoberanie z rôznych destináci súčasne. ActiveMQ podporujú takzvanú hierarchiu destinácií pre fronty aj pre témy. Dovoľuje odoberať správy (výsledky) z témy označenej ako `Hokej.NHL.Washington`. Keby do názvu témy vnesieme wildcard, môžeme napísať `*.*.Washington` a odoberáme výsledky všetkých typov športových zápasov z každej ligy, v ktorej hrajú reprezentanti nášho obľúbeného mesta.

Kompozitné destinácie

Opakom wildcards sú kompozitné destinácie. Umožňujú prispievať do viacerých front či tém súčasne (alebo kombinovane). Wildcards fungujú aj opačne a pomocou kompozitných destinácií je možné nastaviť aj dve rôzne fronty, do ktorých bude správa plnohodnotne odoslaná. Ak existujú fronty `A.X.Bob` a `A.X.Alica`, tak pri prispievaní do kompozitnej destinácie `A.X` sa správa distribuuje do oboch front (Bob aj Alica). Je možné rozdielne názvy tém spájať pomocou čiarky. Štandardne je ešte potrebné špecifikovať prefix podľa toho či ide o frontu, či správu (`queue://` alebo `topic://`).

[5] http://activemq.apache.org/nio-transport-reference.html

[6] Wildcard je zástupca – jeden znak (napríklad *), ktorý môže reprezentovať skupinu znakov, prípadne prázdny reťazec.

Poradenské správy

Anglickým názvom `advisory messages` sú správy generované brokerom nesúce notifikáciu o nejakej zmene stavu brokera. Poskytujú základné informácie o vytvorení a ukončení spojenia producenta/konzumenta, expirovaných správach, brokeroch odosielajúcich správy do cieľových destinácií bez odberateľov a iné. Poskytuje alternatívu k `JMX`. Z dôvodu možnej zvýšenej réžie je možné túto vlastnosť vypnúť.

Virtuálne témy

`JMS` špecifikácia umožňuje eliminovať nedostatky pub-sub pomocou trvácnych odberateľov. Takýto trvalý odberateľ musí mať unikátne klientské ID a meno odberateľa. Pre každé jedno spojenie s trvácnym odberateľom existuje len jediné vlákno a preto nie je spôsob ako obnoviť prenos po nejakej chybe cieľového konzumenta. Štandardné JMS fronty však umožňujú jednoduchú obnovu po chybe konzumenta a tak isto rovnomerné vyváženie zaťaženia jednotlivých komponentov.

Virtuálne témy umožňujú producentovi odosielať správy do normálnych tém ako pri štandardnom pub-sub a konzumentovi prijímať tieto správy ako by ich prijímal z obyčajnej fronty. To je možné vďaka jednoduchým pravidlám pri vytváraní názvov tém. Na označenie virtuálnej témy sa použije vzor `VirtualTopic.<názov_témy>`. Ak chce konzument odberať z virtuálnej témy prihlási sa na odber pomocou špeciálneho prefixu:

`Consumer.<nazov\_konzumenta>.VirtualTopic.<nazov\_temy>.`

Virtuálne témy dovolia na strane konzumenta to čo obyčajné fronty – bojovať o správy. Ak sa vicerí konzumenti prihlásia k odberu správ s rovnakým názvom konzumenta, správy v tejto fronte budú rovnomerne rozdelené medzi nich.

Retroaktívny konzumenti

V niektorých prípadoch môžu byť požiadavky na výkon vyššie ako na zaručenie perzistencie doručenia všetkých správ. Keďže ukladanie správ je drahá operácia, je tu možnosť ako sa vyhnúť perzistovaniu. ActiveMQ ponúka možnosť cachovať určité množstvo správ na zvýšenie spoľahlivosti pre prípad, že retroaktívny konzument nebol schopný v nejakom okamžiku správy prijímať. Konzument pri prihlasovaní sa k odberu nastaví jednoduchý príznak v názve témy (`názov_témy?consumer.retroactive=true`). Na strane brokera je možné nastaviť limit na objem cachovaných dát.

3.4.2 Špeciálne vlastnosti klienta

Exkluzívny konzument

Broker vždy odosiela správy z fronty v poradí `FIFO`. To ale nezaručuje, že správy budú v takom poradí aj spracované. V prípade, že z danej fronty prijíma správy viacero konzumentov, niektorý z nich môže byť rýchlejší, iný pomalší, a tak nie je zaručené poradie spracovania správ. Riešením tohoto problému sú exkluzívni konzumenti. Ak existuje medzi konzumentmi viacero exkluzívnych konzumentov v kombinácii s viacerými klasickými konzumentmi, tak správy prijíma vždy jeden a ten istý exkluzívny konzument. V prípade, že zlyhá, ďalší exkluzívny konzument po ňom preberie zodpovednosť. Keď už existujú iba klasický konzumenti, správy sú opäť rozdelené rovnomerne medzi nich.

Skupiny správ

Skupiny správ sú vylepšením exkluzívnych konzumentov. Namiesto toho aby všetky správy vrámci jednej fronty chodili vždy jednému a tomu istému konzumentovi, pri skupine správ je možné použiť štandardnú JMS hlavičku `JMXSGroupID` na špecifikovanie cieľovej skupiny, do ktorej správa patrí. Správy opäť putujú k jednému a tomu istému konzumentovi vrámci skupiny (vysoká dostupnosť – v prípade poruchy je opäť zvolený ďalší konzument). O doručenie správy správnemu konzumentovi sa stará už broker. Je možné uzavrieť skupiny nastavením vlastnosti `JMSXGroupSeq=-1`. To uzavrie skupinu, čo znamená, že prípadná nová správa do danej skupiny bude už prijatá ďalším konzumentom.

ActiveMQ prúdy

Táto vlastnosť je už zastaralá. Na prenos veľkých súborov je odporúčané používať BLOB správy (viď. 3.4.2). Správy sú rozdelené na malé kúsky, ktoré sú potom jednoduchšie prenositeľné po sieti. Je potrebné, aby bolo zachované správne poradie prijatia správ. Preto je v prípade front silno odporúčané použiť exkluzívneho konzumenta. Je možné posielať prúdy správ aj pomocou tém, ale treba počítať s tým, že ak sa konzument pripojí v polovici odosielania prúdu, nebude mať celú správu a nikdy z nej nič zmysluplné nedostane. Po odoslaní je potrebné uzavrieť prúd aby príjemca vedel, že prenos dát je ukončený, a že obdržal celú správu.

BLOB správy

BLOB (Binary Large Objects) správy prinášajú iný pohľad na prenos veľkých objektov. Samotné objekty nie sú prenášané klasicky pomocou správ, ale pomocou iného mechanizmu (napríklad HTTP či FTP). Producent len odošle informačnú správu o tom, že objekt je dostupný na stiahnutie spolu s pomocnou informáciou o tom, ako ho konzument môže prijať.

3.5 Administrácia a monitorovanie

ActiveMQ ponúka viacero možností administrácie. Pre užívateľa najpohodlnejšia je webová konzola, ktorá umožňuje jednoduchý prístup k jednotlivým vlastnostiam pomocou prehľadného `GUI`. Existujú aj webové konzoly tretích strán, ktoré je tak isto možné použiť s ActiveMQ (napr. *hawtio*).

Java špecifikácia disponuje technológiou `JMX`, ktorá poskytuje štandardné API pre manažment Java aplikácií. Umožňuje pre aplikácie implementovať rozhranie pre správu vystavením funkcionality, ktorá môže byť spravovaná. Tie rozhrania pozostávajú z tzv. `MBeans`. Ďalšou možnosťou pre zbieranie informácií môžu byť už spomínané poradenské správy.

Najprimitívnejšou možnosťou pre administráciu je použitie nástrojov príkazovej riadky. Umožňujú niektoré základné úkony ako zastavenie brokera, zobraziť zoznam brokerov či prehliadanie správ a ďalšie.

Ladeniu aplikícií napomáha možnosť zaznamenávať všetky možné udalosti na strane brokera ale aj na strane klienta. Môžu byť zaznamenávané len do konzoly alebo do súboru.

Kapitola 4

Rabbit MQ

Rabbit MQ je `open-source` midlvér podobne ako ActiveMQ. Rozdielny je však už od základov, pretože je napísaný v jazyku Erlang[1].[14] Je to funkcionálny programovací jazyk, ktorý umožňuje vytvárať masívne škálovateľné aplikácie s dôrazom na vysokú dostupnosť. Využíva sa v telekomunikáciách, elektronickom obchodovaní či práve v instantnom vymieňaní správ. V čase, keď vznikala prvá verzia RabbitMQ, vznikla aj prvá verzia špecifikácie protokolu `AMQP`. Pružnosť Erlangu dovolila jednoducho reflektovať zmeny rozvíjajúceho sa `AMQP` do rozvíjajúceho sa brokera. s odstupom času a rozšírením iných protokolov prišla možnosť za pomoci zásuvných modulov nahradiť štandardný protokol napríklad za `STOMP`, `MQTT` či iné.

Na nasledujúcich riadkoch bude v skratke predstavený protokol, kľúčové vlastnosti nástroja, jeho pohľad na fronty správ, či metódy `pub-sub`, možnosti škálovateľnosti, administrácie a iné.[4]

4.1 Protokol AMQP

`AMQP`[2] je otvorený binárny protokol aplikačnej vrstvy pre `MOM`. Prichádza ako riešenie nedostatku `JMS`. Tou je uzavretosť pre jazyk Java. `JMS` nie je protokol tak ako `AMQP`, ale špecifikácia. Na samotnú komunikáciu potom môže byť v prípade `JMS` použitý ľubovoľný protokol (`AMQP`, `MQTT`, `OpenWire` ako v prípade ActiveMQ, `STOMP` ...). `JMS` predpisuje API a súbor požiadaviek na `MOM`, ktoré musí dodržiavať. Výhodou rozhodnutia zaviazať sa využívať protokol v tomto pohľade je to, že programátor má rozviazané ruky. Podobne ako HTTP či FTP, tak aj `AMQP` len popisuje formát správ posielaných po sieti a to zaručuje interoperabilitu naprieč rôznymi implementáciami, či dokonca rôznymi programovacími jazykmi.[6]

Okrem samotného protokolu popisuje aj AMQ model. Ten definuje sémantiku serveru, spôsob ,akým server zabezpečuje distribúciu správ medzi komponentmi. To dokáže pomocou troch hlavných komponentov:

- ústredňa – prijíma správy od odosielateľa a ďalej ich posiela do fronty správ
- fronta správ – ukladá správy, kým nemôžu byť bezpečne spracované príjemcom (alebo skupinou príjemcov)
- väzba – definuje väzbu medzi ústredňou a frontou správ a poskytuje možnosť smerovania správ

[1] Viac o Erlang na adrese https://www.erlang.org/
[2] AMQP – Advanced Message Queuing Protocol

4.2 Šírenie správ

Správa sa skladá z dvoch častí: štítok a samotný obsah správy. Producent vytvorí správu a odošle ju na server. Server na základe informácií v štítku identifikuje komu má správu odoslať ďalej. Producent môže ešte prípadne špecifikovať názov témy. Štítky teda umožňujú brokerovi smerovať správy. Ku konzumentovi už ale príde len samotný obsah správy bez štítku. Ak producent chce aby o ňom konzument vedel, musí tú informáciu zakomponovať do tela správy.

Správa môže obsahovať ľubovoľný obsah. Či už je to obyčajný textový reťazec, JSON, XML alebo prúd bytov nejakého videa. Dôležité je len aby príjemca vedel ak s danými dátami pracovať, ako ich naspäť serializovať.

Kanál

Na samotný prenos dát medzi producentom a brokerom a tak isto medzi brokerom a konzumentom je použité obyčajné TCP spojenie. Trik spočíva v tom, že ak aplikácia potrebuje viacero spojení so serverom (napr. odosielať/prijímať dáta do/z viacerých front), nepotrebuje pre každé jedno spojenie vytvoriť nové TCP spojenie. To by bolo neefektívne a pre systém drahé. Namiesto toho je pre každé jedno spojenie vytvorený kanál. Ide o virtuálne spojenie vnútri TCP spojenia, ktoré má svoj vlastný identifikátor, aby s ním broker vedel jednoducho pracovať.

Kanál umožňuje osamostatniť svoju obsluhu do samostatného vlákna, vďaka čomu kanály nie sú vôbec zdielané a nemôže medzi nastať premiešanie správ. Z hľadiska výkonu nie je vďaka tomu žiadny problém opakovane vytvárať, či rušiť existujúce kanály.

4.2.1 Fronta

Proces doručenia správy sa skladá z troch častí: prijatie správy na ústredni po odoslaní od producenta, uloženie správy do fronty a následné smerovanie správy až ku konzumentovi, ktorý sa o ňu zaujíma (spomenuté v časti 4.1).

Konzument určí frontu, ku ktorej sa pripojí k odberu pomocou jej mena. Následne má dve možnosti prijímania správ. Prvá je odporúčaná pre všetky systémy, ktoré si zakladajú na vysokej priepustnosti správ. Ide o klasické prihlásenie k odberu (AMQP príkaz `basic.consume`) z fronty kedy po prijatí alebo odmietnutí danej správy automaticky príjme ďalšiu správu vo fronte (môže sa líšiť ak je vo fronte viacero konzumetov). Druhou možnosťou je prijať len jednu jedinú správu (príkaz `basic.get`) a hneď po jej prijatí sa z odberu odhlásiť.

Ako už bolo načrtnuté, v prípade viacerých konzumentov sú pre nich správy rozdelené rovnomerne (štýlom `round-robin`). Ak nie je k dispozícii žiadny konzument, správy čakajú vo fronte.

Po prijatí správy sa očakáva potvrdenie od konzumenta, že túto správu dokázal nie len prijať, ale aj spracovať (`basic.ack`). Ak sa pred potvrdením konzument odhlási, broker rozpozná, že správa nebola spracovaná a odošle ju v poradí ďalšiemu konzumentovi. Naopak ak klient narazí pri spracovávaní na problém a kvôli nemu nemôže odoslať potvrdenie, broker túto správu nepošle nikomu inému. Toto správanie sa dá využiť vo svoj prospech, kedy v prípade náročného spracovávania konzument nie je zavalený obrovským množstvom správ, ktoré nie je schopný spracovať, ale je z `round-robin` poradia až do potvrdenia vyradený.

Ak konzument pri spracovaní narazí na chybu má tri možnosti ako sa vysporiadať s potvrdením/nepotrvrdením správy:

- odpojiť sa z kanála, čo automaticky doručí správu ďalšiemu z konzumentov,
- odmietnuť správu (`basic-reject`) s doručením správy inému konzumentovi (parameter `reque=true`),
- odmietnuť správu (`basic-reject`) bez doručenia správy ďalšiemu konzumentovi a uložením správy do fronty mŕtvych správ (parameter `reque=false`).

4.2.2 Režimy šírenia správ

O spôsobe, akým bude správa doručená rozhodujú ústredne a väzby s frontami. Producent odošle správu do ústredne a broker na základe pravidiel rozhodne, do ktorej fronty bude správa zaradená. Pravidlá sa nazývajú inak smerovacie kľúče. Najjednoduchší spôsob zasielania správ spočíva v tom, že ústredňa nie je špecifikovaná a použije sa tzv. štandardná ústredňa. Smerovanie medzi ústredňou a frontou spočíva len v názve fronty, ktorý je zároveň smerovacím kľúčom. To znamená, že keď užívateľ chce odoslať správu do fronty, špecifikuje jej názov, pomocou ktorého je požadovaná fronta spojená s ústredňou. Ústredňa sa pomocou špecifikovaného kľúča snaží nájsť frontu pre správu, ktorá môže byť v prípade neúspechu (fronta s daným kľúčom neexistuje) dokonca zahodená.

Okrem štandardnej ústredne existujú štyri režimy doručovania správ: priama ústredňa, `Fanout` ústredňa, ústredňa tém a ústredňa, ktorá smeruje podľa hlavičiek.

Fanout ústredňa – fanout exchange

Po štandardnej ústredni je druhá najtriviálnejšia práve `fanout` ústredňa. V skratke by sa dalo napísať, že funguje na princípe `pub-sub`. Umožňuje teda jednu správu smerovať do viacerých front.

V prípade štandardnej správy môže vyzerať odosielanie správy nasledovne:

```
channel.basicPublish("", "hello", ... );
```

kde prvým parametrom je názov ústredne (pre prázdny reťazec je zvolená štandardná ústredňa), druhým je smerovací kľúč, ktorý je názvom fronty. Ostatné parametre nie sú z hľadiska smerovania podstatné a preto bude odteraz táto metóda prezentovaná len s dvomi parametrami.

Po tom ako je deklarovaná ústredňa typu `fanout` môže producent zasielať správy spôsobom kde špecifikuje naopak len názov ústredne a smerovací kľúč nechá prázdny. To zaručí vloženie správy do každej fronty spojenej s danou pomenovanou ústredňou.

Vytvorenie väzby medzi frontou a ústredňou môže byť vykonané pomocou príkazu:

```
channel.queueBind(nazovFronty, NAZOV_USTREDNE, "");
```

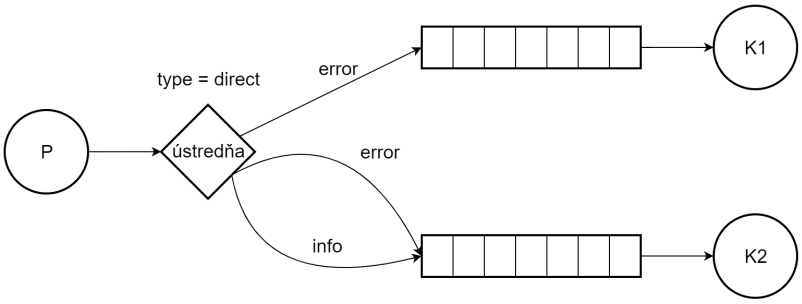
, kde názov fronty môže byť špecifikovaný, ale v prípade, ak aplikácii nezáleží na jej názve, môže byť vygenerovaný brokerom. Ďalší parameter je názov ústredne, ktorej vytvorenie bolo ukázané vyššie. Tretí parameter je väzbový kľúč, ktorý nás pre tento typ ústredne nezaujíma.

Každá takto spojená fronta s ústredňou obdrží rovnakú kópiu zaslanej správy čím je simulovaný `pub-sub` model.

Priama ústredňa – direct exchange

Využitím ďalšieho parametra pri vytváraní väzby medzi frontou a ústredňou je možné dosiahnuť priame smerovanie vrámci špecifikovanej ústredne. Takže je možné rozlišovať prijímanie správ podľa názvu ústredne a podľa smerovacieho kľúča (nový parameter `routingKey`), ktorý budeme v kontexte vytvárania väzby nazývať väzbový kľúč. Tak isto parameter `routingKey` už nebude ignorovaný ani pri odosielaní správ.

Príklad komunikácie pomocou priamej ústredne je zobrazený na obr. 4.1.

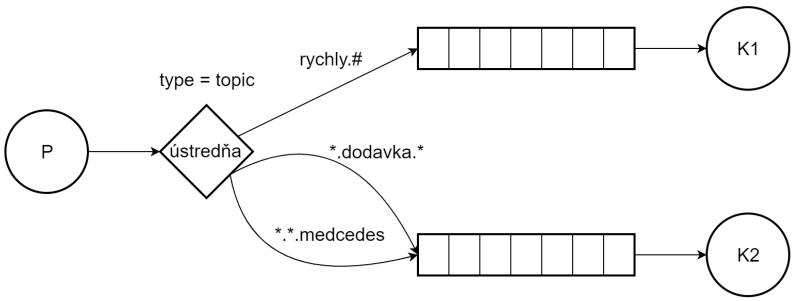


Obr. 4.1: Priama ústredňa

Ústredňa tém – topic exchange

Tento prípad vyžaduje zložitejší smerovací kľúč. Ten pozostáva z viacerých kľúčov oddelených bodkou (napríklad `rychly.dodavka.mercedes`). Väzbový kľúč potom môže byť generalizovaný pre príjem obecnejšej množiny správ pomocou týchto znakov:

- * – hviezdička – nahradí presne jedno slovo (`*.*.mercedes`),
- # – mriežka – nahradí nula alebo viacero slov (`#.mercedes`).



Obr. 4.2: Ústredňa tém

Ústredňa hlavičiek – header exchange

Namiesto smerovacieho kľúča používa na smerovanie argumenty špecifikované v hlavičke. Výhodou je to, že týchto hlavičiek môže byť viac a tým môže byť definovaných viacero

smerovacích kritérií. Správa je zaslaná do fronty, ak pravidlo definované v hlavičke sa zhoduje s hodnotou definovanou v čase vytvárania väzby. Keďže pravidiel v hlavičke môže byť viacero, existuje možnosť rozhodnúť či stačí na zhodu stačí jedno pravidlo (atribút `x-match=any`) alebo musia byť splnené všetky (atribút `x-match=all`).

4.2.3 Ďalšie možnosti

Virtuálny host

Virtuálny host umožňuje separovať ústredne, fronty, skupiny užívateľov a ďalších do oddelených prostredí. To umožňuje bezpečne a pohodlne použiť jeden RabbitMQ server pre rôzne aplikácie zároveň, kde sa nebudú nevhodne miešať jednotlivé ústredne či fronty medzi sebou. Klient si pri vytváraní spojenia zvolí host, na ktorý sa chce pripojiť.

`Trvácne správy` Správy ani fronty nie sú implicitne trvácne. Označenie fronty ako `durable` či spôsob doručenia správy ako perzistentný (`delivery_mode=2`) spôsobí to, že fronta po páde brokera nezanikne a tak isto správa bude uložená na disk pre zvýšenie garancie doručenia. Stále však garancia nie je sto percentná (na garanciu je možné použiť transakcie).

4.3 Výkon a spoľahlivosť

4.3.1 Klastrovanie

Podobne ako ActiveMQ aj RabbitMQ podporuje klastrovanie brokerov. Táto funkcionalita je zabudovaná a pridanie nových brokerov je možné vykonať za behu a vo veľmi krátkom čase.

Štandardne RabbitMQ nepodporuje kopírovanie správy na ostatných brokerov. Hlavným dôvodom je výpočtová náročnosť, rovnako ako zvýšené požiadavky na úložisko dát. Ak by každý broker ukladal 500MB správ, tak pri sieti desiatich brokerov by každý z nich musel mať úložisko o veľkosti 5GB na zálohovanie všetkých správ. Na lepšie pochopenie bude ďalej vysvetlená architektúra pri klastrovaní.

Odhliadnuc od klastrovania, každý jeden RabbitMQ uzol musí ukladať metadáta o svojich frontách, ústredniach, metadáta o väzbách a o virtuálnych hostoch. V prípade klastrovania pribúda nutnosť ukladať metadáta o samotných klastroch. Pri použití viacerých klastrov tak isto pribúda možnosť zvoliť si, či dáta budú ukladané do súboru, čo je štandardné chovanie pri samostatných klastroch, alebo budú kvôli výkonu držané len v pamäti RAM.

Ako už bolo spomenuté, fronta je držaná len v uzle, v ktorom bola vytvorená. Má to nesmierne výhody pri škálovaní sieti uzlov. Všetky ostatné uzly majú len základné informácie o fronte a odkaz na uzol, kde fronta existuje. To umožňuje pri obrovských sieťach distribuovať správy uzlom, ktoré obsahujú správnu frontu. V prípade, že uzol s požadovanou frontou zlyhá, tak správy odosielané do tejto fronty budú stratené (aj pre toto existuje riešenie – zrkadlené fronty), avšak uzol môže byť znovu obnovený a ak fronta bola definovaná ako trvácna, tak správy z nej sa nestratia.

Zatiaľ čo fronty nie sú replikované naprieč uzlami, pri ústredniach je to možné. Ústredňa totižto nie je komponenta ako taká, ale len nejaká vyhľadávajúca tabuľka s referenciami na jednotlivé fronty. Túto tabuľku je preto možné a nutné replikovať na každom jednom uzle. Po pridaní novej ústredne je každý uzol informovaný o zmene a tento záznam si uloží.

Systémy s jedným uzlom podporujú len režim ukladania na disk (sú aj v pamäti aj na disku, ale na disku sú kvôli možnému zotaveniu sa po chybe). V prípade klastrovania je možné použiť uzly aj s podporou RAM pamäte. Dôležitým faktom ale je, že je potrebný minimálne jeden uzol s možnosťou ukladania dát na disk. Ukladanie dát na disk je drahá operácia. V prípade, že tento uzol s diskom vypadne, sieť je stále použiteľná aspoň v obmedzenom režime. Aplikácie ale nemôžu vytvárať nové fronty, ústredne, väzby atď. až do zotavenia uzlu s diskom. Je možné mať viacero uzlov s podporou diskového úložiska – vtedy je systém viac odolný voči výpadkom.

Pre odladenie systému je možné vytvárať klastre na jednom zariadení, no pri nasadení do produkcie sa hodí ich mať na zvlášť zariadeniach. Rozdielnym prístupom oproti ActiveMQ je to, že pri RabbitMQ je výrazne doporučené aby všetci brokeri existovali na jednej sieti, čo neumožňuje prekonávať napríklad geografické prekážky. V opačnom prípade by odozvy systému boli obrovské.

4.3.2 Zrkadlené fronty

Klastrovanie zabezpečuje škálovateľnosť no nezabezpečuje vysokú dostupnosť. To je možné až vďaka explicitnému využitiu zrkadlených front. Takto označené fronty môžu byť buď replikované na všetky uzly, alebo len na presne špecifikované uzly. Ak pri prvom prípade vypadne hlavný uzol, najstarší slave uzol danej fronty sa stane novým master uzlom. Tak isto ak bude pridaný nový uzol do klastru, obdrží kópiu danej fronty.

V druhom prípade treba byť ostražitý. Ak je potreba použiť len nejakú podmnožinu uzlov, tak ich názvy je treba zakódovať do aplikácie, čo pri odstránení uzlu vyžaduje úpravu kódu.

4.3.3 Zotavenie z chýb

K vybudovaniu spoľahlivého systému je okrem zabezpečení brokera potrebné písať aj spoľahlivé klientske aplikácie. V prípade, že uzol, ku ktorému bol klient pripojený nie je kvôli poruche dostupný, klient sa musí sám postarať o znovu-pripojenie do klastru. Exisutje viacero stratégií ako sa znovu pripojiť k uzlu po jeho zotavení, no zaujímavejší prístup je využitie ďalšej komponenty, tzv. `load-balancer`, ktorý sám rozhodne na základe vyťaženia jednotlivých uzlov, ktorý uzol je vhodný na pridelenie. Aplikácii teda stačí vedieť adresu tohoto zariadenia a o viac sa nestará (v prípade potreby vyššej garancie je možné počítať aj s výpadkom pridelovača uzlov a mať ich tak isto viacero).

4.4 Administrácia a monitorovanie

Základným zabudovaným nástrojom pre administráciu RabbitMQ uzla je nástroj `rabbitmqctl`, fungujúci v príkazovom riadku. Dokáže vykonať všetko, čo je potrebné k administrácii uzla. V prípade požiadavku pre možnosť administrácie uzlov viacerými užívateľmi mu však dochádza dych. Problémom je nutnosť prístupu ku `Erlang cookie`, ktorá bola použitá k spusteniu serveru. Server bude pravdepodobne bežať s právami `root` takže na prístup k jeho súborom bude treba mať nastavené určité práva. To môže byť neúnosné pri väčšom počte administrátorov.

Riešením nie len problému s právami je zásuvný modul pre správu (oficiálny názov je `Mangement Plugin`). Ten prichádza hneď v troch rôznych formách. Prvá z nich je webové rozhranie, ktoré umožňuje pohodlný grafický prístup k monitorovaniu a administrá-

cii RabbitMQ uzlu. Webové rozhranie je síce fajn, ale neumožňuje administráciu nijakým spôsobom automatizovať. Preto druhým nástrojom, ktorý tento zásuvný modul ponúka je konzolová verzia webového rozhrania (*RabbitMQ Management CLI*). Poslednou možnosťou je HTTP API, ktorá je dostupná pre vzdialenú administráciu pomocou štandardných HTTP požiadaviek. Pre uľahčenie práce s HTTP API vznikol jednoduchý konzolový nástroj `rabbitmqadmin` napísaný v jazyku Python, ktorý značne skrýva vytváranie HTTP požiadaviek do viac užívateľsky prívetivejšieho šatu. Namiesto poslanie HTTP požiadavku z príkazového riadka v takomto tvare:

```
$ curl -i -u guest:guest http://localhost:55672/api/queues
```

stačí zavolať jednoduchý príkaz

```
./rabbitmqadmin list queue,.
```

ktorý vykoná to isté.

Kapitola 5

Apache Kafka

Rovnako ako ActiveMQ a RabbitMQ aj Kafka je `open-source`. Pôvodne bola interne vyvíjaná spoločnosťou *Linkedin* no od roku 2011 je vyvíjaná pod záštitou *Apache Software Foundation* a bola zverejnená pod licenciou *Apache License v2*. Je vyvíjaná v jazyku Scala[1] a Java.[12]

Kafka bola navrhovaná so zámerom zbierať dáta a sledovať aktivitu (vyhľadávania, zobrazené strány a ďalšie aktivity) užívateľov na stránkach za účelom ich analýzy a spracovania v reálnom čase, rovnako ako aj ich ďalšieho spracovania pomocou `Apache Hadoop`[2] alebo `offline` analýzy v rozsiahlych dátových skladoch.[8] Obrovské množstvo získaných dát a potreba ich spracovávania za behu prinútilo *Linkedin* aby sa vyhli zaužívanej obmedzujúcej špecifikácii `JMS` či protokolu `AMQP`.

Kafka poskytuje lepšiu natívnu replikáciu dát, čo poskytuje vyššiu garanciu doručenia správ. Tak isto umožňuje konzumentom pristupovať k správam v ľubovoľnom poradí. Od základu funguje Kafka na princípe streamovania dát, čo prináša značné výhody pri spracovaní obrovského množstva dát.

Ďalej v tejto kapitole bude priblížený rozdielny princíp fungovania brokerov, odlišné nároky na implementáciu konzumenta, spôsob použitia v štýle klasických systémov pre vymieňanie správ (fronty a `pub-sub`) a možnosti administrácie takéhoto systému.

5.1 Šírenie správ

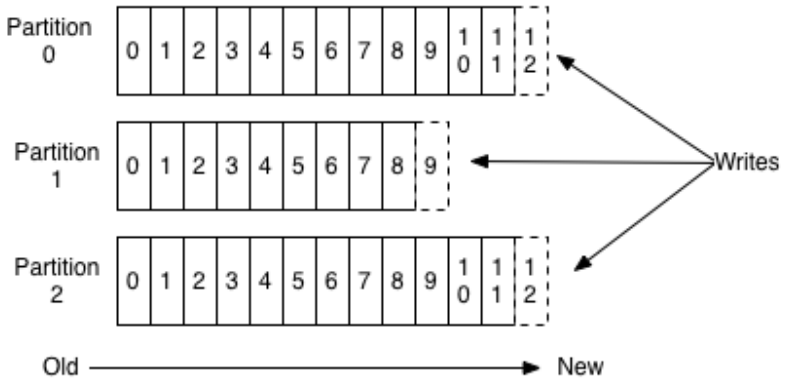
Téma

Dvomi základnými modelmi pre výmenu správ sú fronty a metóda `pub-sub`. Jedinou základnou abstrakciou pre prúd správ, ktorú Kafka ponúka je téma.[2] Každá téma môže mať nula, jedného alebo viacerých konzumentov. Kvôli možnosti lepšej distribúcie výkonu a zodpovednosti sú záznamy jednej témy rozdelené do partícií. Pre každú tému Kafka klaster uchováva log partícií (viď obr. 5.1). Partícia je zoradená sekvencia záznamov. Jej poradie je nemenné. Každý záznam má priradený sekvenčný identifikátor (ofset), ktorý ho unikátne identifikuje. Voľba správneho množstva partícií [3] závisí od špecifického požiadavku na systém.

[1] viac informácií o jazyku Scala dostupných na http://www.scala-lang.org/

[2] viac o nástroji Hadoop na adrese http://hadoop.apache.org/

[3] viac informácií o voľbe množstva partícií od jedného z bývalých architektov nástroja Kafka na adrese https://www.confluent.io/blog/how-to-choose-the-number-of-topicspartitions-in-a-kafka-cluster/



Obr. 5.1: Log partícií [2]

Záznamy sú k dispozícii na odoberanie po vopred stanovenú dobu. Tú určuje politika uchovávania údajov. Ak je nastavená na dva dni, tak po dobu dvoch dní sú záznamy v partícii k dispozícii a po uplynutí tohto času sú z disku odstránené z dôvodu šetrenia miesta. Zároveň však výkon systému Kafka nie je ovplyvnený objemom uchovávaných dát - či už je to 5GB alebo 50GB.

Distribúcia a replikovanie

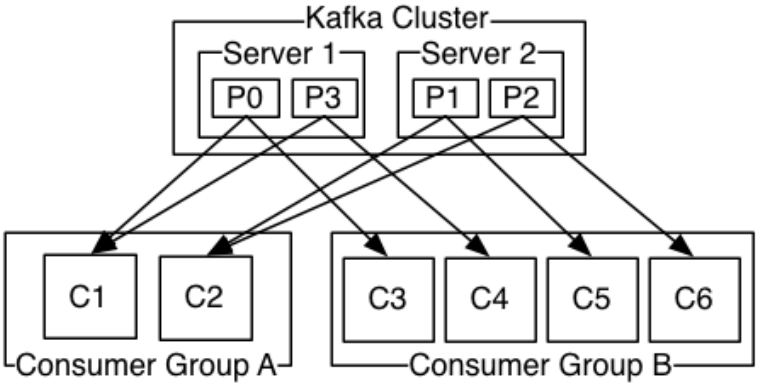
Jednotlivé partície danej témy sú potom rozdelené medzi viacero brokerov. To umožňuje paralelný prístup k záznamom viacerým konzumentom. Pokiaľ existuje viac ako jeden broker, tak nenastane prípad, kedy je jediný broker zodpovedný za celú tému.

Zároveň každá partícia môže byť replikovaná na producentom definované množstvo brokerov. Tým je zaručená vyššia garancia doručenej správy. Pri replikácii rádu `N` sa klaster dokáže intuitívne vysporiadať s výpadkom až `N-1` replík. Ak existuje v klastri viacero replík, len jedna partícia je označená ako líder, ostatné ako sluha. Na zápis a čítanie sa používa len líder a sluhovia sú určený len na zabezpečenie dát. Teda vždy po zmene záznamu v primárnej partícii nastáva distribúcia zmien medzi sluhov. Ak líder zlyhá, automaticky po ňom preberie túto funkciu jeden zo sluhov. Každý klaster môže byť lídrom pre niektoré partície a sluhom pre iné. To umožňuje rovnomerné rozloženie výkonu naprieč klastrom.

Odosielanie a prijímanie správ

Producent je zodpovedný za výber partície danej témy, do ktorej chce pridať záznam. To môže dosiahnuť pomocou automatického prideľovania `round-robin`, alebo môže špecifikovať partíciu napríklad pomocou kľúča v správe.

Konzument je zaujímavejší z hľadiska toho, aké má možnosti. Rôzne modely výmeny správ je možné dosiahnuť pomocou skupín konzumentov. Ak jedna skupina obsahuje viacero konzumentov, obdržia správu len raz a ich spracovanie si medzi sebou rozdelia čo zaručuje dobré rozdelenie výkonu. Tento princíp tak isto pripomína klasické fronty tak ako ich poznáme z predchádzajúcich prípadov. Ak je naopak požiadavok na to, aby každá správa bola spracovaná viacerými konzumentmi, stačí aby každý z nich mal svoju vlastnú skupinu. Toto je analógia ku klasickému `pub-sub` modelu.



Obr. 5.2: Skupiny konzumentov [2]

Ako to funguje v praxi je možné vidieť na obr. 5.2. Je na ňom znázornený Kafka klaster pozostávajúci z dvoch brokerov. Na každom z nich sú dve partície. Ďalej sú tam dve skupiny konzumentov. Jedn má dvoch členov a druhá štyroch. Ak je v skupine rovnaký počet konzument ako partícií v téme, každý číta sekvenčne práve jednu partíciu. Ak je počet konzumentov menší ako počet partícií, môžu niektorí konzumenti čítať záznamy aj viacerých partícií. V opačnom prípade môžu byť niektorí konzumenti bez partície.

Základné API

Pre informáciu je dobré spomenúť, že Kafka poskytuje až štyri rôzne API pre klientské aplikácie, pre pohodlnú prácu. Dve základné sú `Consumer API` a `Producent API`, ktorých činnosť už bola priblížená.

Ďalšia je `Streams API`. Umožňuje transformovať záznamy zo vstupných tém do výstupných tém. Využitie môžu nájsť vždy vtedy, keď vstupné témy zbierajú surové dáta, no reálni konzumenti majú záujem až o spracované dáta. Spracovať by si ich samozrejme mohol klient, ale použitím `Streams API` je možné tieto dáta spracovať raz a následne môžu byť odoberané opäť ľubovoľným množstvom jednoduchých klientov.

Posledná je `Connect API`. Je to nástroj pre jednoduché škálovateľné a spoľahlivé vymieňanie dát medzi celými systémami. Implementíciou konektorov (`Sink` a `Source`) je možné jednoducho naplniť tému priamo celou databázou. Exportovanie dát umožňuje uložiť dáta na nejakom sekundárnom úložisku, či posunúť ich iným systémom pre ďalšie dávkové `offline` spracovanie.

Zookeeper

Zookeeper je centralizovaná služba určená na konfiguráciu, pomenovanie, sprostredkovanie distribuovanej synchronizácie medzi jednotlivými komponentami a ďalšie. Kafka na svoj beh potrebuje `Zookeeper` a bez neho ju nie je možné spustiť.

Za jeho pomoci je napríklad upozornený broker, ktorý bol sluhom pre danú pertíciu po vypadnutí lídra o tom, že sa z neho stáva nový líder. Udržiava informáciu brokeroch, ktorí sú aktívni. Rovnako udržiava informácie o témach samotných. Vie o tom, ktoré témy existujú, koľko majú partícií, ktoré sú repliky a ďalšie špeciálne konfigurácie témy.

5.2 Administrácia a monitorovanie

Jednou z možností administrácie jednotlivých brokerov sú vstavané konzolové nástroje na to určené. Nachádzajú sa v priečinku `/usr/bin`. Je pomocou nich možné vytvárať nové témy, mazať témy, pridávať nové partície (mazanie nie je možné pretože partície môžu obsahovať dáta), vytvoriť jednoduchého konzolového konzumenta či producenta a ďalšie možnosti ako kontrola hodnoty `offset` pre skupiny konzumentov či vykonať kontrolované vypnutie brokera.

Čo sa týka monitorovania, Kafka tak isto ponúka možnosť využiť technológiu `JMX`. Nielen Kafka broker, ale aj `zookeeper` poskytuje niektoré svoje informácie pomocou `MBeans`, čo umožnilo rozvoj viacerých nástrojov zbierajúcich práve dáta pomocou `JMX`. V priebehu času vzniklo mnoho implementácií nástrojov na monitorovanie a administráciu. Toto sú niektoré zaujímavé z nich:

- *Burrow* od spoločnosti *Linkedin* – masívny nástroj pre monitorovanie,
- *kafkat* od spoločnosti *airbnb* – nástroj na manažment,
- *kafka-manager* od spoločnosti *yahoo* – grafický nástroj tiež skôr na manažment.

Kapitola 6

Analýza a návrh

Veľkou výhodou troch predstavených nástrojov je fakt, že sú voľne dostupné. Ktokoľvek si môže vytvoriť doma na svojom počítači inštanciu MQ systému. Takéto riešenie môže byť postačujúce na edukatívne účely či na fázu vývoja aplikácie, kedy nie je kladený veľký dôraz na pokročilejšie možnosti nastavenia daného MQ systému. V prípade, že sa v rámci firemných procesov rozhodnete pri vývoji aplikácií využívať výhody vymieňania správ, ideálne budete potrebovať ďalší server, na ktorom bude váš MQ systém bežať. Bude jednoduchšie vyvíjať aplikácie v tímoch s centrálnym serverom než v prípade, kedy by mal každý programátor takýto systém nakonfigurovaný na svojom počítači.

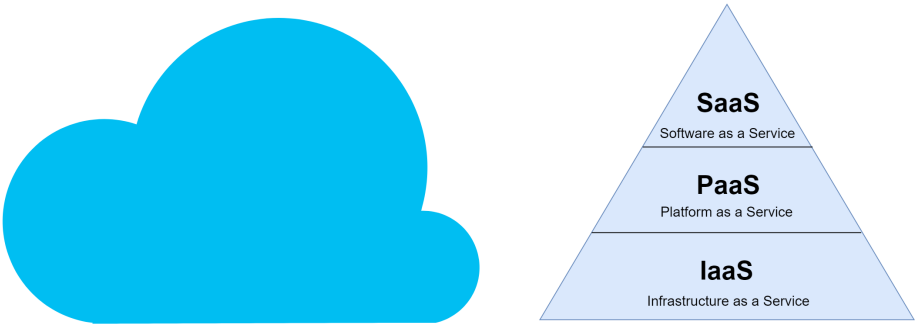
S centrálnym serverom je možné lepšie simulovať reálne nasadenie aplikácie v praxi. Umožní zvoliť to správne nastavenie MQ systému pre optimálny výkon spolu s presnejším nastavením dĺžky dostupnosti vašich správ, prípadne stupňa ich replikácie.

To však prináša zvýšený nárok na zdroje. Či už sú to tie hmotné – ďalší server, alebo tie nehmotné – financie. V neposlednom rade medzi ne patrí aj pracovná sila. O chod takéhoto serveru sa bude musieť starať kvalifikovaná osoba. Nejde len o konfiguráciu, ale aj o monitorovanie, či podporu v prípadoch zlyhania. Server bude vyžadovať aktualizácie. Všetky spomenuté nástroje sú tak isto stále aktívne vyvíjané, doplňované o novú funkcionalitu, či bude nutná náhrada starej funkcionality novým, vhodnejším riešením. Preto, ak chce spoločnosť držať krok s novými technológiami, je vhodné tieto nové aktualizácie v nástrojoch nasadiť aj u seba.

Vývoj aplikácie je jedna vec, no nasadenie do produkcie je niečo úplne iné. Výsledná aplikácia bude potrebovať iný server ako ten, ktorý je využívaný interne na vývoj. To sú ďalšie zdroje navyše.

Menšie spoločnosti, ktoré nemajú vybudovanú takú rozsiahlu infraštruktúru, sú limitované napríklad nedostatkom priestorov či neschopnosťou zabezpečiť ľudské zdroje potrebné pre správu takéhoto systému by ocenili prístup, ktorý ich odbremení od všetkých týchto starostí.

Vhodným riešením je práve využitie MQ systému formou `cloudovej` služby (webovej služby). Takýto prístup rieši nevýhody spomenuté vyššie. Prináša skvelé možnosti škálovateľnosti a celkového prispôsobenia služby na mieru klientovi bez nutnosti hlbšej znalosti o jej internej konfigurácii. Používateľ platí na pravidelnej báze paušálne za to, čo reálne využíva. Poplatok sa môže meniť v závislosti na zvolenú konfiguráciu celej služby. Zároveň konfigurácia nemusí byť vôbec pevná, ale jej nastavenie je možné v čase meniť a prispôsobovať ho tak aktuálnym požiadavkám klienta. Používateľ sa nemusí starať o akékoľvek aktualizácie systému ani sa nemusí vysporadúvať s prípadným zlyhaním hardvéru. O všetky tieto veci sa stará prevádzkovateľ webovej služby.



Obr. 6.1: Cloudová služba

Na obrázku 6.2 je možné vidieť tri základné modely webových služieb: infraštruktúra ako služba (IaaS – Infrastructure as a Service), platforma ako služba (PaaS – Platform as a Service) a softvér ako služba (SaaS – Software as a Service).[11]

Infraštruktúra ako služba

Prvý menovaný je v hierarchii úplne najnižšie. V skratke sa dá napísať, že používateľ má v prenájme virtuálne dátové centrum. Získa tak prístup k vybudovanej hardvérovej a softvérovej infraštruktúre, ktorú potrebuje pre svoju činnosť. Nemusí sa starať o zakúpenie hardvéru, nemusí napríklad infraštruktúru zabezpečovať záložnými zdrojmi proti výpadku energie, pretože tieto úkony spadajú pod zodpovednosť sprostredkovateľa. Zjednodušuje napríklad aj úkony ako zväčšenie diskového priestoru. Počítače bežia často virtuálne na väčších serveroch. Používateľ platí často len za to, čo reálne využije. Poplatky môžu byť účtované na hodinovej, dennej, týždňovej či mesačnej báze, no je možné účtovať poplatky aj napríklad na základe množstva využitého diskového priestoru.

Platforma ako služba

Druhá menovaná poskytuje podobnú službu, no s vyššou úrovňou abstrakcie. Poskytovateľ okrem hardvéru a infraštruktúry poskytuje často aj základné softvérové vybavenie a operačný systém. Jedným zo známych poskytovateľov služieb PaaS je napríklad *Microsoft* so svojím *Microsoft Azure*[1]. Používateľ sa môže plne sústrediť len na vývoj svojej aplikácie. PaaS je teda akási nadmnožina IaaS.

Softvér ako služba

Tretí prípad – softvér ako služba konečne najlepšie predstavuje to, čo by sa mohlo očakávať od MQ systému bežiaceho formou služby. Ten hovorí o tom, že klient môže využívať konkrétnu aplikáciu sprostredkovateľa a nespravuje pritom infraštruktúru serveru, operačného systému, ani úložiska. Výnimkou môže byť malá množina nastavení, ktoré môže užívateľ ovplyvniť. Tieto nastavenia môže aplikovať napríklad voľbou balíčku z portfólia služby, čím má možnosť škálovať svoju službu.

[1] Azure – https://azure.microsoft.com/en-us/overview/what-is-azure/

6.1 Cieľ práce

Po predstavení základných vlastností a niektorých pokročilých vlastností troch spomenutých nástrojov je čas predstaviť, čo má byť výsledkom tejto práce. Jednotlivé riešenia prišli v nie malom časovom rozostupe. Prvý Apache ActiveMQ prišiel už pred takmer pätnástimi rokmi. Úvodná verzia RabbitMQ prišla o tri roky neskôr v roku 2007 a Apache Kafka až v roku 2011. Každý z nástrojov pristupuje k spôsobu, akým implementuje vymieňanie správ po svojom. Tomu faktu dopomáha aj časový rozostup, s akým boli nástroje vydané. Rovnako každý z nástrojov má vlastný prístup k administrácii a monitorovaniu daného MQ systému. Na základe dôkladného preskúmania spoločných a rozdielnych vlastností je zmyslom práce navrhnúť webovú službu. Mala by umožniť vytvárať a spravovať fronty rôznych modelov, pristupovať k týmto frontám spôsobom bežným pri komunikácii formou zasielania správ a v neposlednom rade aj monitorovanie behu MQ systému. Hlavnou výhodou okrem nižších počiatočných nákladov má byť aj fakt, že klient nepotrebuje takmer žiadnu znalosť o tom, ako sa s jednotlivými nástrojmi použitými na pozadí reálne narába. Dôležitým bodom je v tomto prípade navrhnúť službu tak, aby mohla byť používaná s ľubovoľným zo spomenutých nástrojov, prípadne aj ďalšími.

Obr. 6.2: Požiadavka na systém

6.2 Existujúce riešenia

Confluent Cloud

Prvé predstavené existujúce riešenie je platforma *Confluent*. Ide o robustnú platformu, ktorá je vybudovaná nad nástrojom Apache Kafka. Za jej vznikom stoja ľudia zo spoločnosti *Linkedin*. Tí istí ľudia stáli kedysi aj za vznikom samotnej Kafky.

Ponúka dve verzie nástroja: `Open Source` a `Enterprise`. Ako z názvu vyplýva, prvá z nich je dostupná pod verziou OpenSource (konkrétne Apache v2.0). V tejto verzii ponúka základnú funkcionalitu a balíčky potrebné pre využívanie Kafky pre zasielnie správ. Neponúka v nej však plnú funkcionalitu. Chýbajú balíčky zabezpečujúce kontrolu a monitorovanie. Tie sú dostupné len v Enterprise verzii. Navyše platforma Confluent až do prelomu rokov 2017/2018 neponúkala možnosť využívať Confluent formou webovej služby. Tento

projekt je tak v čase písania tejto práce veľmi čerstvý, no vzhľadom na rozšírenie platformy samotnej predurčený na úspech. Viac informácií o službe dostupných na odkaze[2].

Obr. 6.3: Confluent Platforma

CloudAMQP

Služba CloudAMQP [3] by sa dala charakterizovať ako skupina niekoľkých spravovaných RabbitMQ serverov v cloude. Ako bolo spomenuté v kapitole č.4, samotný nástroj je dostupný pod licenciou OpenSource. Používateľ alebo prípadný záujemca, ktorý by chcel nástroj tejto služby spojazdniť na svojom hardvéri má smolu, pretože samotná implementácia nie je zverejnená. Ponúka zákazníkom možnosť vybrať si z niekoľkých balíčkov, ktoré obmedzujú poskytujú používateľovi rôzne stupne obmedzenia. Zároveň ponúka aj základnú neplatenú verziu vhodnú na vyskúšanie a na vývoj.

Obr. 6.4: CloudAMQP

Tu sú niektoré z vlastností, ktoré je možné ovplyvniť voľbou balíčka:

- počet front
- celkový počet správ vo frontách
- dĺžka expirácie správ vo fronte
- maximálny počet správ za sekundu
- maximálny počet aktívnych pripojení
- počet uzlov.

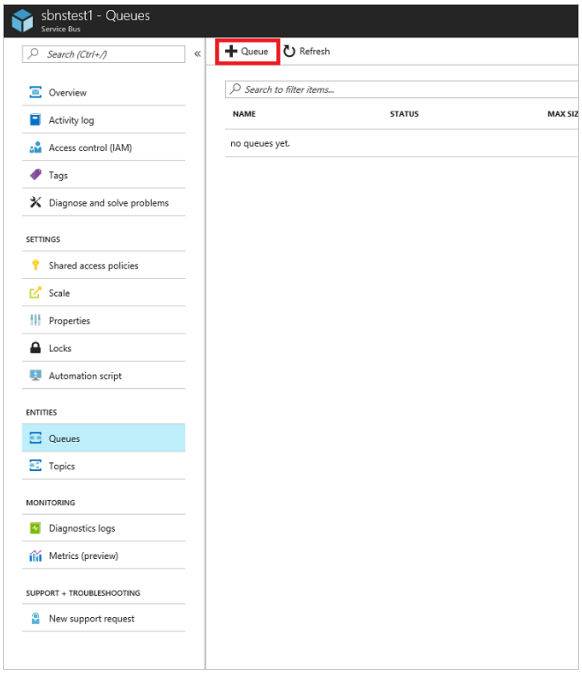
Microsoft a Amazon

Nasledujúce dve služby nezapadajú úplne do koncepcie popisovanej touto prácou, ale ide o „popredných hráčov“ vo svete cloudu a preto je vhodné ich aspoň spomenúť.

Microsoft Azure

Microsoft Azure poskytuje všetky tri modely cloudových služieb: IaaS, PaaS aj SaaS. Využíva sa k vytváraniu, hostovaniu webových aplikácií na datacentrách[4] Microsoftu, s ľubovoľným stupňom škálovania. Jeho súčasťou je samostatný operačný systém `Windows Azure`, ktorý slúži ako portál na spúšťanie a konfiguráciu ďalších služieb.

[2] Viac informácií o Confluent je na adsese https://www.confluent.io/.
[3] CloudAMQP – https://www.cloudamqp.com/docs/index.html
[4] Datacentrá, alebo inak serverovne sú špecializované priestory pre umiestnenie veľkého množstva serverov, ktoré sú určené k nepretržitej prevádzke.



Obr. 6.5: Microsoft Azure – ukážka správy front

Vymieňanie správ umožňuje pomocou svojej služby `Azure Service Bus` a jej REST API rozhrania[5]. Okrem toho ponúka aj klienta v jazyku .NET.

Amazon Web Services – AWS

Služba pre výmenu správ sa nazýva `Amazon Simple Queue Service`[6]. Rovnako ako k službe Serice Bus, je možné pristupovať pomocou klientskych knižníc. Amazon navyše poskytuje balíčky SDK, ktoré umožňujú využívať túto službu z takmer všetkých aktuálne populárnych programovacích jazykov (Java, .NET, Node.js, PHP, Python, Ruby a iné). Umožňuje využiť služby aj pomocou REST API.

Obe tieto služby sú určite veľmi zaujímavé a sprostredkované spoločnosťou s obrovským menom. Vynikajú v možnosti škálovania, v podpore a tak isto aj vo vyhliadkach do budúcna z hľadiska ich ďalšieho vývoja. Ani jedna z nich však nie je OpenSource a ani jedna z nich nefunguje nad žiadnym OpenSource nástrojom.

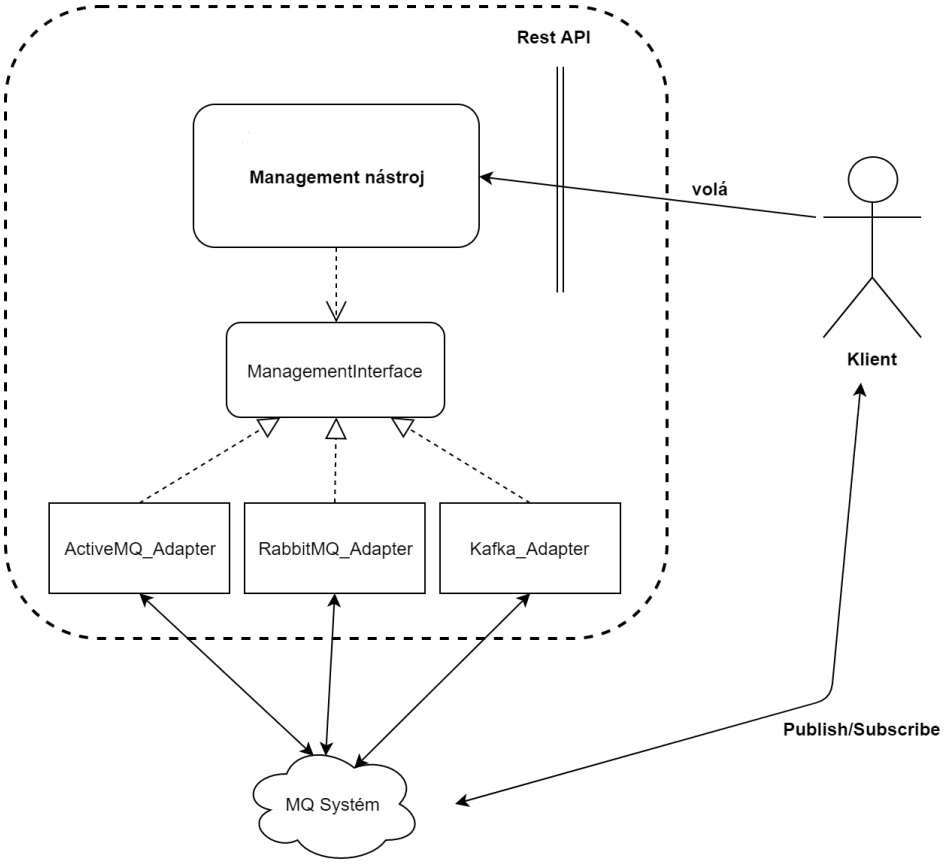
6.3 Návrh

Prvotný návrh

Pôvodný návrh výslednej webovej služby bol značne rozdielny od toho, ako návrh vyzeral po jeho dokončení. Najväčší rozdiel spočíval v tom, že pri prvotnom návrhu nebol kladený až taký dôraz na výsledné nasadenie aplikácie formou webovej služby. Pôvodný návrh je možné vidieť na obrázku č. 6.6. Používateľ cez REST rozhranie pristupuje k `Management` nástroju, ktorý je umiestnený na niektorom zo serverov poskytovateľa. Rovnako ako vo

[5]Service Bus REST API – https://docs.microsoft.com/en-us/rest/api/servicebus/
[6]Amazon SQS – https://aws.amazon.com/documentation/sqs/

finálnom návrhu tento nástroj po implementovaní špecifického adaptéru na `Management Interface` dokáže pracovať s ľubovoľným z troch definovaných nástrojov. Avšak rozdielom je to, že tento nástroj mal byť určený skôr na správu daného MQ systému a nie na celkové použitie. Na obrázku je možné vidieť, že finálna komunikácia s daným brokerom by prebiehala priamo, tzn. mimo `management` systému. Klient by úkony ako `publish`, `subscribe` vykonával priamo nad inštanciou konkrétneho brokera, či už by to bol *RabbitMQ*, ActiveMQ alebo Kafka. Rovnako by mohol priamo bez využitia služby vykonávať aj základné administračné úlohy, ako sú napríklad spravovanie tém či spravovanie front.



Obr. 6.6: Prvotný návrh aplikácie

Takýto prístup by bol tak isto zaujímavým riešením, no neodbremenil by používateľa (programátora) od nutnej technickej znalosti konkrétneho MQ systému. Výsledný nástroj by tak mohol používať na administráciu všetkých troch systémov jedným spôsobom no na samotné vymieňanie správ by bol nútený využiť konkrétne knižnice pre daný nástroj. To môže byť limitujúci faktor v prípade, že pri implementácii distribuovaného systému je použitý jazyk, pre ktorý niektorá z knižníc nemá podporu. Väčším problémom by bol však samotný fakt, že každá z knižníc by si vyžadovala rozdielnu implementáciu aj na strane klienta.

Pri rozhodnutí sa využiť vo svojom systéme hotové riešenie formou webovej služby sa očakáva, že to prinesie okrem výhod spojených z nižšími nákladmi aj nižšie úsilie vynaložené pri implementácii takého systému.

Aby toto riešenie nebolo označené za úplne nesprávne treba vyzdvihnúť jednu jeho vlastnosť, ktorá sa odlišuje od finálneho návrhu. Aj keď za cenu náročnejšej implementácie, ponúka možnosť využiť aj niektoré pokročilejšie vlastnosti konkrétneho systému. To by však prinieslo celú škálu ďalších problémov. Najväčším z nich asi ten, že by bola oveľa väčšia výzva na strane používateľov aj na strane poskytovateľa udržať kompatibilitu medzi jednotlivými nástrojmi.

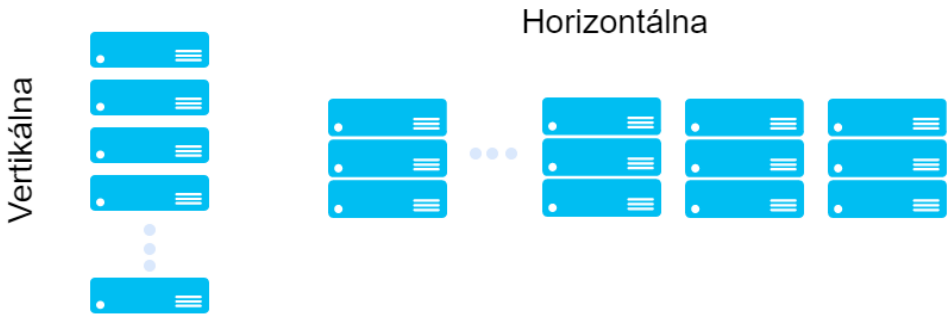
Okrem ťažkopádnej realizácie takéhoto riešenia bol hlavným dôvôdom, prečo bol tento návrh zahodený ten, že neumožňoval dostatočnú možnosť škálovania, čo je jedna zo základných požiadaviek na webovú službu.

Škálovateľnosť

Pred predstavením návrhu systému je ešte dobré priblížiť pojem škálovateľnosť a vysvetliť prečo tak blízko súvisí s webovými službami. Škálovateľný systém je taký systém, ktorý môže byť v prípade zvýšenej záťaže alebo zmeny požiadaviek rozšírený tak, aby nové požiadavky zvládol a neovplyvnil tak svoju funkcionalitu.

Rozlišujeme dva hlavné typy škálovateľnosti (obr. č. 6.8): vertikálna a horizontálna. z týchto dvoch možností je práve vertikálna tá jednoduchšia. Príklad vertikálneho škálovania môže byť navýšenie diskovej kapacity či zrýchlenie disku (výmena HDD za SSD), navýšenie pamäte RAM či výmena procesora za rýchlejší. Tento spôsob je v dnešnej dobe ešte jednoduchší. Pri využívaní služby nejakej zo služieb IaaS (Microsoft Azure, AWS) je možné vertikálnu škálovateľnosť aplikovať veľmi jednoducho, pretože hardvér je virtualizovaný na serveri. Ten disponuje často väčším úložiskom ako aj vyšším výkonom, ktorý je dostupný používateľovi v prípade potreby.

Druhý typ je horizontálna škálovateľnosť. V prípade potreby sa fyzicky zvýši počet serverov, čo pomôže rozložiť záťaž v najlepšom prípade rovnomerne na všetky servery. Dá sa povedať, že tento typ prináša dlhodobejšie riešenie a takisto je vhodnejšie pre prístup veľkého množstva používateľov. Okrem výhod, ktoré prináša treba počítať aj s nevýhodami. Medzi ne patrí zvýšená komplexnosť systému, čo sa prejaví napríklad pri aktualizáciách. Tak isto treba riešiť zdieľanie dát medzi jednotlivými servermi. Pridávanie čím ďalej tým väčšieho množstva serverov môže v istých prípadoch znamenať aj zvýšenú odozvu celého systému pri jednoduchých úkonoch.



Obr. 6.7: Typy škálovateľnosti

Dôvodom, prečo je škálovateľnosť tak úzko spätá s webovými službami je ten, že práve vďaka nej je možné používateľom ponúknuť službu presne podľa ich potrieb. V závislosti od zvolenej konfigurácie sa potom odvíja celková cena služby. Vďaka tomu je možné vytvoriť

pestré portfólio pre danú službu od voľnej verzie až po platenú verziu pre pokročilého používateľa.

Modely výmeny správ

Z analýzy modelov správ troch spomenutých nástrojov vyplynulo ako najideálnejšie riešenie rozhodnúť sa v službe implementovať práve možnosť vymieňania správ pomocou front a modelu `publish/subscribe`. V nástroji ActiveMQ sú obe z nich priamo podporované. RabbitMQ podporuje štyri modely: `Fanout` ústredňa, priama ústredňa, ústredňa tém a ústredňa hlavičiek. Spolu s nimi existuje ešte možnosť využitia základnej ústredne, ktorá sa použije, ak nie je jej názov stanovený (viac informácií v sekcii 4.2.2). Pre implementáciu zvolených modelov je možné zvoliť dva z poskytovaných modelov:

- `fronta` – pomocou základnej ústredni
- `publish-subscribe` – prekvapivo nie pomocou ústredne tém, ale pomocou ústredne typu `fanout` (viac info v 4.2.2)

V prípade Apache Kafka je realizácia oboch modelov najzložitejšia. Stále však nie je nerealizovateľná. Kafka podporuje len jediný model pre výmenu správ – frontu. Našťastie pomocou implementácie na strane klienta je možné dosiahnuť funkcionalitu obdobnú práve modelu `pub-sub`. Slúžia na to pomenované skupiny konzumentov (viac informácií v sekcii 5.1). Štandardne je teda táto funkcionalita implementovaná až na strane konzumenta. V navrhovanom systéme by však konzument nebol koncový klient, ale samotná webová služba. To umožní zachovať konzistentný charakter služby pre všetky nástroje.

Datový typ správy

Návrh podporovaného formátu správy je do značnej miery daný tým, že služba je implementovaná ako REST služba. Na zasielanie správ sa bude využívať dotaz typu `POST`[7]

POST dotaz (rovnako ako aj PUT) obsahuje telo, ktoré je určené na zasielanie dát na server. Typ dát môže byť špecifikovaný v hlavičke. Pre účely tejto aplikácie bude použitý typ `"Content-Type: application/json"`. V tele dotazu sa potom napríklad pre odosielanie správ očakáva jediný `json` element `message` ako je možné vidieť na nasledujúcej ukážke:

```
{
        "message" : "obsah spravy"
}.
```

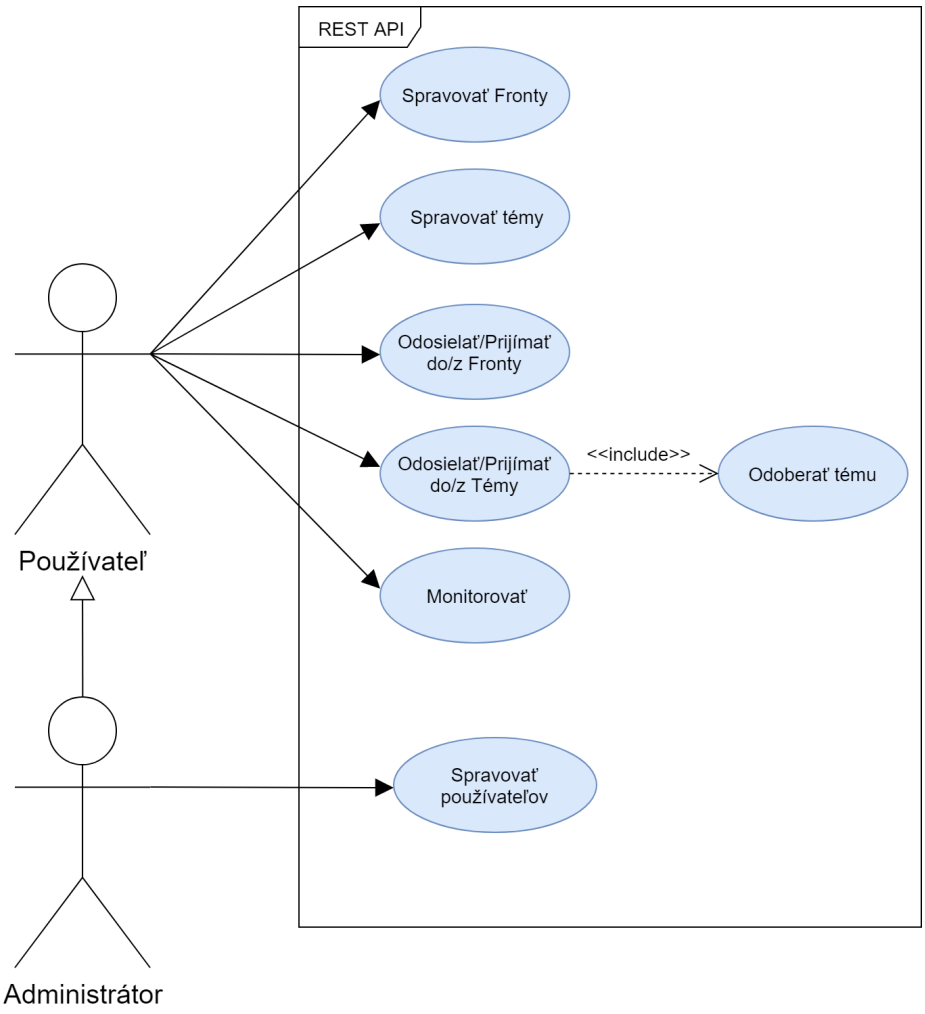
Bude na používateľovi samotnom, čo bude do týchto správ ukladať. Nezáleží na tom, či to budú čísla, `plain-text`, serializovaný objekt vo formáte json, XML štruktúra či prúd bytov nejakého obrázku. Jediným limitom je to, že konzument musí vedieť, aké dáta chce čítať, aby vedel ako má s nimi narábať. Takýto prístup sa zdá byť ako dostatočný a pri vhodnom vytvorení jednotlivých front či tém je možné rozlišovať, ktorá fronta bude obsahovať dáta akého typu.

Maximálna veľkosť správy je potom ovplyvnená maximálnou možnou veľkosťou POST dotazu. Tá je limitovaná nastavením serveru, na ktorom služba beží.

[7] informácie aj o POST dotaze dostupné na adrese https://www.w3schools.com/tags/ref_httpmethods.asp

6.3.1 Prípad použitia

Dôvodom pre zmenu v návrhu bola aj nedostatočná škálovateľnosť služby, ktorá vyplývala z pôvodného návrhu. Nový návrh už teda počíta s možnosťou nastavenia služby podľa viacerých kritérií (viac informácií ďalej v tejto kapitole). Na nasledujúcich riadkoch bude služba predstavená pomocou jednotlivých prípadov použitia.



Obr. 6.8: Diagram prípadov použitia systému

V systéme môžu vystupovať dvaja aktéri: používateľ a administrátor. z názvu aj z diagramu vyplýva, že prípady použitia, v ktorých vystupuje administrátor sú nadmnožinou prípadov použitia, v ktorých vystupuje obyčajný používateľ. Prvé dva základné prípady použitia zobrazujú možnosť spravovania front a tém. Medzi úkony, ktoré je možné vrámci správy vykonať patria tieto:

- vytvorenie fronty/témy
- odstránenie fronty/témy
- vyprázdnenie fronty/témy
- zobraziť všetky fronty/témy.

Ďalšou časťou systému potrebnou k tomu, aby nešlo len o administračný nástroj, ale o komplexný nástroj, ktorý poskytuje možnosť odosielať a prijímať správy pre fronty aj témy. Prijímanie správ z front bude fungovať trochu inak, ako bolo popísané pre jednotlivé modely pri konkrétnych nástrojoch. z diagramu je možné vyčítať, že ak chce používateľ prijímať správy z témy, musí túto tému odoberať. To je veľmi podstatná požiadavka, pretože ak sa používateľ prihlási na odber z nejakej témy, vytvorí sa mu interne fronta, v ktorej sa mu radia správy exkluzívne len pre neho. Nie je možnosť aby prijal správy, ktoré boli zverejnené pod nejakou tému z času pred svojím prihlásením k odberu.

Pri frontách je to trochu inak. Ani pri jednotlivých nástrojoch nebol v spojitosti s frontami použitý výraz „odoberať“. Čítanie správ z fronty funguje v princípe tak, že klient ma vytvorené aktívne spojenie s brokerom a čaká na nové správy vo fronte. Ak je takýchto klientov pripojených na jednu frontu viacero, sú obslúžení v zásade štýlom `round-robin`. Toto je vlastnosť, ktorú je veľmi ťažké implementovať vo webovej službe. Webová služba je štandardne bezstavová. To znamená, že medzi jednotlivými dotazmi od klienta nie je uchovaný žiadny stav a každý takýto požiadavok začína od nuly. Pri REST prístupe je nemožné dosiahnuť to, že klient si požiada o správu z danej fronty jeden krát a broker mu potom správu pošle vždy, keď bude v danej fronte dostupná. z toho dôvodu bol zvolený prístup, kedy sa vôbec nebude riešiť nejaké rovnomerné rozdelenie záťaže medzi jednotlivých klientov. Kto prvý príde, ten prvý berie. Pri fronte aj tak nezáleží, kto danú správu spracuje. Ide o to aby každá správa bola spracovaná jeden krát. Preto ak klient ukončí spracovanie predtým získanej správy, môže si požiadať o ďalšiu a ak je správa dostupná, dostane ju. Rovnako treba počítať s faktom, že narozdiel od témy môže klient prijať aj správu, ktorá je vo fronte uložená už dlhý čas predtým, ako sa klient k tejto fronte prvý krát dostal.

Ďalším prípadom je monitorovanie. Používateľ by nemal byť zaťažený sledovaním aktuálne dostupného množstva serverov. Ak toto číslo klesne pod úroveň, ktorú poskytovateľ zaručuje, poskytovateľ služby o tom bude varovaný a bude na ňom aby vykonal kroky smerujúce k odstráneniu problému. Používateľa však môžu zaujímať informácie o prenesených dátach. Môžu byť použité v nejakých grafoch a dashboardoch zobrazujúcih prenos dát za posledný deň/mesiac/rok. Rovnako bude mať možnosť zistiť, ktorý užívateľ zasiela aké množstvo dát, či cez ktorú z tém/front pretečie najväčšie množstvo dát.

Posledný prípad je dostupný len administrátorovi. Ide o správu používateľov. Môže vytvárať nových používateľov (aj administrátorov), upravovať či mazať existujúcich. V prípade potreby môže zobraziť zoznam všetkých používateľov.

Škálovanie v službe

Diagram prípadov použitia stále nijako nerieši škálovateľnosť. Tá je dôležitá nie len pre používateľa, ale aj pre sprostredkovateľa služby. Práve na základe nej sa sprostredkovateľ zaväzuje v tom, akú minimálnu konfiguráciu plánuje klientovi garantovať. Ak by neexistovalo nejaké obmedzenie nahratých či stiahnutých dát, ťažko by sa prevádzkovateľ mohol pripraviť na to, čo všetko do jednotlivých front zákazník uloží. z toho dôvodu je rozumné zaviesť nejaké obmedzenie prenesených dát. Je vhodné brať na vedomie to, že požiadavka na objem dostupných dát na `upload` bude iná ako na `download`. Pokiaľ by šlo o frontu, tak prenesené nahraté a stiahnuté dáta by boli v pomere 1:1. Dôvodom je, že každá správa z fronty je doručená jediný krát. Iná situácia nastáva pri modeli `pub-sub`. Jeden krát nahraté dáta môžu byť stiahnuté až x krát, kde x predstavuje počet odberateľov danej témy. Preto je vhodné mať možnosť nastaviť v konfigurácii služby zvlášť obmedzenie pre `upload` a zvlášť pre `download`.

Obmedzenie nahratých dát však nie je dostatočné na odhadnutie celkového požadovaného úložiska potrebného pre prevádzkovanie služby. Ak by doba existencie správy vo fronte nebola obmedzená, veľkosť potrebného úložiska by sa mohla zväčšovať do nekonečna, ak by klient správy do fronty len pridával. z toho dôvodu je vhodné mať možnosť obmedziť dobu expirácie správ vo fronte.

Posledným nastavením je počet uzlov v klastri. Tým sa vie klient obrniť proti prípadnej strate dát či dostupnosti klastra pri výpadku jedného alebo viacerých uzlov. Ak si klient zvolí konfiguráciu s klastrom o `N` uzloch, tak takáto konfigurácia sa dokáže vysporiadať so stratou až `N-1` uzlov.

Pre zhrnutie bude mať teda poskytovateľ tieto štyri nastavenia, vďaka ktorým bude môcť ponúkať službu v rôznych balíčkoch:

- `Replication` – stupeň zrkadlenia jednotlivých front (inými slovami počet brokerov)
- `Retention` – doba expirácie správ v jednotkách času (dní)
- `Upload limit` – obmedzenie objemu nahratých dát za deň
- `Download limit` – obmedzenie objemu stiahnutých dát za deň.

6.4 Architektúra REST služby

REST (Representational State Transfer) je architektúra rozhrania navrhnutá pre distribuované systémy postavená na štandarde HTTP. V roku 2000 s ňou prišiel jeden z autorov protokolu HTTP Roy Fielding. Ide o štandard, ktorý definuje množinu obmedzení, vďaka ktorej môžu heterogénne systémy dosiahnuť interoperabilitu, zvyšuje výkonnosť, zlepšuje možnosti škálovania, a takisto aj modifikovateľnosť systému. REST býva použité v architektúre klient-server. Dôležitým faktom je, že REST architektúra je bezstavová, a teda pri novom dotaze nepotrebuje poznať históriu predchádzajúcich dotazov. To je dosiahnuté najmä vďaka prístupu, ktorý je odlišný od webovej služby typu `SOAP`[8]. Zatiaľ, čo SOAP je orientovaný procedurálne, REST je orientovaný dátovo. z toho vyplýva že operácie vykonáva nad dátami. Dáta sú označované ako zdroje a každý zdroj má svoj špecifický URI identifikátor. Príklad identifikátoru pre zdroj `spravy`:

```
http://priklad.com/spravy.
```

Nad takto pomenovaným zdrojom je možné vykonávať operácie z konečnej množiny dostupných operácií zo štandardu HTTP. V tom sa REST tiež odlišuje od SOAP. Medzi HTTP operácie patria: `GET`, `POST`, `PUT`, `DELETE`, `CONNECT`, `OPTIONS`, `TRACE` a `PATCH`. V drvivej väčšine prípadov sa stretávame len s prvými štyrmi metódami známymi aj pod akronymom `CRUD`.

CRUD

Po správnosti ide skôr o určité paradigma, ktoré je bežné pri vytváraní webových aplikácií či aplikácií, ktoré fungujú obecne nad dátami. Jednotlivé písmena z názvu predstavujú:

- `Create` – vytváranie nových zdrojov
- `Read` – čítanie existujúcich zdrojov, nikdy by ich nemalo meniť

[8] Viac o SOAP na adrese https://www.w3.org/TR/soap/

- `Update` – zmena existujúcich zdrojov
- `Delete` – odstránenie existujúcich zdrojov

Nie je nikde presne napísané, ktorý HTTP príkaz by sa mal použiť pre akú CRUD operáciu. Zaužíval sa však už istý postup, ktorý je vhodné pre vyššiu zrozumiteľnosť používať. Operácia CREATE by mala odpovedať dotazu PUT. Ak zdroj z adresy URI ešte neexistuje, bude vytvorený a ak už existuje, bude nahradený novým zdrojom, ktorý sa nachádza v tele dotazu. POST dotaz zodpovedá operácii UPDATE. Ak daný zdroj neexistuje, očakáva sa nejaká vhodná chybová hláška. DELETE dotaz zodpovedá rovnomennej operácii DELETE a GET operácii READ.

Návratové kódy

Dôležitou časťou REST architektúry sú návratové kódy. Umožňujú používateľom vyznať sa v odpovediach od serveru. V prípade, že ich dotazy boli nesprávne, alebo server pri ich vykonávaní zlyhal, sú pomocou týchto kódov so situáciou oboznámení. Existuje päť základných skupín trojciferných kódov:

- `1XX` – informačné správy
- `2XX` – odpoveď o úspechu
- `3XX` – odpoveď informujúca o tom, že k dokončeniu operácie je potrebné presmerovanie
- `4XX` – chyba na strane klienta (nesprávna požiadavka, napr. 404 – zdroj nenájdený
- `5XX` – interná chyba na strane serveru

6.4.1 RESTful API

V tejto časti bude popísané REST rozhranie webovej služby. Toto rozhranie definuje funkcionalitu, ku ktorej má používateľ prístup prostredníctvom HTTP dotazov. Dotazy je možné rozdeliť do troch skupín: dotazy k obsluhe tém, dotazy k obsluhe front a tretia obecnejšia skupina zahŕňajúca dotazy k správe používateľov a monitorovania.

URL	Metóda	Popis
/topics	GET/PUT	vráti zoznam tém/vytvorí novú tému
/topics/{nazov}	DELETE	odstráni danú tému
/topics/{nazov}/purge	POST	zmaže obsah danej témy
/topics/{nazov}/subscribe	POST	prihlási sa na odber k danej téme
/topics/{nazov}/unsubscribe	POST	odhlási sa z odberu danej témy
/topics/{nazov}/publish	POST	pošle správu do danej témy
/topics/{nazov}/receive	GET	pokúsi sa získať správu z danej témy

Tabuľka 6.1: HTTP dotazy pre témy

Ako je možné vidieť v tabuľke č. 6.1 a tabuľke č. 6.2, témy majú k dispozícii niektoré dotazy navyše oproti frontám. Klient si môže zobraziť zoznam všetkých dostupných tém.

Má možnosť vytvoriť novú tému pomocou PUT dotazu `/topics` s jedným povinným parametrom v tele dotazu a tým je názov témy – `name`. Existujúce témy je možné zmazať

pomocou DELETE dotazu `/topics/{nazov}`, kde je priamo v URL možné vidieť povinný parameter, ktorý je identifikátorom témy, ktorú má používateľ záujem zmazať. Telo DELETE dotazu môže obsahovať voliteľný parameter `forceDelete` s predvolenou hodnotou `false`. Téma, ktorá nemá žiadneho odberateľa, okrem používateľa, ktorý o vymazanie požiada, môže byť bez problémov vymazaná. Avšak pre prípad, že téma má viacero odberateľov, je nutné k jej úspešnému odstráneniu definovať spomenutý parameter na hodnotu `true`. Okrem úplného odstránenia témy existuje možnosť zmazať obsah témy pomocou POST dotazu `/topics/{nazov}/purge`. Môže disponovať jediným parametrom, a to názvom fronty priamo v adrese. Vytvorené fronty je potrebné plniť nejakým obsahom. O to sa stará POST dotaz `/topics/{nazov}/publish`. Okrem povinného názvu fronty v adrese musí obsahovať ešte obsah správy v tele dotazu – `message`.

Ak sú zaslané správy do témy bez jediného prihláseného odberateľa, takéto správy sú odkázané na zabudnutie. Klient sa pred prijímaním správ z fronty musí prihlásiť k odberu POST dotazom `/topics/{nazov}/subscribe`. Analogicky k tomu má možnosť sa z odberu odhlásiť opäť POST dotazom `/topics/{nazov}/unsubscribe`. Po prihlásení k odberu môže prijímať správy z témy GET dotazom `/topics/{nazov}/receive`.

Všetky spomenuté dotazy si vyžadujú autentizovaného používateľa, ktorý má oprávnenie minimálne na úrovni `používateľ - user`.

URL	Metóda	Popis
/queues	GET/PUT	vráti zoznam front/vytvorí novú frontu
/queues/{nazov}/	DELETE	odstráni danú frontu
/queues/{nazov}/purge	POST	zmaže obsah danej fronty
/queues/{nazov}/publish	POST	pošle správu do danej fronty
/queues/{nazov}/receive	GET	pokúsi sa získať správu z danej fronty

Tabuľka 6.2: HTTP dotazy pre fronty

Manipulácia s frontami je podľa tabuľky vyššie o niečo jednoduchšia. Väčšina dotazov je rovnakých s tými pre témy. Samozrejme interne tieto dotazy pracujú nad frontami a tak sa bude aj ich implementácia líšiť (viď. kapitola č. 7). Hlavným rozdielom viditeľným na úrovni HTTP dotazov sú chýbajúce dotazy `subscribe` a `unsubscribe`. Tie chýbajú kvôli tomu, že výsledný systém bude prístupný cez bezstavové REST rozhranie. Ako bolo popísané na začiatku tejto kapitoly, keby klient využíva pripojenie na RabbitMQ server priamo cez `RabbitMQ` klienta mohol by sa k odberu z fronty prihlásiť spolu s viacerými klientmi. Tí by boli obsluhovaní v štýle `round-robin`. Pre potreby REST klienta sa zdá ako najvhodnejší prístup „kto prvý príde, ten prvý berie“. Preto je možné vynechať dva spomenuté dotazy. Aj v tomto prípade je vyžadovaný autentizovaný používateľ aspoň s oprávnením `používateľ - user`.

URL	Metóda	Popis
/users	GET/PUT	zobrazí používateľov/vytvorí nového
/users/{menoUzivatela}/	DELETE/POST	odstráni/upraví daného používateľa
/stats	POST	zobrazí štatistiky prenosu

Tabuľka 6.3: Ostatné HTTP dotazy

Poslednou skupinou sú dotazy pre administráciu a monitorovanie. Jediným, kto môže vykonávať dotazy spojené s vytváraním, mazaním a úpravou existujúcich používateľov je

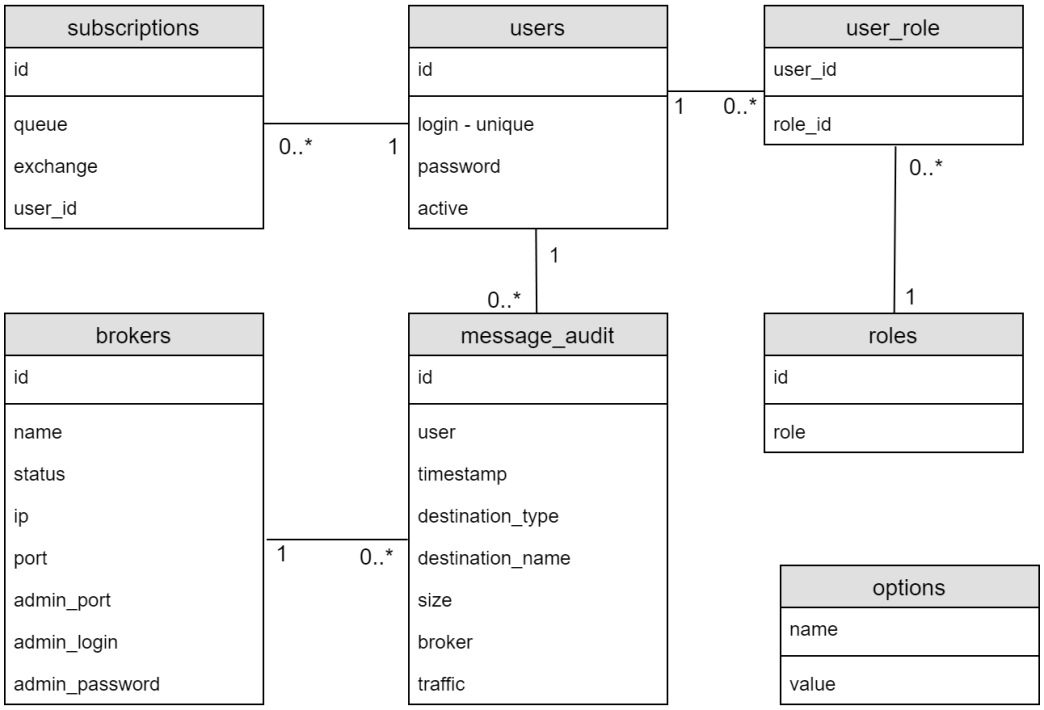
`administrátor - admin`. Administrátor môže zmazať každého používateľa okrem samého seba. To zaručí, že po poslednom vymazaní, ktoré povolí služba zostane k dispozícii aspoň jeden administrátor. Pri vytváraní používateľov sú tri povinné parametre: meno – `username`, heslo – `password` a známa rola – `role`. Pri upravovaní existujúceho používateľa pribudne parameter určujúci, či je používateľ aktívny – `active`. Heslo je v tomto prípade voliteľný parameter.

POST dotaz `/stats` zobrazí používateľovi štatistiku prenosu cez webovú službu. Dokáže teda zistiť objem prenesených dát. Tento filter má možnosť meniť podľa svojich požiadaviek pomocou parametrov v tele dotazu. Dokáže ovplyvniť časové rozpätie pre zisk hodnôt (štandardne sa ráta pre aktuálny deň, kvôli meraniu spotreby), používateľa či to, že štatistika bude obsahovať hodnoty len pre fronty, alebo len pre témy. Na záver môže zvoliť konkrétny názov fronty/témy. Pre lepší prehľad sú možné parametre zobrazené v nasledujúcom zozname:

- user
- from/to – dátum vo formátef `yyyy-MM-dd HH:mm:ss`
- Spring Boot verziu
- destinationType – typ destinácie (`QUEUE/TOPIC`)
- destinationName – unikátny názov destinácie.

6.5 Návrh databázy

Aplikácia bude musieť okrem interného úložiska využívaného nástrojmi disponovať aj ďalším vlastným úložiskom. Keďže v službe budú môcť používatelia vystupovať pod rôznymi rolami, je potrebné týchto užívateľov a role reprezentovať v databáze. K tomu slúži trojica tabuliek `users`, `roles` a `user_role`. Okrem ukladania používateľov je potrebné perzistovať aj informáciu o tom, že nejaký používateľ odberá danú tému. Ako bolo spomenuté vyššie, nejde o klasický odber ako pri použití samotného nástroja ale tento odber je simulovaný na úrovni služby. Odbery sú v tabuľke `subscriptions`. Zoznam všetkých brokerov je pre interné účely ale aj pre účely monitorovania uložený v tabuľke `brokers`. Najväčšia tabuľka čo do počtu riadkov bude v prípade reálneho použitia určite tabuľka obsahujúca logy o odoslaných a prijatých správach (tabuľka `message_audit`). Posledná je tabuľka `options`, ktorá uchováva interné konfigurácie. Diagram vzťahov je znázornený na obrázku č. 6.9.



Obr. 6.9: ER diagram

Kapitola 7

Implementácia

Táto kapitola sa bude zameriavať na konkrétnu problematiku spojenú s implementáciou vyvíjanej aplikácie. V úvode bude predstavená cieľová platforma, pre ktorú je určená. Rovnako budú predstavené tie najzaujímavejšie použité technológie. Ďalej budú ukázané jednotlivé komponenty aplikácie. Bližšie bude predstavená aj implementácia, ktorá sa ukrýva za jednotlivými HTTP dotazmi. Vyzdvihneme niektoré implementované optimalizácie, a takisto úskalia a problémy, ktoré nastali v priebehu implementácie.

7.1 Použité technológie

V čase ranného návrhu aplikácie bola zvažovaná možnosť implementovať výslednú aplikáciu v jazyku C# namiesto jazyka Java. Dôvodom bola lepšia znalosť platformy, väčšie skúsenosti z praxe a v neposlednom rade aktuálne väčšia obľuba jazyka C#. Zvýrazňuje to fakt, že v poslednej dobe bolo predstavených výrazne viac noviniek v jazyku C# ako v jazyku Java.

Napriek osobným preferenciám bola zvolená Java. Aj keď všetky aplikácie, nad ktorými je navrhnovaná výsledná služba ponúkajú klientov pre jazyk C#, administráciu nad nástrojom ActiveMQ je možné najlepšie vykonávať pomocou technológie JMX[1], kvôli ktorej sa hodí zvoliť práve tento jazyk. Rovnako aj Kafka ponúka možnosť administrácie pomocou rozhrania JMX.

Kotlin

Aj keď vyššie bolo napísané, že za implementačný jazyk bola zvolená Java, je to pravda len zčasti. Je to jazyk `Kotlin`, ktorý má k Jave veľmi blízko. Stojí za ním spoločnosť *Jetbrains*, ktorá je autorom série úspešných `IDE`[2] pre celú škálu programovacích jazykov. Medzi IDE patria napr. IntelliJ IDEA, PyCharm, WebStorm, PhpStor, Clion a ďalšie. Tak isto Android Studio je postavené na ich najznámejšom nástroji IntelliJ IDEA. Prvá oficiálna stabilná verzia jazyka `1.0` bola zverejnená 15.2.2016.

Kotlin je staticky typovaný jazyk, ktorý môže bežať na virtuálnom stroji Javy JVM (jedna z možností). Jeho veľkou devízou je fakt, že pri návrhu bol kladený dôraz na spojenie objektovo orientovaného prístupu s tým funkcionálnym. Podobne je na tom aj jazyk `Scala`[3]. Popularita jazyka Kotlin pramení v úsilí jeho tvorcov vytvoriť jazyk, ktorý bude jednoduchý

[1] JMX – Java Management Extensions – technológia umožňujúca správu a monitorovanie aplikácií a častí aplikácií.

[2] IDE – Integrované vývojové prostredie, ktoré spája editor, prekladač, ladiace a mnohé ďalšie nástroje

[3] Viac o jazyku Scala dostupné na adrese https://www.scala-lang.org/

na použitie, prinesie oživenie, ktoré chýbalo Jave a samozrejme nebude tak „ukecaný“ ako Java. Zlomovým momentom pre Kotlin bol okamih, keď spoločnosť *Google* ohlásila, že práve Kotlin sa stane novým oficiálnym jazykom pre vývoj aplikácií pre mobilnú platformu `Android`.

Kotlin nie je určený výhradne len na beh na JVM. Natívne sa prekladá do Java `bytecode` (medzikódu). z dôvodu spätnej kompatibility starších Android zariadení, je plne kompatibilný s verziou `JDK 6`. Okrem virtuálneho stroja môže bežať aj priamo v prehliadači vďaka možnosti kompilácie do jazyku JavaScript (samozrejme bez typických zabudovaných knižníc v JDK). Novinkou je technológia `Kotlin/Native` umožňujúca kompiláciu do natívneho kódu stroja bez nutnosti prítomnosti akéhokoľvek virtuálneho stroja. To umožní zníženie nárokov na hardvér pre beh aplikácií. Aktuálne podporuje tieto platformy: Windows, Linux, MacOS, iOS či Android. Táto technológia je dostupná v neoficiálnej verzii `0.7` (máj 2018) a je stále vo vývoji.

Medzi vlastnosti jazyka, ktoré stoja za zmienku patria napríklad lambda výrazy a `inline` funkcie, rozšírenia pre triedy podobné (`extensions`) v jazyku C#, chytré pretypovanie, používanie šablón v reťazcoch a mnohé ďalšie[4].[10] Jednou z noviniek, ktoré stoja za spomenutie je to, akým spôsobom pristupuje k `null` bezpečnosti. Aby bolo predídené nepríjemným pádom aplikácií z dôvodu `NullPointerException` zvolili prístup, kedy každý objekt štandardne nemôže obsahovať hodnotu `null`. Nasledujúca ukážka skončí chybou pri preklade:

```
var premenna: String
premenna = null // Kompilacna chyba.
```

Ak chceme aby objekt mohol obsahovať `null`, musíme to explicitne špecifikovať pomocou znaku „?“, ktorý nasleduje hneď za typom. Avšak ak už je definovaný typ, ktorý akceptuje hodnotu `null`, kompilátor pri preklade vyžaduje, aby bola takáto hodnota testovaná na to, či `null` neobsahuje:

```
var premenna: String? = "nejaka hodnota"
println(premenna.length(premenna)) // Kompilacna chyba.
```

Toto umožní eliminovať riziko, že aplikácie v produkcii zlyhajú kvôli okrajovým situáciám, ktoré sa nepodarilo pokryť pri implementácii či odchytiť pri testovaní.

Spring Framework

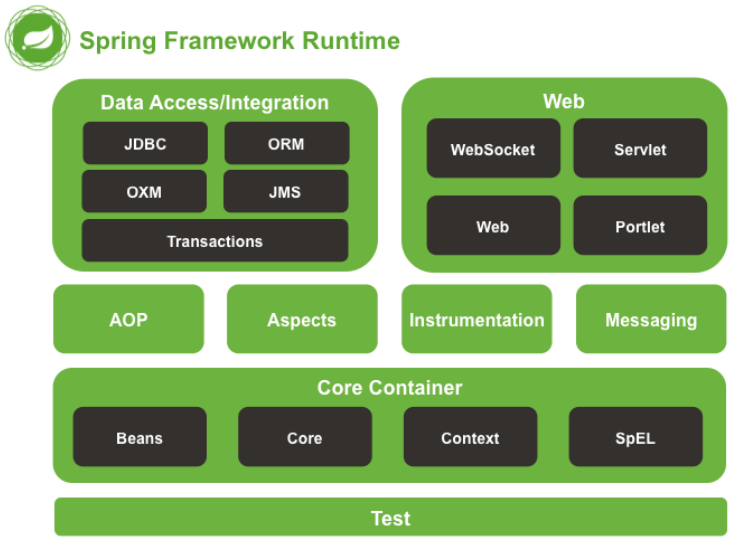
Spring Framework poskytuje infraštrukúru, ktorá pomáha vo vývoji Java aplikácií. Spring sa stará o infraštruktúru a tak sa programátor môže viac zamerať na samotnú funkcionalitu. Je rozdelený do jednotlivých modulov, ktoré sú logicky zoskupené do viacerých skupín, ako je možné vidieť na obrázku č. 7.1. Programátor má možnosť využiť len moduly, o ktoré má záujem a tak výsledná zabalená aplikácia nie je zbytočne veľká po nezahrnutí redundantných balíčkov.

Aj keď je každá Java knižnica plne kompatibilná s jazykom Kotlin, predsa len Kotlin obsahuje niečo navyše. Aby spomínaná `null-safety` plne využila svoj potenciál, potrebuje pri obyčajných Java triedach, ktoré ju samozrejme nepodporujú, aby boli označené príslušnými anotáciami. Naprieč rôznymi knižnicami sú použité rôzne anotácie a preto sa Kotlin snaží podporovať viacero týchto anotácií. Príkladom sú napríklad anotácie `@Nullable` a `@NotNull` z balíčku `org.jetbrains.annotations`. Spring bol o tieto anotácie doplnený a preto je jeho používanie s jazykom Kotlin dostatočne pohodlné.

[4]Zoznam zaujímavých vlastností jazyka Kotlin na adrese https://kotlinlang.org/docs/reference/comparison-to-java.html

IoC a DI

Vítanou vlastnosťou je zabudovaná podpora `IoC` kontajnerov. Veľa ľudí si `IoC`[5] princíp mýli s `dependency-injecton`[6] princípom. IoC je viac abstraktný a popisuje celkový princíp vzťahu medzi triedami vyššej vrstvy a triedami nižšej vrstvy. Hovorí, že triedy vyššej vrstvy by nemali závisieť na triedach nižšej úrovne abstrakcie, ale skôr naopak. Teda všetky závislosti by mali viesť jedným smerom a to od konkrétnej triedy k abstraktnej triede. To umožňuje znížiť počet závislostí a zvýšiť celkovú znovupoužiteľnosť jednotlivých častí systému. Konkrétna implementácia sa môže často meniť, ale tým, že je dodržaný IoC princíp, stačí zmeniť implementáciu konkrétnej triedy bez ovplyvnenia nadradenej abstraktnej triedy. IoC je úzko späté s jedným z piatich `SOLID`[7] pricípov, konkrétne princípu obrátenia závislostí.



Obr. 7.1: Spring architektúra

`Dependency Injection` je konkrétna technika, kde je použitý princíp IoC. Existujú dva hlavné princípy DI. Prvý je konštruktorový DI, kde závislosti su podané konkrétnej triede v dobe vytvárania inštancie tejto triedy. Toto je často automatizované na úrovni DI frameworku, ktorý sám vytvára tieto inštancie. Druhým je `Setter injection`, kde je závislosť vsunutá až po vytvorení konkrétneho objektu pomocou `setter` metódy. V projekte je využitá prvá menovaná.

Spring Boot

Spring Boot je postavený na Spring frameworku. Je to nástroj, ktorý dovolí vyhnúť sa zdĺhavej konfigurácii nového projektu a všetkých jeho súčastí. Pomocou intuitívneho nástroja `Spring Initializr`[8] umožní v priebehu pár minút vytvoriť projekt, ktorý je možné priamo spustiť. Dovolí vybrať z ponúkaných nastavení:

[5] IoC by sa dalo preložiť ako technika obráteného riadenia.
[6] DI – vkladanie závislostí
[7] viac o SOLID princípoch na adrese http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod
[8] Online nástroj dostupný na adrese https://start.spring.io/

- typ projektu – Maven alebo Gradle
- programovací jazyk – Java, Kotlin alebo Groovy
- Spring Boot verziu
- `Group` a `Artefact`, ktoré definujú `namespace` vášho projektu
- rýchle pridávanie balíčkov pomocou intuitívneho vyhľadávača.

Gradle

Tento projekt používa práve Gradle ako svoj nástroj pre automatizáciu zostavovania programu. Je to `OpenSource` nástroj, ktorý namiesto XML konfiguračného súboru používa konfiguráciu pomocou jazyka založeného na jazyku `Groovy`. Okrem iného umožňuje jednoduchú správu použitých balíčkov. Programátor len špecifikuje názov balíčka, ktorý je dostupný v zvolenom online repozitári a Gradle sa sám postará o jeho stiahnutie. Umožňuje vytvárať rôzne úlohy ako napríklad jednoduché vytvorenie spustiteľného archívu `.war` či `.jar`.

MySQL a Spring Data JPA

Výsledná aplikácia si nevystačí len s interným úložiskom dát, ktorým disponuje každý z nástrojov. Okrem interných informácií potrebných k výmene správ pomocou samotných nástrojov je potrebné ukladať aj ďalšie dáta požadované k chodu služby. Ako perzistentné úložisko dát bol zvolený relačný databázový systém MySQL.

JPA je skratka pre Java Persistence Api (Java Api pre perzistenciu). Nie je to žiadny hotový nástroj, ale len špecifikácia pre prístup, perzistovanie a prevod dát medzi relačnou databázou a reprezentujúcim objektom v Jave (v Kotline). JPA definuje aj vlastný jazyk pre písanie dotazov nad databázou `JPQL`.

`Spring Data JPA` je teda konkrétna implementácia tejto špecifikácie. Je súčasťou Spring frameworku. Odstraňuje veľké množstvo zbytočného kódu pri implementácii vrstvy pristupujúcej k dátam. Okrem toho aj minimalizuje množstvo nutných úkonov pre prístup k dátam, a tým znižuje šancu, že programátor urobí chybu. Pre použitie stačí použiť anotáciu `@EnableJpaRepositories` v konfiguračnom Java súbore.

Samotné dotazy na databázu sa potom tvoria jednoducho. Základné dotazy ako `findAll` či `findAllById` sú definované už v základe. Programátor musí vytvoriť len jednu triedu pre entitu t.j. model, ktorý odpovedá tabuľke a potom jedno rozhranie, ktoré je potom interne vo frameworku použité pre načítanie dát. Zaujímavosťou je, že vo väčšine prípadov nemusí písať žiadne dotazy v nijakej verzii jazyka SQL. Stačí ak do v rozhraní vytvorí metódy, ktoré vhodne pomenuje. Tu je ukážka takých metód:

- `findByExchange(exchange: String): List<Subscription>`
- `findByExchangeAndQueue(exchange: String, queue: String): List<Subscription>`
- `findByExchangeAndQueueAndUser(exchange: String, queue: String, user: User): Subscription?`
- `findAllByTimestampGreaterThanEqualAndTimestampLessThanEqual(fromDate: Date, toDate: Date): List<MessageAudit>`

Okrem inteligentných dotazov vygenerovaných z názvov metód je možné písať dotazy aj ručne v tele anotácie `@Query()` priamo nad metódou v rozhraní. Tu je ukážka takej metódy spolu s dotazom v anotácii:

```
@Query(value ="SELECT Sum(size) FROM message_audit WHERE (timestamp
       BETWEEN ?1 AND ?2) AND traffic = ?3", nativeQuery = true)
fun getData(fromDate: Date, toDate: Date, traffic: String): Int?
```

Podľa toho či je podľa návratového typu očakávaná kolekcia, alebo len jediný objekt framework rozhodne, či vráti prvý vyhovujúci záznam, alebo všetky záznamy v kolekcii.

7.2 Implementačné detaily

V tejto časti budú popísaná konkrétna implementácia dôležitých súčastí systému. Postupne bude predstavená implementácia výmeny správ, správa brokerov, logovanie, autentifikácia a autorizácia, a na záver správa používateľov.

7.2.1 Výmena správ

Výmena správ a spravovanie jednotlivých modelov pre komunikáciu je jadrom aplikácie. Hlavným prístupovým bodom pre všetky dotazy nad správami, frontami či témami je trieda `MessagingService` v balíčku `com.messagingservice.service`. Trieda je označená anotáciou `@RestController` čím je zaregistrovaná ako komponenta spravovaná Spring frameworkom. Vďaka tomu môže Spring využiť svoju vlastnosť a to automatické vytváranie takýchto komponent s využitím DI (`dependency-injection`). Okrem stereotypu `@RestController` (alebo `@Controller`) rozoznáva aj ďalšie stereotypy: `@Component`, `@Service` a `@Repository`.

Konštruktor tejto triedy očakáva viacero parametrov, ktorých inštancie sú automaticky vytvorené Spring frameworkom. Dnes už nie je potrebné označovať premenné pomocou anotácie `@Autowired` a stačí len v konštruktore použiť triedy označené jedným z vyššie spomenutých stereotypov. Aby však dokázal nájsť vhodných adeptov pre DI, je v hlavnej triede (v triede s funkciou `main`) pridaná anotácia `@SpringBootApplication` s definovaným zoznamom balíčkov, ktoré má pri inicializácii prehľadávať.

Asi najdôležitejším parametrom v konštruktore je `mqProvider`, ktorý očakáva objekt implementujúci rozhranie `IMqProvider`. Práve toto rozhranie je predpisom pre implementácie jednotlivých nástrojov. Teoreticky stačí implementovať jedinou triedou toto rozhranie a program bude použiteľný s ľubovoľným nástrojom. Pre prípad nástroja RabbitMQ ho implementuje trieda `RabbitMqProvider` v balíčku `com.messagingservice.mq.rabbitmq`.

Väčšinu základnej manipulácie so správami je možné vykonať pomocou RabbitMQ klienta pre jazyk Java. V niektorých prípadoch je však využitý `management-plugin`. Je to zásuvný modul, ktorý beží priamo na brokerovi. Ten ponúka možnosť vykonávať ďalšie úpravy pomocu HTTP Api. Tento zásuvný modul teda musí bežať na každom brokerovi. Aby trieda `RabbitMqProvider` nebola príliš veľká, bola snaha z nej čo najviac funkcionality presunúť do samostatných tried. Obsluha zásuvného modulu pre správu je osamostatnená v triede `AdminConnector` v balíčku `com.messagingservice.mq.rabbitmq.admin`. Implementácia tejto triedy je samozrejme špecifická pre tento nástroj. Kafka ani ActiveMQ rovnakým zásuvným modulom nedisponujú, no dá sa povedať, že by obsahovali podobnú triedu, ktorá by podobné úkony vykonávala nad JMX rozhraním.

7.2.2 Správa brokerov

Používateľ si môže nakonfigurovať službu, kde bude mať k dispozícii viac ako jedného brokera. Väčšie množstvo brokerov vyžaduje implementáciu mechanizmu na ich správu. Po spustení aplikácie sa vytvorí pre každého aktívneho brokera inštancia triedy `RabbitBrokerConnection`. Tá v sebe zabaľuje kanál využívaný v RabbitMQ klientovi a zároveň aj inštanciu triedy `AdminConnector` pre pripojenie k zásuvnému modulu pre správu brokera. Pri vytváraní inštancie tejto triedy sa otvára kanál medzi službou a brokerom.

Pre rovnomerné vytaženie dostupných brokerov sa pri osbluhovaní požiadaviek brokeri striedajú. Namiesto implementovania obsluhy štýlom `round-robin` bolo zvolený náhodný výber. z dlhodobého hľadiska pri použití rovnomerného rozloženia pravdepodobnosti je výsledok takmer rovnaký. Rozhranie predpisuje aj implementáciu metódy `setNextAvailableBroker`. Tá je volaná z triedy `MessagingService` pred každou obsluhou konkrétneho dotazu. Je na konkrétnej implementácii pre daný nástroj, čo bude v tele tejto metódy vykonané.

Nastaveniu ďalšieho dostupného brokera prechádzajú dve veci: obnovenie nečinných brokerov a kontrola aktuálneho stavu a počtu brokerov. Očakáva sa veľké množstvo dotazov od klienta. Berieme preto do úvahy fakt, že klient bude generovať aj niekoľko dotazov za sekundu je zbytočné vykonávať tieto kontroly pri každom dotaze. Sprostredkovateľ služby má možnosť ovplyvniť dobu medzi jednotlivými kontrolami pomocou záznamu v tabuľke options:

- `dead_brokers_recovery_interval_ms` – interval pre kontrolu neaktívnych brokerov
- `sufficient_brokers_number_test_interval_ms` – interval pre kontrolu dostatočného množstva aktívnych brokerov.

Broker mohol byť v statuse `OFFLINE` z rozličných príčin. Na to aby sa broker dostal opäť do statusu `ONLINE`, je potrebné aby poverená osoba na strane sprostredkovateľa služby vykonala potrebné kroky k náprave chyby a nastavila status brokera na `READY`. V aplikácii je zabudovaný mechanizmus, ktorý po uplynutej dobe týchto brokerov oživuje a zaradí ich opäť medzi zoznam aktívnych brokerov. Okrem toho im nastaví status na `Online`.

Vždy po uplynutí druhého z intervalov a po zaslaní ľubovoľného nového dotazu sa vykoná aj kontrola dostupného počtu aktívnych brokerov. Ak je počet nižší ako garantuje služba, sú vygenerované varovania do interných logov a sprostredkovateľ je povinný túto situáciu začať riešiť.

7.2.3 Obmedzovanie služby

Ako už bolo spomenuté v návrhu, zákazník bude mať možnosť konfigurovať svoju služby na základe troch vlastností: počet brokerov, dĺžka doby expirácie správy a množstvo prenesených dát. Tieto vlastnosti sa dajú nastaviť opäť pomocou príslušného záznamu v tabuľke `options`:

- `replication_level` – stupeň replikácie (počet brokerov)
- `retention_hours` – doba expirácie v hodinách
- `data_download_limit_mb` a `data_upload_limit_mb`

Ak počet uzlov klastra klesne pod hodnotu definovanú prvou z menovaných nastavení, sú generované upozornenia pre prevádzkovateľa služby. Doba expirácie je nastavená na úrovni správ. Najviac pozornosti si zaslúži implementácia denného obmedzenia prenesených dát. Aby nebolo nutné po každom odoslaní/prijatí správy spúšťať dotaz nad databázou je kontrola s databázou vykonávaná opäť až po uplynutí doby definovanej v tabuľke `options` (`data_transferred_refresh_interval_ms`). To samozrejme úplne nezabráni zákazníkovi prekročiť limit prenesených dát, no prevádzkovateľ si môže túto dobu nastaviť podľa svojich preferencií. Skracovanie intervalu bude mať za následok zvyšovanie odozvy systému a predlžovanie naopak jemne zvýšené nároky na úložisko v prípade prekročenia limitu. Základný interval je nastavený na jednu minútu.

7.2.4 Logovanie

Logovanie v aplikácii sa dá rozdeliť na dva typy. Prvým je interné logovanie udalostí, chýb a varovaní potrebných k rýchlej identifikácii prípadných problémových situácií. Druhým je logovanie za účelom zaznamenať históriu o používateľmi prenesených dátach.

V druhom prípade sú logy uložené v databázi v tabuľke `messge_audit`. Záznamy sa vytvárajú pri odosielaní a pri prijímaní správ. Aby nemalo logovanie tak výrazný dopad na výkon a záznamy sa neukladali pri každej odoslanej/prijatej správe bol implementovaný mechanizmus, ktorý tomu umožní predísť. Na implementáciu je interne využitá fronta. Po vytvorení záznamu sa záznam odošle do fronty. Táto fronta nie je replikovaná, stačí ak bude na jednom brokerovi. Rozhranie `IMqProvider` obsahuje dve metódy `registerMessageLogQueue` a `registerMessageAuditLogListener`, ktoré vyžadujú aby bola implementovaná funkcionalita zaregistrovania tejto fronty a rovnako zaregistrovanie konzumenta tejto fronty. Tento odoberateľ je reprezentovaný triedou `RabbitMessageAuditListener`. Interne už vytvára plnohodnotný odber danej fronty a nie tak ako je implementovaný v prípade front a tém sprístupneným cez REST rozhranie. Pri vytvorení a zaregistrovaní obdrží inštanciu triedy `MessageAuditLogger`, ktorá je zaregistrovaná ako Spring komponenta. Práve `MessageAuditLogger` v sebe obsahuje tú časť mechanizmu, ktorá zabezpečí, že logy nemusia byť ukladané po jednom. Obsahuje kolekciu logov, ktorá je po naplnení sprostredkovateľom definovaným počtom záznamov (`message_audit_batch_size`) naraz uložená do databázy. Zaregistrovaný poslucháč fronty je zaregistrovaný na jedného z dostupných brokerov. Aplikácia počíta aj s výpadkom tohoto brokera. Je zaregistrovaný poslucháč typu `messageAuditConnectionClosedListener`, ktorým je trieda `RabbitMqProvider` a pri vyvolaní udalosti `messageAuditConnectionClosed()` vytvorí spojenie s novým brokerom a tak odber logov z fronty môže pokračovať ďalej.

Na interné logy je využitá knižnica `log4j` vo verzii `2.1.1`. Tá umožňuje zaregistrovať viacero koncových bodov, ktoré smerujú logy podľa svojho zamerania. Môže to byť konzola, súbor či napríklad databáza. Každému koncovému bodu je možné nastaviť, ktoré z logov sa doňho pošlú (informačné, varovania či chyby). Informačné logy budú vypísané len do konzoly z dôvodu šetrenia výkonu kvôli príliš častému zapisovaniu do súboru či databázy.

7.2.5 Autentifikácia a autorizácia

Na prihlásenie používateľov bola zvolená základná metóda `Basic Access Authentication` (jednoduché overenie prístupu). Jej bezpečnosť nie je veľmi vysoká no pre účely tejto práce sa javí toto riešenie ako postačujúce. Používateľ teda pripája svoje meno a heslo k dotazu. Autentizácia je implementovaná pomocou dvoch tried v balíčku `com.messaging-service.auth`. Prvou je `SecurityConfiguration`, ktorá obsahuje samotnú konfiguráciu

zabezpečenia. Stanovuje, ktoré dotazy vyžadujú autentizáciu a nastavuje typ autentizácie pomocou zvoleného vstupného bodu (entry point). Vstupný bod `MqBasicAuthEntryPoint` je druhou spomenutou triedou. Definuje, aký typ autentifikácie bude pre služby zvolený.

Tento typ autentizácie by bol v prípade ďalšieho vývoja určite zmenený za nejakú bezpečnejšiu alternatívu, napr. `OAuth 2.0`[9].

Stupeň autorizácie potrebný na vykonanie jednotlivých dotazov je definovaný priamo pri nich v triede `MessagingService` pomocou anotácie `@PreAuthorize()`. V tele anotácie sú vymenované role, ktoré majú k tomuto prístupovému bodu prístup. Ak majú práva obe role `user` aj `admin` anotácia bude vyzerať takto:

```
@PreAuthorize("hasAnyRole('admin', 'user')").
```

Samotný `management-plugin` logicky ponúka možnosť povoliť prístup len autentifikovaným používateľom. Prihlasovacie meno a heslo je uložené v databázi v tabuľke `brokers`. Heslo samozrejme nie je uložené ako `plain-text`, ale je zašifrovaný symetrickou šifrou AES. Použitá je metóda je `CBC` – `Cipher Block Chaining`. Funguje na princípe kedy sa na každý šifrovaný blok textu pred zašifrovaním aplikuje `XOR` s predchádzajúcou zašifrovanou časťou a až potom je znovu zašifrovaný kľúčom. Absenciu zašifrovaného bloku textu pri šifrovaní prvého bloku rieši IV (inicializačný vektor), ktorý sa použije pri operácii XOR namiesto zašifrovaného textu. Na šifrovanie je použitý 256 bitový kľúč.

7.2.6 Správa používateľov

Správu používateľov môže vykonávať len administrátor. Jej implementácia sa nachádza v triede `MessagingService`. Administrátor môže vymazať ľubovoľného používateľa (aj administrátora) s výnimkou samého seba. To zabezpečí, že v systéme zostane za každých okolností minimálne jeden administrátor.

7.3 Ďalšie zaujímavosti

Správa nastavení

Aplikácia poskytuje viacero možností svojej konfigurácie tak, aby čo najlepšie vyhovovala spôsobu, akým ju jej prevádzkovateľ chce používať. Prvou možnosťou je zmena nastavení v databázi v tabuľke `options`, ktorá už bola spomenutá vyššie v tejto kapitole. Spring framework má tak isto niektoré nastavenia, ktoré môže používateľ zmeniť v prípade záujmu. Medzi základné z nich využívané aj v tejto aplikácii patrí napríklad nastavenie zdroju dát, ktorým je `MySQL` databáza spolu s prístupovým menom a heslom. Používateľ ďalej môže nastaviť čístlo portu, na ktorom bude bežať táto služba. Kompletný zoznam bežných nastavení je možné vidieť na tomto odkaze[10].

Spomenuté nastavenia sú priamo súčasťou projektu a v tomto prípade sa nachádzajú v adresári `resources`. Tento súbor je však pri nasadení aplikácie do obehu spolu so zvyškom preložených zdrojových súborov zabalený do spustiteľnej knižnice. To by znamenalo, že pri zmene nastavení by bolo znovu nutné zo zdrojových súborov vytvoriť spustiteľný archív. Našťastie existuje možnosť tieto nastavenia zmeniť aj pri hotovom balíčku. To je možné či už pomocou argumentov pri spustení programu, alebo pomocou externého súboru `application.properties`. Ten môže byť priložený k archívu napríklad v adresári `config`.

[9] Viac informácií o OAuth na adrese https://oauth.net/2/

[10] https://docs.spring.io/spring-boot/docs/current/reference/html/common-application-properties.html

V prípade, že tento súbor existuje, tak nastavenia v ňom definované majú prednosť pred tými špecifikovanými v rovnomennom súbore, ktorý je súčasťou archívu.

Okrem nastavení pre Spring framework aplikácia očakáva aj ďalšie nastavenia, ktoré ovplyvňujú jej chod. Získava odtiaľ napríklad kľúč použitý pri šifrovaní algoritmom AES. Aby bolo možné jednoducho zmeniť nástroj, ktorý je interne použitý pre výmenu správ, pomocou nastavenia hodnoty `messageQueueProvider` je možné zmeniť tento nástroj. V rámci tejto práce bol využitý nástroj RabbitMQ a teda štandardne má toto nastavenie hodnotu `rabbit`.

Určenie použitého nástroja

V predchádzajúcej časti bolo spomenuté, že je možné ovplyvniť výber nástroja zmenou nastavenia v externom konfiguračnom súbore. Skôr v tejto kapitole bola reč o `dependency-injection` a o tom, že je využitá schopnosť Spring frameworku automaticky vytvárať tieto závislosti. Toto vytváranie závislostí funguje tak, že framework na predom špecifikovaných miestach v adresárovej štruktúre projektu hľadá triedu, ktorá mu vyhovuje. Obyčajne by takáto trieda mala byť len jedna, aby ju vedel presne určiť.

V tomto prípade framework hľadá triedu ktorá implementuje rozhranie IMqProvider. Takéto triedy sú však v projekte dve a po implementácii ďalších nástrojov ich môže byť kľudne viac. Prvou je samozrejme `RabbitMqProvider` a druhou triedou, ktorá je v projekte len z dôvodu lepšej ukážky, je trieda `DummyMqProvider`.

Tento problém je možné vyriešiť vďaka tomu, že disponuje možnosťou podmieneného vytvárania závislostí. Spring umožňuje viacero spôsobov ako to dosiahnuť. V tejto aplikácii je výber triedy rozhodnutý podľa hodnoty nastavenia z konfiguračného súboru. Triedu, ktorej inštancia má byť vytvorená len za určitých podmienok (hodnoty v konfiguračnom súbore) je potrebné označiť anotáciou napríklad takto:

```
@ConditionalOnProperty(
        name = ["messageQueueProvider"],
        havingValue = "rabbit"
).
```

Parametrami je názov nastavenia, ktoré by malo existovať v konfiguračnom súbore a hodnota pri ktorej bude vytvorená práve táto trieda.

Kapitola 8

Testovanie

Táto kapitola sa sústredí na testovanie celej aplikácie. V prvej časti bude zmapovaná základná funkcionalita v niekoľkých testovacích prípadoch. Bude popísané, čo sa v jednotlivých prípadoch testuje a ako by mali vyzerať očakávané výsledky. V druhej časti bude predstavené ako si aplikácia vedie pri rôznych konfiguráciách a pri reálnejších príkladoch nasadenia.

Na testovanie bol použitý OpenSource nástroj *Apache JMeter*[1](použitý vo verzi 4.0). Umožňuje jednoducho simulovať rôzne úrovne záťaže a sledovať pri tom výkon aplikácie. Napríklad je možné simulovať viacerých používateľov súčasne a na základe toho sledovať, aký vplyv to malo na výkon celej aplikácie. Nástroj sa zameriava hlavne na testovanie webových aplikácií, SOAP či REST webových služieb, databáz cez JDBC konektor a ďalších.

Program disponuje IDE, ktoré zjednodušuje testovanie. Podporuje zásuvné moduly a väčšina funkcionality dostupná po nainštalovaní aplikácie je implementovaná formou zásuvných modulov.

Prvá fáza testovania by sa dala pomenovať aj ako prezentácia toho, ako aplikácia funguje. Pozostáva s niekoľkých sád dotazov. Tieto sady sú jednotlivo popísané tak, aby bolo jednoducho po ich spustení zistiť, či program pracuje správne, alebo nie. Popísanú sadu testov je možné nájsť v prílohe B.

V prípade testovania zaťaženia aplikácie pri najčastejšie vyskytujúcich sa scenároch sú najzaujímavejšie práve štyri dotazy. Ide o odosielanie a príjem správ z/do front/tém. Pre tieto prípady je pripravený jednoduchý test, kedy sa pri rôznych konfiguráciách vyskúša rýchlosť odozvy aplikácie pre rozdielne množstvo používateľov.

Pre všetky štyri spomenuté prípady boli vykonané nasledujúce testy:

- 1 broker; 1 používateľ, 1000 odoslaných aj prijatých správ
- 1 broker; 5 používateľov, každý 200 odoslaných aj prijatých správ
- 1 broker; 10 používateľov, každý 100 odoslaných aj prijatých správ
- 1 broker; 20 používateľov, každý 50 odoslaných aj prijatých správ.

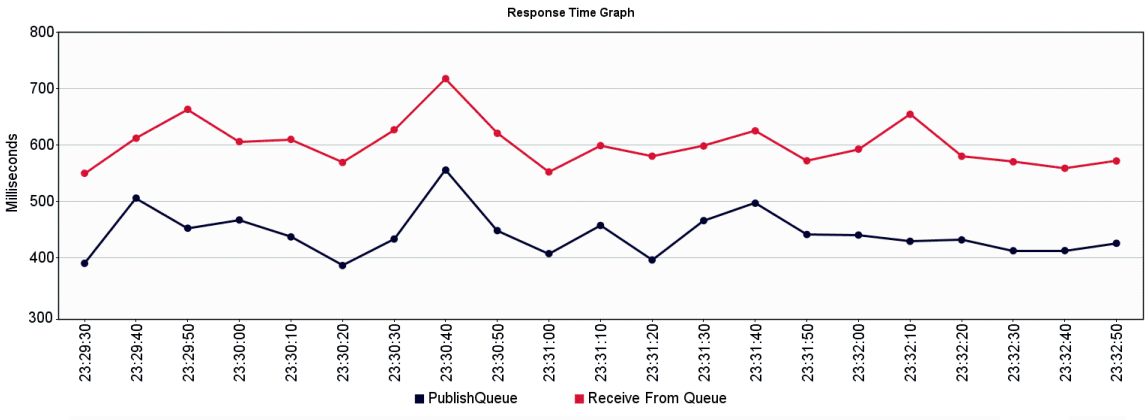
Testovanie bolo vykonávané na počítači s nasledujúcou konfiguráciou:

- `Procesor:` Intel Core i5-6200U CPU @ 2.30 GHz (2 fyzické jadrá/4 logické jadrá)
- `RAM:` 8192 MB DDR3

[1]Viac informácií dostupných na adrese https://jmeter.apache.org/

- `Disk:` Samsung SSD 850 EVO 500GB.

Rovnaké prípady boli zopakované aj pri konfigurácii so stupňom replikácie dva a tri čo znamená, že klaster sa skladá z dvoch, prípadne troch brokerov, na ktorých sú správy duplikované. Na obrázku č. 8.1 je zobrazené, ako na odozvu systému vplývajú spomenuté faktory. Klaster a klient bežali na jednom a tom istom zariadení. Počet rôznych používateľov bol simulovaný pomocou programu `JMeter`, ktorý tak jednotlivé dotazy simultánne vykonáva na takom počte vláken, ktoré odpovedá počtu používateľov pre daný testovaný prípad.



Obr. 8.1: 3 brokeri; 5 používateľov; každý 200 odoslaných aj prijatých správ

Na obrázku je vidieť len jeden z celkovo šestnástich testovaných prípadov. Ostatné prípady je možné nájsť v prílohe B.

8.1 Zhodnotenie výsledkov

Ako je možné vidieť na obrázkoch s výsledkami testov či už priamo v texte, alebo v prílohe A, odozva aplikácie a rýchlosť závisí na počte brokerov. V nemalej miere však záleží aj na počte používateľov pripojených k systému. Treba podotknúť, že pri testovaní bol klaster vytvorený len na jedinom počítači. V reálnom použití bude tento klaster vytvorený z viacerých počítačov ideálne na jednej sieti. Rovnako väčšie množstvo prihlásených používateľov bolo simulované nástrojom `JMeter` pomocou viacerých vlákien. To má tak isto vplyv na výkonnosť aplikácie, ktorá beží na rovnakom počítači.

Pri jednom jedinom brokerovi je možné vidieť, že napríklad dotaz pre odoslanie správy do fronty je zhruba dva-krát rýchlejší, ako dotaz pre čítanie z témy. z hľadiska úkonov na strane aplikácie sú tieto operácie podobné. V oboch prípadoch sa testuje, či daná téma existuje, vytvára sa záznam v audit logu o prenesených dátach. Dôvodom je zložitejšia operácia čítania zo samotnej fronty, keďže to nie je implementované pomocou trvácneho odberateľa, ktoré je typické pri použití klastra RabbitMQ brokerov bez použitia REST služby.

Je možné vidieť, že s narastajúcim počtom používateľov a brokerov sa doba čakania na odpoveď predlžuje. Pri jednom brokerovi a jednom používateľovi trval dotaz na odoslanie správy do fronty priemerne 110 ms. Pri dvadsiatich používateľoch to bolo už 750 ms. Pri dvadsiatich používateľoch a troch brokeroch to testovanej konfigurácii trvalo v priemere až 1.5 sekundy.

Po tomto testovaní bolo vykonané ešte jedno testovanie. Testovaná zostava bola vymenená za výkonnejšiu a to za účelom zistenia, aký vplyv to bude mať na výsledky testov. Druhá zostava je drahšia a zároveň výkonnejšia:

- `Procesor:` Intel Core i7-6820HQ CPU @ 2.70 GHz (4 fyzické jadrá/8 logických jadier)
- `RAM:` 16384 MB DDR4
- `Disk:` Samsung SSD 512 GB M2.

Výsledky získané za použitia druhej zostavy boli oveľa lichotivejšie (viď tabuľka 8.1). Je možné vidieť, že napríklad pri použití jedného brokera dosahuje výkonnejšia zostava pri 40 používateľoch stále lepšie výsledky ako slabšia zostava pri 20. Dokonca v tabuľke je aj prípad, ktorý by sa dal označiť za nie až taký častý v prípade menších aplikácií, a to so 100 používateľmi. Odozva je tam už síce vyššia, no stále použiteľná. Dá sa ešte očakávať, že server, na ktorom bude táto aplikácia bežať, bude ešte výkonnejší. Okrajovým prípadom to bolo označené preto, že je očakávané skôr veľké množstvo spracovávaných správ ako veľké množstvo aktívnych používateľov. Zároveň takéto využitie nie je možné vylúčiť.

Je na prevádzkovateľovi ako sa mu podarí optimalizovať jeho klaster. Má na to viacero možností. Môže znížiť počet zásahov do databázy zvýšením veľkosti dávky logov, ktoré sa zapisujú do databázy. Zároveň existujú ďalšie spôsoby optimalizácie výkonu takéhoto systému. Príkladom môže byť experimentovanie s veľkosťou `pool`[2] JDBC pripojení. Príliš vysoké množstvo môže zvýšiť odozvu systému. Príliš nízke množstvo nemusí stačiť pre potreby aplikácie. Konkrétne nastavenie závisí priamo na strojoch, na ktorých služba beží.

Testovaná konfigurácia	i5 6200U	i7 6820HQ
1 broker; 5 používateľov	170 ms	100 ms
1 broker; 20 používateľov	725 ms	315 ms
1 broker; 40 používateľov	–	635 ms
1 broker; 100 používateľov	–	1400 ms
2 brokeri; 5 používateľov	380 ms	240 ms
2 brokeri; 20 používateľov	1400 ms	800 ms
3 brokeri; 5 používateľov	420 ms	280 ms
3 brokeri; 20 používateľov	1500 ms	900 ms
3 brokeri; 40 používateľov	–	1800 ms

Tabuľka 8.1: Porovnanie výkonu oboch zostáv

[2]Je to skupina dostupných voľných pripojení, ktoré sú pre aplikáciu k dispozícii.

Kapitola 9

Záver

Úvodná kapitola sa zameriava na všeobecné predstavenie problematiky výmeny správ v distribuovaných systémoch. Boli predstavené modely komunikácie ako komunikácia pomocou výmeny súborov, synchrónne zasielanie jednoduchých správ medzi komponentami, vzdialené volanie procedúr RPC či zdieľané úložisko dát, ktoré predchádzali zavedeniu výmeny správ pomocou front. Porovnali sme výhody a nevýhody, aké so sebou prináša takáto forma asynchrónnej komunikácie.

V nasledujúcich troch kapitolách boli postupne predstavené nástroje ActiveMQ, RabbitMQ a Apache Kafka. Dôraz bol kladený na priblíženie princípu, akým tieto nástroje pristupujú k výmene správ. Porovnali sme rozdiely a spoločné vlastnosti jedonotlivých modelov výmeny správ v každom z nástrojov. Následne sme priblížili niektoré kľúčové vlastnosti a možnosti škálovania daných nástrojov.

V ďalšej kapitole je po teoretickom úvode predstavený cieľ tejto práce, ktorým je vytvorenie služby poskytujúcej možnosť a správu výmeny správ v technológii cloud computing. Je priblížený princíp poskytovania softvéru formou služby. Predstavili sme niektoré existujúce riešenia spolu s vyzdvihnutím ich dobrých vlastností a s poukázaním na vlastnosti, ktoré narozdiel od vyvíjanej aplikácie chýbajú. Bol predstavený prvotný návrh, ktorý nekládol dôraz na možnosti škálovateľnosti služby. Finálny návrh bol prezentovaný s vyzdvihnutím rozdielov oproti prvotnému návrhu. Spolu s diagramom prípadov použitia sú prezentované konkrétne požiadavky na systém. Jednotlivo je navrhnutý spôsob realizácie vybranej množiny vlastností, ktoré sme označili za implementovateľné pre všetky tri nástroje. Spolu s návrhom databázy je ukázané REST rozhranie služby.

Kapitola sedem sa venuje implementácii aplikácie pre zvolený nástroj – RabbitMQ. Veľkou devízou tejto práce je voľba mladého a populárneho programovacieho jazyka Kotlin. Ďalej sú vymenované a popísané použité nástroje pri implementácii, popísaný spôsob vytvárania inštancií tried pomocou DI. V zvyšku kapitoly sú predstavené konkrétne implementačné detaily a výzvy pri implementácii. Zároveň sú tam priblížené zaujímavé časti implementácie ako napríklad voľba použitého nástroja pomocou podmienenej DI či optimalizácie spojené s logovaním dát.

Posledná kapitola obsahuje popis postupu pri testovaní celej aplikácie. Porovnáva výsledky získané testovaním výkonu programu na klastroch vytvorených na dvoch výkonnostne rozdielnych počítačoch.

Do budúcna by si aplikácia určite zaslúžila nasadenie do reálneho použitia, na základe ktorého by bolo možné vykonať ďalšie optimalizácie implementácie pre vyšší výkon. Ďalšou zaujímavou funkcionalitou by bola možnosť kompletnej automatickej migrácie všetkých modelov a správ pri výmene použitého nástroja. To je možné zatiaľ vykonať len manuálne

prevádzkovateľom služby. Ako bolo spomenuté pri implementáci, v prípade ďalšieho vývoja by došlo aj k výmene použitého typu autentifikácie napríklad za typ OAuth 2.0. Po vyladení celého systému a po nasadení do produkcie by sa mohlo pokračovať aj s implementáciou adaptérov pre ďalšie nástroje.

Výhodou aplikácie popísanej v tejto práci je jej charakter webovej služby a jej jednoduchosť použitia bez nutnosti hlbších znalostí o tom, ako takýto model výmeny správ funguje. Voľba atraktívneho programovacieho jazyka Kotlin a voľnosť licencie Apache v2.0 sú dobrým odrazovým mostíkom pre ďalší vývoj aplikácie.

Literatúra

[1] ActiveMQ Features Specification, The Apache Software Foundation. [Online; navštívené 27.12.2017].
URL http://activemq.apache.org/features.html

[2] Apache Kafka Documentation, The Apache Software Foundation. [Online; navštívené 6.1.2018].
URL https://kafka.apache.org/documentation/

[3] Java Message Service (JMS), Oracle. [Online; navštívé 15.12.2017].
URL http://www.oracle.com/technetwork/java/jms/index.html

[4] RabbitMQ Documentation, Pivotal Software. [Online; navštívené 5.1.2018].
URL https://www.rabbitmq.com/documentation.html

[5] Java EE 6 Tutorial, Oracle. 2010, [Online; navštívené 15.12.2017].
URL https://docs.oracle.com/cd/E19798-01/821-1841/bnces/index.html

[6] Boschi, S.; Santomaggio, G.: *RabbitMQ CookBook*. Packt Publishing, 2013, ISBN 978-1-84951-650-1.

[7] Eugster, P. T.; Felber, P. A.; Guerraoui, R.; aj.: The Many Faces of Publish/Subscribe. *ACM Computing Surveys*, ročník 35, č. 2, Jún 2003: s. 114–131.

[8] Garg, N.: *Learning Apache Kafka*. Packt Publishing, druhé vydání, 2015, ISBN 978-1-78439-309-0.

[9] Hophe, G.; Woolf, B.: *Enterprise Integration Patterns: Designing, Building and Deploying Messaging Solutions*. Addison-Wesley Professional, 2003, ISBN 978-0321200686.

[10] Jemerov, D.; Isakova, S.: *Kotlin in Action*. Manning Publications, 2017, ISBN 1617293296.

[11] Mell, P. M.; Grance, T.: SP 800-145. The NIST Definition of Cloud Computing. Technická zpráva, Gaithersburg, MD, United States, 2011.

[12] Narkhede, N.: First Apache release for Kafka is out! Január 2012, [Online; navštívené 10.1.2018].
URL
https://engineering.linkedin.com/kafka/first-apache-release-kafka-out

[13] Snyder, B.; Bosanac, D.; Davies, R.: *ActiveMQ in Action*. Manning Publications Co., 2011, ISBN 978-1-933988-94-8.

[14] Videla, A.; J.W.Williams, J.: *RabbitMQ in Action: Distributed Messaging for Everyone*. Manning Publications Co., 2012, ISBN 9781935182979.

Prílohy

Príloha A

Obsah CD

Priložené CD obsahuje nasledujúce súbory a adresáre

- /text/dp.pdf – elektronická verzia písomnej správy
- /text/tex/ – zdrojové súbory technickej správy a použité obrázky
- /src/ – zdrojové súbory aplikácie
- /app/ – .jar súbor s funkčnou aplikáciou
- /test/ – testovacia sada spolu s výstupom výkonového testu
- /manual/manual.pdf – návod na inicializáciu a jednoduchý popis použitia
- initDb.sql – inicializačný skript databázy

Príloha B

Testovacie sady

V skratke tu budú predstavené jednotlivé testovacie sady spolu s očakávaným výstupom. Testovacie sady sú dostupné v prílohe A v zložke /test/vystup_testov. Testy sú určené pre testovací nástroj `JMeter` popisovaný v tejto práci.

Sada 1

Sada 1 sa zameriava na ukážku funkcionality okolo front.

Názov dotazu	Úspech	Popis
GetQueues	✓	Na prázdnej databázi je očakávaný prázdny zoznam.
Publish_NonExisting_Queue	X	Upozorní na to, že fronta neexistuje.
Receive_NonExisting_Queue	X	Upozorní na to, že fronta neexistuje.
Create_Queue	✓	Vytvorí novú frontu.
Receive_EmptyQueue	X	Upozorní na to, že fronta je prázdna.
Publish_NewQueue	✓	Odošle správu do novej fronty.
Receive_Queue	✓	Prijme správu.
Receive_EmptyQueue	X	Upozorní na to, že fronta je prázdna.

Tabuľka B.1: Scenár 1

Sada 2

Druhá sada sa zameriava na ukážku funkcionality okolo tém.

Názov dotazu	Úspech	Popis
GetTopics	✓	Na prázdnej databázi je očakávaný prázdny zoznam.
Publish_NonExisting_Topic	X	Upozorní na to, že téma neexistuje, alebo nie je odoberaná.
Receive_NonExisting_Topic	X	Upozorní na to, že téma neexistuje, alebo nie je odoberaná.
Create_Topic	✓	Vytvorí novú tému.
Receive_EmptyTopic_ WithoutSubscription	X	Upozorní na to, že téma neexistuje, alebo nie je odoberaná.
Subscribe_EmptyTopic	✓	Prihlási na odber danej témy.
Receive_EmptyTopic_ WithSubscription	X	Upozorní na prázdnu tému.
Publish_EmptyTopic	✓	Odošle správu do predtým prázdnej témy.
Receive_Topic	✓	Prijme správu.
Receive_EmptyTopic	X	Upozorní na prázdnu tému.

Tabuľka B.2: Scenár 2

Sada 3

Tretia sada sa zameriava tak isto na témy. Konkrétne ukazuje kroky potrebné k tomu, aby používateľ bol skutočne schopný čítať správy z nejakej témy.

Názov dotazu	Úspech	Popis
Create_Topic	✓	Vytvorí novú tému.
Publish_TopicWithout Subscriber_#1 #2 #3	✓	Úspešne odošle správu do témy bez odoberateľov.
Subscribe_Topic	✓	Prihlási na odber danej témy.
Receive_EmptyTopic	X	Upozorní na prázdnu tému.
Publish_TopicWith Subscriber_#1	✓	Úspešne odošle správu do témy s odoberateľom.
Receive_Topic	✓	Prijme správu.

Tabuľka B.3: Scenár 3

Sada 4

V tejto sade je priblížená validácia mena fronty na vyhradené názvy, ktorým je v tomto prípade názov fronty pre internú správu logov (`internal_message_audit`). Všetky tieto dotazy skončia chybou.

Názov dotazu	Popis
Create_Queue_SpecialName	Zlyhá pri vytvorení novej témy.
Purge_Queue_SpecialName	Zlyhá pri mazaní obsahu špeciálnej témy.
Delete_Queue_SpecialName	Zlyhá pri vymazaní špeciálnej témy.
Publish_Queue_SpecialName	Zlyhá pri posielaní správy do špeciálnej témy.
Receive_Queue_SpecialName	Zlyhá pri čítaní správy z špeciálnej témy.

Tabuľka B.4: Scenár 4

Sada 5

Táto sada predstaví možnosti konfigurácie dotazu pre štatistiky prenosu dát. Okrem predstavených dotazov je jednotlivé parametre možné kombinovať a vďaka tomu má používateľ možnosť ich profilovať presne podľa potreby.

Názov dotazu	Popis
DefaultStats	Celkové prenesené dáta za dnešný deň.
StatsUser	Dnes prenesené dáta daného používateľa.
StatsAllQueues	Dnes prenesené dáta pre všetky fronty.
StatsQueue	Dnes prenesené dáta konkrétnej fronty.
StatsDate	Celkové prenesené dáta od dátumu do dnes.

Tabuľka B.5: Scenár 5