Západočeská univerzita v Plzni
Fakulta aplikovaných věd
Katedra informatiky a výpočetní techniky

Diplomová práce

Srovnání implementací message brokerů

Plzeň 2020 David Bohmann

Místo této strany bude zadání práce.

Prohlášení

Prohlašuji, že jsem diplomovou práci vypracoval samostatně a výhradně s použitím citovaných pramenů.

V Plzni dne 29. dubna 2020

David Bohmann

Abstract

The goal of this thesis is to compare selected message brokers from performance, security and developer friendliness points of view. Thesis analyses existing studies and based on them it provides new analysis with practical use of brokers. The main benefit of this application is comparison of brokers in environment which is close to real use-case.

The application is developed in programming language Java with use of SpringBoot and Apache Camel frameworks. Message brokers are tested in Amazon Web Services cloud environment. The created application allows user to easily test performance of message brokers and configure its security.

Abstrakt

Cílem této práce je porovnat zvolené implementace message brokerů z pohledu výkonnosti, bezpečnosti a vývojářské přívětivosti. Práce analyzuje existující studie a na jejich základě představuje novou analýzu pro praktické použití brokerů. Přínosem této aplikace je vyhodnocení brokerů v prostředí blížícímu se reálnému použití.

Aplikace je vyvíjena v programovacím jazyce Java za použití frameworků Spring Boot a Apache Camel. Testování message brokerů probíhá v cloudovém prostředí Amazon Web Services. Vytvořená aplikace umožňuje jednoduše otestovat výkonnost message brokeru a konfigurovat jeho zabezpečení.

Obsah

1 Úvod

Ve světě, kde se každým dnem zvyšuje počet zařízení připojených k internetu společně s množstvím přenesených dat, čelí vývojáři stále častěji problému úspěšné výměny dat mezi jednotlivými systémy. V případě integrace více systémů je vhodným řešením použití prostředníka namísto přímé komunikace mezi systémy. Jako prostředník mezi danými systémy lze použít právě message broker.

Message broker představuje nástroj pro zajištění spolehlivého předávání zpráv mezi aplikacemi. Zajišťuje oddělení aplikací, které o sobě nemusí kvůli předání zpráv vzájemně vědět a stačí jim vědět pouze o message brokeru. S rostoucím počtem integrací systémů se message brokery stávají nedílnou součástí komunikace.

Tato práce si klade za cíl porovnat existující implementace message brokerů z pohledu výkonnosti, bezpečnosti a vývojářské přívětivosti. Cílem práce je vybrat message brokery pro porovnání, důkladně se s jednotlivými implementacemi seznámit a navrhnout aplikaci, která umožní objektivně porovnat jejich výkonnost a bezpečnost.

V kapitole 2 se seznámíme se samotným konceptem message brokeru a existujícími implementacemi. Následně si v kapitole 3 přiblížíme existující studie porovnávající výkonnost a bezpečnost brokeru.

Na závěr teoretické části si v kapitole 4 představíme technologie použité při vývoji aplikace.

V praktické části práce si v kapitolách 5 a 6 představíme konkrétní implementace message brokerů zvolené pro porovnávání a seznam kritérií, na základě kterých bude uskutečněno vyhodnocení.

V kapitole 7 podrobně prozkoumáme vytvořenou aplikaci, běhové prostředí a podmínky pro optimalizaci testování výkonnosti. Závěrem v kapitole 8 vyhodnotíme naměřené hodnoty výkonnosti, možnosti konfigurace bezpečnosti a vývojářskou přívětivost.

2 Message broker

V této kapitole se seznámíme s konceptem message brokeru, způsoby výměny zpráv a jednotlivými implementacemi. Message broker je architektonický vzor pro validaci, konverzi a přeposílání zpráv. Broker používá různé protokoly pro zjištění formátu zprávy. Součástí protokolů jsou informace o tom, jak by zpráva měla být přenesena, přijata a zpracována [8].

Message broker zajišťuje následující důležité úkoly:

- Přeposlat zprávu do jedné nebo více destinací.
- Konvertovat zprávu.
- Provést spojení více zpráv nebo naopak rozdělení zprávy při odesílání od uživatele do cílové destinace. Při odpovědi uživateli opět zprávu spojit či rozdělit zpět.
- Komunikovat s externím úložištěm pro trvalé uložení zprávy.
- Přijmout zprávu z webových služeb.
- Zvládat nenadálé události a chyby.
- Poskytnout přeposílání zpráv na základě obsahu pomocí principu Publish - subscribe [7].

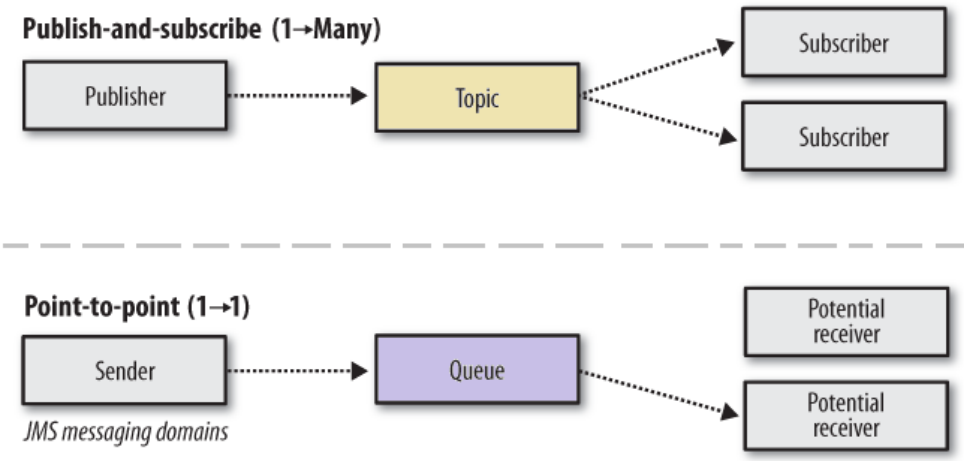
2.1 Fronta u message brokeru

Než se pustíme do dalších podrobností, je třeba definovat si pojem *Queue (Fronta)* z pohledu message brokerů.

Frontu lze popsat jako abstraktní datový typ na principu FIFO (First In, First Out - První dovnitř, první ven). Ve frontě lze ukládat objekty tím způsobem, že objekt, který se do fronty dostane nejdříve, také frontu nejdříve opustí.

Fronta z pohledu message brokerů funguje na stejném principu. Objekty ukládající se do fronty jsou jednotlivé zprávy. Použitím fronty message broker zajišťuje, že zprávy dorazí ve správném pořadí.

Nejjednodušším příkladem je použití message brokeru mezi dvěma systémy. Message broker přijímá zprávy od první aplikace a zařazuje je do fronty, ze které jsou následně zprávy posílány do druhé aplikace. Tomuto způsobu předávání zpráv se říká Point-To-Point [3].



Obrázek 2.1: Schéma fronty a tématu, zdroj: [3]

2.2 Téma u message brokeru

Druhým důležitým pojmem je *Topic (Téma)*.

Téma funguje rozdílně než fronta. Pro názorné pochopení rozdílu slouží Obr. 2.1. Zatímco fronta zprostředkovává komunikaci mezi jedním odesílatelem zprávy a jedním příjemcem, k jednomu tématu se může přihlásit více „odběratelů“ a všichni obdrží zprávu, kterou do tématu vloží „přispěvatel“. Důležité je uvědomit si také situaci, kdy k tématu nemusí být přihlášen žádný odběratel. Různé brokery řeší tuto situaci rozdílně, může se stát, že zpráva se jednoduše ztratí, nebo naopak čeká, dokud se odběratel nepřihlásí k odběru.

Téma tedy slouží pro komunikaci ve stylu 1:N (1 přispěvatel, 0-N odběratelů), narozdíl od fronty, která používá styl komunikace 1:1 (1 odesílatel, 1 příjemce).

Tomuto stylu předávání zpráv se říká Publish-Subscribe [3].

2.3 Vzory výměny zpráv

V předchozí části jsme si vysvětlili hlavní principy message brokerů a nyní si představíme vzory pro výměnu zpráv. Message brokery používají nejčastěji vzory *Point-To-Point*, dále *Publish-Subscribe* a v některých případech i *Pipeline*.

Point-To-Point vzor je implementován pomocí fronty zpráv. Jedná se o komunikaci dvou systémů, z nichž jeden zprávu do fronty vloží a druhý ji zpracuje. Jedná se o nejjednodušší způsob komunikace. Broker zpravidla neřeší, kolik systémů do fronty zprávy vkládá a kolik konzumuje, zpráva by

měla být doručena právě jednou (konzument může běžet na více vláknech a pouze jedno zpracuje zprávu). Pokud naopak v danou chvíli není k frontě připojen žádný konzument, broker by měl zprávu udržet, dokud se nepřipojí.

Publish-Subscribe je vzor implementován pomocí tématu. Jedná se o způsob oddělení množiny přispěvatelů a odběratelů. Přispěvatel do tématu vloží zprávu a neví, kteří odběratelé zprávu zpracují. Hlavním rozdílem oproti Point-To-Point je to, že zpráva z brokeru nezmizí po zpracování prvním odběratelem, ale obdrží ji všichni, kteří jsou v době přidání zprávy do tématu připojeni. Problém nastává ve chvíli, kdy k tématu není připojen žádný odběratel a přispěvatel publikuje zprávu. Broker nemůže při použití tohoto vzoru zaručit doručení zprávy.

Pipeline je vzor, kdy z fronty může konzumovat zprávy více odběratelů. Jedná se tedy vlastně o kombinaci předcházejících vzorů, zaručuje doručení zprávy a zároveň lze zprávu poslat více odběratelům zároveň. Pro implementaci se může používat výraz *Fan-out*.

2.4 Přehled implementací message brokerů

S rozmachem message brokerů a jejich zvětšující se oblíbeností se samozřejmě rozrůstá i množství jejich implementací. V rámci této sekce se podrobně seznámíme s detaily následujících implementací.

- Apache ActiveMQ,
- RabbitMQ,
- Apache Kafka,
- Amazon Simple Queue Service,
- ZeroMQ.

2.4.1 Apache ActiveMQ

ActiveMQ je open-source message broker napsaný v Javě a je používán jako Java Message Service (JMS) klient. Zahrnuje komunikaci mezi více než jedním klientem nebo serverem, podporuje standard JMS 1.1. ActiveMQ používají open-source projekty jako Apache CXF či Apache Camel [19].

Kromě „klasického ActiveMQ“ Apache aktivně vyvíjí oddělený produkt a pravděpodobně budoucího nástupce zvaného Artemis. Apache ActiveMQ Artemis je kompletně přepsaná verze ActiveMQ 5 zahrnující prvky dalšího

brokeru HornetQ (broker vyvíjený společností RedHat). Tento broker implementuje JMS standard podle specifikace 2.0.

Artemis je postaven na neblokující architektuře a poskytuje možnosti jako například clustering, používání databází jako úložnou vrstvu virtuální paměti JMS, ukládání journalů nebo správu cache.

ActiveMQ je používán spíše v menších projektech, kde není třeba vypořádávat se s obrovským množstvím dat. Tento broker se stal populárním díky své výkonnosti a podpoře mnoha protokolů.

Představuje vhodné řešení pro vývojáře v Javě, jelikož je postaven na JMS standardech a podporuje asynchronní komunikaci. Umožňuje vývojáři vytvořit rychlý a stabilní proces pro výměnu zpráv v rámci vyvíjeného softwaru.

ActiveMQ je široce kompatibilní, lze ho kombinovat s dalšími nejvíce používanými brokery, jako je Kafka, RabbitMQ, či AmazonSQS. K tomu je zde možnost používat Amazon MQ, což je implementace ActiveMQ poskytovaná v cloudu Amazon Web Services (AWS).

Vzhledem k tomu, že je ActiveMQ postaven nad JMS API, procesy vytváření, přenosu a zpracování zpráv jsou definovány unifikovaným JMS standardem. ActiveMQ klient lze implementovat kromě Javy i v dalších programovacích jazycích, například Python, Node.js, či Ruby [2].

ActiveMQ je snadno konfigurovatelný a umožňuje jednoduše používat vzory výměny zpráv. Podporuje připojení přes mnoho používaných protokolů, kromě OpenWire jsou to například WebSockety, AMQP, REST, STOMP, MQTT a další [21].

Architektura ActiveMQ zahrnuje následující prvky:

- Message broker,
- klient (odesílatel),
- příjemce.

ActiveMQ umožňuje posílat zprávy dvěma různými způsoby:

- Fronta - zprávy z fronty přijímá pouze jeden příjemce.
- Téma - zprávu může přijímat více příjemců.

2.4.2 RabbitMQ

RabbitMQ je jedním z nejvíce používaných open-source message brokerů. Jedná se o jeden z nejstarších open-source message brokerů, vznikl v Rabbit

Technologies Ltd. v roce 2007. Mezi vývojáři je RabbitMQ oblíbený z důvodu velkého množství knihoven, vývojářských nástrojů a jednoduchých instrukcí.

RabbitMQ byl původně implementovaný pro použití protokolu AMQP (Advanced Message Queuing Protocol) pro ukládání, směrování a zasílání zpráv a zajištění spolehlivosti a bezpečnosti. Následně došlo k začlenění dalších protokolů, například MQTT (Message Queuing Telemetry Transport) protokolu používanému zejména pro komunikaci chytré elektroniky, nebo STOMP (Streaming Text Oriented Messaging Protocol) [18].

RabbitMQ je vyvíjený ve funkcionálním programovacím jazyce Erlang. RabbitMQ je známý jako jeden z „tradičních“ message brokerů, který nabízí široké využití. RabbitMQ používají jak vývojáři malých projektů a startupů, tak vývojáři entreprise řešení.

Nejčastější komunikace s RabbitMQ je pomocí protokolu AMQP, který zajišťuje flexibilitu při zakomponování message brokeru i v softwaru vyvíjeném v jiných jazycích, než je Java. Existuje mnoho knihoven pro užití RabbitMQ ve většině používaných programovacích jazycích. Mezi další jazyky, které bezproblémově implementují RabbitMQ, patří například .NET, PHP, Python, Ruby, JavaScript, Go, Elixir, Objective-C nebo Swift. Největší výhodou RabbitMQ je rozhodně velké množství pluginů a knihoven pro takto široké pole programovacích jazyků [16].

Jakožto zástupce „klasických message brokerů“ pro běžné použití je RabbitMQ založen na komunikačním vzoru Publish-Subscribe. Zasílání zpráv mezi systémy může být jak synchronní, tak asynchronní [25].

RabbitMQ nabízí následující možnosti:

- Podpora různých protokolů, ukládání zpráv do front, změny routování do front, různé typy výměny zpráv.
- Clustering zajišťuje vysokou dostupnost a průchodnost.
- Kompatibilita s většinou používaných programovacích jazyků.
- Jednoduché spuštění v privátních i veřejně dostupných cloudech.
- Podpora autentikace a autorizace, podpora TLS a LDAP.

RabbitMQ zaručuje doručení zprávy mezi odesílatelem a příjemcem, pokud je příjemce připojen. Broker kontroluje stav zprávy a ověřuje, zda bylo doručení úspěšně dokončené. Broker předpokládá, že příjemci zpráv jsou připojeni k brokeru. RabbitMQ zaručuje, že zprávy dorazí v pořadí, v jakém byly publikovány.

Jak již bylo řečeno, největší výhodou je velké množství knihoven a pluginů. Další plus je velmi kvalitní škálovatelnost. RabbitMQ má kvalitně zpracovanou dokumentaci a definovaná pravidla použití a nabízí použití různých vzorů pro výměnu zpráv. Zejména vhodné použití je pro následující vzory:

- Point-to-point - přímá výměna mezi odesílatelem a příjemcem.
- Topic - všichni příjemci obdrží zprávu publikovanou do Topicu.
- Fan-out - všichni příjemci připojeni k frontě obdrží zprávu.

Zde můžeme vidět nevýhodu RabbitMQ: pokud není příjemce připojen k fan-out frontě, zpráva bude ztracena. Řešení tohoto problému nabízí například Apache Kafka.

2.4.3 Apache Kafka

Apache Kafka je message broker napsaný v jazycích Scala a Java, který začal vznikat v roce 2010, kdy vývojáři ve společnosti LinkedIn začali řešit integraci velkého množství dat v jejich infrastruktuře s dalšími systémy. Tradiční message brokery dostupné v té době nepostačovaly požadavkům LinkedInu, byly pro ně moc pomalé a těžkopádné. V LinkedInu se rozhodli přijít s vlastním řešením. Oproti stávajícím implementacím vývojáři cílili na vysokou škálovatelnost, výkon a odchytávání chyb.

Kafka začal být populární díky své kompatibilitě s mnoha systémy. Může se jednat jak o webové a desktopové aplikace, tak i o mikroservisy, monitorovací a analytické systémy, SQL a NoSQL databáze a mnoho dalších zdrojů. V dnešní době je stávající vývoj pod záštitou Apache [18].

Kafku dnes používají společnosti jako Netflix, eBay, Uber, The New York Times, PayPal nebo Pinterest.

Kafku lze použít při vývoji data-driven aplikací a při správě složitých back-endových systémů. Kafka si drží svou oblíbenost zejména díky tomu, že umí následující:

- Odesílat a přijímat zprávy ze streamů s výbornou škálovatelností a výkonností.
- Trvale ukládat streamy zpráv a distribuovat data napříč mnoha uzly pro velmi vysokou dostupnost systému.
- Zpracovávat streamy dat hned po přijetí, což umožňuje agregaci, úpravu dat, spojovat či rozdělovat data, atd.

Apache Kafka funguje velmi rozdílně oproti většině klasických message brokerů. Základem celého brokeru je „distribuovaný commit log“. Obsahuje časovou stopu a data v něm už nelze měnit, pouze přidávat další data na konec logu [4]. Použitím commit logu Kafka zaručuje doručení zpráv ve správném pořadí.

Mezi další koncepty Apache Kafky patří:

- Témata (topics) - uložené streamy záznamů.
- Záznam - informace ve formátu klíč + hodnota, společně s časovou známkou.

Interakce mezi klientem a serverem je implementovaná přes binární protokol specifický pro Kafku. Kafka nepodporuje použití tradičních protokolů, jako například AMQP či OpenWire.

Jak již bylo řečeno, message brokery nabízejí dva základní způsoby výměny zpráv:

- Point-To-Point (pomocí fronty),
- Publish-Subscribe (pomocí tématu).

Každý způsob má své výhody a nevýhody. Výhoda prvního způsobu je možnost jednoduše škálovat zpracování, ale je vykoupena tím, že zprávu nelze doručit více příjemcům. Druhý způsob umožňuje posílat naráz data více příjemcům, ale naopak je problémem škálovatelnost. Kafka dokáže zkombinovat tyto dva způsoby zpracování zpráv tak, aby získal výhody obou způsobů a eliminoval nevýhody.

Tím, že Kafka kombinuje nejen funkce zasílání zpráv, ale i ukládání a zpracování, není tradičním message brokerem. Jedná se spíše o velmi silnou streamovací platformu schopnou zvládnout biliony zpráv za den. Kafku lze využít jak pro ukládání a zpracování historických dat, tak pro real-time systémy. Tento broker je možné použít jak pro vývoj streamovacích aplikací, tak pro streamování dat [20].

Další věci, v čem se Kafka liší od tradičních message brokerů:

- Zprávy nemají unikátní ID, ale jsou identifikovány podle offsetu v logu.
- Systém nekontroluje příjemce každého tématu nebo zprávy.
- Kafka neudržuje indexy ani nepovoluje „random-access“ (náhodný přístup), pouze doručí zprávy v pořadí, začínající na daném offsetu.
- Kafka nemaže zprávy ani nepoužívá buffer pro zprávy v uživatelském prostoru.

Z předchozího detailního popisu je zřejmé, že Kafka je velmi rozdílný oproti tradičním brokerům a bude vyžadovat velké množství práce pro správné nastavení. Při vývoji uživatel může pocítit nedostatek pluginů a nástrojů pro snadné použití v kódu. Oproti tomu se jedná o perfektní open-source řešení pro real-time systémy a projekty, kde je potřeba zpracovávat obrovské množství dat (100 000 zpráv za sekundu či více).

2.4.4 Amazon Simple Queue Service

Amazon Simple Queue Service (SQS) je příklad message brokeru v cloudových službách, konkrétně Amazon Web Services (AWS).

Velkou výhodou je snadná konfigurovatelnost, SQS odstraňuje potíže spojené s nastavováním a během middlewaru na vlastních serverech a umožňuje vývojářům soustředit se na samotnou práci. SQS umožňuje posílat, ukládat a přijímat zprávy mezi softwarovými komponentami, jak v cloudu, tak mimo cloud [1].

Amazon SQS lze používat přes interaktivní konzoli, v příkazové řádce nebo se lze napojit z SDK třetích stran.

Amazon SQS umožňuje používat dva typy front:

- Standardní fronta umožňuje maximální propustnost, která je ale vykoupena tím, že zprávy nemusí dorazit přesně ve správném pořadí a mohou dorazit jednou až vícekrát.
- FIFO frontu lze použít pokud chceme u zpráv mít zaručeno, že dorazí ve správném pořadí a právě jednou. Toto naopak snižuje propustnost.

Amazon SQS přináší oproti klasickým message brokerům běžícím na jednom serveru výhody cloudového prostředí. AWS zajišťuje infrastrukturu nutnou pro běh brokeru, zároveň zajišťuje její vysokou přístupnost a škálovatelnost. Odstraňuje nutnost výdajů na nákup infrastruktury před začátkem vývoje a nutnost instalace a konfigurace message brokeru. Odstiňuje vývojáře od nutnosti správy prostředí. Fronty v SQS jsou dynamicky vytvářené a automaticky škálovatelné dle potřeby aplikace.

Amazon SQS lze použít k zasílání citlivých dat při použití SSE (server-side encryption - šifrování na straně serveru) k zašifrování těla zpráv. AWS umožňuje integraci Amazon SQS s AWS Key Management Service (KMS) pro centrální uložení klíčů které chrání zprávy v SQS. Mezi zajímavé funkce AWS KMS patří například logování každého použití šifrovacího klíče.

Zajištěním automatické škálovatelnosti lze Amazon SQS použít pro zasílání „téměř neomezeného“ množství dat s „téměř neomezenou“ úrovní

propustnosti beze ztráty zpráv a bez potřeby dostupnosti připojených systémů. Amazon SQS jako prostředník mezi jednotlivými systémy zajišťuje, že tyto systémy mohou běžet nezávisle na sobě, čímž zvyšuje celkovou odolnost systému proti chybám. Umožňuje ukládat redundantní kopie každé zprávy v různých datacentrech s různou dostupností tak, aby byla data kdykoliv dostupná.

Amazon SQS je řešením pro dynamické škálování cloudových aplikací dle potřeby. SQS škáluje v závislosti na ostatních běžících cloudových aplikacích, čímž odstiňuje uživatele od plánování kapacit a dohledu.

Amazon tvrdí, že SQS nemá „žádný“ limit pro počet zpráv v jedné frontě a standardní fronta nabízí „téměř neomezenou“ průchodnost.

Dalším rozdílem oproti klasickým brokerům je cena, která je počítána dle opravdového používání fronty. Naproti tomu klasické brokery musejí být stále připravené na nejvyšší nápor a platí se pevná částka za konstantní provoz.

2.4.5 ZeroMQ

ZeroMQ je příkladem velmi jednoduché knihovny pro posílání zpráv. Jedná se o nízkoúrovňovou knihovnu vyvinutou v jazyce C++. ZeroMQ poskytuje fronty zpráv, které nepotřebují samotný message broker [9]. Jedná se o rychlé a jednoduché řešení pro použití v distribuovaných či paralelních aplikacích.

ZeroMQ používá vzory pro posílání zpráv definované v ZeroMQ message queueing library [10]. Tato knihovna poskytuje takzvané sockety (zobecněná verze socketů z UNIXových systémů či síťových socketů), které mohou reprezentovat M-N propojení mezi koncovými uzly. Sockety vyžadují informaci o tom, který vzor pro výměnu zpráv použít pro optimalizaci přenosu. Mezi základní vzory z knihovny ZeroMQ patří následující:

- Request-Reply: Propojí množinu klientů s množinou služeb. Tento vzor se používá pro volání vzdálených procedur, či distribuci úloh.
- Publish-Subscribe: Propojuje množinu „Přispěvatelů“ s množinou „Odběratelů“. Tento vzor se používá pro distribuci dat.
- Pipeline: Propojuje uzly ve stylu fan-out / fan-in, může mít několik kroků a smyček. Tento vzor se používá zejména pro paralelizaci úloh.
- Exclusive Pair: Propojí pouze dva sockety mezi sebou. Speciální užití nízkoúrovňového vzoru pro specifické případy užití.

2.4.6 Další implementace

Kromě výše zmíněných implementací existuje spousta dalších, některé z nich v krátkosti zmíníme zde:

- Redis - Remote Dictionary Server. Jedná se o distribuované úložiště dat typu „Klíč - Hodnota“ udržující všechny informace v paměti. Lze používat jako databázi, cache, nebo message broker.
- IronMQ - message broker cílící na cloudová řešení, chlubí se skvělou výkonností a škálovatelností.
- Microsoft Azure Service Bus - cloudová verze message brokeru pro prostředí Microsoft Azure.
- Eclipse Mosquitto MQTT Broker - Message broker používaný pro IoT, používá velmi jednoduchý protokol MQTT.

3 Způsoby porovnávání message brokerů

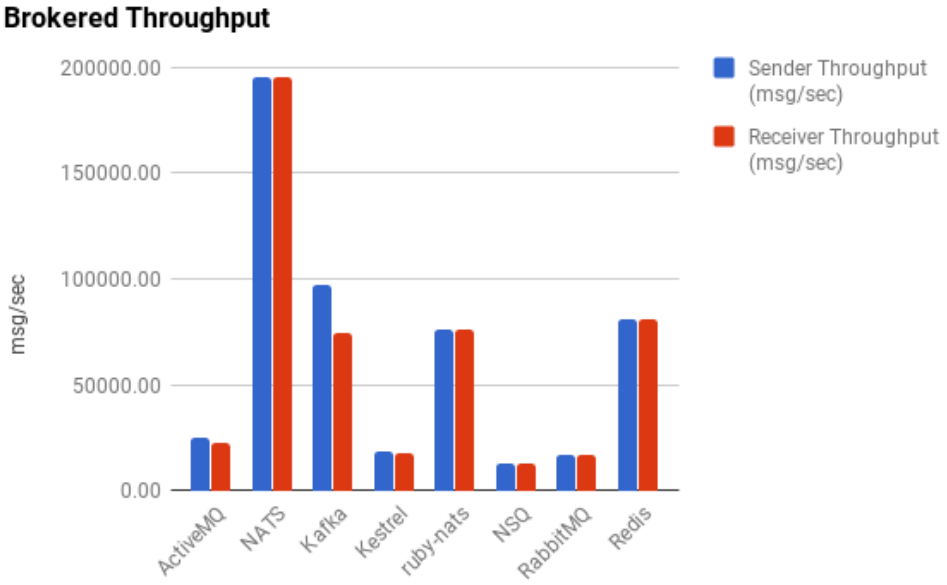
V této kapitole se podrobněji podíváme na principy a techniky měření pro vyhodnocování message brokerů. Zaměříme se na veřejně dostupné analýzy, ze kterých lze v diplomové práci vycházet a na zhodnocení jejich závěrů. Na základě těchto zdrojů navrhneme, jakým způsobem budeme testovat message brokery v rámci diplomové práce.

3.1 Výkonnost message brokerů

Většina benchmarků se naprosto logicky soustředí na výkonnost, propustnost a latenci, jelikož to je něco, co je jednoznačně měřitelné a pro naprostou většinu uživatelů jsou tyto informace rozhodujícím faktorem při výběru message brokeru. Samotný princip benchmarkování je komplikovaný, protože testovací podmínky velmi těžko budou odpovídat reálným podmínkám v provozu.

Podíváme se podrobně na analýzu Tylera Treata, který pro benchmarkování message brokerů napsal několik programů a analýz. V roce 2014 v článku Dissecting Message Queues [23] se soustředí na několik aspektů, mezi nimi charakteristiky API, jednoduchost spuštění a spravování, či výkonnostní parametry. V jeho analýze se nacházejí i message systémy, které nepotřebují samotný broker jako prostředníka mezi systémy. Z message brokerů fungujících jako prostředník mezi systémy zahrnuje v analýze následující:

- ActiveMQ,
- NATS,
- Kafka,
- Kestrel,
- NSQ,
- RabbitMQ,
- Redis,
- ruby-nats.

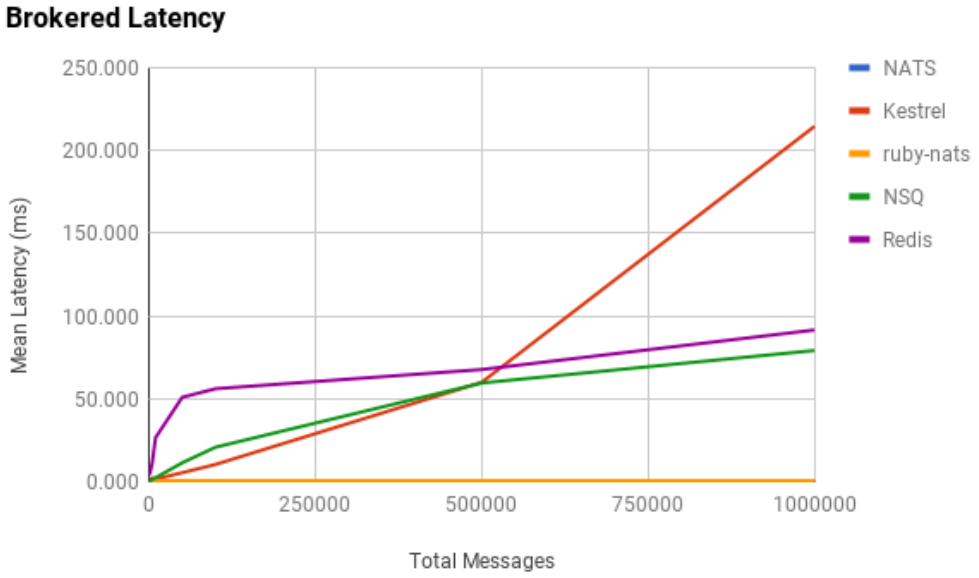


Obrázek 3.1: Graf propustnosti message brokerů dle Treata, zdroj: [23]

Mezi výkonnostní ukazatele Treat řadí propustnost a latenci. Při testování propustnosti brokeru posílá z jednoho systému 1 000 000 zpráv o velikosti 1 kB do druhého, a měří, kolik zpráv je schopný odeslat za 1 sekundu a kolik je schopen přijmout. Graf na Obr. 3.1 ukazuje, že pro většinu brokerů je propustnost při odesílání velmi podobná propustnosti přijímání zpráv, nicméně rozdíly v propustnosti jednotlivých brokerů jsou velmi velké a polovina testovaných brokerů má propustnost nižší, než 25 000 zpráv za sekundu.

Druhou klíčovou metrikou výkonnosti je latence. Latence označuje dobu potřebnou k poslání zprávy z jednoho bodu do druhého. Dalo by se předpokládat, že latence je převrácená hodnota propustnosti, ale není to přesně takto. Latence není uniformní a pro každou zprávu je jiná. Zatímco propustnost je měřena pro příjemce a odesílatele zvlášť, latence zahrnuje oba. Lze předpokládat, že latence bude růst v závislosti na počtu zasílaných zpráv.

Na Obr. 3.2 jsou zobrazeny grafy latencí pro testované brokery NATS, Kestrel, ruby-nats, NSQ a Redis. Na Obr. 3.3 jsou zobrazeny grafy latencí pro testované brokery ActiveMQ, Kafka a RabbitMQ. Grafy jsou celkem 2, protože rozdíly mezi brokery jsou i několik tříd. Zajímavé je pozorovat, že například pro Redis skokově vzroste latence už u malého počtu zpráv, ale pak roste lineárně velmi pomalu. U dalších brokerů lze pozorovat konstantní latenci nehledě na počet procházejících zpráv. Ostatní brokery splňují teoretický předpoklad a s počtem zpráv rostou téměř lineárně, lze pozorovat například u ActiveMQ, který má zároveň nejhorší hodnoty.



Obrázek 3.2: Grafy latence message brokerů NATS, Kestrel, ruby-nats, NSQ a Redis dle Treata, zdroj: [23]

Treat poskytuje pro naměřené hodnoty následující vysvětlení [23]:

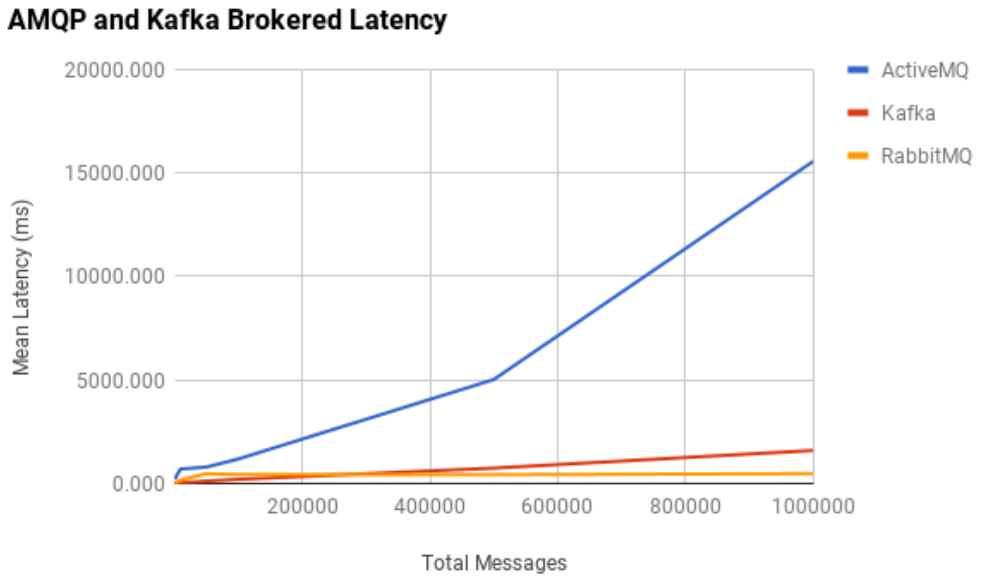
ActiveMQ a RabbitMQ jsou implementacemi AMQP a zaručují doručení zprávy pomocí několika mechanismů, které se projevují na zhoršené latenci:

- Ukládání zpráv na disk, což zaručuje, že se zpráva neztratí při restartu brokeru.
- Poslání potvrzení doručení zprávy.
- Možnost synchronního doručování zprávy.
- Možnost replikace front v clusteru jako ochranu proti ztrátě zpráv.

Kafka používá k ukládání zpráv commit log, který po oproti ActiveMQ či RabbitMQ po odeslání zprávy nemaže a udržuje ho po stanovenou dobu. Pokud se zpráva nedoručí, lze i po určité době odeslat znovu. Pro škálování používá nástroj ZooKeeper, který snižuje režii Kafky, zřejmě proto mohou být hodnoty latence o něco nižší oproti ActiveMQ a RabbitMQ.

NATS a ruby-nats jsou velice jednoduché brokery nevhodné pro využití v enterprise systémech, které nevyužívají ukládání zpráv na disk, nevyužívají ani TLS či SSL zabezpečení.

Kestrel neřeší možnost clusteringu a nemá ochranu proti selhání systému. Spoustu práce, které enterprise brokery zajišťují, nechává Kestrel na vývojáři (například dělení zpráv, možnost clusteringu).



Obrázek 3.3: Grafy latence message brokerů ActiveMQ, Kafka a RabbitMQ dle Treata, zdroj: [23]

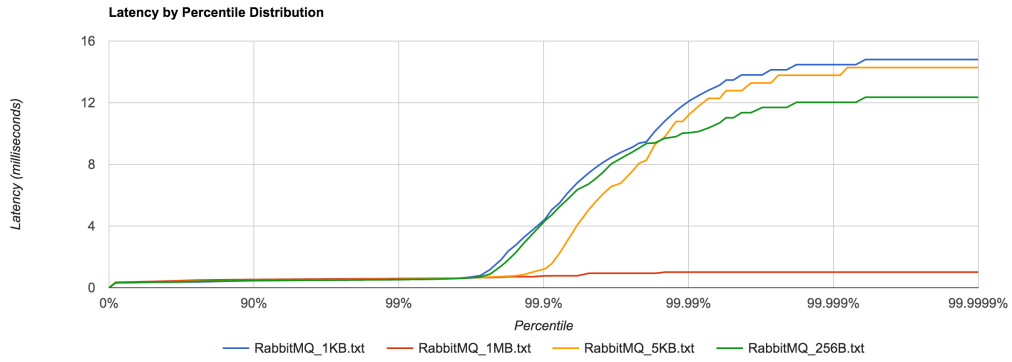
Redis dle Treata není vhodný nástroj pro distribuovaný message-broker. Přestože je jednoduchý na použití a velmi rychlý, má omezené možnosti použití.

NSQ je možné použít jako message-broker pro real-time systémy. Lze velice jednoduše použít v clusteru a potvrzuje doručení zprávy, ale neukládá zprávy na disk a nezaručuje doručení zpráv ve správném pořadí.

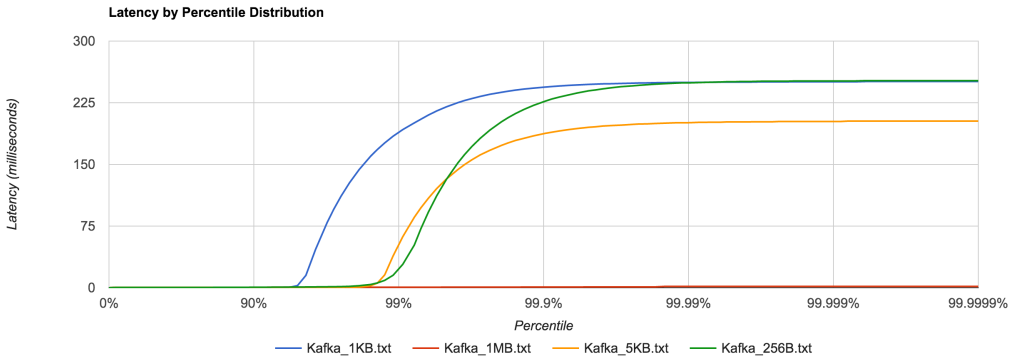
Závěrem této analýzy je, že nelze nalézt optimální řešení pro všechny situace, každý broker má své přednosti vykoupené některými zápory. Volba je závislá na tom, co zrovna daný vývojář od message brokeru potřebuje.

Ačkoliv nám tato analýza dává základní přehled o tom, jak rychlé message brokery jsou, má dva velké nedostatky. Porovnává „neporovnatelné“, zahrnuje jak enterprise message brokery, tak i jednoduchá řešení, která jsou pro enterprise aplikace nepoužitelná. Druhý nedostatek je, že u latence dodává pro porovnání pouze hodnotu průměru, která má malou vypovídající hodnotu. Latence není uniformní, ale není to ani hodnota, která má rovnoměrné rozložení. Bez hodnoty směrodatné odchylky a hodnoty různých percentilů je hodnota průměru tedy nepoužitelná.

Sám Treat si tohle následně uvědomil a v dalším článku se soustředí na podrobnější analýzu latence [24]. Zde zmiňuje, že některé benchmarky se soustředí pouze na latenci zpracování zprávy, ale nikoliv na latenci samotného doručení, která zahrnuje i dobu čekání zprávy ve frontě na doručení.



(a) Latence dle percentilu pro RabbitMQ



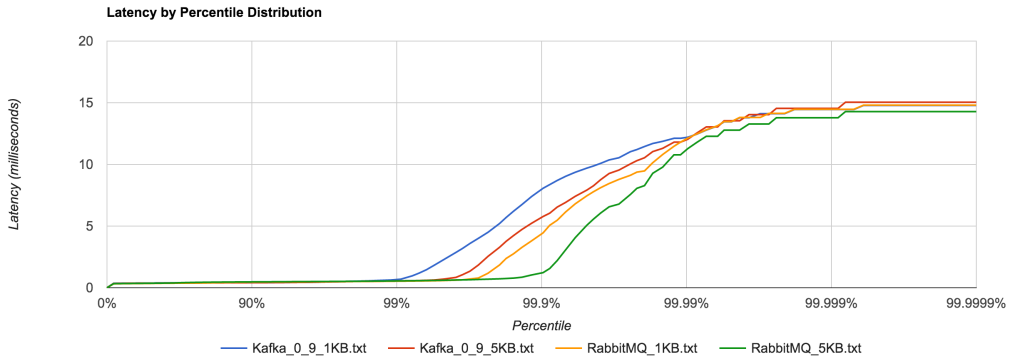
(b) Latence dle percentilu pro Apache Kafka 0.8.2.2

Obrázek 3.4: Grafy latence RabbitMQ a Apache Kafka dle percentilu dle Treata, zdroj: [24]

V analýze se Treat soustředí na 4 message brokery, z nichž zde uvedeme pouze 2, které lze považovat za kandidáty pro enterprise řešení (Kafka a RabbitMQ) a brokery NATS a Redis z důvodů popsaných výše vynecháme.

Způsob benchmarkování byl v této analýze následující: Program posílá požadavky na server se „stálou“ frekvencí a měří čas od zadání požadavku po obdržení odpovědi. Pro testování zvolil Treat následující konfigurace:

- Zpráva o velikosti 256 B, 3 000 požadavků za sekundu.
- Zpráva o velikosti 1 kB, 3 000 požadavků za sekundu.
- Zpráva o velikosti 5 kB, 2 000 požadavků za sekundu.
- Zpráva o velikosti 1 kB, 20 000 požadavků za sekundu.
- Zpráva o velikosti 1 MB, 100 požadavků za sekundu.



Obrázek 3.5: Porovnání latence RabbitMQ a Apache Kafka 0.9, zdroj: [24]

Pro každou konfiguraci probíhalo testování 30 sekund.

Při testování byla u RabbitMQ vypnuta funkce perzistentního ukládání dat, což má velký vliv na výkonnost a tím i latenci. U Apache Kafka možnost vypnout ukládání dat na disk není, protože Kafka na tomhle principu má postavený celý systém (již zmíněný commit log), nicméně ukládání dat je asynchronní a nemělo by mít velký vliv na výkonnost. V následujících grafech (Obr. 3.4) lze vidět hodnoty latence pro různé velikosti zpráv a frekvence (zde je u zprávy s velikostí 1kb počítáno s hodnotou 3 000 požadavků za sekundu). V grafech je použito logaritmické měřítko pro zvýraznění nejvyšších hodnot latence (90. percentil, 99. percentil, 99.9 percentil, atd.).

Autor se v analýze dále soustředí na porovnávání latencí při různých velikostech zpráv a různých frekvencích zasílání požadavků [24]. Zjišťuje, že jak RabbitMQ, tak Kafka zvládají velmi dobře zpracovávat velké zprávy posílané s nižší frekvencí, oproti tomu se latence výrazně zvyšuje při zasílání velkého množství malých zpráv, u Apache Kafka již kolem 94. percentilu, zatímco u RabbitMQ až u 99. percentilu. Treat předpokládá, že velice dobré zpracování velkých zpráv je způsobeno bufferováním zpráv u RabbitMQ, nicméně u Apache Kafka důvod neuvádí.

Následně probíhá porovnávání na různých verzích Apache Kafka (0.8.2.2 a 0.9.0.0), zde Treat uvádí několik důkazů o velkém zrychlení zpracování velkého počtu malých zpráv, kdy se Kafka latencí vyrovnává RabbitMQ, viz graf na Obr. 3.5. Treat předpokládá, že ke skokovému zlepšení mezi jednotlivými verzemi došlo díky změně nastavení flushování zpráv z cache na perzistentní disk.

Analýzu výkonnosti message brokerů můžeme shrnout následovně: Je nutné oddělit propustnost a latenci. Ačkoliv se může zdát, že tyto hodnoty jsou na sobě závislé, opak je pravdou. Propustnost zpráv při přijímání

je u většiny testovaných brokerů velmi podobná propustnosti při odesílání zprávy. U latence se nelze spolehnout na základní statistiky, jako je průměrná hodnota, ale je třeba důkladně se zaměřit na okrajové hodnoty. Zároveň není vhodné brát benchmarky, kde se broker zatíží na 100%, jako rozhodující faktor, jelikož v reálném použití k takovému případu moc často nedojde.

3.2 Bezpečnost message brokerů

Dalším kritériem, pomocí kterého lze porovnávat message brokery, je bezpečnost. Zajímá nás tedy, jestli lze zprávy procházející přes message broker nějakým způsobem zachytit a odposlouchávat. Velmi zajímavou přednášku ohledně bezpečnosti message brokerů přednesl v květnu 2019 Petr Stuchlík na konferenci OWASP [22].

Pan Stuchlík v přednášce do detailu popisuje problémy, které nalezl při testování brokerů a protokolů, které brokery používají. Soustředí se na RabbitMQ, Kafku, ActiveMQ a další brokery postavené na JMS a na protokol MQTT používaný pro IoT. Pro vyhledání počtu zařízení používající daný broker používá nástroj Shodan, což je vyhledávací nástroj pro služby a počítače připojené k internetu.

Jak již víme, Kafka používá ZooKeeper, který slouží jako služba pro správu clusteru (správa konfigurace, pojmenování uzlů, synchronizace mezi uzly, atd.). ZooKeeper pro Kafku zajišťuje konfiguraci témat či správu odesílatelů a příjemců. ZooKeeper je ale také snadným přístupovým bodem pro nabourání se do Kafky a odposlech zpráv, jedná se o centrální bod celého clusteru a po nabourání tohoto bodu již není velký problém nabourat se do zpráv procházejících přes Apache Kafka.

Zookeeper je většinou vystaven na portu TCP/2181. Pomocí několika jednoduchých příkazů lze zjistit, zda na tomto portu ZooKeeper opravdu běží, podrobnosti o jeho prostředí a jaké brokery jsou na ZooKeeper napojeny. Následně pan Stuchlík představuje Python script, který dané příkazy provede a předloží uživateli informace o žádaném brokeru (IP adresu, port, a další). S těmito informacemi lze již zjistit detaily o jednotlivých tématech a konzumovat zprávy z vybraného tématu. Pan Stuchlík představuje další Python script, který umožní celý proces zautomatizovat a dokládá tím, že nabourání se do Apache Kafka je opravdu velmi snadné. Dle Shodanu je vystaveno na internet cca 43 000 instancí Apache Kafka [22].

Samotná dokumentace Kafky zmiňuje, že předpokládá běh brokeru v zabezpečené síti za firewallem a tudíž není vhodné vystavovat broker na internet. Kafka umí používat bezpečnostní certifikáty, ale nedělá to defaultně.

Pomocí přídavných pluginů, které ale nejsou součástí brokeru, podporuje ACL (Access Control List). Doporučeno je nevystavovat port, na kterém běží ZooKeeper, na internet. Další důležitá poznámka je, že Kafka neukládá zprávy ve frontách trvale (oproti tématům), to znamená, že pokud se podaří do fronty nabourat zvenčí, nejen, že se podaří zprávu odposlechnout, ale zpráva bude v tu chvíli z fronty „přijata“ a nedorazí očekávanému příjemci.

Další část prezentace je věnovaná brokeru RabbitMQ [22]. RabbitMQ byl vytvořen zejména pro komunikaci přes AMQP, ale používá i další protokoly (MQTT, WebSocket, STOMP). Dle Shodanu je vystaveno na internet cca 6 000 instancí RabbitMQ.

Na portu 15672 se lze přihlásit k RabbitMQ monitorovací konzoli. V přednášce se můžeme dozvědět, že do 12% z cca 4 500 testovaných instancí bylo možné přistoupit bez autentikace, nebo s vyplněním defaultních přihlašovacích údajů (guest / guest). V této konzoli lze vidět přehled všech front a témat včetně detailních podrobností. V tuto chvíli nic nebrání vytvořit odběratele, který bude zprávy z těchto front a témat odebírat. Některé instance nepoužívají TLS, přestože ho RabbitMQ podporuje, tudíž zprávy přijdou bez jakéhokoliv šifrování v plaintextu.

Pro zvýšení bezpečnosti je doporučeno upgradovat RabbitMQ na verzi 3.4.0 a vyšší (v předchozích verzích jsou známé bezpečností problémy, přesto 99% veřejně přístupných brokerů je nižší verze). Další doporučení jsou vynutit používání TLS 1.2 či vyšší, nepoužívat konzoli pro správu v produkčním prostředí a správná konfigurace a zabezpečení veřejně dostupných portů.

Dále se v prezentaci pan Stuchlík zaměřuje na MQTT, protokol pro komunikaci mezi chytrými zařízeními [22]. Tento protokol podporuje mnoho brokerů. Jedná se o protokol, který využívají chytré domácnosti, chytrá auta, Arduino počítače, či například sdílené bicykly. Vzhledem ke své povaze se jedná o co nejjednodušší protokol s jednoduchým principem Publish-Subscribe. Přes MQTT lze přenášet například JPEG obrázky z kamer, informace o vozidle (při připojení na řídící jednotku vozidla), GPS souřadnice z trackerů v autobusech, či dokonce zvukové záznamy z mikrofonu.

Přes MQTT mohou procházet velmi citlivé osobní údaje, které nemusí být správně zabezpečené. Pro zvýšení zabezpečení se doporučuje používat autentikace a TLS. MQTT může podporovat i symetrické či asymetrické šifrování zprávy mezi odesílatelem a odběratelem.

Poslední, na co se v přednášce podíváme, je bezpečnost JMS [22]. Zjištění veřejně vystavených brokerů není příliš jednoduché, protože každý používá lehce rozdílnou implementaci JMS. Lze ale zjistit, zda jsou brokery vystaveny na internet přes MQTT (například ActiveMQ používá jak MQTT, tak JMS).

Následně je možné vyzkoušet, zda má daný broker vystavený i OpenWire port, typicky 61616.

JMS přenáší zprávy v serializované podobě specifické pro Javu, tudíž teoreticky by bylo potřeba pro deserializaci znát Java třídy všech DTO (Data transfer object). Samozřejmě v praxi existuje další Python script, který nám umožní tohle obejít a dokáže ze serializované podoby zprávy zjistit co nejvíce.

Pan Stuchlík se v přednášce věnuje i bezpečnosti dalších brokerů a protokolů a ukazuje konkrétní příklady, které pro potřeby diplomové práce již není třeba uvádět. Závěrem analýzy bezpečnosti message brokerů je, že prostředky pro zabezpečení u většiny z nich existují, nicméně se příliš nevyužívají a tím se message broker stává slabým místem vhodným pro útok hackera. Většinou ovšem broker běží v uzavřené síti a není vystaven na internet, tudíž se o jeho bezpečnost starají jiné prvky.

Další zajímavou studii na bezpečnost JMS přináší Gursev Singh Kalra ze společnosti McAfee soustředící se na bezpečnostní software. V článku *A Pentesters Guide to Hacking ActiveMQ-Based JMS Applications* [14] se zabývá zabezpečením a možnostmi proniknutí do ActiveMQ. Po krátkém popsání vlastností ActiveMQ představuje čtyři bezpečnostní rizika:

1. ActiveMQ od verze 5.8.0 má monitorovací konzoli zabezpečenou pomocí defaultních přihlašovacích údajů, které jsou velmi často nezměněné.

2. Monitorovací konzole běží nad nezašifrovaným HTTP protokolem.

3. ActiveMQ před verzí 5.8.0 nepoužívala vůbec žádnou autentikaci pro přístup k monitorovací konzoli.

4. Transportní protokoly a monitorovací konzole defaultně poslouchají na IP adrese 0.0.0.0 (všechny IP adresy), což způsobí, že se lze připojit odkudkoliv.

Jako další riziko uvádí autor list známých odhalených bezpečnostních nedostatků. Autor se dále soustředí na samotné možnosti zabezpečení, popisuje, že ActiveMQ v základu nevyžaduje TLS a že hesla jsou v konfiguračních souborech uložena nezašifrována. Zároveň uvádí, že defaultně používaná databáze pro ukládání zpráv KahaDB nepodporuje šifrování uložených dat [14].

Na závěr textu autor přináší podrobné návody na otestování zabezpečení fronty a tématu, přičemž cílem je neautorizovaně přečíst zprávy ve frontě či tématu. Studie může sloužit jako velmi dobrý návod pro penetrační testování aplikací používající JMS.

Další studie *How to silently capture RabbitMQ messages* se zaměřuje na bezpečnost RabbitMQ. Autor Quentin Kaiser popisuje své zkušenosti z penetračního testování, kdy u všech RabbitMQ instancí vystavených na internet vždy byla vystavena i monitorovací konzole a často se setkal buď s defaultními přihlašovacími údaji, či byly údaje ve veřejně přístupném konfiguračním souboru. Autor zde popisuje podrobný způsob, jakým lze zjistit potřebné údaje z monitorovací konzole či vystaveného API (list klientů, front, témat, použitých protokolů, atd.).

V další části představuje modely pro odchytávání zpráv tak, aby připojení klienti nepoznali, že jsou zprávy odchytávány. Ve chvíli, kdy je k frontě připojen reálný konzument a připojí se další, který bude chtít odchytávat zprávy, odchycená zpráva by po odebrání z fronty byla smazána a nebyla by doručena na původní místo odeslání. Nástroj pro odchytávání zpráv tak musí zajistit poslání zprávy zpět do fronty, aby ji mohl obdržet i reálný konzument. Tento přístup ovšem nemusí odchytit úplně všechny zprávy, zejména ve frontě s vysokou průchodností zpráv.

Jednodušší je odchytávání zpráv, pokud broker používá fan-out nebo téma, v těchto případech zprávy dorazí všem konzumentům.

Závěrem autor představuje svůj program, který vše popsané dokáže vykonat automaticky pouze s poskytnutím IP adresy daného brokeru. Na závěr předkládá alarmující statistiky, kdy při testování téměř 5 000 vystavených brokerů na internet více než 18 % vystavuje monitorovací konzoli a z nich téměř 60 % nemá změněné defaultní přihlašovací údaje. Když se na čísla podíváme jinak, tak k 1 z 10 RabbitMQ instancí vystavených na internet se lze bez problému přihlásit za použití údajů *guest / guest*.

4 Použité technologie

Než se pustíme do praktické části diplomové práce, krátce si představíme několik nástrojů a technologií, které jsou použity v rámci vytvořené aplikace a pro testování message brokerů.

4.1 Docker

Začátkem si zavedeme pojmy používané při práci s Dockerem:

- *Docker* je služba pro vytváření a běh kontejnerů.
- *Kontejner* je spuštěné prostředí/služba vycházející z image s volitelně připojenými volumes.
- *Image* je název pro připravené prostředí nebo službu. Image je složen z několika vrstev.
- *Volume* je úložiště mimo kontejner napojené na kontejner.
- *Image* vrstva reprezentuje část image, vycházející z kroků v Dockerfile tvořící společně Image.
- *Dockerfile* je soubor pokynů podobný bash skriptu k vytvoření Image.
- *Docker-compose* je nástroj pro správu kontejnerů a jejich propojení.

V posledních letech zažívá Docker velký vzestup. Docker si lze představit jako odlehčený virtuální stroj blíže navázaný na hostitelský operační systém. Výhodou Dockeru je velké zjednodušení a zrychlení vývojářské práce. Každý jednotlivý vývojář nemusí od nuly rozbíhat celé prostředí, ale stáhne si připravený Docker Image a může začít pracovat. Zároveň tím urychluje vydání aktuální verze vyvíjené aplikace na záležitost několika sekund [15].

Kromě zrychlení Docker přináší i větší bezpečnost. Jednotlivé kontejnery jsou od sebe naprosto oddělené a vývojář má naprostou kontrolu nad tím, co běžícímu kontejneru umožní. Docker je důležitou součástí „Continous Integration“ procesu, podporuje propojení s nástroji automatizujícími build, jako například Jenkins.

Mezi nevýhody Dockeru patří menší rychlost, než spouštení aplikace na samostatném serveru. Docker stále nezvládne všechno, například spuštění

aplikací s grafickým prostředím, nebo kopírování dat mezi hostitelským počítačem a kontejnerem.

Životní cyklus Docker kontejneru je následující: Vývojář sepíše Dockerfile, což je posloupnost příkazů pro vytvoření Docker Image. Příkazem `docker build` se z Dockerfile souboru vytvoří Image a následujícím příkazem `docker push` ho lze nahrát do veřejného repozitáře.

V případě, že uživatel chce daný Image používat, příkazem `docker pull` stáhne danou verzi Image k sobě. Následně lze z Image vytvořit běžící kontejner za pomocí příkazu `docker start`. K tomuto příkazu lze přidávat různé parametry (vystavení portů, připojení volumes, jméno kontejneru a mnoho dalších specifických pro každý Image). Zastavení běhu kontejneru lze docílit příkazem `docker stop`.

Dále existuje nástroj Docker-compose, který umožňuje uložit konfiguraci kontejneru do `yaml` souborů. Tímto není potřeba při každém spuštění vypisovat jednotlivé parametry, ale spuštění a zastavení kontejneru lze docílit jednoduchými příkazy `docker-compose start/stop`. Dále přináší možnost provázat více kontejnerů (například server a databáze, běžící každý v jiném kontejneru). Toto přináší možnost vytvoření kompletního běhového prostředí složeného z mnoha komponent.

Docker v rámci práce použijeme pro spuštění jednotlivých message brokerů. Toto řešení zvolíme zejména z důvodu jednoduchosti spuštění a snadné replikovatelnosti vytvořeného prostředí (vývoj probíhá na lokálním PC, testování výkonnosti v cloudu).

4.2 Amazon Web Services

V této části se podrobněji podíváme na cloudové služby, konkrétně na Amazon Web Services.

Cloud se stal jedním z největších buzzwordů posledního desetiletí. Pro spoustu začínajících startupů se stal základním stavebním kamenem a v posledních letech své produkty převádějí do cloudu i velké společnosti, například banky.

Podíváme se na několik důvodů, proč je pro firmy cloud dobrým řešením. Cloud nevyžaduje kapitálové investice do serverového hardware a software vyžadující odepisování. Uživatelé dokáží předvídat provozní náklady. Tyto náklady se odvíjejí od požadavků, rostou nebo se snižují s počtem uživatelů a odebíraných služeb. Růst společnosti je možné dosáhnout s nižšími náklady na IT infrastrukturu. Cloud umožňuje rychlejší nasazení technologií. Zároveň zajišťuje větší bezpečnost dat a nižší prostoje. Uživatelé používající

cloud musí řešit méně manuálních aktualizací, mají zaručené automatické aktualizace na nové verze.

Použití cloudu uvolní pracovníky firem od rutinní práce s instalací, správou a zálohováním serverů a umožňuje soustředit se na rozvoj IT infrastruktury a jejího efektivního využívání uživateli. Firmy mohou využít kombinaci cloudového řešení s již existujícím, místně nasazeným řešením. IT infrastruktura může okamžitě reagovat na potřeby růstu společnosti. Použitím cloudu se zvýší produktivita zaměstnanců díky využívání nejnovějších verzí aplikací a serverových produktů. Zároveň se zlepší spolupráce a sdílení prostředků díky efektivnějšímu využívání aplikací. Mobilita obchodníků bude efektivnější díky zlepšení přístupu k datům během obchodních cest. Nižší náklady na IT a tedy i celkové provozní náklady umožní snížit ceny vlastních produktů a získat tak konkurenční výhodu.

Po vyčerpávajícím výčtu výhod a důvodů, proč by firmy měly používat cloudová řešení zde zmíníme i několik nevýhod a překážek, na které je třeba brát ohled. Je třeba si uvědomit, že citlivá data už nebudou uložena na firemním vlastním serveru, ale na serveru vlastněném cloudovým poskytovatelem, pravděpodobně v jiné zemi. Zde mohou platit odlišné zákony o nakládání s daty, což může způsobit, že již nebudou stoprocentně zabezpečena. Firma je odkázána na potřebné internetové připojení, bez nějž s cloudem pracovat nelze. Pokud dojde k výpadku serverů, musíme spoléhat na poskytovatele, že výpadek vyřeší.

Mezi největší poskytovatele cloudových řešení patří tyto společnosti:

- Amazon Web Services (AWS),
- Microsoft Azure,
- Google Cloud Platform,
- IBM Cloud (dříve Bluemix),
- Oracle,
- Alibaba Cloud.

Pro potřeby diplomové práce a otestování message brokerů zvolíme Amazon Web Services (AWS). AWS byl spuštěn v roce 2006. Jedná se o jeden z nejznámějších cloudových poskytovatelů pro tvorbu webových aplikací pomocí integrovaných webových služeb. Nabízí velice rozsáhlé množství IaaS a PaaS služeb. AWS nabízí podrobnou administrátorskou správu služeb pomocí webového portálu [26].

Velkou výhodou AWS je, že umožňuje upravovat a přizpůsobovat téměř jakoukoliv poskytovanou službu. Možnosti úprav jsou tak detailní, že pro nováčka je téměř nemožné se ve změti možností a služeb vyznat. Naštěstí AWS poskytuje velice podrobnou dokumentaci a články pro demonstraci funkcionality.

O jednotlivých službách AWS použitých v diplomové práci si řekneme dále v kapitole 7, kde si popíšeme i přesný způsob spuštění služby.

4.3 Apache Camel

Ve chvíli, kdy píšeme integrační projekt, potýkáme se s problémem, že máme mnoho aplikací na mnoha platformách s mnoha způsoby přenosu dat. Pokud bychom chtěli, aby spolu aplikace komunikovali napřímo, systém začne být velmi brzy velmi nepřehledný. Jako vývojáři bychom museli vymýšlet a optimalizovat způsoby, jakým spolu dané aplikace mohou komunikovat. Mnohem jednodušším řešením je použít prostředníka, se kterým budou aplikace komunikovat. Ideálně prostředníka, který dokáže jednoduše implementovat všechny způsoby, jakými lze mezi aplikacemi komunikovat. Zde přichází Apache Camel.

Apache Camel je open-source Java framework, který usnadňuje integrování dalších systémů do vyvíjené aplikace. Usnadňuje zakomponování message brokerů a v praxi se ve většině projektů setkáme právě s ním [13].

Camel je jako integrační nástroj využíváný z několika důležitých důvodů:

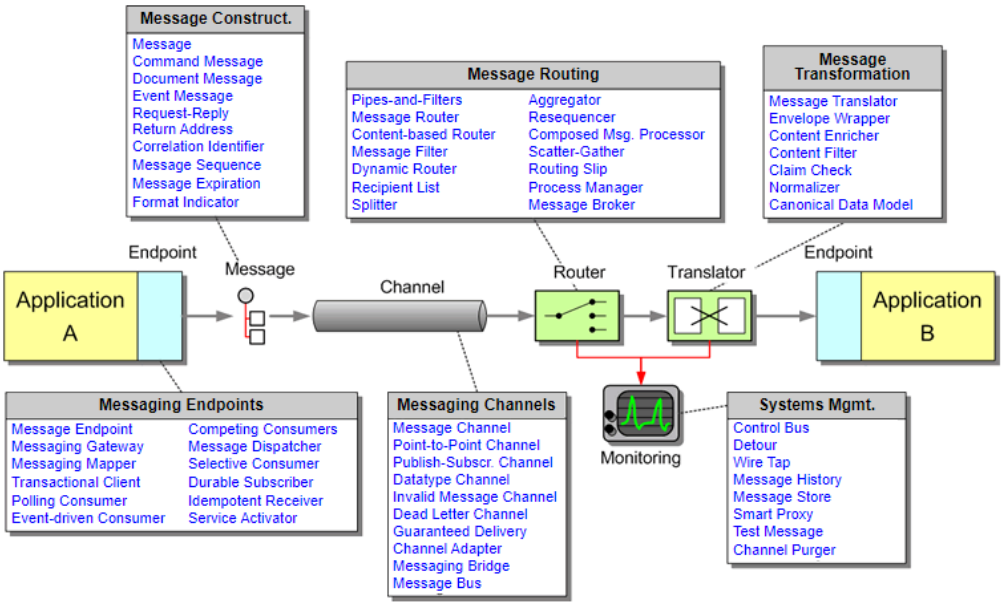
- Nabízí konkrétní implementaci všech široce používaných integračních vzorů (Enterprise Integration Patterns - EIP).
- Nabízí připojení k obrovské škále transportních protokolů.
- Nabízí nástroje pro jednoduché propojení EIP a transportních protokolů pomocí různých DSL (Domain Specific Language).

Vzhledem k vlastnostem a možnostem Camelu se jedná o ideální řešení při psaní enterprise aplikací. Camel bude centrálním prvkem aplikací Producer a Consumer.

4.3.1 Integrační vzory

Důvodem k vytvoření integračních vzorů je to, že samotný proces integrace více služeb je velmi složitý a náročný. Integrační vzory dávají návod na to, jak nejlépe řešit dané problémy. Je definováno 65 integračních vzorů [12].

Vzory lze rozdělit do několika částí. Přehledné rozdělení je vidět na Obr. 4.1.



Obrázek 4.1: Enterprise integrační vzory, zdroj: [11]

Na obrázku je vidět, že zpráva při průchodu z aplikace A do aplikace B prochází několika fázemi, a pro každou je definováno několik integračních vzorů. První skupina definuje samotné koncové body reprezentující aplikace A a B (Endpointy). Další skupina se zaměřuje na konstrukci zprávy. Když je zpráva vytvořena, je třeba definovat, jakým způsobem bude poslána k aplikaci B. Na to se zaměřuje skupina vzorů „Messaging Channels“, definuje se zde například, zda se jedná o Frontu či Téma, lze definovat Dead Letter Queue a další. Následující skupina vzorů pro „Router“ (směrovač) umožňuje například přeposílat zprávu na základě informací v hlavičce či těle a spojovat či rozdělovat zprávy. Do skupiny „Translator“ (překladač) patří vzory, které mění obsah samotné zprávy. Zde lze přidávat či mazat properties a hlavičky, měnit obsah těla a další. Poslední skupinou jsou vzory pro správu a monitorování systému, lze například dohledat kompletní historii zprávy od té doby, co byla vytvořena.

Camel implementuje více jak 40 těchto integračních vzorů [6]. Tyto vzory jsou v Camelu definované jako „Processory“. Všechno, co se se zprávou děje mezi přijetím od aplikace A a odesláním do aplikace B, obstarávají Processory. Kromě EIP Camel obsahuje další zajímavé Processory a v případě potřeby si vývojář může jednoduše vytvořit vlastní Processor.

4.3.2 Transportní protokoly

Transportní protokoly, jak již název napovídá, slouží k přenosu zpráv. Apache Camel umožňuje připojovat tzv. „komponenty“, které zaobalují transportní protokoly a umožňují Camelu napojit se na jiný systém. Pro jednoduché vystavení komponenty do zbytku Camelu slouží „Endpoint“. Endpoint je v Camelu definován pomocí URI. Celý proces průchodu zprávy vždy začíná tím, že Camel obdrží zprávu, a to právě z Endpointu. Zároveň většinou končí tím, že je zpráva poslána do jiného Endpointu.

Camel v době psaní tohoto textu obsahuje 332 komponent (většinu z nich jako připojitelný modul). Mezi základní implementované komponenty patří zcela jistě základní http, jms, či ftp, ale Camel nabízí připojení komponent například pro Git, spoustu AWS cloudových služeb, Dropbox, spoustu Google služeb (Google mail, Google drive, a další), Facebook, Salesforce, a spoustu dalších. Lze vytvořit mockovací endpoint pro testování zprávy, zprávu lze uložit jako soubor na disk nebo zapsat do logu, či ji pravidelně generovat pomocí Quartz časovače [6].

Každá komponenta má definováno, zda je možné z ní zprávy přijímat, nebo do ní zprávy odesílat, nebo oboje. Například testované message brokery umožňují přijímání i odesílání zprávy. Naopak například komponenta Timer umožňuje pouze zprávy generovat.

4.3.3 Propojení EIP a transportních protokolů

O snažší propojení mezi těmito dvěma částmi se snaží použití DSL. DSL znamená Domain Specific Language, Camel umožňuje používat více DSL. EIP a zároveň i Camel používají jako základní stavební kámen tzv. „Exchange“ - zprávu. Zpráva je objekt obsahující hlavičku, tělo a další parametry a informace. Camel kromě jiného pro zprávy definuje tzv. „Routes“ - cesty. Cesta začíná tím, že Camel obdrží zprávu. Následně lze se zprávou provádět téměř cokoliv, co umožňují EIP.

Tyto cesty není nutné definovat pouze v Java DSL, ale i Spring DSL (definováno v XML), nebo Scala či Groovy DSL. Je potřeba si uvědomit, že i pokud je cesta definována v Javě, nejedná se o vykonávaný kód, ale o pouhou specifikaci toho, jaké operace se zprávou provést. O samotné vykonávání se starají již zmíněné Processory v případě práce se zprávou a transportní protokoly v případě přijímání či odesílání [6].

4.4 Postman

Postman je multiplatformní aplikace, která původně vznikla jako rozšíření do prohlížeče. Slouží k návrhu a interakci s HTTP API. Ale umožňuje i psaní automatických testů, mock server, monitoring, tvorbu dokumentace.

V Postmanovi lze definovat HTTP požadavky, přičemž můžeme zahrnout parametry, hlavičky, možnosti autorizace, tělo zprávy, či další skripty. Postman umožňuje jednotlivé požadavky shromažďovat do kolekcí. V rámci kolekce lze vytvářet proměnné, které jsou následně v kolekcích použité. Kromě toho lze vytvořit různá prostředí (například `develop`, `preproduction`, `production`), ve kterých se hodnota proměnných liší, ale není třeba měnit celý požadavek.

5 Výběr implementací message brokerů

V předchozích dvou kapitolách jsme se seznámili s konceptem a implementacemi message brokerů a také s již existujícími studiemi na jejich porovnávání. V této kapitole si popíšeme konkrétní implementace message brokerů, které budou použity pro závěrečné porovnávání. V následující kapitole 6 si popíšeme kritéria pro jejich porovnání. Při výběru brokerů můžeme vzít v potaz následující faktory.

Prvním faktorem je možnost použití jako komponenty v enterprise řešení. Již jsme si představili několik brokerů, které jsou pro tento typ použití nevhodné a různé důvody proč. Jako základní požadavky můžeme brát robustnost, škálovatelnost, možnost způsobu zabezpečení, možnost ukládání zpráv na disk, zaručení doručení zpráv ve správném pořadí a kontrola doručení a případné přeposlání zprávy.

Dalším faktorem je možnost napojení na broker přes Apache Camel, nástroj popsaný v předchozí kapitole, který vývojáři výrazně ulehčuje komunikaci s brokery a definuje jednotlivé úkony, které je třeba se zprávou provézt.

Jako další faktor bychom mohli chtít použít žebříček použití, ale vzhledem k tomu, že většina běžících brokerů není vystavena veřejně na internet, je těžké zjistit přesná čísla. Většina článků ovšem uvádí skupinu nejpoužívanějších brokerů, ze které lze vycházet.

Dále můžeme vzít v potaz osobní zkušenosti z praxe, přičemž brokery použité v praxi jsou téměř stejné jako ty uvažované mezi skupinou nejpoužívanějších.

Nakonec bychom mohli zvolit alespoň jedno cloudové řešení. S nástupem cloudu v posledních letech bude potřeba takového řešení čím dál větší a je dobré zjistit, jak si povede oproti stávajícím zaběhlým řešením.

Na základě těchto kritérií pro porovnávání zvolíme následující message brokery:

- Apache ActiveMQ (enterprise řešení, nativní pro Apache Camel, JMS standard).
- RabbitMQ (nejčastěji používaný broker v enterprise řešení, AMQP protokol).

- Apache Kafka (vhodné pro clusterové řešení, použití commit logu).
- Amazon SQS (příklad cloudového řešení, běžící v AWS).

Aby bylo možné porovnávat brokery objektivně, budou muset běžet všechny ve stejném prostředí. Vzhledem k výběru cloudového řešení bude třeba i ostatní brokery spouštět v cloudu. Pro testování zvolíme cloudovou službu od Amazonu - Amazon Web Services (AWS).

5.1 Specifikace zvolených implementací

Zde si popíšeme specifikace implementací brokerů zvolených pro testování. V této sekci popíšeme pouze běhové prostředí, detaily ohledně Dockeru jsme si popsali v předchozí kapitole a jednotlivé použité služby Amazonu si vysvětlíme podrobně v kapitole 7.

5.1.1 Apache ActiveMQ

Apache ActiveMQ má dvě verze, klasickou a Artemis. Pro testování budeme používat klasickou verzi ActiveMQ 5.15.11 a verzi Artemis 2.11.0. Obě tyto verze poběží v dockerizované verzi v AWS.

Kromě toho vyzkoušíme službu od Amazonu, nazvanou Amazon MQ, postavenou na ActiveMQ. Tato služba podporuje jako poslední verzi ActiveMQ 5.15.10.

Apache nenabízí svoji vlastní verzi dockerizované ActiveMQ, proto budeme využívat Docker Image rmohr/activemq (odkaz na docker hub `https://hub.docker.com/r/rmohr/activemq/`) pro klasickou verzi ActiveMQ a vromero/activemq-artemis (odkaz na docker hub `https://hub.docker.com/r/vromero/activemq-artemis`) pro ActiveMQ Artemis. Oba tyto Docker Image budou spuštěny v Amazon ECS na FARGATE serveru s konfigurací 8 GB RAM, 2 vCPU.

AmazonMQ nabízí pouze několik možností serverů, na kterých lze broker spustit. Zvolíme instanci EC2 serveru typu m5.large s konfigurací 8 GB RAM a 2 vCPU.

5.1.2 RabbitMQ

Pro testování RabbitMQ budeme používat nejnovější verzi 3.8.2 v dockerizované verzi v AWS.

Amazon nenabízí vlastní službu pro RabbitMQ, proto si budeme muset vystačit s použitím Dockeru.

Pro RabbitMQ budeme používat Docker Image od Bitnami, konkrétně bitnami/rabbitmq (`https://hub.docker.com/r/bitnami/rabbitmq`). Tento Docker Image bude spuštěn v Amazon ECS na FARGATE serveru s konfigurací 8 GB RAM, 2 vCPU.

5.1.3 Apache Kafka

Apache Kafka je aktuálně ve verzi 2.4.0, tuto verzi budeme používat v Dockeru v AWS.

Kromě toho vyzkoušíme službu od Amazonu nazvanou Amazon MSK, postavenou na Apache Kafka.

Pro Apache Kafka budeme používat Docker Image od Bitnami, konkrétně bitnami/kafka (odkaz na docker hub `https://hub.docker.com/r/bitnami/kafka`). Tento Docker Image bude spuštěn v Amazon ECS na FARGATE serveru s konfigurací 8 GB RAM, 2 vCPU.

Amazon MSK nabízí pouze několik možností serverů, na kterých lze broker spustit. Zvolíme instanci EC2 serveru typu m5.large s konfigurací 8 GB RAM a 2 vCPU.

5.1.4 Amazon Simple Queue Service

Amazon SQS je služba běžící přímo v Amazonu, která běží samostatně a není třeba používat dockerizované prostředí. Vyzkoušíme jak klasickou frontu, tak FIFO frontu.

Amazon SQS nemá příliš mnoho možností konfigurace, není například možné zvolit typ a velikost serveru, na kterém chceme Amazon SQS používat. Nicméně Amazon zaručuje dostatečnou škálovatelnost, aby broker zvládl nápor a reagoval v rozumném čase.

6 Hodnotící kritéria

Z předchozí kapitoly již víme, na jaké implementace message brokerů se budeme soustředit. Nyní si na základě článků zmíněných v kapitole 3 specifikujeme, jak budeme vybrané message brokery porovnávat. Samotné porovnání můžeme rozdělit podle několika základních kategorií, přičemž v každé kategorii půjde o několik srovnávacích kritérií. Základní tři kategorie jsou:

- Výkonnost,
- bezpečnost,
- vývojářská přívětivost.

U každé kategorie podrobně rozebereme, na základě jakých kritérií budeme brokery porovnávat. Kritéria porovnání vychází jak z již zmíněných analýz v kapitole 3, tak z vlastních zkušeností s message brokery a z konzultací s panem Šimkem z firmy Intraworlds, který byl technickým konzultantem této práce.

6.1 Výkonnost

Výkonnost je nejčastěji zmiňovaná vlastnost message brokerů, kterou lze objektivně hodnotit a použít k porovnání. Do této kategorie patří již analyzovaná průchodnost a latence. Oba pojmy jsme si podrobně vysvětlili v kapitole 3. Průchodnost zde značí počet zpráv, který broker zvládne přijmout či odeslat za jednotku času, latence značí dobu mezi odesláním a přijetím zprávy. Obě hodnoty budeme měřit následujícím způsobem. Do brokeru pošleme velké množství zpráv a budeme čekat na jeho zpracování. Toto budeme provádět s různými velikostmi zpráv a to v následujících konfiguracích:

- 100 000 zpráv o velikosti 1 kB,
- 100 000 zpráv o velikost 16 kB,
- 100 000 zpráv o velikosti 64 kB,
- 100 000 zpráv o velikosti 256 kB.

Aplikace pro odesílání zpráv v tomto případě do fronty odešle všechny zprávy, změří dobu od odeslání první zprávy do odeslání poslední zprávy, ze

které zjistí průchodnost brokeru při přijímání zpráv. Každá zpráva bude ve své hlavičce obsahovat informaci o čase odeslání. Při přijetí zprávy se tato hodnota odečte od aktuálního času a daný rozdíl bude značit latenci doručení. Zároveň na straně příjemce zpráv budeme měřit čas od doručení první zprávy po doručení poslední. Tímto bude změřena průchodnost brokeru při odesílání zpráv.

6.2 Bezpečnost

Při porovnávání bezpečnosti message brokerů se budeme zabývat zabezpečením v uzavřené lokální síti, ale i zabezpečením při vystavení brokeru na internet.

Prvním hodnotícím kritériem bude možnost šifrovat zprávy při posílání mezi jednotlivými body (klient a server) a použití SSL/TLS certifikátů.

Dále nás bude zajímat, jakým způsobem se klient autentikuje při připojení k brokeru.

Následně bude třeba hodnotit, jestli broker podporuje autorizaci a autentikaci pro přihlášení do monitorovací konzole a jestli jde zabezpečit i ostatní vystavené porty.

Porovnáme u brokerů jejich základní konfiguraci - jak moc je broker zabezpečený bez zásahu a konfigurace od vývojáře, čili co může uniknout, pokud broker konfiguruje někdo, kdo s ním neumí.

Posledním kritériem bude možnost vystavení brokeru na internet. U jednotlivých brokerů uvedeme, zda je toto řešení vůbec doporučeno, případně co je potřeba provézt za kroky, aby byl vystavený broker zabezpečený.

6.3 Vývojářská přívětivost

Zřejmě nebude možné najít studii, která objektivně porovnává vývojářskou přívětivost, protože to už z principu není možné. Přívětivost pro vývojáře je subjektivní pojem a každý vývojář má jinou představu o tom, co je pro něj dostatečně přívětivé.

V této práci se nicméně zkusíme podívat na několik kritérií, podle kterých bude možné v této kategorii hodnotit co nejvíce objektivně.

Prvním kritériem bude konektivita, tedy přes jaké protokoly lze s danými brokery komunikovat.

Dalším kritériem bude pracnost počáteční konfigurace a následná správa brokeru. Zde popíšeme, co všechno je potřeba pro první spuštění brokeru a jak náročné je následně broker spravovat.

Následně můžeme u brokerů porovnat možnost použití monitorovacích konzolí. Zde nás bude zajímat, zda broker vůbec monitorovací konzoli poskytuje, zda existuje případné řešení třetí strany, co všechno konzole umí a jak těžké je ji používat. Toto kritérium zahrnujeme zejména proto, že s brokery velmi často musí pracovat člověk, který programovat neumí a monitorovací konzole je pro něj jediný způsob, jak přečíst informace.

Další kritérium je možnost spuštění brokerů v Docker prostředí. Zde nás bude zajímat kvalita existujících řešení, jejich použitelnost v praxi a možnosti nastavení.

Kromě Dockeru bude dalším kritériem pro porovnání možnost spuštění brokeru v cloudu. Kromě spuštění brokeru na klasickém cloudovém serveru nás bude zajímat, jestli největší poskytovatelé cloudových řešení mají pro daný broker vlastní řešení.

7 Program pro ověření sady hodnotících kritérií

V této kapitole podrobně prozkoumáme vyvinutý program pro otestování výkonnosti brokeru. Zároveň si díky poznatkům z předchozích kapitol uvedeme, co vše lze nakonfigurovat tak, aby byl broker zabezpečený, a to jak na straně brokeru, tak na straně klienta. Zatímco výkonnost je hodnota, kterou program dokáže změřit, bezpečnost bude pouze záležitostí různých konfigurací.

Celý projekt se skládá z několika částí:

- Broker - samostatně běžící message broker (jednotlivé typy popsané v předchozích kapitolách).
- Producer - samostatně běžící aplikace odesílající zprávy směrem do brokeru. Aplikace při odesílání měří průchodnost zpráv.
- Consumer - samostatně běžící aplikace přijímající zprávy z brokeru. Aplikace měří průchodnost zpráv při přijímání a a zároveň latenci jednotlivých zpráv.
- Initializer - aplikace, která spustí test pro danou konfiguraci a daný broker.

Jak jsme si již řekli, jednotlivé brokery poběží buď v dockerizované verzi, nebo přímo jako nativní služba v prostředí Amazon Web Services.

Aplikace Consumer a Producer jsou programy napsané v Javě, běžící jako Springboot aplikace postavené na frameworku Apache Camel. Při testování tyto aplikace poběží jako služba v AWS Elastic Beanstalk.

Aplikace Initializer má pouze na starost vyvolat událost v Producer aplikaci, která začne posílat zprávy s danou konfigurací do daného brokeru.

V dalších částech kapitoly si představíme strukturu celého projektu. Poté si postupně přiblížíme, co vlastně Docker je, jak funguje a jaké přináší výhody a nevýhody. Následně si podrobněji představíme Amazon Web Services a možnosti spuštění jednotlivých částí projektu v AWS prostředí a detailní postup konfigurace. Opět si uvedeme výhody a nevýhody tohoto řešení. Podíváme se na framework Apache Camel a jeho možnosti při práci s message brokery. Podrobně projdeme aplikace Producer a Consumer. Následně zhodnotíme možnosti spouštění testů a případné návrhy na vylepšení.

7.1 Struktura projektu

Projekt obsahuje 3 základní složky:

- `camel` - složka obsahující Java projekt zahrnující aplikace Producer a Consumer, více v části 7.3,
- `docker` - složka obsahující docker-compose soubory pro spuštění brokerů v rámci Docker prostředí, více v části 7.2,
- `postman` - složka obsahující kolekci a prostředí pro požadavky pro Postman aplikaci, více v části 7.4.

Pro vývoj použijeme verzovací nástroj Git. Celý projekt lze nalézt na `https://github.com/bombic94/diplomka`.

7.2 Běhové prostředí

V rámci diplomové práce poběží jak message brokery, tak vytvořená aplikace v Amazon Web Services. Použijeme celkem čtyři Docker image:

- `rmohr/activemq`,
- `vromero/activemq-artemis`,
- `bitnami/rabbitmq`,
- `bitnami/kafka`.

Bitnami je společnost zabývající se dodáváním balíčků zahrnujících cílený software, bohužel nemá Docker Image pro ActiveMQ ani Artemis, nicméně Image pro RabbitMQ a Apache Kafka využijeme. Pro ActiveMQ využijeme nejvíce používaný open-source Image od uživatele rmohr. Pro ActiveMQ Artemis využijeme open-source Image uživatele vromero. Ke každému z těchto Imagů vytvoříme docker-compose `yaml` soubor, kterým lze kontejner jednoduše spustit.

Budeme používat následující služby poskytované AWS:

- Amazon EC2,
- Amazon ECS,
- Amazon Elastic Beanstalk,
- Amazon SQS,

- Amazon MQ,
- Amazon MSK.

Jednotlivé části vytvořeného programu poběží na různých službách AWS. Aby byly zajištěny co nejlepší podmínky pro testování, je potřeba spuštěné služby zakomponovat do VPC (Virtual Private Cloud). VPC vytvoří uzavřenou skupinu služeb na privátní síti, ve které jdou tvořit takzvané „podsítě“ (subnets). Privátní síť lze vytvořit pro jeden Region (geografická oblast), přičemž nejbližší region je eu-central-1 se servery ve Frankfurtu.

7.2.1 Amazon EC2

Amazon Elastic Compute Cloud (Amazon EC2) je jedním z centrálních prvků AWS cloudové platformy. EC2 umožňuje uživatelům pronajímat si virtuální servery, na kterých může běžet jejich aplikace. EC2 umožňuje škálovatelné spuštění aplikací, poskytuje uživateli webové služby, skrz které uživatel může spustit takzvaný Amazon Machine Image (AMI). AMI dokáže automaticky nakonfigurovat virtuální stroj (tzv. „instanci“) obsahující žádaný software. Uživatel má možnost vytvářet, spouštět a spravovat instance běžících serverů dle potřeb, přičemž platí za čas běhu serveru. EC2 umožňuje uživatelskou kontrolu nad tím, v jaké geografické lokaci daná instance běží, což umožní optimalizaci latence.

Mnoho služeb, které Amazon nabízí, běží právě nad EC2, některé z nich si zmíníme dále. EC2 nabízí několik typů serverů:

- Server pro základní použití.
- Optimalizované pro výpočetní výkon.
- Optimalizované pro práci s pamětí.
- Optimalizované pro rychlost výpočtu.
- Optimalizované pro práci s úložištěm.

7.2.2 Amazon ECS

Amazon Elastic Container Service (Amazon ECS) je nástroj umožňující spouštět kontejnery v cloudu. Kontejnery lze spustit buď nad serverem EC2, nebo nad nástrojem AWS Fargate. Fargate je nástroj, který uživateli umožní spustit kontejner bez potřeby vytvářet pro něj server. Při použití Fargate uživatel platí pouze za zdroje, které opravdu využívá, a ne za celý server.

Kromě toho Fargate dostatečně zabezpečuje běžící kontejner a uživatel má naprostou kontrolu nad tím, na jakých portech je aplikace vystavená na internet.

Pro spouštění Docker kontejnerů v cloudu je ECS vhodnou volbou zejména díky usnadnění nasazení a konfigurace. Zde si uvedeme podrobný postup, který použijeme při spouštění brokerů v ECS.

Tak jako jsme si definovali službu běžící v Dockeru pomocí docker-compose souboru, tak v ECS je na velmi podobném principu možné definovat službu běžící v ECS pomocí „Task Definition“. Zde lze navolit, jaký Docker Image budeme používat, na jakých portech bude služba vystavena na internet, nastavení prostředí a mnoho dalšího.

Běžící Docker kontejner má v ECS ekvivalent ve formě tzv. „Tasku“ (služby). Task je definovaný pomocí Task Definition. Dá se říct, že Task je běžící instance Task Definition.

Pro spuštění Tasku v ECS je nejdříve potřeba vytvořit „Cluster“. Cluster je zjednodušeně řečeno skupina EC2 instancí. Ve vytvořeném Clusteru lze spouštět jednotlivé služby - „Services“.

Service může obsahovat jeden nebo více Tasků definovaných jedním Task Definiton. To znamená, že v Service může běžet více instancí daného Task Definition.

Prvním krokem pro spuštění brokeru v ECS je tedy vytvořit Cluster.

Po vytvoření Clusteru lze přistoupit k definici Task Definitions, kde pro každý broker bude třeba vytvořit jednu definici. V definici je třeba určit jméno, použitý Docker Image, zda je Task vhodný pro použití v EC2 či Fargate. Je třeba nastavit porty, na kterých bude běžící Task vystaven. Každý Task Definition má jednotlivé revize. Každá konfigurace je uložena pod danou revizí a při editaci Task Definition se vytvoří revize nová, přičemž předchozí jsou ukládány.

Ve chvíli, kdy máme připraven Cluster a Task Definiton, můžeme vytvořit Service. Při vytváření je nutné zvolit, jaký Task Definition zvolit pro spuštění Tasku v Service, o jakou se jedná revizi, kolik instancí Tasku (v rámci testování brokerů stačí jeden Task), název Service a typ serveru (EC2 či Fargate). Následně je potřeba nakonfigurovat vytvoření Security Group. Ta určuje, zda bude služba zakomponována do VPC, zda může být vystavena na internet a seznam portů, přes které se lze na službu připojit. Dále lze při vytvoření služby definovat další možnosti, které v rámci diplomové práce nepokryjeme (např. Load Balancing nebo Route53). Posledním krokem je samotné spuštění Service. Při spuštění se postupně vytváří jednotlivé Tasky (v tomto případě pouze jeden). V detailu spuštěného Tasku lze vidět logy, dále informace o veřejné i privátní IP adrese, které budeme potřebovat pro

napojení na broker pro programy Producer a Consumer.

7.2.3 Amazon Elastic Beanstalk

Amazon Elastic Beanstalk je služba sloužící pro jednoduché spuštění aplikací napsaných v programovacích jazycích Java, .NET, PHP, Node.js, Python, Ruby, nebo Go. Pro tyto aplikace jsou připraveny nakonfigurované servery, jako například Apache Tomcat, Apache HTTP server, Nginx a další.

Jedná se o tzv. orchestrační službu, což znamená, že spouští aplikace, které mohou vyžadovat více AWS služeb (například server ve spojení s databází, úložištěm, notifikační službou, pokročilým logováním, load balancingem, atd). Elastic Beanstalk si lze představit jako abstraktní vrstvu nad serverem a OS, uživatel zde již vybírá předkonfigurovaný server, na kterém může jeho aplikace běžet (v našem případě Java ve spojení s Apache Tomcat).

V rámci diplomové práce použijeme Elastic Beanstalk pro spuštění aplikace Producer a Consumer (detaily o nich v další kapitole). Obě aplikace můžeme spustit jako samostatnou `jar` aplikaci. Pro aplikaci je nejdříve potřeba vytvořit prostředí. Prvním krokem je zvolit, o jaký typ prostředí se jedná (webový server, nebo tzv. „Worker“ - prostředí pro dlouhotrvají procesy). Po zvolení webového serveru je potřeba nadefinovat jméno, název domény pro registraci v DNS a platformu, na které aplikace poběží. V našem případě je zvolená platforma Java.

Následně je třeba vytvořit samotnou aplikaci. Aplikace lze jednoduše nahrát jako `jar` soubor z počítače. Při běhu serveru lze nahrávat nové verze aplikace, přičemž po nahrání se server restartuje a nasadí novou verzi, není tedy třeba vytvářet pro každou novou verzi nové prostředí. Elastic Beanstalk si automaticky jednotlivé aplikace verzuje.

Po nahrání aplikace již lze přistoupit ke spuštění s přednastavenými parametry, nebo přejít do pokročilé konfigurace a zde si parametry upravit. Možností je opravdu hodně, lze upravovat například běžící SW serveru, typ serveru, úložiště, databázi, zabezpečení, automatické aktualizace a mnoho dalšího. V našem případě je potřeba upravit typ instance serveru, místo přednastaveného typu t2.micro aplikace spustíme na t2.medium. Instance t2.medium má 2 virtuální jádra a 4 GB RAM, což bude pro běh aplikací Producer a Consumer bohatě stačit. Další částí, kterou je třeba upravit, je zapojení do sítě. Je potřeba aplikaci zakomponovat do stejné VPC a podsítě, v jaké jsou spuštěné brokery, aby na sebe části aplikace viděly v rámci lokální sítě. Je třeba vystavit veřejnou IP adresu, aby se na aplikace dalo provolat z internetu, posílat požadavky na spuštění testů a číst výsledky.

7.2.4 Amazon SQS

O Amazon SQS jsme si již teoreticky řekli mnoho v kapitole 4. V tomto případě AWS poskytuje Simple Queue Service jako předpřipravenou službu s minimální konfigurací, pro spuštění Fronty stačí zadat název a typ (standardní nebo FIFO).

Frontu lze před spuštěním upravit v pokročilém nastavení. Lze nastavit doba viditelnosti zprávy ve frontě po odeslání, doba udržení zprávy, která nebyla nikdy odeslána, maximální velikost zprávy a další. Lze definovat tzv. „Dead Letter Queue“, kam se zprávy zařadí, když se je nepodařilo odeslat a dojde k pokusu o opakované odeslání. Následuje možnost zvolit si šifrování zpráv na straně serveru a případné vytvoření šifrovacího klíče. Následně je fronta spuštěna a ve webové konzoli lze vidět veškeré informace a vytvořenou Frontu případně upravovat či smazat.

7.2.5 Amazon MQ

Amazon MQ je služba postavená na ActiveMQ, která umožňuje vytvořit plnohodnotný message broker v AWS cloudu. Podobně jako Amazon SQS cílí na co největší jednoduchost spuštění a odstínění vývojáře od konfigurace.

Při vytváření brokeru si lze vybrat mezi jednou běžící instancí nebo clusterem brokerů. Cluster zajišťuje, že pokud u jedné instance brokeru dojde z nějakého důvodu k výpadku, její úlohu může převzít další instance. Následně si lze zvolit, zda bude broker optimalizován pro propustnost nebo pro „zachování zpráv“. To znamená, že zprávy budou redundantně ukládány napříč několika zónami a sdíleny mezi brokery v clusteru. Aby byly podmínky co nejvíce podobné brokerům spuštěným v ECS kontejnerech, zvolíme kombinaci jedné běžící instance s optimalizací pro propustnost.

Následuje konfigurace zvoleného brokeru. Je potřeba specifikovat název, typ instance serveru a přístupové údaje. Typ instance bude v tomto případě mq.m5.large, který má 2 virtuální jádra a 8 GB RAM. Přístupové údaje slouží jak pro přístup do konzole, tak pro přihlášení aplikací Producer a Consumer k brokeru a zasílání zpráv.

V detailním nastavení lze zvolit verzi ActiveMQ, kterou má broker používat (poslední a zvolená je 5.15.10). Kromě dalších možností konfigurace je třeba zakomponovat broker do VPC. Dále je možné vystavit broker na internet (umožní přístup do monitorovací konzole).

7.2.6 Amazon MSK

Amazon Managed Streaming for Apache Kafka (MSK) je, jak již název napovídá, služba postavená na Apache Kafka. Amazon se stejně, jako v případě MQ či SQS, snaží o co nejjednodušší spuštění samotné služby a odstraňuje problémy při konfiguraci služby.

Amazon MSK používá clustery. Prvním krokem je tedy vytvoření clusteru. Kromě názvu je třeba zvolit, jakou verzi Apache Kafka chceme používat, doporučená je verze 2.2.1, kterou zvolíme. Následně můžeme konfigurovat nastavení clusteru, nebo ponechat základní přednastavené hodnoty od AWS.

Další část je zapojení do VPC, zde musíme cluster zakomponovat do VPC, aby komunikoval s ostatními částmi aplikace. Poté musíme vybrat typ serveru, na kterém jednotlivé instance clusteru poběží. Zvolíme kafka.m5.large, který má 2 virtuální jádra a 8 GB RAM. Následně můžeme nakonfigurovat TLS pro šifrování zpráv a způsob jakým bude broker přijímat zprávy (pouze šifrované, pouze nešifrované, nebo oboje).

Posledním krokem je nastavení monitorování clusteru a následně můžeme cluster spustit.

7.3 Aplikace pro porovnávání message brokerů

Po krátkém představení frameworku Apache Camel se již dostáváme k vytvořené aplikaci pro porovnávání message brokerů, která je na něm postavena. Aplikace je psaná v programovacím jazyce Java (verze 1.8.231) a je postavena na Spring Boot (verzi 2.2.2). Pro sestavení aplikace je použit buildovací nástroj Maven. Aplikace používá Camel ve verzi 3.0.0.

Aplikace je rozdělena na 3 moduly:

- `common`,
- `producer`,
- `consumer`.

Modul `common` obsahuje třídy, které jsou použity jak v modulu `producer`, tak `consumer`. Jedná se o Enum obsahující všechny použité brokery (jejich název a URI daného endpointu). Dále třídu `BaseProcessor`, která obsahuje metody pro zpracování zprávy (Exchange), které využívají Processory ve zbývajících modulech. Poslední třídou je `IOUtils`, která se stará o ukládání naměřených výsledků do souboru.

7.3.1 Odesílání zpráv

Modul `producer` se stará o produkování zpráv a odesílání do brokeru. Zároveň má na starost měřit propustnost brokeru při přijímání zpráv.

O spuštění aplikace se stará vstupní třída `CamelProducerApplication`. Dále projekt obsahuje dva balíčky, `route` a `processor`. Třídy v balíčku `route` definují již zmíněné cesty pro zprávy. Každá třída vytvářející cestu - Route musí implementovat `RouteBuilder`, který definuje důležitou metodu `configure()`. V této metodě jsou definovány jednotlivé kroky, od přijetí zprávy až po její odeslání. Zde bude uveden kód jedné cesty pro zprávu. Všechny třídy zde uvedené fungují principiálně stejně, přičemž každá je rozdílná pouze v pojmenování brokeru.

Kód 7.1: Kód AmazonSQSRoute

```
1  @Component
2  public class AmazonSQSRoute extends RouteBuilder {
3
4      @Override
5      public void configure() throws Exception {
6          from("servlet:amazonsqs")
7                  .routeId("AmazonSQS in")
8                  .convertBodyTo(String.class)
9                  .removeHeaders("*", "iterationNumber", "size")
10                 .log(LoggingLevel.INFO, "Start sending messages")
11                 .process("startProcessor");
12     }
13 }
```

Projdeme si jednotlivé řádky kódu 7.1. Na prvním řádku je anotace `Component`. Tato anotace říká Springu, aby ji autodetekoval při spuštění aplikace. V rámci Camel projektu to znamená, že Spring při spuštění vytvoří AmazonSQSRoute s popsanou konfigurací.

Metoda `configure()` je přepisována z děděné třídy `RouteBuilder`. Zde jsou definovány jednotlivé kroky dané cesty. Začínáme definováním kroku `from("servlet:amazonsqs")`, tento krok říká, že se Camel napojí na komponentu servlet, která zachycuje http požadavky na endpointu "amazonsqs". Na tento endpoint budeme posílat žádosti z části aplikace Initializer pro spuštění testu.

Další krok určuje identifikátor cesty, zvolil jsem systém pojmenování „Název brokeru, směr“. Směr je buď in, nebo out, dle toho, zda zprávu do brokeru posíláme či přijímáme.

Následuje metoda pro konverzi těla zprávy do Stringové reprezentace. Může se stát, že tělo přijde jako pole bytů, s čímž si některé brokery nedokážou poradit a zároveň chceme s tělem v programu zacházet jako se Stringem.

V dalším kroku Camel odstraní zbytečné hlavičky, které přijal a zanechá pouze ty potřebné (podrobnosti dále).

Následuje ukázka logování v rámci definice cesty, informace pro uživatele, že začíná proces posílání zpráv.

Posledním krokem cesty je předání zprávy třídě `StartProcessor`. Za zmínku stojí, že Processor stačí uvést takto ve Stringové reprezentaci, není třeba Autowirovat samotný Processor, Camel už se o propojení postará sám. V této cestě není definováno volání `to(endpoint)`, protože o to se stará samotný processor, nicméně ve většině případů endpoint specifikujeme.

Nyní se můžeme přesunout k třídě `StartProcessor`, která se stará o zasílání zpráv do brokeru. Každý uživatelem definovaný Processor musí implementovat Camel interface `Processor` a implementovat metodu pro práci se zprávou `process(exchange)`. V této metodě lze specifikovat vlastní část procesu se zprávou, kterou Camel sám o sobě neposkytuje. Cílem pro otestování je několikrát poslat zprávu dané velikosti do brokeru (zde se dostáváme k hlavičkám `iterationNumber` a `size` specifikující počet iterací a velikost zprávy).

`StartProcessor` vyextrahuje ze zprávy dané hlavičky a vytvoří objekt `ProducerTemplate` - Camel implementaci EIP pro odesílání zpráv. Processor dokáže ze zprávy vyčíst ID cesty (například v předchozím případě „AmazonMQ in“). Na základě toho zvládne zjistit, o jaký message broker se jedná. Následně uloží aktuální čas jako čas začátku odesílání zpráv.

Poté přichází na řadu cyklus o zadaném počtu iterací, kde v každé iteraci zprávě vygenerujeme unikátní identifikátor (musíme si uvědomit, že Camel na začátku obdržel pouze jeden objekt zprávy ze servlet komponenty a tuto zprávu posílá stále dokola, nevytváří novou. Aby broker mohl zprávy od sebe rozeznat, je třeba jim přiřadit unikátní ID) a následně do hlavičky uložíme pořadí zprávy (číslo iterace) a čas odeslání zprávy (aktuální čas).

Nakonec přichází na řadu samotné odeslání zprávy pomocí této metody `template.send(endpoint, exchange)`. Endpoint je definovaný pomocí URI, který je uložen v enumu `MessageBroker`. O jaký broker se jedná, víme pomocí ID cesty.

V případě, že iterací je více než jedna, procesor změří průchodnost zpráv odečtením aktuálního času od času začátku odesílání a vydělením počtu iterací výsledným časem. Následuje uložení výsledků do souboru.

7.3.2 Přijímání zpráv

Modul `consumer` zajišťuje přijímání zpráv, měření latence jednotlivých zpráv a měření propustnosti při odesílání zpráv z brokeru.

Struktura aplikace je velice podobná aplikaci `producer`. Spuštění aplikace je docíleno vstupní třídou `CamelConsumerApplication`. Opět projekt obsahuje balíčky `route` a `processor`. Opět platí, že třídy definující cesty v balíčku `route` jsou strukturou velice podobné a liší se pouze v použitém message brokeru. V následujícím kódu 7.2 si opět projdeme jednotlivé řádky.

Kód 7.2: Kód AmazonSQSOutputRoute

```
1  @Component
2  public class AmazonSQSOutputRoute extends RouteBuilder {
3
4      @Override
5      public void configure() throws Exception {
6          from(MessageBroker.AmazonSQS.getEndpointUri())
7                  .routeId("AmazonSQS out")
8                  .aggregate(constant(true), new
                       CustomAggregationStrategy())
9                      .completionSize(100000)
10                     .completionTimeout(3600000)
11                 .log(LoggingLevel.INFO, "Generating results")
12                 .process("resultsProcessor")
13                 .to("mock://out");
14     }
15 }
```

Metoda `configure()` začíná definicí endpointu `from(uri)` obsahující URI endpointu AmazonSQS (reálně: "aws-sqs:standardAwsQueue"). Následuje definice ID cesty.

Poté přichází na řadu definice agregování zpráv, detaily budou rozebrány dále. U agregování definujeme agregační strategii, počet zpráv pro dokončení, případně timeout pro dokončení (co nastane dříve).

Následuje logování informace o tom, že se vytváří výsledky a předání zprávy procesoru `resultsProcessor`. Poslední krok ukazuje, že i pokud není potřeba zprávu nikam dále posílat, lze ji poslat do mockového endpointu.

V balíčku `processor` se nachází 2 třídy: třída `ResultsProcessor` a třída `CustomAggregationStrategy`.

Nejdříve se podíváme na strategii pro agregaci zpráv. Třída musí implementovat `AggregationStrategy` (Camel implementace EIP pro agregaci zpráv) a implementovat metodu `aggregate(oldExchange, newExchange)`,

která ze dvou zpráv vytvoří jednu na základě definovaných pravidel a tuto zprávu vrací jako návratovou hodnotu. Metoda `aggregate` se zavolá při přijetí každé zprávy. Postup agregace je následující:

1. Zjistí se aktuální čas - čas přijetí zprávy.
2. Z hlavičky zprávy se zjistí číslo zprávy a čas odeslání zprávy.
3. Vypočte se rozdíl mezi časem přijetí a časem odeslání, což značí latenci zprávy v brokeru.
4. Pokud se jedná o první přijatou zprávu (objekt `oldExchange` je null):
   (a) Vytvoří se nový List latencí a přidá se do něj první naměřená hodnota.
   (b) Tento list se uloží do těla zprávy (již není třeba tělo o dané velikosti).
   (c) Vytvoří se hlavička, do které se uloží čas přijetí zprávy - jedná se zároveň o čas začátku celého procesu přijímání zpráv.
   (d) Návratovou hodnotou je `newExchange`.
5. Pokud se nejedná o první přijatou zprávu:
   (a) List latencí se získá z těla `oldExchange` a přidá se do něj další naměřená hodnota.
   (b) Pokud je velikost listu stejně velká jako počet iterací (přijaty všechny zprávy):
      i. Nastaví se property říkající, že agregace je kompletní - přepíše i podmínky v definici cesty.
      ii. Vypočítá se propustnost na základě informace o času přijetí první zprávy uloženého v hlavičce a času přijetí poslední zprávy.
      iii. Propustnost se uloží do hlavičky zprávy.
   (c) Návratovou hodnotou je `oldExchange`.

Po proběhnutí agregace se výsledná zpráva předává ke zpracování třídě `resultsProcessor`. Zde dochází ke generování výsledků. Procesor nedělá nic zásadně zvláštního, pouze ze zprávy vyextrahuje informace v hlavičce (původní velikost těla zprávy, počet iterací, průchodnost) a informace v těle (List všech latencí) a zavolá metody třídy `IOUtils` pro uložení výsledků. Třída zároveň generuje základní statistiky a vypisuje je do konzole.

7.3.3 Konfigurace

Camel umožňuje konfigurovat jednotlivé komponenty přímo v kódu, nebo pomocí konfiguračních souborů, což se hodí zejména při kombinaci Camel a Spring-boot. Spring-boot umožňuje uvést všechny parametry aplikace (*properties*) v souboru `application.properties` nebo `application.yaml`, v závislosti na formátu, který chceme použít. Vzhledem k velkému množství parametrů je přehlednější použít `application.yaml` soubor. Tento soubor odděluje jednotlivé úrovně zanoření pomocí odsazení a není třeba opakovat celou cestu pro konfiguraci každé vlastnosti. Aplikace Consumer a Producer mají až na drobné odlišnosti konfigurace velmi podobné, proto zde projdeme pouze jeden z těchto souborů:

Kód 7.3: Konfigurace application.yaml

```
camel:
  component:
    activemq:
      broker-u-r-l: tcp://localhost:61616
      username: ********
      password: ********
      async-consumer: true
      concurrent-consumers: 20
      max-concurrent-consumers: 20
      delivery-persistent: true
      request-timeout: 600000
      use-message-i-d-as-correlation-i-d: true
    aws-sqs:
      configuration:
        access-key: *****
        secret-key: *****
        region: EU_CENTRAL_1
        message-attribute-names: All
        attribute-names: All
        concurrent-consumers: 20
```

V kódu 7.3 vidíme, že pro každý broker je konfigurace odlišná. U každého brokeru kromě AWS SQS je základ stejný - specifikujeme URL (adresu a port), uživatelské jméno a heslo pro autentikaci klienta u brokeru. U všech brokerů definujeme delší timeout na požadavek, je zde nastavený na 10 minut (600 000 milisekund) pro případ, že by broker byl zahlcen. U všech brokerů je nastaven počet paralelně běžících vláken pro příjem zpráv (`concurent-consumers`) na 20. V případě RabbitMQ nelze tuto hodnotu de-

finovat zde, ale přímo v url Endpointu. V dalších detailech se již konfigurace liší.

V případě ActiveMQ se jedná o konfiguraci jak pro klasickou ActiveMQ, tak pro verzi Artemis. Kromě výše zmíněných vlastností je zde třeba explicitně nadefinovat persistenci zprávy, zda bude příjemce zpráv přijímat zprávy asynchronně (u ostatních brokerů jsou obě hodnoty automaticky nastavené na `true`). Z důvodů specifických pro JMS je potřeba nastavit parametr, který umožní používat ID zprávy jako korelační ID (jak jsme již zmínili, Camel posílá stále stejnou zprávu - stejné korelační ID, ale nastavuje ji pokaždé různé ID).

V případě AWS SQS není třeba uvádět ani url brokeru, ani přihlašovací údaje. Camel zjistí konfiguraci brokeru dle přihlašovacích klíčů, které lze vygenerovat pro uživatele majícího účet na AWS. Následně je třeba určit, v jakém Regionu broker běží. Konfigurace pro jména attributů říká, aby AWS SQS broker přijímal a vracel všechny hlavičky ve zprávě, ve výchozím nastavení je maže a nastavuje své.

Kód 7.4: Konfigurace application.yaml

```
    kafka:
      configuration:
        consumer-request-timeout-ms: 600000
        brokers: localhost:29092
        securityProtocol: SASL
        ssl-truststore-password: *******
        ssl-truststore-location: {location}
        ssl-keystore-password: ********
        ssl-keystore-location: {location}
    rabbitmq:
      hostname: localhost
      port-number: 5672
      username: ********
      password: ********
      request-timeout: 600000
server:
  port: 5000
```

Navazující kód 7.4 zobrazuje konfiguraci Kafky a RabbitMQ. V případě brokeru Kafka je potřeba nakonfigurovat, aby broker používal SASL protokol a následně cesty a hesla k `.jks` souborům obsahujícím informace potřebné pro zabezpečený SSL přenos.

V konfiguraci pro RabbitMQ není třeba nic přidávat, počet vláken pro

příjem je potřeba nadefinovat v URL Endpointu a ostatní hodnoty pro konfiguraci (persistence zpráv, atd.) jsou ve výchozím nastavení konfigurovány dle našich požadavků.

Poslední nastavení určuje port, na kterém celá aplikace bude vystavena. Je potřeba nastavit 5 000, jelikož na tomto portu Elastic Beanstalk vystavuje běžící aplikace na internet.

7.3.4 Spouštění testů

V této sekci si uvedeme princip spouštění testů. Testy se spouští provoláním HTTP požadavku směrem na aplikaci Producer. Ten poslouchá na daných endpointech za pomoci komponenty `servlet`. Pro vytvoření a uložení HTTP požadavků využijeme program Postman.

V rámci diplomové práce pro posílání požadavků do aplikace Producer vytvoříme jednu kolekci obsahující 4 požadavky (zprávy s tělem různých velikostí - 1 kB, 16 kB, 64 kB, 256 kB).

Dále vytvoříme 4 různá prostředí (1 prostředí pro každý broker), lišící se URL adresou, na kterou budeme posílat požadavky. Poté budeme používat jednu globální proměnnou, a to je počet iterací. Pro otestování základní funkčnosti nemá smysl posílat hned 100 000 požadavků, ale pouze několik, abychom zjistili, zda vše funguje, jak má.

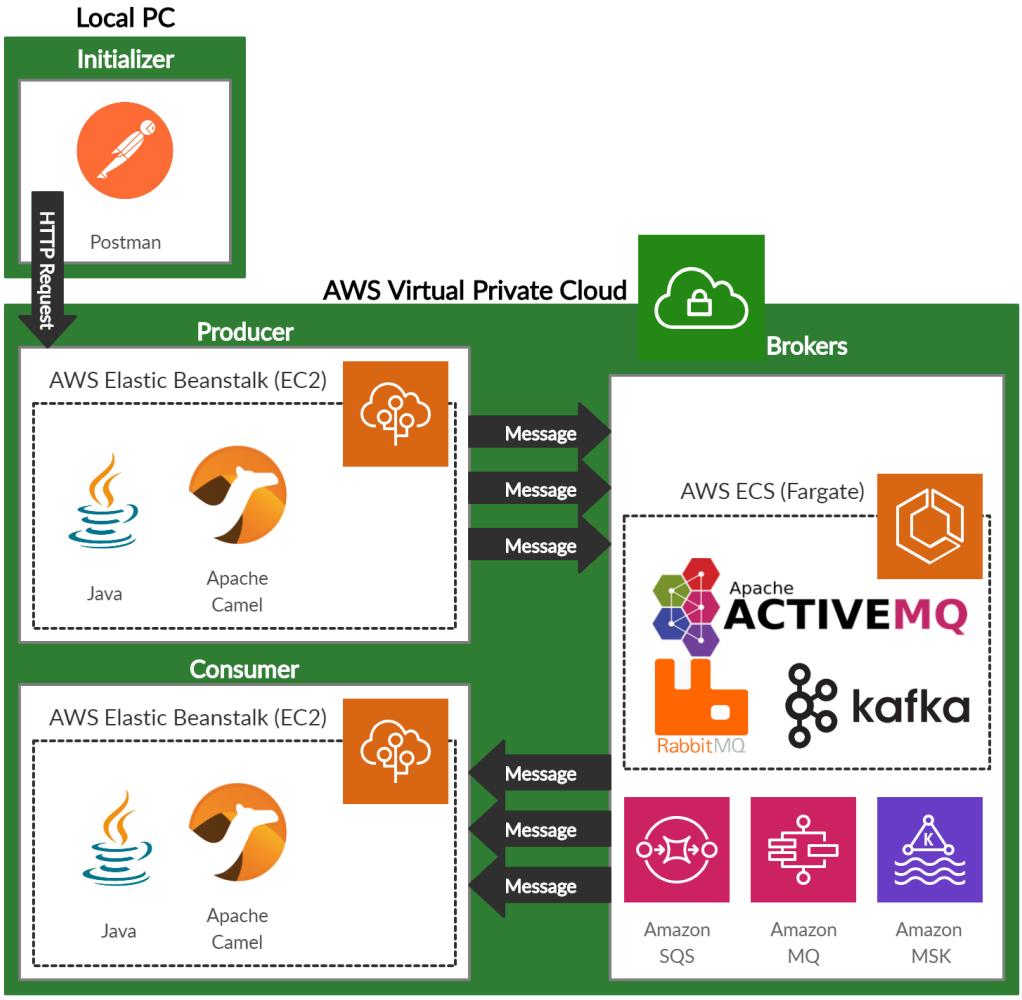
Posláním požadavku z Postman aplikace se spustí proces generování zpráv pro daný broker a testování. Určitě bychom dokázali vymyslet automatizovaný způsob testování, nicméně výhoda použití Postmanu je naprostá kontrola a velké možnosti úprav samotné zprávy (hlavičky, tělo, parametry).

7.4 Možnosti benchmarkování

V předchozích sekcích jsme se seznámili s jednotlivými částmi celé aplikace pro testování. Krátce jsme si představili použité technologie a jejich praktické využití v rámci projektu. Zde si povíme, jak jednotlivé části zapadají dohromady, představíme si celou strukturu projektu a upřesníme si možnosti benchmarkování.

Pro lepší představu propojení jednotlivých částí slouží obrázek 7.1. Na obrázku vidíme, že většina programu při testování poběží v rámci cloudových služeb AWS. Jediná část běžící na lokálním PC je Initializer. Jak jsme si již popsali, Initializer používá nástroj Postman, který pro každý test pošle HTTP požadavek do aplikace Producer.

V rámci AWS Virtual Private Cloud (VPC) běží zbývající 3 části programu (Producer, Consumer a testované brokery).



Obrázek 7.1: Schéma aplikace

Producer a Consumer (Java programy postavené na frameworku Apache Camel) každý běží na vlastním EC2 serveru pod správou Elastic Beanstalk.

Producer posílá zprávy do message brokeru a Consumer je z message brokeru přijímá. Message brokery běží buď jako předpřipravené Docker Image nakonfigurované a spuštěné v rámci Amazon ECS na Fargate serveru (Apache ActiveMQ, RabbitMQ, Apache Kafka, každý broker má vlastní server), nebo jako nativní AWS služby (Amazon SQS, Amazon MQ, Amazon MSK), v tomto případě se o samotný běh stará Amazon v případě Amazon SQS, nebo jsou služby spuštěné na EC2 serveru v případě Amazon MQ a Amazon MSK.

Pojďme si to celé představit na konkrétním příkladu: Chceme otestovat výkonnost brokeru ActiveMQ pro 100 000 zpráv o velikosti 1 kB. V Postmanovi zvolíme prostředí ActiveMQ, které obsahuje URL adresu, na které po-

slouchá Producer (servlet zpracovává požadavky na Endpointu, kterým začíná cesta `ActiveMQ in`). V požadavku nastavíme hlavičky „počet iterací“ (`iterationNumber`) na 100 000 a „velikost“ (`size`) na 1 kB a pošleme požadavek s tělem o velikosti 1 kB.

V tuto chvíli Producer zaregistruje nový požadavek s dvěma hlavičkami vyplněnými Postmanem. Zpráva přišla na Endpoint pro ActiveMQ, proto projde cestou `ActiveMQ in`. Na konci cesty je zpráva předána třídě `StartProducer`. Zde je ve smyčce zpráva 100 000 krát poslána do message brokeru ActiveMQ. Producer uloží hodnotu propustnosti při přijímání zpráv.

Následně zprávy procházejí samotným brokerem, v našem případě ActiveMQ, ze kterého je konzumuje aplikace Consumer. Zprávy jsou agregovány, dokud se nedosáhne počtu zpráv uvedených v hlavičce (`iterationNumber`). Po přijetí poslední zprávy třída `ResultsProcessor` vygeneruje výsledky latence a propustnosti při odesílání zpráv.

Po proběhnutí celého procesu můžeme v Postmanovi spustit další test (jiný broker, jiná velikost zpráv, jiný počet zpráv).

Nyní, když jsme si vysvětlili průběh benchmarkování, musíme před samotným zahájením testování a analýzou výsledků vzít v potaz několik faktorů.

K měření uplynulého času použijeme metodu `System.nanoTime()`, ve chvíli, kdy nepoužíváme žádné pokročilé nástroje pro MicroBenchmarking, bude tato metoda dostačující. Je ale třeba si uvědomit, že samotné zavolání této metody zabere nějaký čas, který nebude pokaždé stejný.

Při použití Apache Camel sice realizujeme reálné použití brokeru v praxi, ale určitě bude jeho použití znamenat režii navíc. Ve chvíli, kdy Apache Camel použijeme pro všechny brokery, můžeme předpokládat, že režie navíc bude přibližně stejná (přidávání hlaviček, agregace, atd.).

Všechny brokery běží v AWS prostředí ve VPC společně s aplikacemi Producer a Consumer proto, abychom se vyhnuli možnosti znehodnocení z důvodu různých rychlostí připojení. Zároveň bude velmi zajímavé porovnat dockerizovanou službu oproti AWS službě používající stejný broker.

Další krok, který sice sníží výkonnost, naopak ale povede k větší bezpečnosti a reálnosti použití, je zapnutí persistence zpráv, abychom viděli, jakou výkonnost má broker v reálném použití v praxi.

8 Ověření hodnotících kritérií

V této kapitole vyhodnotíme porovnávané brokery. Porovnáme naměřené hodnoty výkonnosti (propustnosti a latence) pomocí vytvořeného programu popsaného v kapitole 7. Následně se zaměříme na možnosti zabezpečení a závěrem na uživatelskou přívětivost a poznatky získané v průběhu práce s message brokery. Nakonec u každého brokeru shrneme klady a zápory.

8.1 Naměřené hodnoty výkonnosti

Oproti plánovaným 8 konfiguracím message brokerů se podařila objektivně změřit výkonnost pouze 5 z nich. Podařilo se otestovat všechny brokery postavené na Docker Images běžící v ECS clusteru (ActiveMQ, Artemis, RabbitMQ, Kafka) a AWS Simple Queue Service ve standardní verzi.

Nepodařilo se změřit následující konfigurace brokerů:

- AWS Simple Queue Service FIFO. Podařilo se navázat připojení, nicméně nepodařilo se přijmout více než 1 zprávu.
- ActiveMQ běžící jako cloudová služba Amazon MQ. Podařilo se navázat připojení a začít odesílat zprávy, nicméně po 3. odeslané zprávě došlo k vypršení timeoutu (nastavený na 10 minut).
- Kafka běžící jako cloudová služba Amazon MSK. Nepodařilo se navázat připojení ani z vytvořené aplikace, ani z aplikací třetích stran.

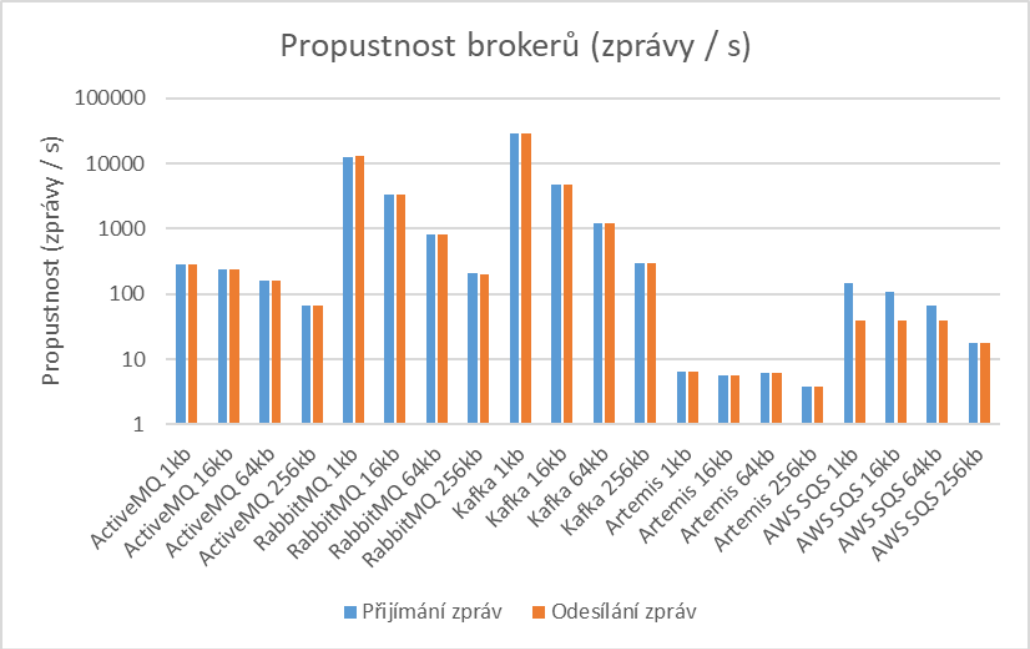
Z důvodu malé rychlosti některých brokerů u nich došlo ke snížení počtu zasílaných zpráv, a to z 100 000 na 10 000. Jedná se o broker Artemis běžící v ECS clusteru a AWS Simple Queue Service ve standardní verzi.

Pro brokery ActiveMQ, RabbitMQ a Kafka běžící v ECS clusteru proběhlo testování dle předpokládaného plánu, tedy testování na poslaných 100 000 zpráv. V této kapitole se podíváme na několik základních grafů, zbylé grafy a tabulky s konkrétními hodnotami budou uvedeny v příloze.

8.1.1 Propustnost

Podíváme se nejdříve na rozdíly v propustnosti brokerů, kde byly rozdíly velice markantní.

Při pohledu na graf na Obr. 8.1 vidíme, že dle očekávání je nejvyšší propustnost při nejmenší velikosti zprávy. Osa propustnosti je v logaritmickém



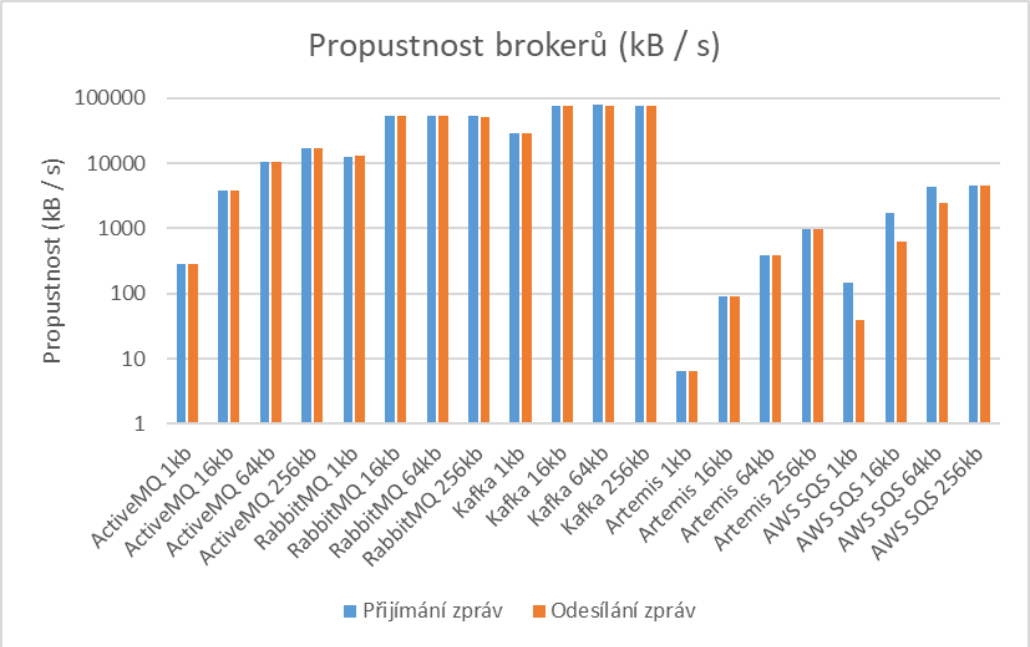
Obrázek 8.1: Propustnost testovaných brokerů ve zprávách za sekundu - logaritmické měřítko

měřítku, jelikož rozdíly jsou v řádu několika tříd. Graf propustnosti v lineárním měřítku je v příloze na Obr. A.8. Vidíme, že až na výjimky je hodnota propustnosti při přijímání zpráv téměř stejná jako při odesílání.

Jako jasný vítěz v propustnosti zpráv vychází v tomto případě Apache Kafka. Broker dosáhl při posílání zpráv o velikosti 1 kB průměrné propustnosti téměř 30 000 zpráv za sekundu.

Druhým nejrychlejším brokerem byl RabbitMQ, který byl sice při posílání zpráv o velikosti 1 kB více než dvakrát pomalejší, naproti tomu u zpráv větší velikosti dosahoval propustnosti cca 70% Apache Kafky.

Vidíme, že ostatní brokery dosáhly hodnot propustnosti o několik tříd níže. Překvapením je ActiveMQ, která za Kafkou a RabbitMQ zaostává při testování na zprávách malých velikostí, ale propustnost se příliš nesnižuje s větší velikostí zprávy. Toto je pravděpodobně způsobeno nastavením synchronního posílání zpráv. ActiveMQ používá JMS standard, který při původním nastavení používá některé funkce zásadně negativně ovlivňující výkonnost. Kromě používání MessageID (pro každou zprávu broker generuje identifikátor) jde hlavně o funkci „Auto Acknowledge“, která na každou přijatou zprávu generuje potvrzení a posílá ho zpět straně generující zprávu. Tímto je většina času strávena síťovou komunikací a ne zatížením brokeru. Grafy zobrazující propustnost ActiveMQ jsou na Obr. A.10 a A.11, konkrétní



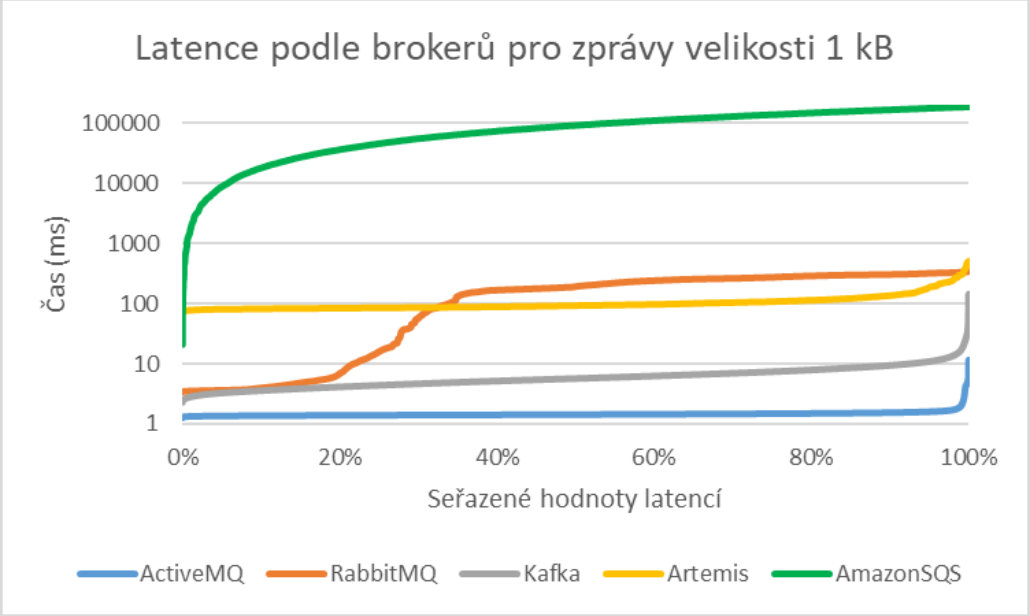
Obrázek 8.2: Propustnost testovaných brokerů v kB za sekundu - logaritmické měřítko

hodntoy jsou uvedeny v Tab. B.1.

Ještě větším překvapením je propustnost brokeru Artemis. Artemis jakožto nástupce ActiveMQ vykazuje mnohem horší hodnoty při identickém nastavení jako ActiveMQ. Může jít o nedokonalou optimalizaci brokeru, což může být důvodem, proč Apache stále podporuje používání klasické verze ActiveMQ. Grafy zobrazující propustnost Artemis jsou na Obr. A.18 a A.19, konkrétní hodnoty lze najít v Tab. B.4.

Velice zajímavý je pohled na hodnoty brokeru Amazon SQS. Vidíme, že broker zprávy ve většině případů přijímá mnohem rychleji, než je stíhá odesílat. U velikosti zpráv 1, 16 a 64 kB je propustnost při odesílání téměř konstantní (40 zpráv / s), zatímco propustnost při přijímání postupně klesá dle velikosti zpráv. Může se zdát, že broker nestíhá zpracovávat více než takto malé množství zpráv. U velikosti zpráv 256 kB jsou již hodnoty menší a téměř stejné. Grafy zobrazující propustnost Amazon SQS jsou na Obr. A.16 a A.17, naměřené hodnoty jsou uvedeny v Tab. B.5.

Druhý zajímavý pohled nám přinese graf na Obr. 8.2 zobrazující propustnost dat v kB / s. Osa propustnosti je opět uvedena v logaritmickém měřítku, v lineárním měřítku lze graf vidět na Obr. A.9. Zde můžeme vidět pravděpodobné vysvětlení poklesů propustností v závislosti na velikosti zprávy u RabbitMQ a Apache Kafka. Oba brokery při velikosti zpráv 16 kB dosáhly pravděpodobně svého maxima pro množství zpracovaných dat za



Obrázek 8.3: Latence testovaných brokerů pro velikost zprávy 1 kB - logaritmické měřítko

sekundu a na této hodnotě zůstaly až do velikosti zpráv 256 kB, konkrétně v případě Kafky cca 77 MB / s a v případě RabbitMQ 52 MB / s. Grafy zobrazující propustnost RabbitMQ jsou na Obr. A.12 a A.13, grafy zobrazující propustnost Kafky jsou na Obr. A.14 a A.15. Konkrétní naměřené hodnoty najdeme v Tab. B.2 (RabbitMQ) a B.3 (Kafka).

Oproti tomu vidíme, že brokery používající JMS standard se zvětšující se velikostí zprávy zvládaly zpracovávat více dat, což nahrává teorii o velké režii při synchronním zpracování zpráv a brokery nejsou naplno vytíženy.

Co se týče Amazon SQS, vidíme, že se zvětšující se velikostí zprávy propustnost dat roste. Zdá se, že broker nedosáhl svého maxima v množství zpracovaných dat, ale v počtu zpráv.

8.1.2 Latence

Nyní se zaměříme na latenci testovaných message brokerů. V grafu na Obr. 8.3 vidíme seřazené hodnoty naměřených latencí pro zprávy o velikosti 1 kB (čas latence je uveden v milisekundách v logaritmickém měřítku). V tomto případě ještě brokery nedosáhly svých možností, co se týče datového přenosu, a propustnost je většinou nejvyšší.

Vidíme, že u většiny brokerů je trend celkem podobný a odpovídá zjištěním při analýze propustnosti. Za povšimnutí rozhodně stojí náhlý růst kolem 99. percentilu, kde je vidět, že několik zpráv došlo s latencí řádově vyšší, než

je průměrná hodnota.

ActiveMQ má nejnižší hodnoty latence, což je pravděpodobně způsobeno tím, že broker není vytížený z důvodu synchronního zpracování. Jednoduše řečeno broker zpracovává málo zpráv na to, aby byl vytížen a zprávu zvládne velmi rychle poslat dál. Průměrné hodnoty latence se pohybují okolo 1,5 ms v případě zprávy velikosti 1 kB a okolo 7,5 ms v případě zprávy velikost 256 kB. Graf hodnot latence pro ActiveMQ je na Obr. A.1.

Hodnoty brokeru ActiveMQ Artemis mají podobný trend, ale hodnotově jsou o třídu jinde, průměrná latence se pohybuje kolem 100 ms, přestože má broker minimální propustnost. Graf hodnot latence pro Artemis je na Obr. A.5.

Velice zajímavou křivku vytváří hodnoty latence u RabbitMQ. Vidíme, že část zpráv zvládne projít brokerem velmi rychle (v řádu jednotek ms), poté ale křivka prudce vzroste a zprávy mají zpoždění až stovky milisekund, nicméně není zde vidět růst na konci křivky, RabbitMQ tedy stabilně drží hodnotu latence (maximální hodnota je přibližně dvakrát větší než medián, naproti tomu u ostatních brokerů se jedná i o desetinásobky). Záhadou je ovšem náhlý růst kolem 30% naměřených hodnot. Graf hodnot latence pro RabbitMQ je na Obr. A.2.

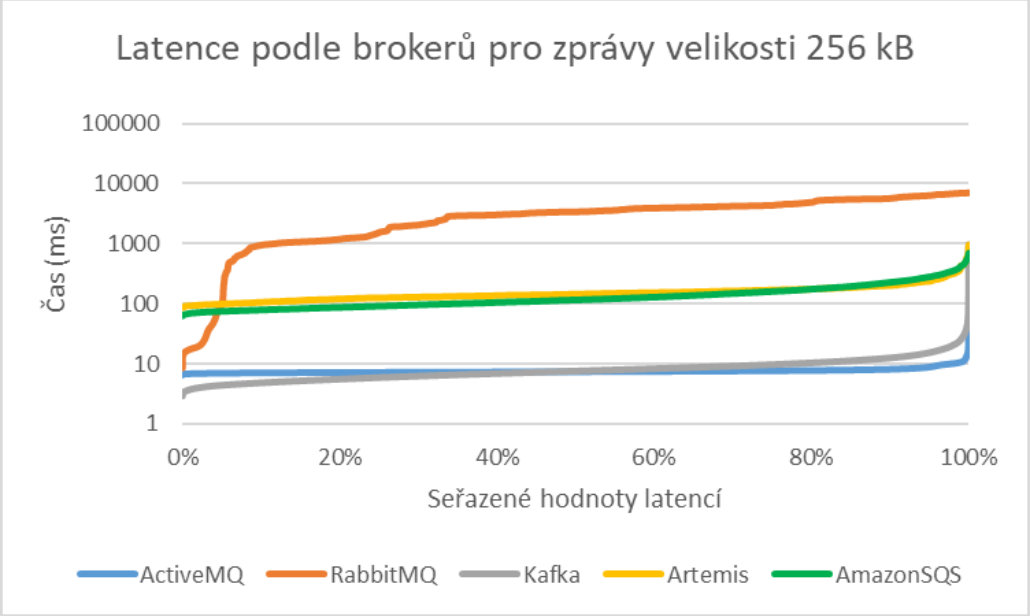
Kafka si navzdory nejvyšší naměřené propustnosti stále drží velmi kvalitní hodnotu latence. Průměrná hodnota se zde pohybuje kolem 6 ms, ale křivka roste rychleji než například v případě ActiveMQ či Artemis. Graf hodnot latence pro Apache Kafka je na Obr. A.3.

Kapitolou sama o sobě je Amazon SQS. Křivka latencí je naprosto odlišná od ostatních brokerů, což je způsobeno zejména tím, že broker nestíhá zpracovávat a odesílat zprávy tak rychle, jako je přijímá. V případě testování se zprávami o velikosti 1 kB zvládá SQS přijímat přibližně 150 zpráv za sekundu, ale odesílat pouze 40 zpráv za sekundu. Průměrná latence tímto vzrostla na neuvěřitelných 93 sekund, přičemž křivka rostla až k maximální hodnotě 189 sekund. Graf hodnot latence pro Amazon SQS je na Obr. A.4.

Pojďme se nyní podívat na latenci při posílání zpráv s největší velikostí - 256 kB. V grafu na Obr. 8.4 si můžeme všimnout několika zásadních rozdílů oproti latencím při testování se zprávami o velikosti 1 kB.

Předně, křivka u Amazon SQS již následuje trend ostatních brokerů. Toto je způsobeno tím, že při zpracování takto velkých zpráv už broker nezvládal přijímat více než 40 zpráv za sekundu a hodnoty průchodnosti při přijímání a odesílání se sjednotily (cca 17,5 zpráv za sekundu). Zprávy tedy nezůstavají ve frontě v brokeru a latence je několikanásobně nižší.

Artemis má velmi podobný trend jako v minulém případě, křivka ale roste lehce strměji, průměrná hodnota latence vzrostla ze 106 ms na 156 ms.



Obrázek 8.4: Latence testovaných brokerů pro velikost zprávy 256 kB - logaritmické měřítko

ActiveMQ si stále drží velmi kvalitní hodnoty latence, průměrná hodnota se nachází těsně pod 7,6 ms a až na strmý růst na konci křivky si drží stabilní latenci. Stále můžeme předpokládat, že to je z důvodu nízké vytíženosti brokeru z důvodu synchronního zpracování zpráv.

RabbitMQ jde i v tomto případě oproti trendu ostatních a jeho latence velice rychle vyroste nad hodnotu 1 sekundy a postupně šplhá až k 10 sekundám, průměrná hodnota zde přesahuje 3,2 sekundy. Takto velký růst lze dát za vinu naprostému vytížení brokeru, datová propustnost je podle předchozích grafů na maximu a RabbitMQ nestíhá zprávy odesílat včas.

Překvapením je zde Apache Kafka. Tento broker stejně jako RabbitMQ pravděpodobně dosáhl maximální datové propustnosti, nicméně na hodnotách latence se to téměř vůbec neprojevilo. Kolem 99. percentilu začínají hodnoty latence strmě růst až k maximální hodnotě 968 ms, nicméně průměrná hodnota je velmi nízko kolem 8,5 ms.

Zbylé grafy lze nalézt v příloze, pro velikost zprávy 16 kB na Obr. A.6 a pro velikost zprávy 64 kB na Obr. A.7.

8.2 Konfigurace zabezpečení

V této části kapitoly zhodnotíme u každého brokeru jejich možnosti zabezpečení. Začneme šifrováním zpráv, podíváme se na možnosti autentikace

klienta, zabezpečení monitorovací konzole, uvedeme si, jak je broker zabezpečený „out of the box“, tedy bez našeho zásahu a s jakými podmínkami lze broker vystavit na internet.

8.2.1 Šifrování zpráv

Všechny 4 testované brokery podporují šifrování zpráv a komunikaci přes zabezpečený SSL protokol. Pro používání šifrování zpráv je potřeba vytvořit certifikát, ke kterému má jak broker, tak klient přístup, a k tomuto certifikátu musí obě strany znát heslo.

Pro každý z brokerů je nastavení používání TLS/SSL velmi komplikované, zahrnuje samotné generování a podepisování certifikátu, poskytnutí certifikátu oběma stranám a samotnou konfiguraci oběma stranám. Vzhledem k podmínkám používání protokolu pro šifrování zpráv je logické, že žádný z brokerů nativně zprávy nešifruje, nicméně v dokumentaci ke každému brokeru je použití šifrování zpráv důrazně doporučeno pro produkční prostředí.

8.2.2 Autentikace

ActiveMQ umožňuje zabezpečení brokeru pomocí přídavných pluginů. Nejčastější použití je použití JAAS pluginu. V tomto případě se pro konfigurace autentikace uvede v konfiguračním souboru, pojmenovaném např. `login.config`. Zde je možné uvést pravidla pro uživatele a skupiny uživatelů definované v souborech `users.properties` a `groups.properties`, nebo také jak se chovat k nepřihlášeným uživatelům. Další možností je použít Simple Login plugin, kde stačí uvést přihlašovací jméno a heslo. V tomto pluginu lze povolit připojení anonymních uživatelů [5]. Broker Artemis zdědil autentikační mechanismy od ActiveMQ, tudíž změn je zde minimum. Stejně jako v případě ActiveMQ je pro autentikaci použit JAAS plugin.

V případě RabbitMQ je anonymní přihlášení povoleno pouze z localhostu, v případě potřeby přihlásit se anonymně z jiného počítače je potřeba upravit konfigurační soubor. RabbitMQ nabízí 4 vestavěné autentikační mechanismy:

- PLAIN - defaultně povoleno pro server a klienty.
- AMQPLAIN - verze PLAIN se zpětnou kompatibilitou, defaultně povoleno pro server.
- EXTERNAL - externí mechanismy autentikace, v přídavných pluginech, např. X.509 certifikát.

- RABBIT-CR-DEMO - ukázka „challenge-response“ autentikace, bezpečnostně ekvivaletní PLAIN.

RabbitMQ podporuje pomocí pluginů jak jednoduchou autentikaci pomocí přihlašovacího jména a hesla, tak pomocí X.509 certifikátů. V případě použití certifikátu je nutná další konfigurace, povolení SSL autentikačního mechanismu na straně brokeru a povolení přihlašovacího mechanismu `EXTERNAL` na straně klienta. V tuto chvíli RabbitMQ ignoruje poskytnuté přihlašovací údaje a používá certifikát [17].

Kafka od verze 0.9 přidává nové bezpečnostní prvky, momentálně dovoluje použití následujících SASL mechanismů pro autentikaci klientů, či brokerů mezi sebou:

- GSSAPI (Kerberos)
- PLAIN
- SCRAM-SHA-256
- SCRAM-SHA-512
- OAUTHBEARER

Kafka stejně jako ActiveMQ pro SASL konfiguraci používá JAAS. V konfiguračních souborech lze nastavit parametry pro jednotlivé použité pluginy. Kromě konfigurace autentikace klientů musí Kafka nakonfigurovat autentikaci mezi brokerem a Zookeeperem.

Dále lze využít připojení klientů pomocí SSL certifikátu (postup generování bude zmíněn níže). Není třeba znát přihlašovací jméno a heslo, ale je potřeba mít certifikát a heslo k němu. Kafka umožňuje povolit připojení i anonymním klientům bez certifikátu, což je potřeba explicitně nastavit a nedoporučuje se [4].

Pro připojení k Amazon Simple Queue Service musí klient znát klíče `access-key` a `secret-key` a zároveň region, ve kterém služba běží. Tyto klíče umožňují přistoupit k většině služeb od AWS. SQS umožňuje připojení anonymních uživatelů, toto je třeba nastavit v případě dané fronty, stačí v sekci zabezpečení povolit připojení všem klientům.

8.2.3 Zabezpečení monitorovací konzole

Všechny brokery poskytující monitorovací konzoli umožňují (a doporučují) použít přihlašovací jméno a heslo. Apache Kafka monitorovací konzoli neposkytuje, nicméně existují nástroje třetích stran (například Conductor) pro

monitorování brokeru, tento nástroj musí také znát certifikát a heslo k jeho použití. Do Amazon SQS konzole se lze dostat v rámci AWS konzole, kde se uživatel musí autorizovat svým AWS účtem.

ActiveMQ v klasické i Artemis verzi má konzoli zabezpečenou defaultními hodnotami (*admin / admin*), které je velmi doporučeno při vystavení konzole na internet změnit. Do verze 5.8.0 se lze do ActiveMQ monitorovací konzole dostat bez přihlašovacích údajů.

RabbitMQ má také defaultní přihlašovací údaje, které jsou v tomto případě *guest / guest* a dokumentace doporučuje tyto hodnoty změnit.

8.2.4 Základní konfigurace

Většina brokerů v základu příliš zabezpečená není. ActiveMQ, Artemis i RabbitMQ mají zabezpečenou monitorovací konzoli pomocí defaultních přihlašovacích údajů. Kafka monitorovací konzoli nemá, nicméně pokud nepoužívá šifrování, lze se připojit pomocí nástrojů třetích stran bez nutnosti autentizace. Pro použití TLS pro šifrovanou komunikaci je třeba vygenerovat a podepsat certifikát.

ActiveMQ má defaultně vystavené porty 61616 pro OpenWire protokol a 5672 pro AMQP protokol bez zabezpečení. Monitorovací konzole s přihlašovacími údaji *admin / admin* je vystavena na portu 8161. Verze Artemis má stejné defaultní nastavení, jako klasická verze ActiveMQ, kromě defaultního vystavení AMQP protokolu.

RabbitMQ defaultně vystavuje porty 5672 pro AMQP protokol, 15672 pro monitorovací konzoli a 25672 pro clustering. Na portu 4369 potom běží epmd - Erlang Port Mapper Daemon (tato informace není příliš zmiňována a může se jednat o další bezpečností riziko). Defaultní přihlašovací údaje do monitorovací konzole jsou *guest / guest*. Klient se může přes AMQP protokol připojit se stejnými přihlašovacími údaji.

Apache Kafka potřebuje Zookeeper, běžící na portu 2181. Kafka dále vystavuje port 9092 pro připojení klientů. Zprávy jsou v nešifrované podobě a v základním nastavení může kdokoliv přispívat či odebírat zprávy z tématu.

Zde má malou výhodou Amazon SQS. Služba je vázána k danému uživateli a bez znalosti jeho přístupového a tajného klíče nelze k službě přistoupit. Zároveň při úvodním nastavení fronty (kterou musí uživatel před používáním projít) lze jednoduše zvolit použití šifrovaného připojení, přičemž šifrovací klíč lze získat z Amazonu.

8.2.5 Vystavení na internet

V některých případech se může stát, že budeme potřebovat vystavit message broker na internet. Broker může například přijímat zprávy přes MQTT prokol od chytrých zařízení, nebo lze broker využít pro distribuci úloh mezi různými servery mimo lokální síť, například přes STOMP protokol.

Při potřebě vystavit broker na internet můžeme vycházet ze základní konfigurace a uvažovat, co bude potřeba přidat, či změnit. V základním nastavení by nebylo příliš rozumné broker na internet vystavit.

Předně bychom potřebovali změnit defaultní přihlašovací údaje. Toho lze buď dosáhnout v samotné monitorovací konzoli, nebo v konfiguračních souborech (budou zmíněny níže). V případě cloudových služeb lze přihlašovací údaje nastavit při vytváření služby.

Dalším krokem je šifrování zpráv mezi klienty, tzv. „end-to-end“ šifrování. Amazon nabízí SSE (Server Side Encryption), což je zabezpečení, které každý objekt zašifruje unikátním klíčem. Tento klíč je zašifrovaný hlavním klíčem, pro zabezpečení Amazon využívá šifrování AES-256. V případě cloudových poskytovatelů se dá předpokládat, že zabezpečení bude na úrovni, na kterou snaha jedince o vlastní zabezpečení bez pokročilých zkušeností a let praxe nemá šanci dosáhnout.

V případě brokerů běžících na vlastní insfrastruktuře se ale o zabezpečení budeme muset postarat sami. Pro „end-to-end“ šifrování je potřeba nakonfigurovat komunikaci mezi klienty a brokerem prostřednictvím TLS/SSL vrstvy. Způsob je pro všechny brokery velmi podobný:

1. Vygenerovat klíč a certifikát. Nástroj `keytool` dokáže vygenerovat tzv. „keystore“, soubor, kde je uložený privátní klíč certifikátu.

2. Vytvořit certifikační autoritu k podepsání certifikátu. Nástroj `openssl` dokáže vytvořit CA (Certificate Authority) a následně vygenerovat „truststore“, soubor, ve kterém se bude CA nacházet. K tomu opět poslouží nástroj `keytool`.

3. Podepsat vygenerovaný certifikát vytvořenou certifikační autoritou. K tomu je potřeba vyexportovat certifikát z „keystoru“, podepsat ho pomocí CA a následně certifikát i CA importovat do „keystoru“.

K oběma souborům při generování uživatel vytvoří heslo. Vytvořené soubory a hesla broker i klient zahrne do své konfigurace. Posledním krokem je na obou stranách povolit připojení pomocí SSL a lze komunikovat pomocí šifrovaného spojení.

Dále bychom měli uvažovat, zda je u produkčního prostředí vystaveného na internet nutnost mít dostupnou monitorovací konzoli, která je jedním z nejzranitelnějším míst brokeru.

Většina brokerů se v základní konfiguraci váže na IP adresu 0.0.0.0. To v praxi znamená, že poslouchá na jakékoliv IP adrese. Kromě velmi výjimečných případů je tedy vhodné v konfiguraci specifikovat seznam IP adres, které mají povolení se k brokeru připojit.

Po splnění těchto základních předpokladů už by šlo uvažovat o vystavení message brokeru na internet. I přesto je u brokerů doporučeno provozování za zabezpečeným firewallem v uzavřené síti a vystavení na internet se pokud možno vyhnout. Jednou z alternativ vyzkoušených v praxi je například monitorování brokeru pomocí JMX.

8.3 Vývojářská přívětivost

Nakonec zhodnotíme vývojářskou přívětivost, přičemž prvním parametrem hodnocení budou protokoly, přes které lze s brokery komunikovat. Následně projdeme náročnost počáteční konfigurace, monitorovací konzole a možnosti spuštění brokeru v Dockeru a cloudových službách.

8.3.1 Podporované protokoly

ActiveMQ v klasické i Artemis verzi v základu používají JMS používající OpenWire protokol. Kromě toho ActiveMQ umožňuje využívat protokoly STOMP, AMQP a MQTT. Další možností je publikovat či konzumovat zprávu přes REST API. V minulosti ActiveMQ umožňoval i podporu XMPP, protokol, na kterém byl postaven například oblíbený komunikační nástroj Jabber. Artemis kromě OpenWire, STOMP, AMQP, MQTT umožňuje navíc využívat HornetQ jako součást „core API“, kam lze přistupovat přímo bez použití JMS.

RabbitMQ nativně pro komunikaci používá AMQP protokol, nicméně podporuje i STOMP a AMQP. Další možností je publikování zprávy přes HTTP za použití WebSocketů.

Kafka se vydává úplně jinou cestou a místo JMS či AMQP používá binární protokol založen na TCP, který je optimalizován pro efektivitu a zprávy seskupuje do skupin pro zrychlení přenosu.

Amazon SQS žádné z výše zmíněných protokolů nepoužívá a v případě potřeby použití těchto protokolů doporučuje použít službu Amazon MQ. Komunikace se službou probíhá pomocí webových služeb přes SOAP protokol.

8.3.2 Počáteční konfigurace

Počáteční konfigurace brokerů velmi závisí na tom, jakou implementaci brokeru používáme. V rámci diplomové práce jsme používali předpřipravené Docker Image, jejichž velkou výhodou je spuštění jedním příkazem. Nutná konfigurace je zde tedy ve všech případech minimální.

O co nejjednodušší konfiguraci se snaží i cloudové služby. Zde nepotkáme složité konfigurační soubory, ale jednoduchý formulář vysvětlující dané pojmy a uvádějící příklady, kdy je vhodné dané nastavení použít. Po několika kliknutích tedy můžeme běžící službu používat.

O jiný případ se jedná, pokud chceme používat samotný message broker. V případě ActiveMQ (klasické i Artemis) jde o stažení programu v komprimovaném archivu a rozbalení do žádané složky. Následně lze příkazem `activemq start` spustit klasickou verzi. Základní konfigurační soubor se nazývá `activemq.xml`. V případě Artemis je nejdříve vytvořit instanci brokeru (`artemis create broker`) a tu následně spustit (`artemis run`). Základní konfigurační soubor se nazývá `broker.xml`. V obou případech lze broker konfigurovat i jiným způsobem, například přes příkazovou řádku, přesto se doporučuje používat xml konfigurační soubory.

V případě RabbitMQ je pro spuštění potřeba broker nainstalovat. Pro Windows, Linux i Mac OS X nabízí RabbitMQ instalační soubor, po instalaci na Windows se služba automaticky nastartuje, na Linuxu a Mac OS X je potřeba server s brokerem manuálně spustit. Konfiguraci lze měnit ve dvou konfiguračních souborech, `rabbitmq.conf` pro základní nastavení brokeru a pluginů a `advanced.conf` pro pokročilé nastavení.

Kafka vyžaduje stažení komprimovaného archivu a rozbalení do žádané složky. Dalším krokem je spuštění Zookeperu. Spustit lze zavoláním skriptu (`zookeeper-server-start`) s povinným parametrem, kterým je cesta k souboru s properties. Po spuštění Zookeeperu lze spustit stejným postupem Kafku (`kafka-server-start`), opět s předložením cesty k souboru s properties. V případě pouštění více uzlů v clusteru je potřeba Kafku spustit vícekrát, pokaždé s jiným properties souborem definujícím port, na kterém uzel bude poslouchat. Konfigurační soubory mají `.properties` formát a vždy je třeba uvést cestu při spouštění serveru.

Co se týče připojení klienta, který byl ve všech případech implementovaný jako Camel komponenta, v případě většiny brokerů stačilo zadat do konfigurace IP adresa, port a přihlašovací údaje a propojení proběhlo v pořádku. Výjimku v tomto případě tvoří Amazon SQS, tento broker potřeboval v nastavení uvést přístupový a tajný klíč a region. Jediný problém, který se nepodařilo vyřešit, bylo propojení Camelu a Amazon MSK, v tomto případě

se nepodařilo na broker napojit ani s pomocí dokumentace od Amazonu, Kafky a Camelu.

8.3.3 Monitorovací konzole

Jak jsme si již řekli, všechny brokery kromě Apache Kafky nabízejí monitorovací konzoli. V případě Kafky je pro monitoring použít nástroje třetích stran.

ActiveMQ poskytuje konzoli běžící na portu 8161, kde můžeme najít seznam existujících front či témat s detaily jako název, zprávy ve frontě, počet odběratelů, počet přijatých a odeslaných zpráv. Nad každou frontou či tématem lze z konzole provádět různé operace, jako promazat čekající zprávy, smazat frontu, či poslat zprávu. ActiveMQ konzole poskytuje informace o odběratelích přihlášených k jednotlivým tématům a informace o jednotlivých připojeních přes různé protokoly. Konzole nabízí i poslat zprávu do fronty či tématu zprávy přes OpenWire protokol.

ActiveMQ Artemis používá monitorovací konzoli postavenou na systému Hawtio. Hawtio je modulární webová konzole nabízející několik desítek modulů pro monitoring dle potřeb systému. Zároveň lze velmi jednoduše vytvořit vlastní modul. Artemis konzole zde běží jako modul poskytující přehled připojených klientů a vytvořených připojení (vláken), seznam producentů a konzumentů a seznam front. Konzole u každého přehledu poskytuje detailní informace.

RabbitMQ konzole běží nativně na portu 15672. Konzole poskytuje základní souhrnný přehled, kde uživatel najde základní informace i s grafy (grafy pro počet zpráv a průchodnost za posledních 10 minut, globální hodnoty pro počet aktivních připojení, konzumentů, front a kanálů). Dále zde jsou detaily uzlů, na kterých broker běží, včetně detailních informací o vytížení. Následujíc další grafy o síťovém vytížení a přehled vystavených portů. Konzole nabízí exportovat či importovat definici brokeru. Na dalších záložkách konzole nabízí detailní přehled připojených klientů s informacemi o IP adrese, protokolu, uživatelském jménu a procházejících datech. Následuje přehled front a kanálů (pokud z fronty čte zprávy 20 konzumentů, má fronta 20 kanálů).

Pro monitoring Apache Kafka je potřeba použít jeden z nástrojů třetích stran, jedním příkladem je nástroj Conduktor, desktopová aplikace pro Windows, Linux i Mac OS X. Aplikace po připojení k brokeru nabízí základní přehled zobrazující seznam uzlů brokeru, aktivních témat a odběratelů a informace o zabezpečení. V dalších záložkách lze najít detailnější informace k informacím ze základního přehledu. Nástroj nabízí i manuální vytvoření

nového producenta či konzumenta či přijímat zprávy z existujících témat.

Amazon SQS lze monitorovat v rámci AWS konzole. Konzole nabízí jednoduchý přehled front s informacemi o typu fronty, počtu čekajících a zpracovaných zpráv. Po rozkliknutí fronty lze vidět detailní nastavení fronty a informace pro připojení. Další záložky nabízejí přehledné grafy o počtu přijatých a odeslaných zpráv, dat, průměrné velikosti zpráv, zpoždění a další. Konzole nabízí promazání zpráv ve frontě, poslání zprávy, zobrazení zpráv ve frontě a konfiguraci či smazání fronty.

8.3.4 Spuštění v Dockeru a cloudu

Všechny brokery kromě Amazon SQS je možné spustit v dockerizované verzi, což bylo demonstrováno i v rámci této diplomové práce. Apache nenabízí své Docker Image pro ActiveMQ ani Artemis, tudíž je třeba si vystačit s implementacemi třetích stran. RabbitMQ nabízí svou vlastní implementaci pro Docker, kromě toho lze použít certifikovaný Image od Bitnami. Bitnami nabízí i Image pro Apache Kafka. Kvalita jednotlivých řešení se liší, zejména u ActiveMQ a Artemis jsou možnosti nastavení omezené, nicméně pro otestování výkonnosti tato řešení stačila.

Popsali jsme si i možnosti spuštění brokerů v cloudovém prostředí. Prozkoumaným případem vyzkoušeným v diplomové práci bylo spuštění Docker Image pomocí Amazon ECS. V tomto případě všechno fungovalo bezproblémově a dá se zjednodušeně říci, že co lze spustit v Dockeru, lze spustit i v AWS.

Kromě toho AWS nabízí již dříve zmíněné služby Amazon MQ a Amazon MSK poskytující ActiveMQ a Kafku jako službu. Bohužel ani jednu službu se nepodařilo řádně otestovat. Na podobném principu jako Amazon SQS funguje i Amazon SNS (Simple Notification Service), která narozdíl od SQS neposkytuje fronty, ale témata (funguje na principu Publish/Subscribe).

Google Cloud nabízí dvě své služby, Cloud Task jakožto službu pro práci s frontami a Pub/Sub pro práci s tématy.

Microsoft Azure nabízí službu Azure Service Bus fungující jako plnohodnotný broker podporující práci s frontami a tématy nad protokolem AMQP 1.0.

Příkladem jedné z dalších možností je využití předpřipravených služeb od Bitnami pro běh v cloudu. Bitnami nabízí použít ActiveMQ, RabbitMQ a Kafku jako službu pro AWS, Microsoft Azure a Google Cloud.

8.4 Klady a zápory porovnávaných implementací

Na úplný závěr si krátce shrneme zjištěné klady a zápory testovaných message brokerů a uvedeme, pro řešení jakých problémů je daná implementace vhodná. Vzhledem k velice podobným možnostem zabezpečení může hrát hlavní roli při rozhodování rychlost, na druhou stranu samotný message broker velmi pravděpodobně nebude „úzkým hrdlem“ aplikace. Roli může hrát uživatelská zkušenost, možnosti monitoringu, či jiné specifické vlastnosti důležité pro daný projekt.

ActiveMQ je nativně používaná a zaobalená v programu Apache Camel, kde se stará o přenos zpráv v interním prostředí. Pokud použijeme Apache Camel jako integrační nástroj, který s dalšími systémy komunikuje přes jiné protokoly a k samotnému message brokeru nepotřebuje přistupovat nikdo další, jedná se o ideální řešení. Jinak se použití ActiveMQ hodí na jednodušší projekty a tam, kde je třeba použít JMS.

Ještě asi nedozrál čas na nutný přechod z klasické verze ActiveMQ na verzi Artemis, dle zkušeností z testování je ještě na čem pracovat a nahrazení klasické ActiveMQ v blízké době nehrozí.

Pro větší projekty je zřejmě vhodnější použít RabbitMQ či Apache Kafka. RabbitMQ nabízí celkem jednoduchou konfiguraci a ve většině případů poskytuje vše, co vývojář i zákazník potřebuje. Apache Kafka je oproti tomu řešení, které málokdo dokáže využít naplno. Nabízí mnoho vlastností nad rámec klasických message brokerů, s tím ale jde ruku v ruce náročná konfigurace a správa. Použít Apache Kafka se vyplatí pravděpodobně až v těch nejnáročnějších projektech, kde je rychlost naprosto klíčová a náklady jdou stranou.

Amazon SQS je služba, kterou se vyplatí vyzkoušet, pokud zbytek infrastruktury běží taktéž v cloudu a nemáme zapotřebí posílat velké množství zpráv. Do 1 milionu zpráv měsíčně je služba poskytována zdarma. I přesto, že se jedná o cloudovou službu, samotné nastavení a zprovoznění nebylo jednoduché a naměřené výsledky tristní. FIFO verze fronty zvládla zpracovat pouze 1 zprávu, pravděpodobně se jednalo o špatné nastavení, bohužel dokumentace Camelu ani Amazonu v tomto případě nebyla užitečná.

Další velká nevýhoda zjištěná při testování Amazon SQS je, že Camel (konkrétně Consumer aplikace) používá aktivní polling (v pravidelných intervalech se doptává, zda existuje zpráva ke zpracování) a toto se v Amazon SQS tváří jako velké množství prázdných přijatých zpráv. Vzhledem k limitovanému počtu bezplatných zpráv se cena služby brzy vyšplhala do desítek

eur a testování se značně prodražilo. Tato kombinace tedy rozhodně nebude dobrým řešením.

Ani další služby poskytované Amazonem nepřinesly pozitivní výsledky, k Amazon MSK (cloudové verzi Apache Kafky) se nepodařilo vůbec připojit a Amazon MQ (cloudová verze ActiveMQ) zvládla zpracovat pouhé 3 zprávy.

Souhrn kladů a záporů pro testované message brokery je uveden v Tab. 8.1.

Message broker	Klady	Zápory
ActiveMQ	Nativní broker v Apache Camel. Dostačující pro menší projekty. Přehledná monitorovací konzole.	Nízká propustnost při synchronní komunikaci. Nevhodné pro větší projekty. Nutnost konfigurace zabezpečení.
Artemis	Nástupce ActiveMQ, novější verze protokolů (JMS 2.0 a HornetQ). Nativní podpora Dockeru. Monitorovací konzole v Hawtio.	Velmi špatné hodnoty výkonnosti. Stále nedosahuje kvalit ActiveMQ. Nutnost konfigurace zabezpečení.
RabbitMQ	Vysoká průchodnost. Podpora mnoha protokolů a programovacích jazyků. Jednoduchá konfigurace. Přehledná monitorovací konzole.	Rychle rostoucí hodnoty latence. Nutnost konfigurace zabezpečení.
Kafka	Nejlepší hodnoty výkonnosti. Vhodné řešení pro velké projekty. Podpora clusteringu.	Náročná konfigurace. Nutnost dalšího prvku - Zookeeper. Nutnost konfigurace zabezpečení.
Amazon SQS (klasická fronta)	Jednoduchá konfigurace a spuštění. Detailní monitorovací konzole. Platba dle počtu zpráv. Amazon zajišťuje šifrování zpráv.	Nízká průchodnost a vysoká latence v řádu desítek sekund. Spojení s Camelem generuje velké množství prázdných zpráv. Nezaručuje doručení zpráv ve správném pořadí. Zpráva může přijít vícekrát.
Amazon SQS (FIFO fronta)	Nepodařilo se otestovat (dorazila pouze jedna zpráva).	
Amazon MQ (ActiveMQ v AWS)	Nepodařilo se otestovat (dorazily pouze tři zprávy).	
Amazon MSK (Kafka v AWS)	Nepodařilo se propojit s testovací aplikací.	

Tabulka 8.1: Souhrn kladů a záporů porovnávaných message brokerů

9 Závěr

V rámci diplomové práce se nám podařilo porovnat message brokery na základě předem stanovených kritérií. V kapitole 2 jsme si představili samotný koncept message brokeru a důkladně prozkoumali některé z existujících implementací. Seznámili jsme se s již existujícími analýzami výkonnosti a bezpečnosti message brokerů.

Následně jsme si v kapitole 3 stanovili seznam implementací k porovnání a přesná kritéria, na základě kterých se porovnání uskutečnilo.

Podařilo se vyvinout aplikaci a nakonfigurovat prostředí, které dokáže objektivně vyhodnotit výkonnost message brokerů v reálném použití. Pronikli jsme do frameworku Apache Camel, který je používán na integračních projektech, a do moderních technologií, jako je například Docker či Amazon Web Services. V kapitole 7 je podrobně popsán popis prostředí vytvořeném právě v Amazon Web Services.

V analýze výsledků v kapitole 8 jsme zhodnotili výkonnost daných message brokerů. Je potřeba si uvědomit, že naměřené výsledky mohly být ovlivněny mnoha faktory a jedná se o hodnoty naměřené pro specifické prostředí. Následně jsme si důkladně popsali možnosti konfigurace zabezpečení a uvedli některé z faktorů ovlivňující vývojářskou přívětivost.

Výsledky a zhodnocení potvrzují, že každý broker může být vhodný pro použití v jiné situaci. V případě menších projektů vyvíjených v Javě může být optimálním řešením ActiveMQ, zatímco v případě velkých projektů bychom využili Apache Kafku. V případě zakomponování malého brokeru v AWS lze zvolit Amazon SQS. Jako nejuniverzálnější řešení lze vybrat RabbitMQ. Velkým přínosem práce je zjištění, že cloudové služby, i přes nabízené zjednodušení procesů, mohou vývoj komplikovat a není vhodné je používat za každé situace.

Tímto se podařilo splnit všechny cíle uvedené v zadání práce.

Literatura

[1] Amazon. *Amazon Simple Queue Service Documentation* [online]. 2020. [cit. 2020/04/01]. Dostupné z: `https://aws.amazon.com/sqs/features`.

[2] Apache. *Apache ActiveMQ Documentation* [online]. 2020. [cit. 2020/04/01]. Dostupné z: `https://activemq.apache.org/components/classic/documentation`.

[3] Apache. *How does a Queue compare to a Topic* [online]. 2020. [cit. 2020/04/01]. Dostupné z: `http://activemq.apache.org/how-does-a-queue-compare-to-a-topic.html`.

[4] Apache. *Apache Kafka Documentation* [online]. 2020. [cit. 2020/04/01]. Dostupné z: `https://kafka.apache.org/documentation/`.

[5] Apache. *Security* [online]. 2020. [cit. 2020/04/01]. Dostupné z: `https://activemq.apache.org/security`.

[6] Apache. *Apache Camel Documentation* [online]. 2020. [cit. 2020/04/01]. Dostupné z: `https://camel.apache.org/components/latest`.

[7] Apshankar, K. *Web services business strategies and architectures*. Expert Press, 2002. ISBN 9781430253563.

[8] Ejsmont, A. *Web scalability for startup engineers : tips & techniques for scaling your Web application*. McGraw-Hill Education, 2015. ISBN 9780071843669.

[9] Hintjens, P. *ØMQ - The Guide* [online]. ZeroMQ, 2020. [cit. 2020/04/01]. Dostupné z: `http://zguide.zeromq.org/page:all`.

[10] Hintjens, P. *ØMQ - The Guide - Messaging Patterns* [online]. ZeroMQ, 2020. [cit. 2020/04/01]. Dostupné z: `http://zguide.zeromq.org/page:all#Messaging-Patterns`.

[11] Hohpe, G. *Enterprise Integration Patterns* [online]. 2019. [cit. 2020/04/01]. Dostupné z: `https://www.enterpriseintegrationpatterns.com/patterns/messaging/`.

[12] Hohpe, G. *Enterprise integration patterns : designing, building, and deploying messaging solutions*. Addison-Wesley, 2004. ISBN 0321200683.

[13] Ibsen, C. *Camel in Action*. Manning, 2018. ISBN 9781617292934.

[14] Kalra, G. S. *A Pentesters Guide to Hacking ActiveMQ-Based JMS Applications* [online]. 2014. [cit. 2020/04/01]. Dostupné z: https://www.mcafee.com/enterprise/en-us/assets/white-papers/wp-pentesters-guide-hacking-activemq-jms-applications.pdf.

[15] Nickoloff, J. *Docker in action.* Manning, 2019. ISBN 9781617294761.

[16] Pivotal. *RabbitMQ Documentation* [online]. 2020. [cit. 2020/04/01]. Dostupné z: https://www.rabbitmq.com/documentation.html.

[17] Pivotal. *Authentication, Authorisation, Access Control* [online]. 2020. [cit. 2020/04/01]. Dostupné z: https://www.rabbitmq.com/access-control.html.

[18] Samofal, V. *Introduction to Message Brokers: Part 1: Apache Kafka vs RabbitMQ* [online]. Freshcode, 2019. [cit. 2020/04/01]. Dostupné z: https://freshcodeit.com/blog-introduction-to-message-brokers-part-1-apache-kafka-vs-rabbitmq.

[19] Samofal, V. *Introduction to Message Brokers: Part 2: ActiveMQ vs Redis Pub/Sub* [online]. Freshcode, 2019. [cit. 2020/04/01]. Dostupné z: https://freshcodeit.com/freshcode-post/introduction-to-message-brokers-activemq-vs-redis-pub-sub.

[20] SCOTT, D. *Kafka in Action.* Manning, 2020. ISBN 9781617295232.

[21] Snyder, B. *ActiveMQ in Action.* Manning, 2011. ISBN 9781933988948.

[22] Stuchlík, P. *The Messageing Menagerie* [online]. 2019. [cit. 2020/04/01]. Video of live presentation https://vimeo.com/346065668. Dostupné z: https://www.owasp.org/images/d/dd/The_Messaging_Menagerie.pdf.

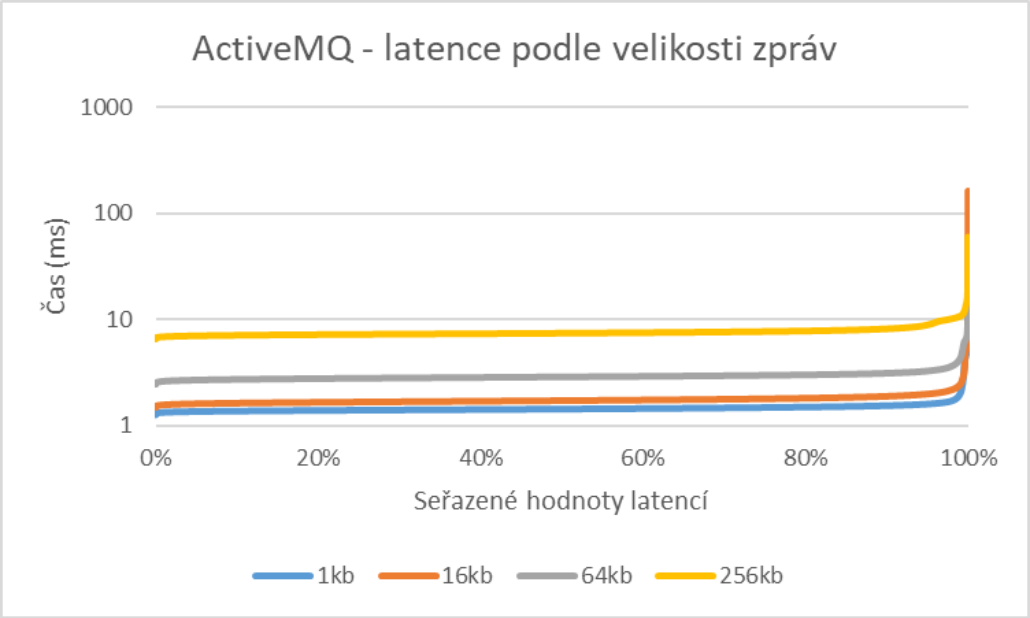
[23] Treat, T. *Dissecting Message Queues* [online]. 2014. [cit. 2020/04/01]. Dostupné z: https://bravenewgeek.com/dissecting-message-queues.

[24] Treat, T. *Benchmarking Message Queue Latency* [online]. 2016. [cit. 2020/04/01]. Dostupné z: https://dzone.com/articles/benchmarking-message-queue-latency.

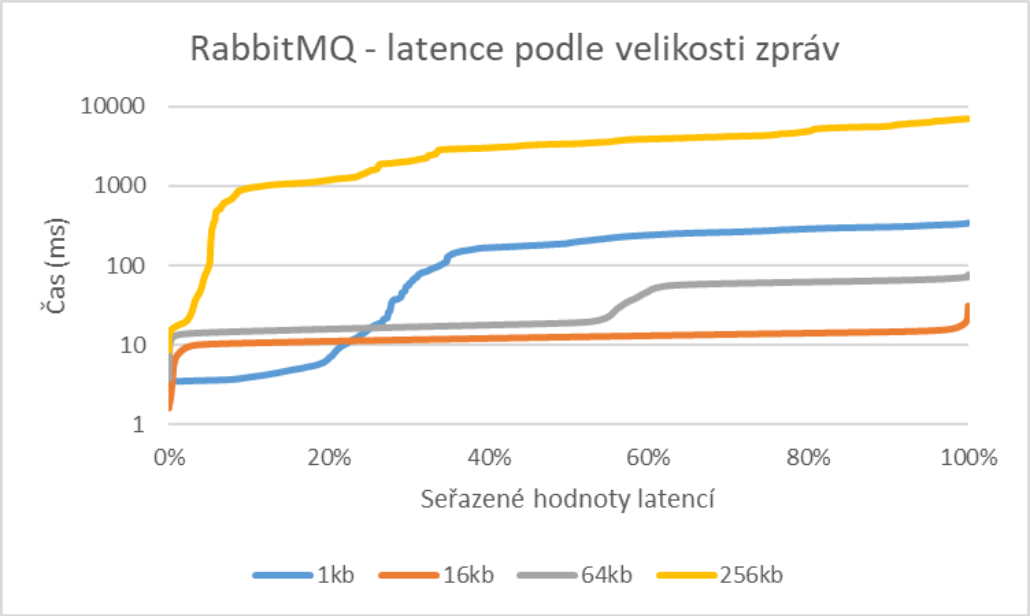
[25] Videla, A. *RabbitMQ in action.* Manning, 2012. ISBN 9781935182979.

[26] Wittig, M. *Amazon Web Services in Action.* Manning, 2019. ISBN 9781617295119.

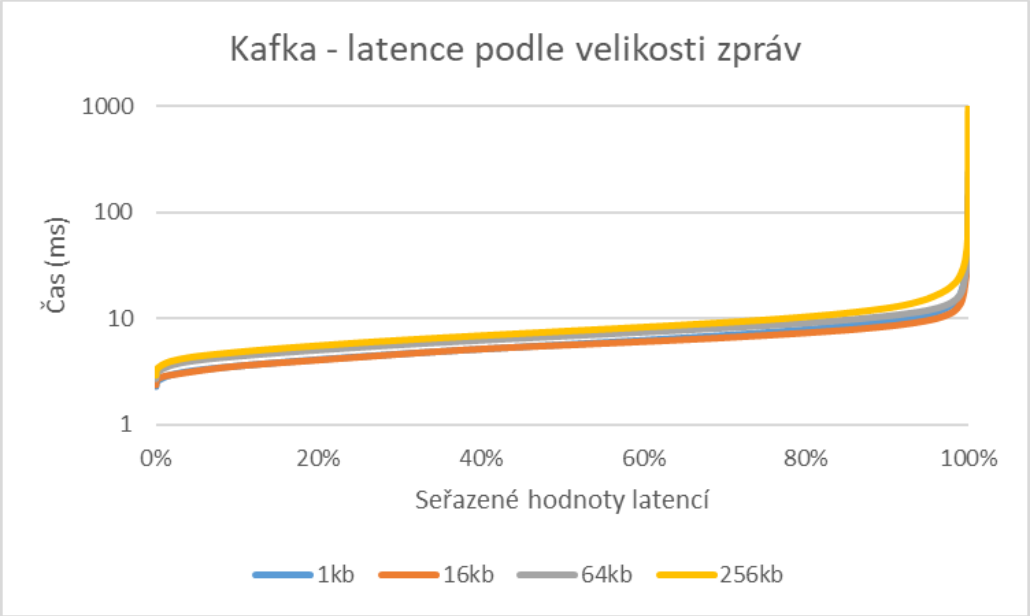
A Grafy



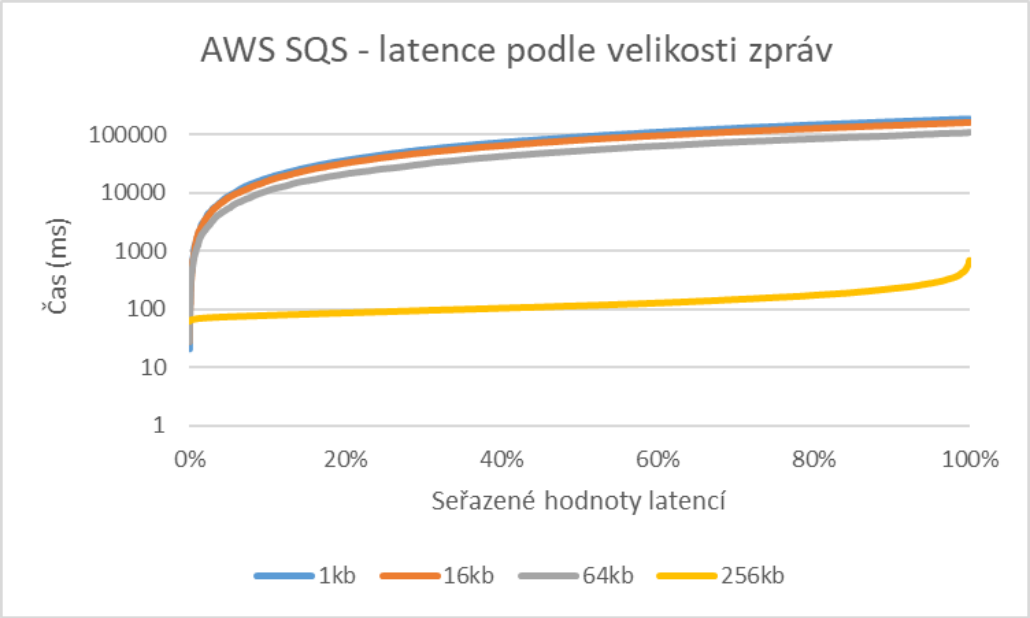
Obrázek A.1: Latence ActiveMQ dle velikosti zpráv - logaritmické měřítko



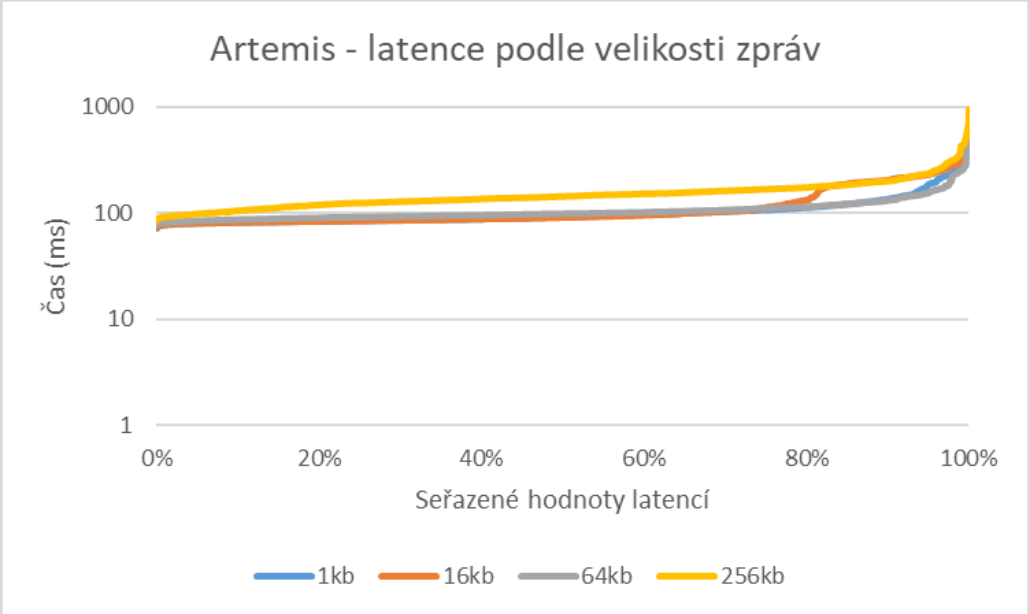
Obrázek A.2: Latence RabbitMQ dle velikosti zpráv - logaritmické měřítko



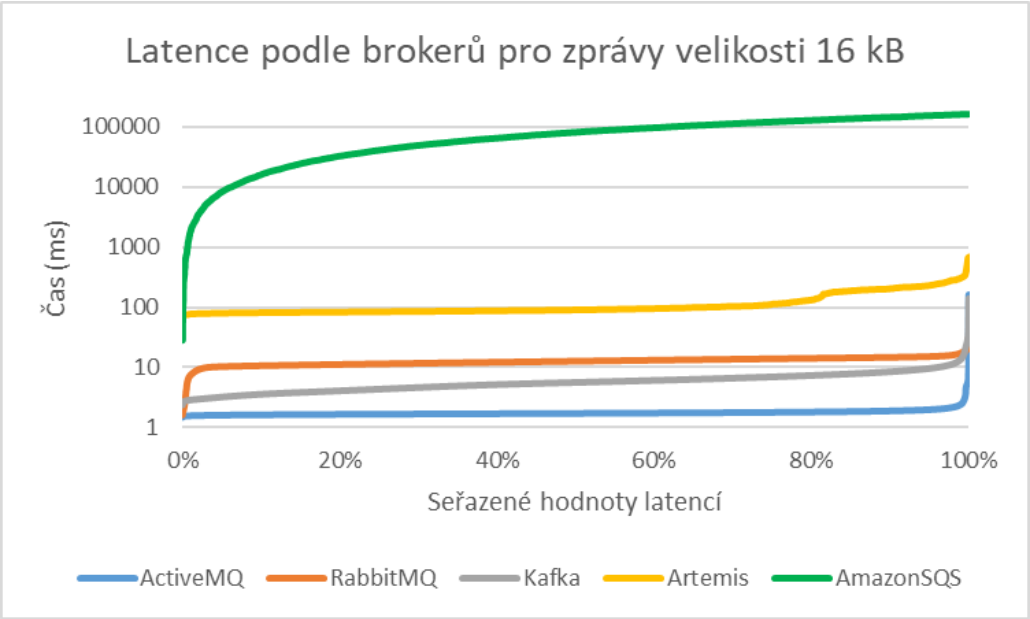
Obrázek A.3: Latence Kafky dle velikosti zpráv - logaritmické měřítko



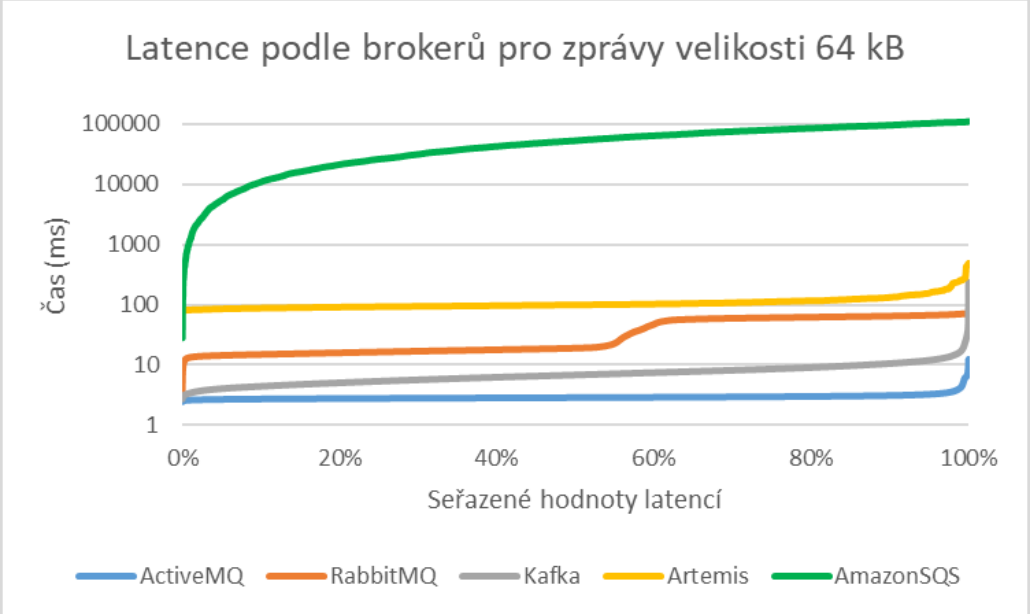
Obrázek A.4: Latence Amazon SQS dle velikosti zpráv - logaritmické měřítko



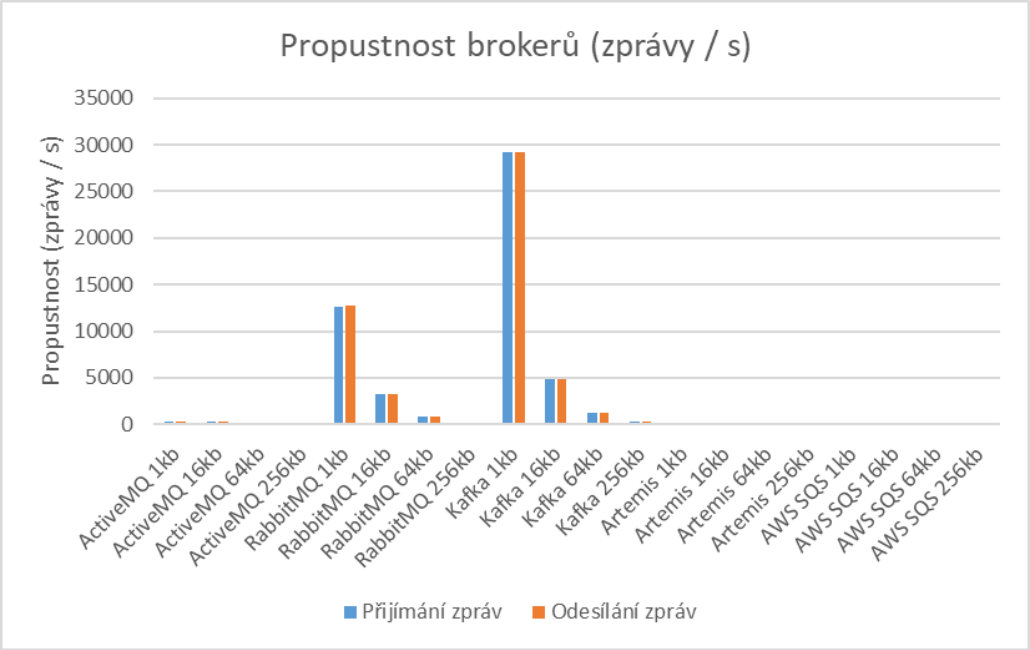
Obrázek A.5: Latence Artemis dle velikosti zpráv - logaritmické měřítko



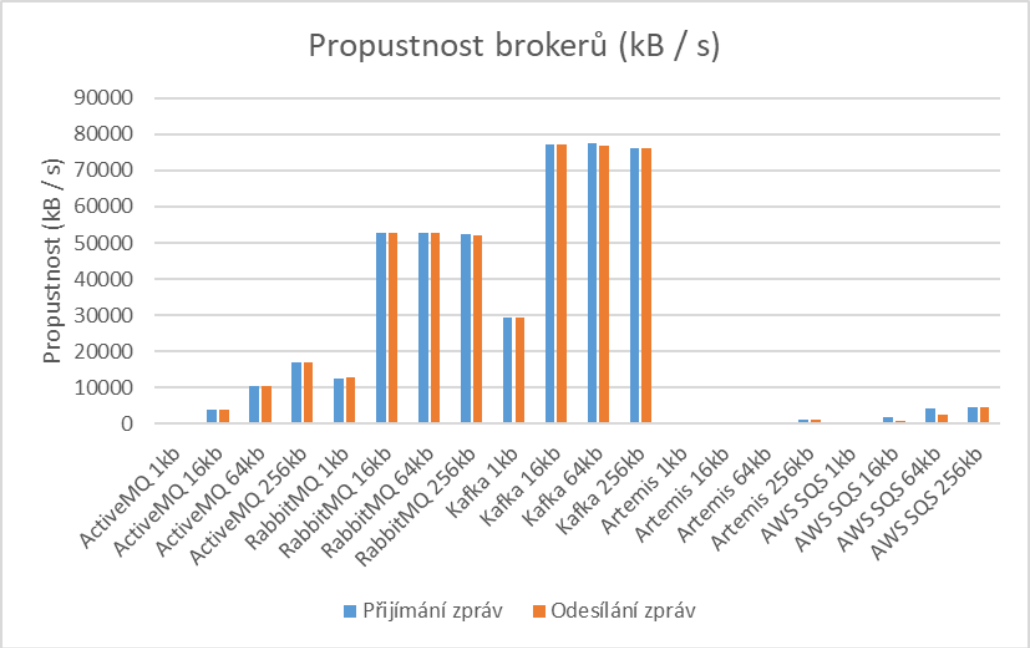
Obrázek A.6: Latence testovaných brokerů pro velikost zprávy 16 kB - logaritmické měřítko



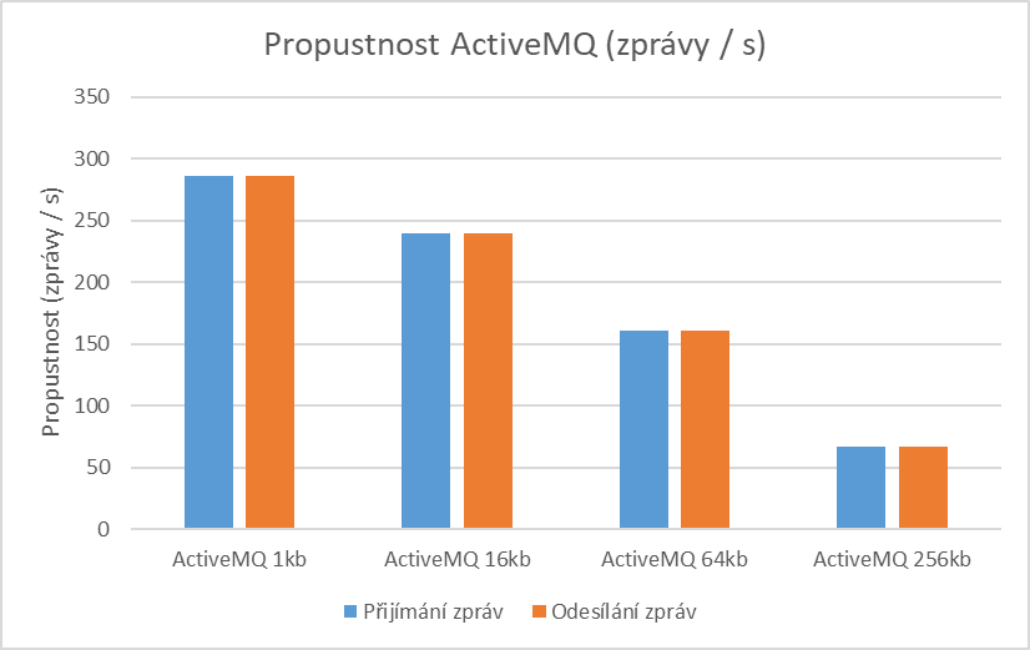
Obrázek A.7: Latence testovaných brokerů pro velikost zprávy 64 kB - logaritmické měřítko



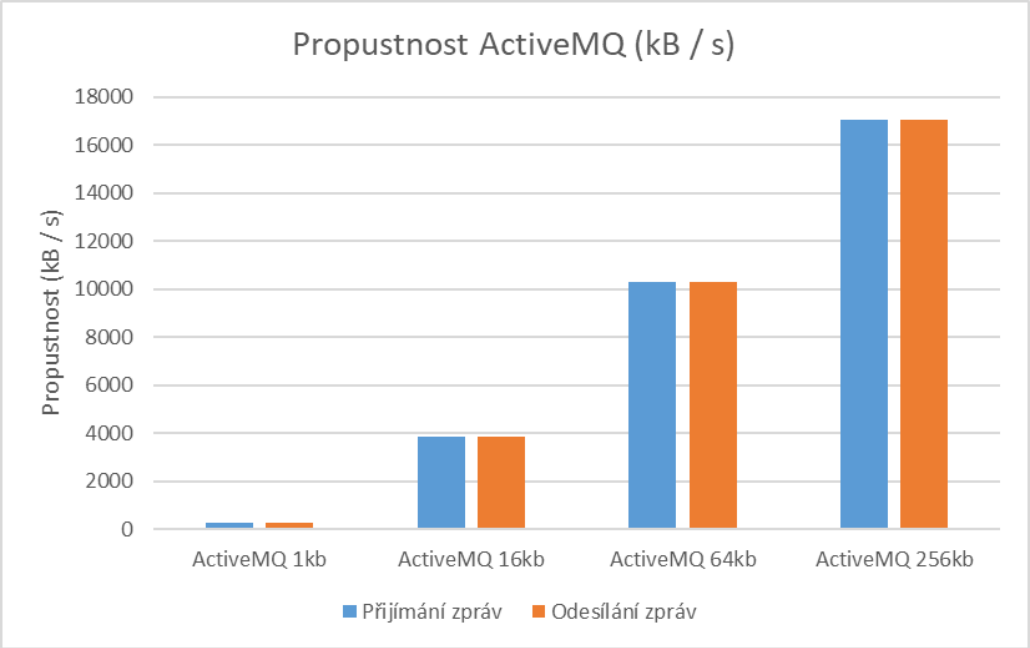
Obrázek A.8: Propustnost testovaných brokerů ve zprávách za sekundu



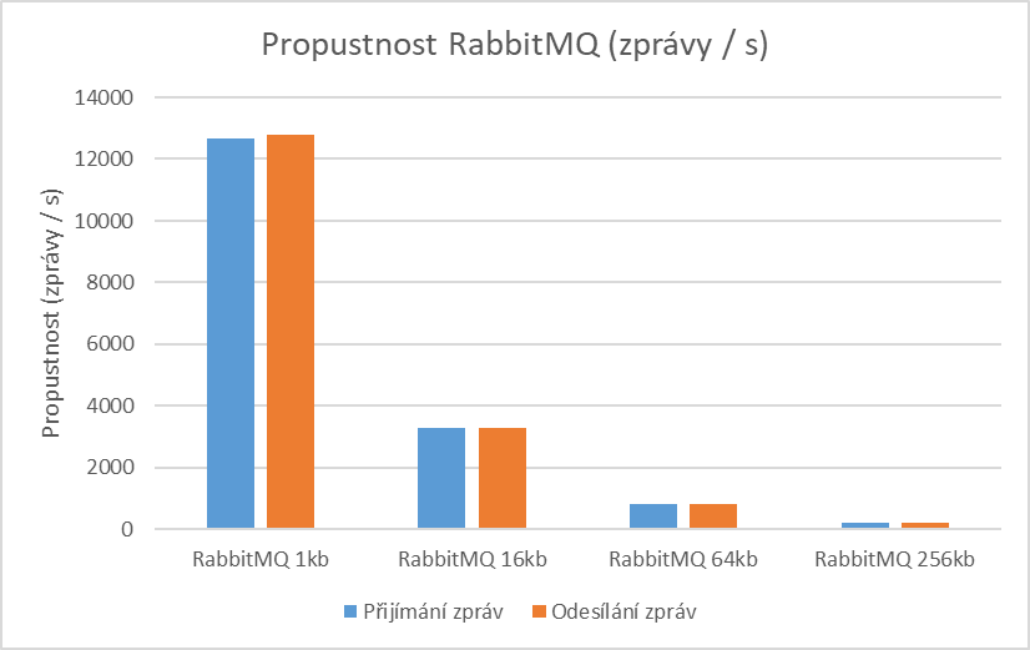
Obrázek A.9: Propustnost testovaných brokerů v kB za sekundu



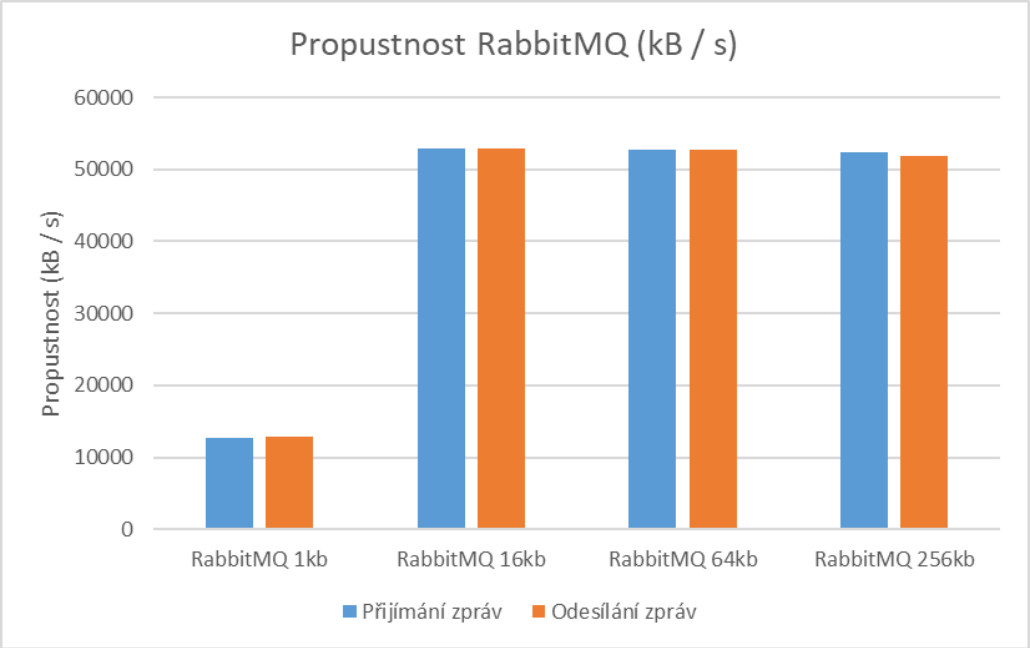
Obrázek A.10: Propustnost ActiveMQ ve zprávách za sekundu



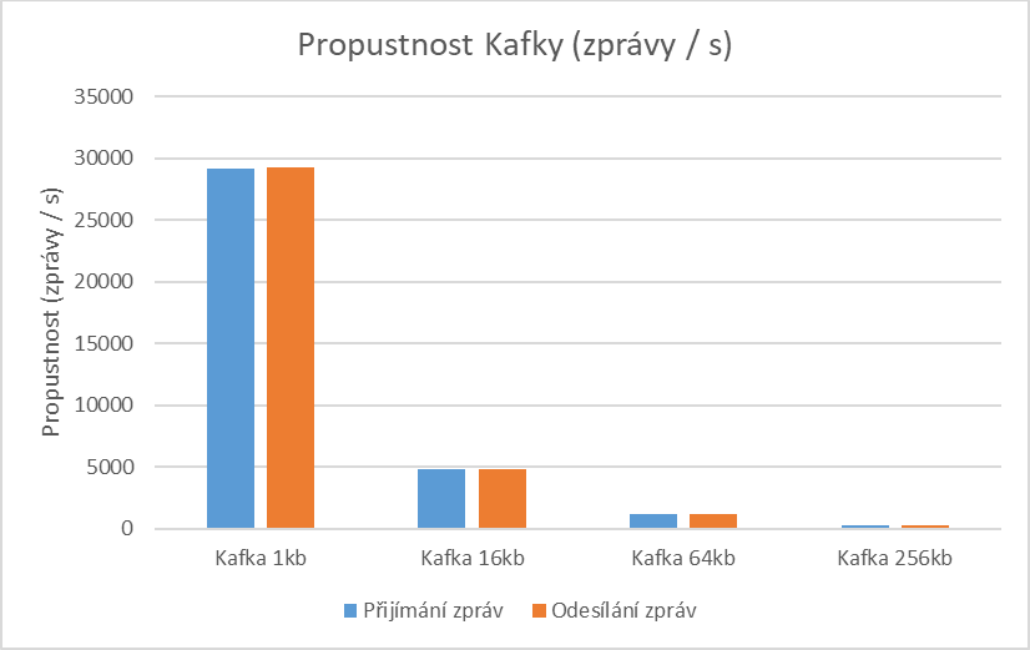
Obrázek A.11: Propustnost ActiveMQ v kB za sekundu



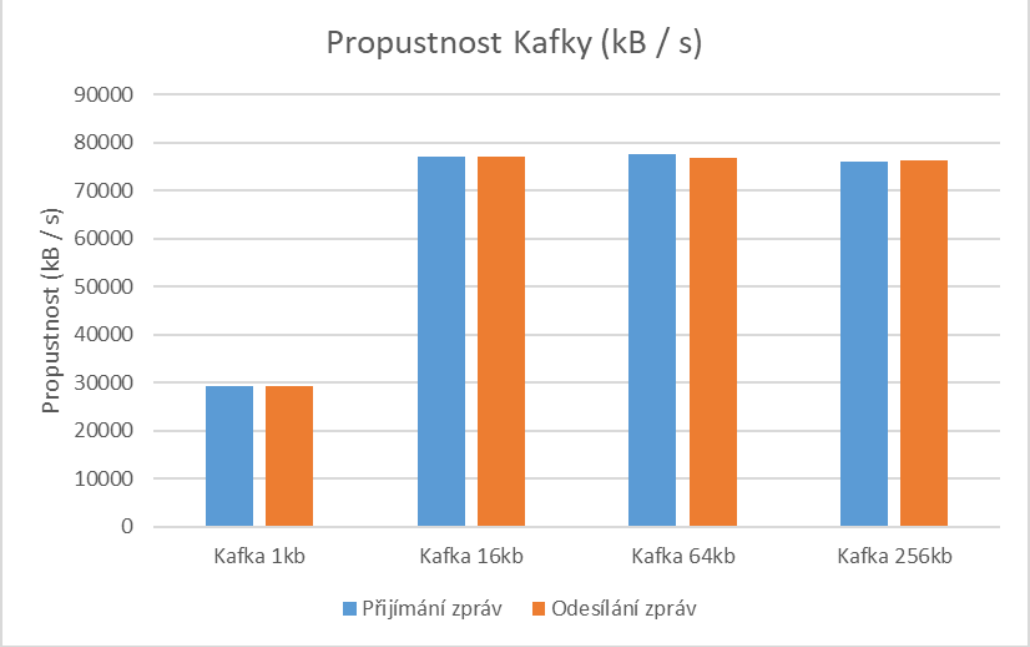
Obrázek A.12: Propustnost RabbitMQ ve zprávách za sekundu



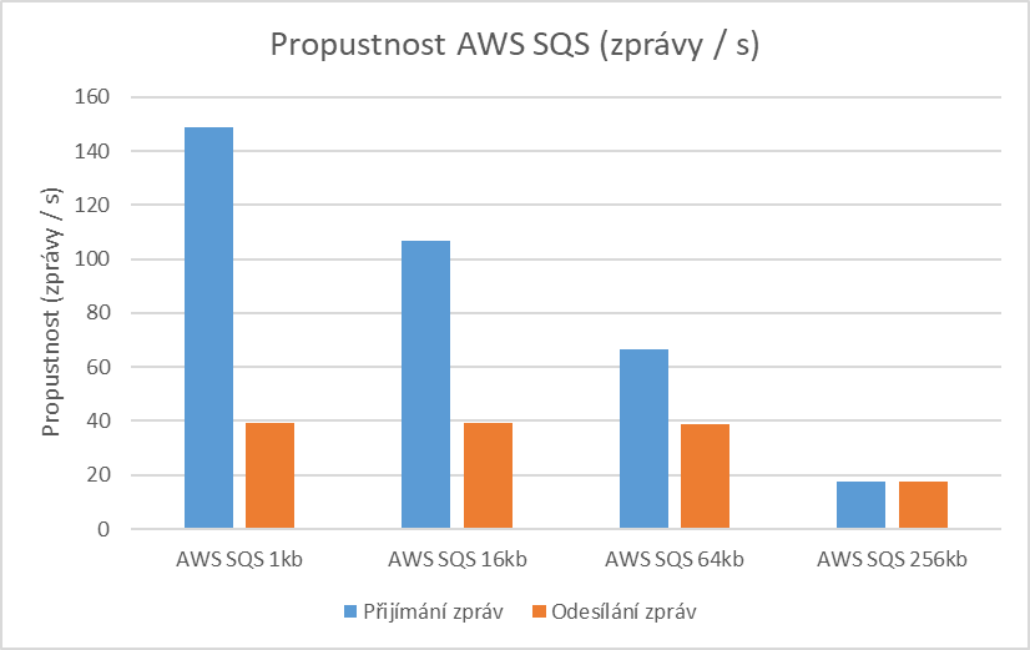
Obrázek A.13: Propustnost RabbitMQ v kB za sekundu



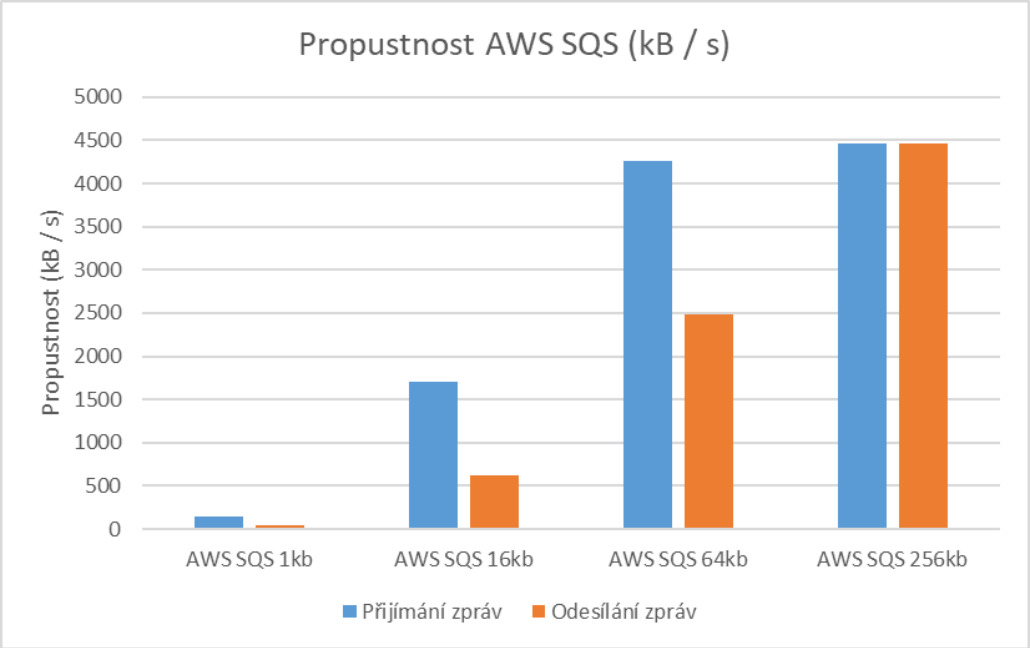
Obrázek A.14: Propustnost Kafky ve zprávách za sekundu



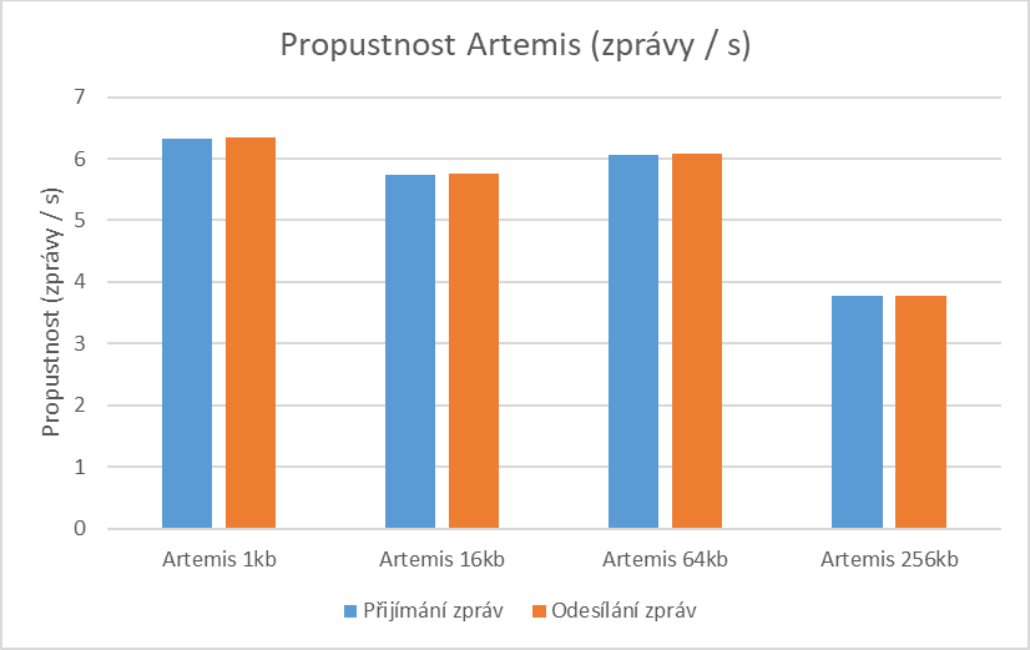
Obrázek A.15: Propustnost Kafky v kB za sekundu



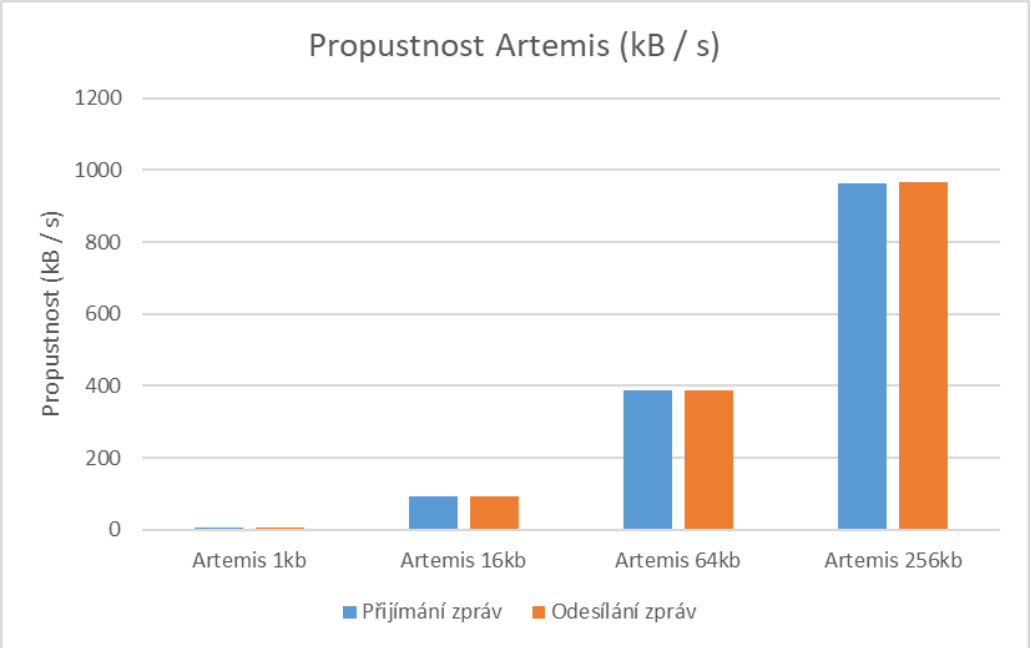
Obrázek A.16: Propustnost Amazon SQS ve zprávách za sekundu



Obrázek A.17: Propustnost Amazon SQS v kB za sekundu



Obrázek A.18: Propustnost Artemis ve zprávách za sekundu



Obrázek A.19: Propustnost Artemis v kB za sekundu

B Tabulky

Broker	ActiveMQ			
Velikost zprávy	1kb	16kb	64kb	256kb
Počet zpráv	100000	100000	100000	100000
Latence (ms)				
Průměr	1.46	1.76	2.92	7.58
Směrodatná odchylka	0.26	0.58	0.37	0.84
Medián (50 percentil)	1.43	1.71	2.86	7.39
Maximum	11.55	161.46	12.47	59.66
Minimum	1.26	1.48	2.45	6.53
75 percentil	1.47	1.78	2.97	7.67
25 percentil	1.39	1.66	2.78	7.21
90 percentil	1.54	1.88	3.10	8.12
99 percentil	1.97	2.42	4.22	10.67
99.9 percentil	5.07	5.79	7.18	15.52
Propustnost				
Přijímání (zprávy / s)	286.25	239.66	160.75	66.69
Odesílání (zprávy / s)	286.45	239.69	160.76	66.69
Přijímání (kB / s)	286.25	3834.50	10287.75	17071.86
Odesílání (kB / s)	286.45	3835.11	10288.77	17072.67

Tabulka B.1: Naměřené hodnoty latencí a propustnosti pro ActiveMQ

Broker	RabbitMQ			
Velikost zprávy	1kb	16kb	64kb	256kb
Počet zpráv	100000	100000	100000	100000
Latence (ms)				
Průměr	169.80	12.64	35.78	3285.03
Směrodatná odchylka	119.00	1.88	22.12	1833.69
Medián (50 percentil)	192.50	12.65	18.99	3387.17
Maximum	342.99	31.13	77.07	7021.65
Minimum	3.40	1.62	3.81	8.57
75 percentil	274.69	13.89	60.92	4346.37
25 percentil	15.85	11.43	16.41	1531.37
90 percentil	306.17	14.68	64.85	5665.47
99 percentil	333.22	17.34	70.26	6918.39
99.9 percentil	341.67	21.58	71.89	7000.45
Propustnost				
Přijímání (zprávy / s)	12655.44	3303.20	823.16	204.50
Odesílání (zprávy / s)	12806.82	3304.42	822.84	202.67
Přijímání (kB / s)	12655.44	52851.17	52682.02	52351.26
Odesílání (kB / s)	12806.82	52870.78	52661.44	51883.75

Tabulka B.2: Naměřené hodnoty latencí a propustnosti pro RabbitMQ

Broker	Apache Kafka			
Velikost zprávy	1kb	16kb	64kb	256kb
Počet zpráv	100000	100000	100000	100000
Latence (ms)				
Průměr	6.30	5.96	7.34	8.52
Směrodatná odchylka	3.46	2.75	3.52	6.37
Medián (50 percentil)	5.67	5.60	6.81	7.58
Maximum	144.17	139.60	236.85	967.99
Minimum	2.29	2.37	2.72	2.92
75 percentil	7.37	6.94	8.57	9.71
25 percentil	4.38	4.33	5.35	5.91
90 percentil	9.38	8.45	10.56	12.54
99 percentil	16.72	13.64	16.84	25.53
99.9 percentil	36.92	31.36	35.93	53.40
Propustnost				
Přijímání (zprávy / s)	29214.96	4813.18	1211.92	297.04
Odesílání (zprávy / s)	29226.23	4813.30	1202.10	297.69
Přijímání (kB / s)	29214.96	77010.84	77562.57	76041.70
Odesílání (kB / s)	29226.23	77012.83	76934.68	76209.70

Tabulka B.3: Naměřené hodnoty latencí a propustnosti pro Apache Kafka

Broker	Artemis			
Velikost zprávy	1kb	16kb	64kb	256kb
Počet zpráv	10000	10000	10000	10000
Latence (ms)				
Průměr	106.30	116.60	108.65	155.61
Směrodatná odchylka	44.18	60.32	34.81	59.98
Medián (50 percentil)	92.69	90.46	99.59	144.75
Maximum	511.78	685.19	492.63	948.88
Minimum	72.24	73.25	79.25	86.39
75 percentil	108.24	112.98	112.40	169.08
25 percentil	85.24	84.19	92.82	125.34
90 percentil	137.14	204.35	133.27	201.45
99 percentil	305.42	313.75	254.76	433.28
99.9 percentil	511.75	685.15	492.60	948.55
Propustnost				
Přijímání (zprávy / s)	6.32	5.74	6.06	3.77
Odesílání (zprávy / s)	6.34	5.76	6.07	3.77
Přijímání (kB / s)	6.32	91.77	387.90	964.36
Odesílání (kB / s)	6.34	92.15	388.61	966.14

Tabulka B.4: Naměřené hodnoty latencí a propustnosti pro Artemis

Broker	Amazon SQS			
Velikost zprávy	1kb	16kb	64kb	256kb
Počet zpráv	10000	10000	10000	10000
Latence (ms)				
Průměr	93073.59	81304.66	53754.69	138.16
Směrodatná odchylka	54307.76	46764.29	31254.97	71.33
Medián (50 percentil)	92520.73	81649.09	53193.45	115.66
Maximum	188504.56	163906.24	111104.76	697.18
Minimum	20.91	28.38	28.16	62.95
75 percentil	139421.36	121294.43	80562.24	160.21
25 percentil	46099.99	40955.28	26287.57	91.16
90 percentil	168539.15	145616.29	96328.49	226.89
99 percentil	186568.13	161922.60	108149.85	422.42
99.9 percentil	188281.91	163569.16	110593.83	598.66
Propustnost				
Přijímání (zprávy / s)	148.72	106.88	66.46	17.42
Odesílání (zprávy / s)	39.22	39.10	38.86	17.42
Přijímání (kB / s)	148.72	1710.01	4253.29	4458.55
Odesílání (kB / s)	39.22	625.65	2486.80	4458.79

Tabulka B.5: Naměřené hodnoty latencí a propustnosti pro Amazon SQS

C Uživatelská příručka

Výsledný projekt se skládá z několika samostatně spustitelných modulů. Celé řešení lze spustit na lokálním nebo v cloudovém prostředí (Amazon Web Services). V této příručce bude popsáno spuštění na lokáním prostředí, v případě spuštění v AWS bude poskytnut návod s odkazy na dokumentaci AWS.

C.1 Struktura projektu

Projekt se skládá z následujících složek:

- `camel` - Složka obsahující Java projekt obsahující 3 moduly:
  - `camel-common` - Projekt obsahující třídy s obecnou funkcionalitou používanou oběma aplikacemi.
  - `camel-consumer` - Spring Boot aplikace konzumenta.
  - `camel-producer` - Spring Boot aplikace producenta.
- `docker` - Složka obsahující `docker-compose` soubory pro spuštění jednotlivých brokerů.
- `postman` - Složka obsahující požadavky na server generované Postmanem.

C.2 Lokální prostředí

V případě rozběhnutí aplikace na lokálním prostředí jsou zapotřebí následující prerekvizity:

- Java 1.8 a vyšší,
- Maven verze 3.5.1 a vyšší,
- Docker,
- Postman.

Prvním krokem je spuštění brokeru. Broker lze jednoduše spustit v Docker kontejneru. Ve složce `docker` lze nalézt složky pro brokery ActiveMQ, RabbitMQ, Kafka, Artemis. V každé složce se nachází `docker-compose.yml` soubor s konfigurací daného Docker kontejneru. Pro spuštění brokeru je zapotřebí spustit příkaz `docker-compose up`.

Pozor na kombinaci brokerů ActiveMQ a Artemis, vystavují stejné porty a nemohou s danou konfigurací běžet zároveň. Mapování portů lze změnit v `docker-compose` souboru, Camel spring-boot konfigurace ale nepovoluje připojení k více instancím brokeru zároveň.

V případě Amazon SQS je třeba spustit službu v AWS prostředí a do konfiguračních souborů konzumenta a producera uvést potřebné údaje (access-key a secret-key).

Dále je třeba spustit konzumenta. Ve složce `camel/camel-consumer` je třeba spustit příkaz `mvn spring-boot:run`, který spustí aplikaci. Konzument se připojuje k brokerům nadefinovaným v `application.yml` konfiguračním souboru (nachází se ve složce `src/main/resources`). V případě, že nějaký broker neběží, aplikace se bude snažit pravidelně připojit a selhání logovat do konzole, je proto vhodné část konfigurace neběžícího brokeru odstranit.

Po úspěšném spuštění konzumenta přichází na řadu spuštění producenta. Aplikace producenta se nachází ve složce `camel/camel-producer`. Stejným způsobem, tedy příkazem `mvn spring-boot:run` lze dosáhnout spuštění aplikace. Producent nekonzumuje z fronty, tudíž o neexistujícím spojení informuje uživatele až ve chvíli, kdy se snaží do brokeru poslat zprávu. Opět je vhodné nepoužívané brokery odstranit z konfiguračního souboru.

Ve složce `postman` lze najít kolekci požadavků a definice prostředí pro požadavky spouštějící testy. V programu Postman lze dané soubory naimportovat a upravovat prostředí. V globální konfiguraci lze definovat počet generovaných zpráv, defaultní hodnota je 100 000. URL jednotlivých brokerů směřuje na adresu `localhost:5000`, adresu lze změnit v prostředí daného brokeru.

Po odeslání požadavku se spustí testování pro daný broker. Aplikace producenta i konzumenta průběžně logují průběh testování. Po dokončení testování lze najít detailní výsledky ve složce `camel/out`.

C.3 Cloudové prostředí

Návod pro cloudové prostředí se zaměřuje na spuštění v Amazon Web Services (AWS). V cloudu poběží jak broker (v EC2), tak producent s konzu-

mentem (v Elastic Beanstalk). Prvním krokem je tedy mít vytvořený účet v AWS. Všechny služby poběží v rámci VPC.

V případě brokeru je nejprve potřeba vytvořit Cluster (návod `https://docs.aws.amazon.com/AmazonECS/latest/userguide/create_cluster.html`).

Následně lze přistoupit k vytváření Task Definitons pro jednotlivé brokery (návod `https://docs.aws.amazon.com/AmazonECS/latest/userguide/create-task-definition.html`). Task Definition obsahuje stejné informace, jako `docker-compose` soubor, tudíž při vytváření je dobré vycházet z něho.

Posledním krokem je spustit Service na základě vytvořeného Task Definition (návod `https://docs.aws.amazon.com/AmazonECS/latest/userguide/create-service.html`).

Po spuštění lze v ECS konzoli vyhledat informace o přidělené IP adrese služby (většinou jsou vystavené 2, jedna v rámci VPC, jedna na internet). Adresu je třeba zadat do konfigurace aplikací producent a konzument.

V případě aplikací producent a konzument je postup totožný. Nejprve je třeba vytvořit spustitelný soubor příkazem `mvn clean install`. V Elastic Beenstalk je potřeba vytvořit nové prostředí a v něm vytvořit server z vytvořené aplikace (návod `https://docs.aws.amazon.com/elasticbeanstalk/latest/dg/GettingStarted.CreateApp.html`). Při vytváření je třeba nahrát do AWS vytvořenou aplikaci a následně v pokročilých nastaveních zapojit server do VPC. Po několika minutách bude server spuštěn a v logu lze ověřit, zda se správně připojil na broker.

Stejně jako v případě běhu na lokálním prostředí, testy jsou spouštěny z Postmana. Před spuštěním je potřeba v konfiguraci změnit URL serveru. Po dokončení testování se výsledky nacházejí na běžícím serveru, pro jejich získání je třeba se připojit před SSH.

D Seznam zkratek

- `ACL` Access Control List
- **AES** Advanced Encryption Standard
- **AMI** Amazon Machine Image
- **AMQP** Advanced Message Queuing Protocol
- **API** Application Programming Interface
- **AWS** Amazon Web Services
- **CA** Certificate Authority
- **DNS** Domain Name System
- **DSL** Domain Specific Language
- **DTO** Data Transfer Object
- **EC2** Elastic Compute Cloud
- **ECS** Elastic Container Service
- **EIP** Enterprise Integration Patterns
- **EPMD** Erlang Port Mapping Daemon
- **FIFO** First In, First Out
- **GPS** Global Positioning System
- **HTTP** Hypertext Transfer Protocol
- **IBM** International Business Machines
- **ID** Identity Documentation
- **IoT** Internet of Things
- **IP** Internet Protocol
- **IT** Information Technology
- **JAAS** Java Authentication and Authorization Service

- **JMS** Java Message Service
- **JMX** Java Management Extensions
- **JPEG** Joint Photographic Experts Group
- **KMS** Key Management Service
- **LDAP** Lightweight Directory Access Protocol
- **MQ** Message Queue
- **MQTT** Message Queuing Telemetry Transport
- **MSK** Managed Streaming for Apache Kafka
- **OS** Operační Systém
- **OWASP** Open Source Foundation for Application Security
- **PC** Personal Computer
- **RAM** Random Access Memory
- **REST** Representational State Transfer
- **SASL** Simple Authentication and Security Layer
- **SNS** Simple Notification Service
- **SOAP** Simple Object Accesss Protocol
- **SQL** Structured Query Language
- **SQS** Simple Queue Service
- **SSE** Server Side Encryption
- **SSH** Secure Shell
- **SSL** Secure Sockets Layer
- **STOMP** Streaming Text Oriented Messaging Protocol
- **SW** Software
- **TCP** Transmission Control Protocol
- **TLS** Transport Layer Security

- **URI** Uniform Resource Identifier
- **URL** Uniform Resource Locator
- **vCPU** virtual Central Processing Unit
- **VPC** Virtual Private Cloud
- **XML** Extensible Markup Language
- **YAML** YAML Ain't Markup Language