Mendelova univerzita v Brně
Provozně ekonomická fakulta

Clustering a load balancing serveru pro zpracování řeči

Diplomová práce

Vedoucí práce:
Ing. Jan Přichystal, Ph.D.

Bc. Miroslav Trnka

Brno 2017

Touto cestou bych rád poděkoval panu Ing. Janu Přichystalovi, Ph.D. za jeho cenné rady a připomínky během tvorby této práce. Také bych rád poděkoval firmě Phonexia s.r.o. za umožnění vzniku této práce. Rovněž své rodině a přátelům za neustálou podporu v dosažení vzdělání.

Čestné prohlášení

Prohlašuji, že jsem tuto práci: **Clustering a load balancing serveru pro zpracování řeči**
vypracoval samostatně a veškeré použité prameny a informace jsou uvedeny v seznamu použité literatury. Souhlasím, aby moje práce byla zveřejněna v souladu s § 47b zákona č. 111/1998 Sb., o vysokých školách ve znění pozdějších předpisů, a v souladu s platnou *Směrnicí o zveřejňování vysokoškolských závěrečných prací.*
Jsem si vědom, že se na moji práci vztahuje zákon č. 121/2000 Sb., autorský zákon, a že Mendelova univerzita v Brně má právo na uzavření licenční smlouvy a užití této práce jako školního díla podle § 60 odst. 1 Autorského zákona.
Dále se zavazuji, že před sepsáním licenční smlouvy o využití díla jinou osobou (subjektem) si vyžádám písemné stanovisko univerzity o tom, že předmětná licenční smlouva není v rozporu s oprávněnými zájmy univerzity, a zavazuji se uhradit případný příspěvek na úhradu nákladů spojených se vznikem díla, a to až do jejich skutečné výše.

V Brně dne 31. prosince 2016 .....

Abstract

Trnka, M. Load balancing and clustering of server for speech processing. Diploma thesis. Brno: Mendel University in Brno, 2017.

This paper deals with the possibilities for load balancing and clustering of an existing server for speech processing. The paper analyzes problems of load balancing and clustering. There are also described the concepts of network programming and options for I/O processing. A new design of a load balancer is created, fully customized for the needs of speech processing server. This newly designed load balancer is implemented and thoroughly tested.

Keywords

Load balancing, clustering, network programming, Netty, speech processing

Abstrakt

Trnka, M. Load balancing a clustering serveru pro zpracování řeči. Diplomová práce. Brno: Mendelova univerzita v Brně, 2017.

Tato práce se zabývá možnostmi load balancingu a clusteringu existujícího serveru pro zpracování řeči. V práci je rozebrána problematika load balancingu a clusteringu. Dále jsou zde popsány koncepty síťového programování a možnosti zpracování I/O. Rovněž je zde vytvořen návrh nově vzniklého load balanceru přizpůsobeného na míru potřebám serveru pro zpracování řeči. Tento nově navržený load balancer je implementován a podrobně testován.

Klíčová slova

Load balancing, clustering, síťové programování, Netty, zpracování řeči

Obsah

Seznam obrázků

Seznam tabulek

1 Úvod a cíl práce

1.1 Úvod

Jednou z populárních disciplín dolování dat (data mining) je bezesporu i zpracování řeči. Jedná se o pluri-disciplinární obor, využívající poznatků věd přírodních, technických i humanitních, který zaujímá své místo v mnoha oblastech lidské činnosti.

Jednou z firem, která vytváří technologie pro zpracování řeči, je firma sídlící v Brně s názvem Phonexia s.r.o. Portfolio technologií, které tato firma nabízí, je poměrně veliké a nechybí mezi nimi takové technologie, jakými jsou přepis řeči do textu, hledání klíčových slov nebo identifikace mluvčího. Jedním ze způsobů, jak firma Phonexia s.r.o. tyto technologie nabízí, je server pro zpracování řeči. Tento server zapouzdřuje řečové technologie, které zpřístupňuje pomocí univerzálního REST API.

Proces zpracování řeči je náročnou záležitostí na výpočetní výkon a liší se napříč technologiemi. Naštěstí jsou již v této době natolik výkonné stroje obsahující několikajádrové procesory, které celý proces zvládají v přijatelném čase. Problém ovšem nestává tehdy, kdy je potřeba zpracovat velké množství nahrávek za určitou časovou jednotku, nejčastěji den. Zde už může nastat situace, že i při využití nejvýkonnějšího HW není možné během jednoho dne toto množství zpracovat na jednom stroji. Je nutné tedy rozdělit práci mezi více fyzických strojů. Také z ekonomického hlediska může pořizování nejnovějšího hardwareu vyjít daleko dráž, než nákup více, méně výkonných strojů.

Poptávka po technologiích firmy Phonexia s.r.o. vzrůstá a často se jedná o požadavky na zpracování velkého množství dat. V současné době však firma nedisponuje možnostmi, jak by mohla takové řešení realizovat čistě svými produkty. A z tohoto důvodu vznikla tato práce.

1.2 Cíl práce

Cílem této práce je tedy navrhnout takové řešení, které by dokázalo efektivně load balancovat a clusterovat server pro zpracování řeči. Využitím tohoto řešení, spolu s několik servery pro zpracování řeči, by mělo vést k uspokojení všech požadavků na zpracování velkého množství dat.

2 Metodika

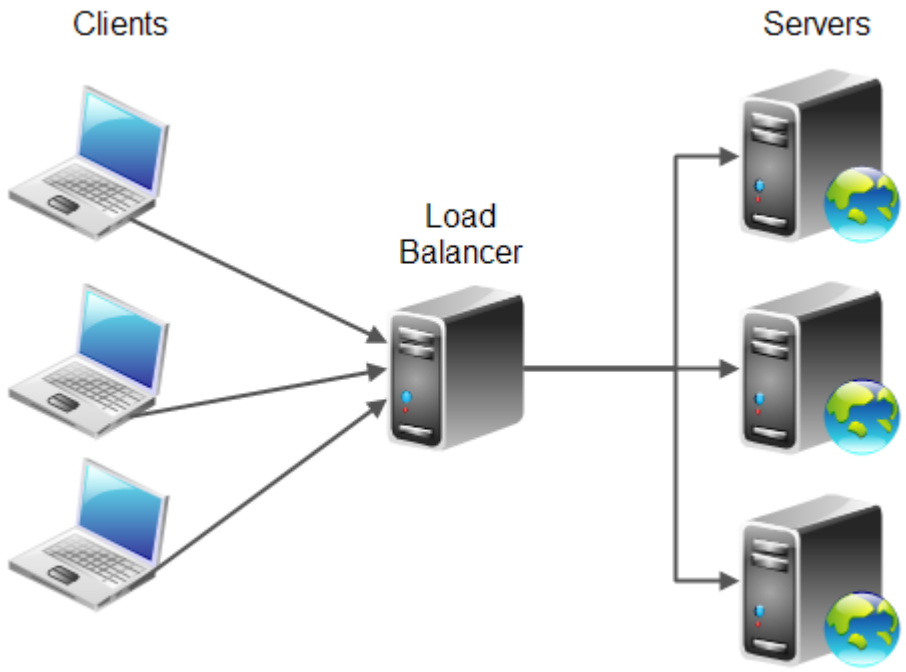
Aby mohlo vzniknout nějaké řešení, je potřeba si stanovit kroky, jak k takovému řešení dospět.

V první řadě je důležité seznámit se s problematikou load balancingu a clusteringu, pochopit úskalí, které tato doména obnáší a seznámit se s možnostmi, jak by mohlo výsledné řešení vypadat. Jelikož se nebude jednat o řešení s obecným účelem použití, je třeba se seznám s tím, jak server pro zpracování řeči funguje. Toto je naprosto klíčové, neboť bez plného porozumění jeho funkcionality a chování nelze navrhnout řešení postavené na míru potřebám serveru. Po seznámení se s funkcionalitou serveru by nemělo být složité vytvořit seznam požadavků, které by výsledné řešení mělo splňovat, a pustit se do zkoumání již existujících řešení. Dle výsledku předešlého kroku bude cíl práce naplněn pomocí již existujícího řešení, nebo bude implementováno nové softwareové řešení, kterému bude předcházet vytvoření abstraktního návrhu. Různá řešení mohou být různě výkonná, proto by v následující fázi mělo proběhnout důkladné testování, jehož výsledky by měly být podrobeny analýze. Na základě analýzy by pak měla vzniknout charakteristika chování. Závěrem by mělo dojít k celkovému zhodnocení práce.

Dojde-li k implementaci nového softwareového řešení, bude vývoj řízen pomocí agilní metodiky Extrémního programování. Vývoj bude probíhat ve 14-ti denních vývojových cyklech. Já i firma Phonexia s.r.o. budeme společně zastávat funkci zákazníka. Budeme tedy společně definovat požadavky a priority na software a následně již vzniklé funkcionality revidovat. Já sám budu zastávat pozici vývojáře. Budu navrhovat výsledný produkt spolu s jeho implementací a testováním.

3 Load balancing

Jedná se o proces distribuce síťového provozu mezi několik serverů. Celý proces load balancingu (rozložení zátěže) je pro koncového uživatele neprůhledný. Často celý síťový provoz je díky load balancingu rozložen mezi desítky až stovky koncových serverů. (Bourke, 2001) Proces load balancingu probíhá na několik vrstvách ISO/OSI modelu.[1]



Obrázek 1: Load balancing (Jenkov, 2016)

3.1 DNS-Based

(Bourke, 2001) zmiňuje jako jednu z prvních technik load balancingu tzv. DNS round robin. Podobně tuto problematiku popisuje i (Roth, 2008). Tato technika využívala funkcionality DNS, která umožňuje přiřadit více IP adres k jednomu hostname. Každý DNS záznam má tzv. A záznam[2], který mapuje doménové jméno (např. https://www.seznam.cz/) k IP adrese (77.75.77.53). Obvyklé je, že pouze jedna adresa je přiřazena k doménovému jménu. Tento A záznam může vypadat například takto:

[1] ISO/OSI model – `http://www.studytonight.com/computer-networks/complete-osi-model`

[2] A záznam – `https://napoveda.active24.cz/idx.php/32/100/article/`

```
https://www.seznam.cz/  IN      77.75.77.53
```

Použitím DNS round robin je možné nastavit více IP adres k jednomu doménovému jménu, což má za následek distribuci provozu mezi uvedené IP adresy. Pokud by jedeno doménové jméno mělo 3 adresy 77.75.77.53, 77.75.77.54 a 77.75.77.55, pak by A record vypadal takto:

```
https://www.seznam.cz/  IN      77.75.77.53
                        IN      77.75.77.54
                        IN      77.75.77.55
```

Přestože se daný způsob může jevit velmi jednoduchým a lehce použitelným, má však dle (Bourke, 2001) několik omezení, kterými jsou např.:

- nepředvídatelné rozložení provozu (zejména kvůli Caching problému) – kvůli snížení zátěže DNS serverů, se využívá nejrůznějších cache pamětí. Ukládáním do těchto cache pamětí má vést ke snížení počtu požadavků na překlad doménového jména. Při prvním dotazu na překlad doménového jména je odpověď DNS serveru uložena do této paměti s danou dobou platnosti. Platnost může trvat několik hodin, avšak i několik dní. Problém nastává tehdy, když při dalších požadavcích na překlad doménového jména je IP adresa vybrána z cache paměti a není kontaktován DNS server, který by navrátil IP adresu určenou rozložení zátěže.
- nepřítomnost opatření pro tolerance chyb
- pomalá rychlost propagování změn – v případech, kdy dochází k nečekanému nárůstu síťového provozu, nastává potřeba rychlého přidání nových serverů. Každá změna DNS záznamu však zabere jistý čas, aby se rozdistribuovala, což činí škálování infrastruktury tímto způsobem velmi obtížným.

3.2 Layer 4 (packet-based)

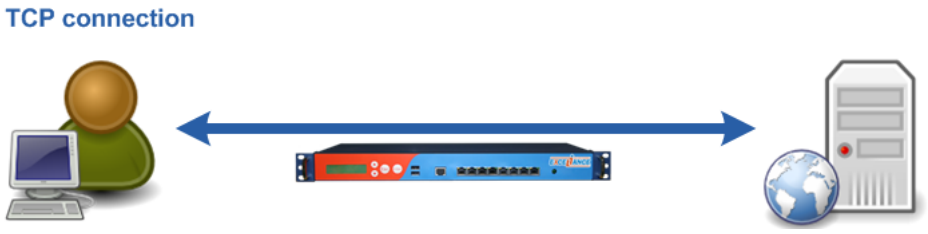
Proces load balancingu využívá, jako základ pro rozhodnutí, jak distribuovat klientské požadavky mezi skupinu koncových serverů, informací dostupných ve 4. vrstvě ISO/OSI modelu – transportní vrstvě. Jedná se o zdrojovou a cílovou IP adresu a číslo portu, uvedenou v hlavičce paketu bez toho, aby se zohledňoval obsah paketu. Rozhodování probíhá paket po paketu.

Je časté, že Layer 4 load balancery jsou dedikovaná hardwarová zařízení s proprietární load balancing softwarem přímo od výrobce. Pro zvýšení rychlosti bývají NAT operace prováděny specializovanými čipy raději než softwarem. (What Is Global server load balancing?, 2016)

Dle (Load balancing Frequently Asked Questions, 2016) se Layer 4 load balancing využívá v níže uvedených módech. Obdobně popisuje stejné módy i (Tarreau, 2006).

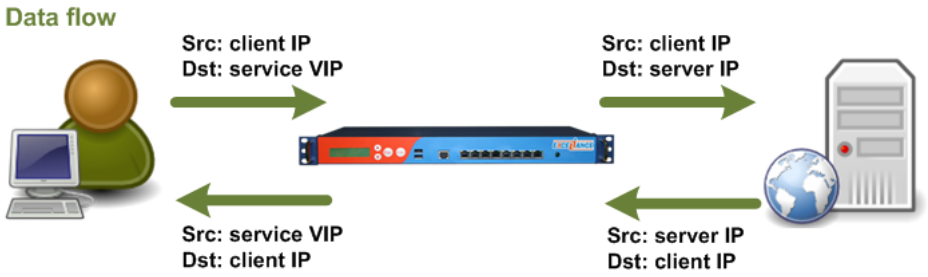
3.2.1 NAT or routed mode

V tomto módu load balancer routuje síťový tok mezi uživatelem a serverem tak, že paketům mění cílovou IP adresu. V případě forwardování odpovědí dochází také ke změně zdrojové IP adresy.



Obrázek 2: NAT or routed mode – TCP spojení (Load balancing Frequently Asked Questions, 2016)

TCP spojení je vytvořeno mezi klientem a serverem. Load balancer pouze zajišťuje to, že požadavky klienty jsou forwardovány vždy na stejný server.



Obrázek 3: NAT or routed mode – datový tok (Load balancing Frequently Asked Questions, 2016)

Výhody:

- rychlý load balancing
- snadné nasazení

Nevýhody:

- aby byly NAT operace korektně provedeny, musí být load balancer nastaven jako gateway pro backend servery
- výstupní propustnost sítě je limitována výstupní propustností load balanceru

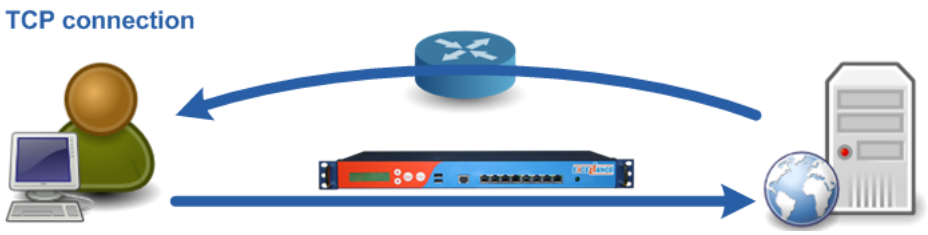
Kdy využít tento mód:

- když je potřeba velmi nízké odezvy
- když není potřeba využívat vlastností, které poskytuje Layer 7 load balancing

- když se výstupní propustnost load balanceru nestane v nejbližší době slabým místem infrastruktury

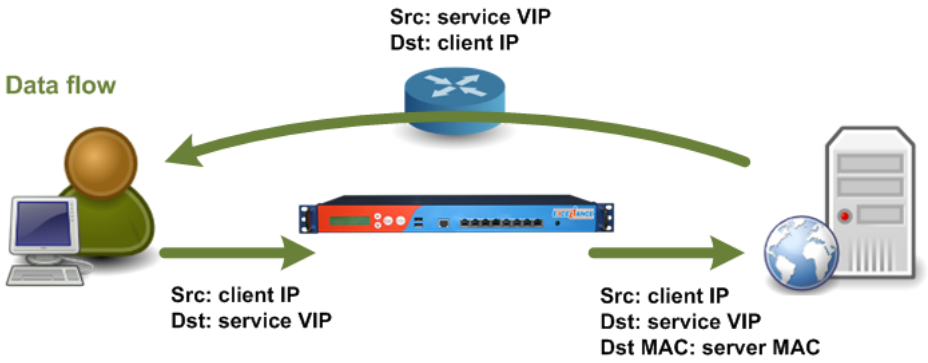
3.2.2 Direct Server Return or Gateway mode (DSR)

V tomto módu load balancer routuje pakety směrem k backend serverům s tím, že nemění nic jiného krom MAC adresy. Backend servery přijmou a zpracují příchozí požadavky a odpovědi jsou odeslány zpět klientům bez toho, aby prošly přes load balancer. Aby backend servery mohly přijímat klientské požadavky, je potřeba, aby VIP adresa[3] služby byla nastavena na všech backend serverech na loopback.



Obrázek 4: Direct Server Return mode – TCP spojení (Load balancing Frequently Asked Questions, 2016)

Jak je obvyklé u Layer 4 load balancingu, TCP spojení je navázáno přímo mezi klientem a serverem.



Obrázek 5: Direct Server Return mode – datový tok (Load balancing Frequently Asked Questions, 2016)

Výhody:

- velmi rychlý load balancing mód
- výstupní propustnost load balanceru není slabým místem infrastruktury

Nevýhody:

[3]Virtual IP – `http://www.webopedia.com/TERM/V/virtual_IP_address.html`

- VIP adresa služby musí být na všech backend serverech nastavena na loopback a nesmí odpovídat na ARP požadavky

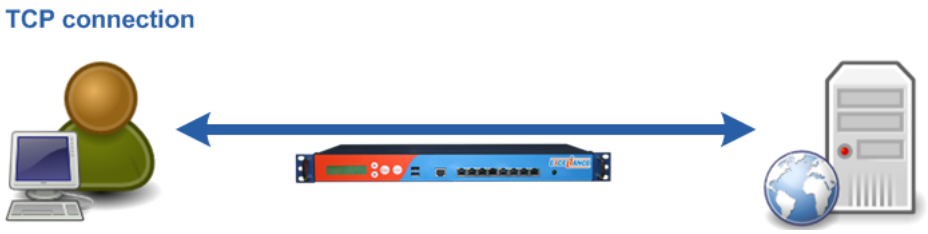
Kdy využít tento mód:

- když je potřeba velmi nízké odezvy
- když není potřeba využívat vlastností, které poskytuje Layer 7 load balancing
- když se může výstupní propustnost load balanceru stát slabým místem infrastruktury

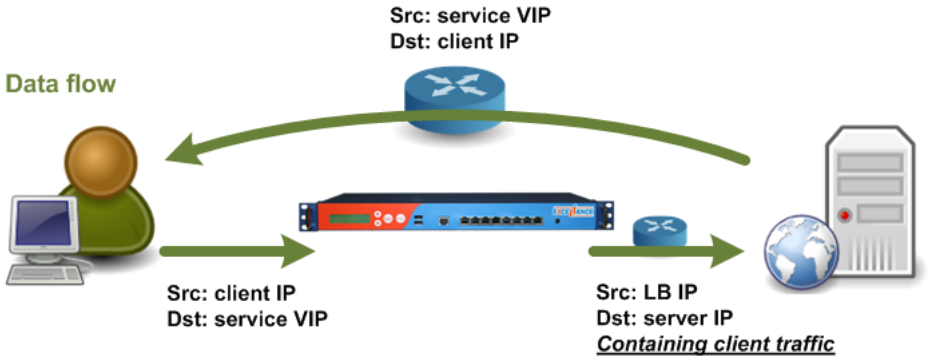
3.2.3 IP Tunnel mode

Tento mód je velmi podobný předchozímu s tím rozdílem, že síťový tok mezi load balancerem a backend serverem může být dále routován. Load balancer v tomto případě zapouzdří klientské požadavky do IP tunelu, který propojuje backend server s load balancerem.

Backend server poté přijme zapouzdřené požadavky, rozbalí je a zpracuje s tím, že odpověď poté forwarduje přímo klientovi.



Obrázek 6: IP tunnel mode – TCP spojení (Load balancing Frequently Asked Questions, 2016)



Obrázek 7: IP tunnel mode – datový tok (Load balancing Frequently Asked Questions, 2016)

Výhody:

- backend servery se mohou nacházet v několika datacentrech

- výstupní propustnost load balanceru není slabým místem infrastruktury

Nevýhody:

- backend servery musí podporovat IP tunneling

Kdy využít tento mód:

- když routování je jediný způsob, jak se připojit k backend serverům
- když není potřeba využívat žádnou z vlastností, které poskytuje Layer 7 load balancing

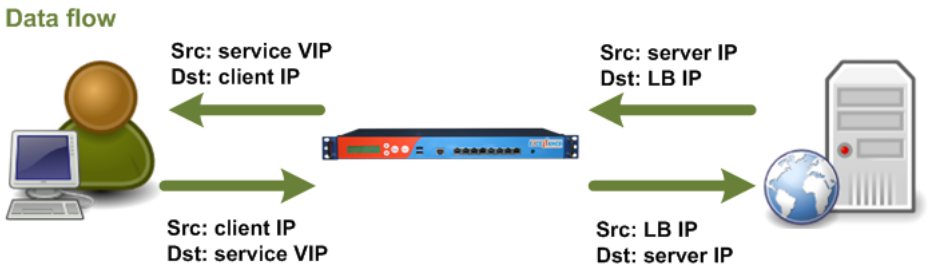
3.3 Layer 7 (application-based)

Proces load balancingu využívá stejných informací jako v případě Layer 4 (zdrojovou a cílovou IP adresu a číslo portu, vše uvedeno v hlavičce paketu) spolu s informacemi, které lze získat z aplikačních protokolů (v případě Internetu je HTTP dominantním protokolem na této vrstvě). Pakety jsou znovu sestaveny do podoby aplikačního protokolu, dle jeho obsahu je uskutečněno load balancing rozhodnutí (v případě HTTP se může jednat o URL, typ dat která jsou přenášena nebo o informace uložené v HTTP cookie). Layer 7 load balancing se využívá v níže uvedených módech. (Load balancing Frequently Asked Questions, 2016; What Is Global server load balancing?, 2016)

3.3.1 Proxy mode

Tomuto módu se také občas říká „reverse-proxy“ mód. Load balancer je uprostřed všech transakcí mezi klientem a serverem. Sám udržuje dvě oddělená TCP spojení:

- s klientem: load balancer zaujímá roli serveru – přijímá požadavky a forwarduje odpovědi
- se serverem: load balancer zaujímá roli klienta – forwarduje požadavky a přijímá odpovědi



Obrázek 8: Proxy mode – TCP spojení (Load balancing Frequently Asked Questions, 2016)

Obrázek 9: Proxy mode – datový tok (Load balancing Frequently Asked Questions, 2016)

Výsledkem tohoto módu je to, že klient komunikuje pouze s load balancerem, stejně tak jako server. Nikdy tedy nedojde k přímému navázání spojení mezi mezi klientem a serverem.
Výhody:

- load balancer nikterak nemění průchozí pakety
- bezpečnost – backend server není přímo dostupný
- dovoluje inspekci a validaci protokolů
- lze load balancovat služby, i když se nacházejí na stejné podsíti

Nevýhody:

- backend servery vidí pouze IP adresu load balanceru
- je o něco pomalejší než Layer 4 load balancing

Kdy využít tento mód:

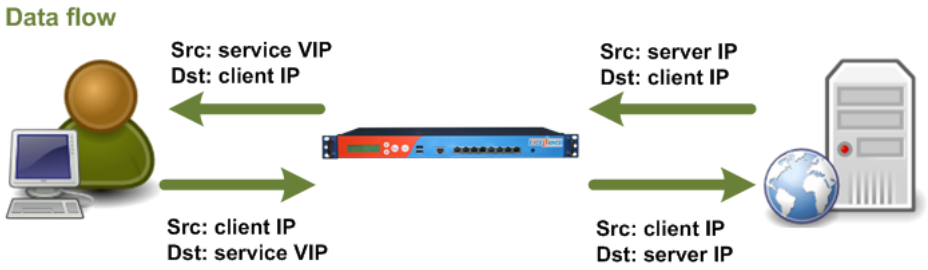
- když je potřeba aplikační logiky
- k ochraně backend serverů
- když běžící služba nepotřebuje znát klientskou IP adresu

3.3.2 Transparent proxy mode

Obdobně jako Proxy mode, load balancer udržuje dvě oddělená TCP spojení:

- s klientem: load balancer zaujímá roli serveru – přijímá požadavky a forwarduje odpovědi
- se serverem: load balancer zaujímá roli klienta – forwarduje požadavky a přijímá odpovědi

Rozdíl oproti Proxy mode je ten, že load balancer navazuje spojení s backend servery za použití klientské IP adresy jako zdrojové. V tomto případě musí být load balancer nastaven jako default gateway pro backend servery, jinak by odpovědi od serverů byly klientem zahozeny.



Obrázek 10: Transparent proxy mode – TCP spojení (Load balancing Frequently Asked Questions, 2016)

Obrázek 11: Transparent proxy mode – datový tok (Load balancing Frequently Asked Questions, 2016)

Výhody:

- backend server vidí IP adresu klienta
- bezpečnost – backend server není přímo dostupný
- dovoluje inspekci a validaci protokolů

Nevýhody:

- je o něco pomalejší než Layer 4 load balancing
- klient a server musí být v dvou různých podsítích

Kdy využít tento mód:

- když load balancovaná služba potřebuje znát IP adresu klienta již na síťové úrovni (např. anti-spam service)
- když je potřeba aplikační logiky
- k ochraně backend serverů

3.4 Global Server Load Balancing (GSLB)

Jedná se o inteligentní způsob distribuce síťového provozu mezi více datových center či serverů, které se geograficky nacházejí na různých místech. (What Is Global server

load balancing?, 2016; Bourke, 2001) Dle obou dvou zmíněných autorů se výhody GSLB dají shrnout do těchto bodů:

- spolehlivost a dostupnost – GSLB může vést ke zvýšení spolehlivosti a dostupnosti v momentech, kdy dojde k výpadku serverů nebo sítě. V případě, že je např. část země mimo dodávky elektrické energie, může load balancer routovat síťový provoz do jiných částí země. Často se využívá active-passive schématu – obsah je poskytován pouze z jednoho geografického zdroje s tím, že data jsou duplikována mezi jeden či více pasivních geografických zdrojů, které poskytují obsah až tehdy, kdy je hlavní zdroj nedostupný.
- výkon – dodáváním obsahu ze serverů, které se nacházejí nejblíže ke klientu, snižuje síťovou odezvu a také pravděpodobnost vzniku různých síťových problémů.
- vyhovění regulacím a bezpečnostním požadavkům – v mnoha odvětvích panuje velké množství regulací a bezpečnostních opatření. GSLB může přispět k tomu, aby tato opatření byla dodržena. GSLB může být nakonfigurován například tak, že global server load balancer bude forwardovat pouze požadavky do data centra nacházejícího se v určité zemi, pokud vznikly v této zemi.
- dodávání lokalizovaného obsahu – využitím geografických informací o klientu, může global server load balancer forwardovat požadavky na servery, které hostují obsah relevantní k zemi klienta.

3.5 Load balancing metody

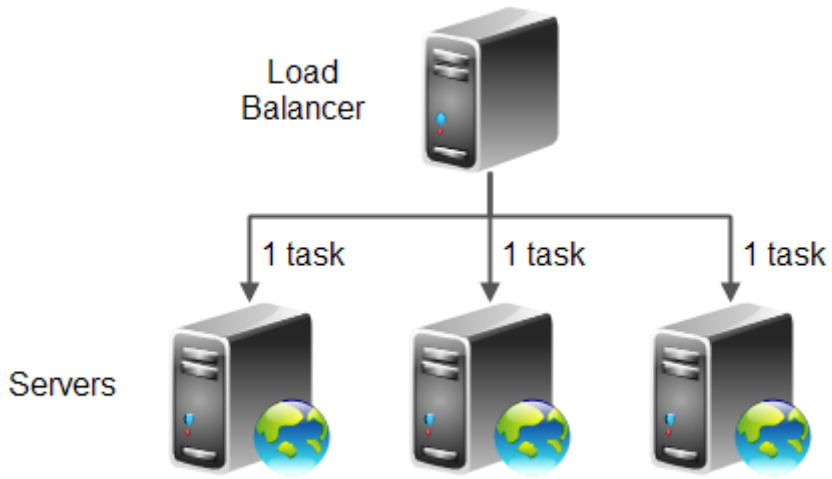
Existuje několik load balancing metod, jak distribuovat síťový provoz či požadavky klienta. Níže uvedený seznam metod je souhrnem těch nejčastěji používaných a často dostupných již v existujících řešeních.

3.5.1 Náhodně

Tato metoda distribuuje zátěž mezi servery v clusteru tak, že si náhodně vybere jeden server, kterému předá příchozí požadavek. Přestože tato funkcionalita je často nabízena existujícími load balancery, je zde otázka toho, jestli existuje vůbec nějaká využitelnost pro tuto metodu. (MacVittie, 2010; Jenkov, 2016)

3.5.2 Round Robin

Každý požadavek je distribuován směrem k dalšímu serveru v řadě. Tak dochází k rovnoměrnému rozložení zátěže. Tato metoda je velmi jednoduchá a lehce implementovatelná. (Membrey et al., 2012)



Obrázek 12: Round Robin metoda (Jenkov, 2016)

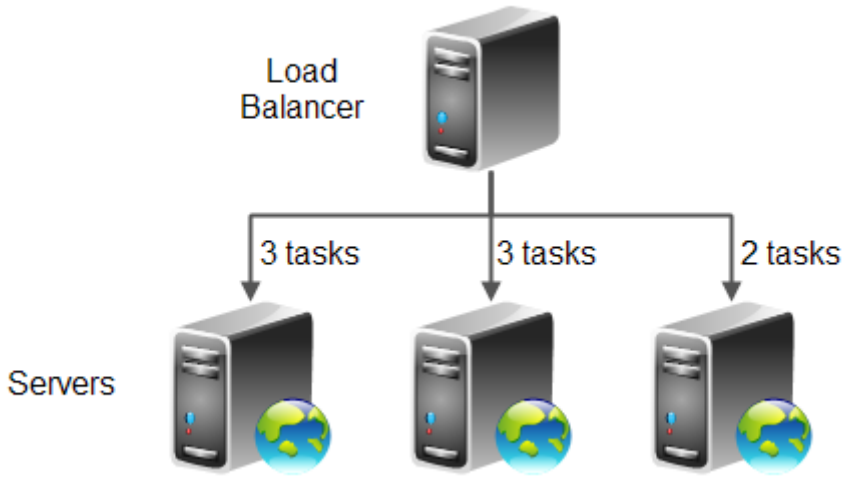
Toto rovnoměrné rozložení požadavků je vhodné použít tam, kde servery v clusteru mají stejné výpočetní kapacity a příchozí úlohy vyžadují statisticky stejný čas na vykonání.

Metoda ignoruje rozdíly v množství práce, která musí být vykonána, aby daný požadavek byl zpracován. To znamená, že v případě, kdy každý server má stejný počet požadavků, může nastat situace, kdy jeden server má více náročných požadavků (větší náročnost na HW prostředky) než ostatní. Tato situace může nastat díky náhodnosti příchozích požadavků. Často se také poté stává, že vzniklý rozdíl se sám vyrovná, neboť velmi vytížený server může také obdržet sadu nenáročných (nízké nároky na HW prostředky) požadavků.

I přesto, že tato metoda distribuuje jednotlivé požadavky rovnoměrně mezi servery v clusteru, neznamená to, že musí dojít k opravdovému rozložení zátěže. (Jenkov, 2016)

3.5.3 Ohodnocený Round Robin

Metoda vycházející z klasické Round Robin metody, s tím rozdílem, že distribuuje příchozí požadavky mezi servery v clusteru za použití stanovených vah. Serverům jsou nastaveny váhy, které zvýhodňují (znevýhodňují) servery při rozhodování, kam příchozí požadavky poslat. Použití tohoto schématu je vhodné tam, kde servery v clusteru nemají stejné výpočetní kapacity. (Membrey et al., 2012) Jako příklad lze uvést situaci, kdy jeden ze tří serverů má pouze 2/3 výpočetní kapacity zbylých dvou. Nastavením vah 3, 3, 2 se zapřičiní to, že první server obdrží 3 požadavky, druhý server obdrží 3 požadavky a třetí server obdrží 2 požadavky pro každých 8 příchozích požadavků. (Jenkov, 2016)



Obrázek 13: Ohodnocený Round Robin (Jenkov, 2016)

Tak jako v předešlém případě opět platí to, že toto schéma je vhodné použít tam, kde příchozí požadavky jsou statisticky stejně náročné na výpočetní čas.

3.5.4 Nejrychlejší odezva

Požadavky jsou distribuovány na základě toho, který ze serverů v clusteru má nejmenší odezvu. Využití této metody může být vhodné v prostředích, kde se servery nacházejí v různých logických sítích. Jistou nevýhodou této metody je to, že odezva v aktuálním momentě nemusí být stejná za několik sekund poté. Vhodnějším řešením by bylo udržovat průměrnou hodnotu odezvy serverů. (MacVittie, 2010)

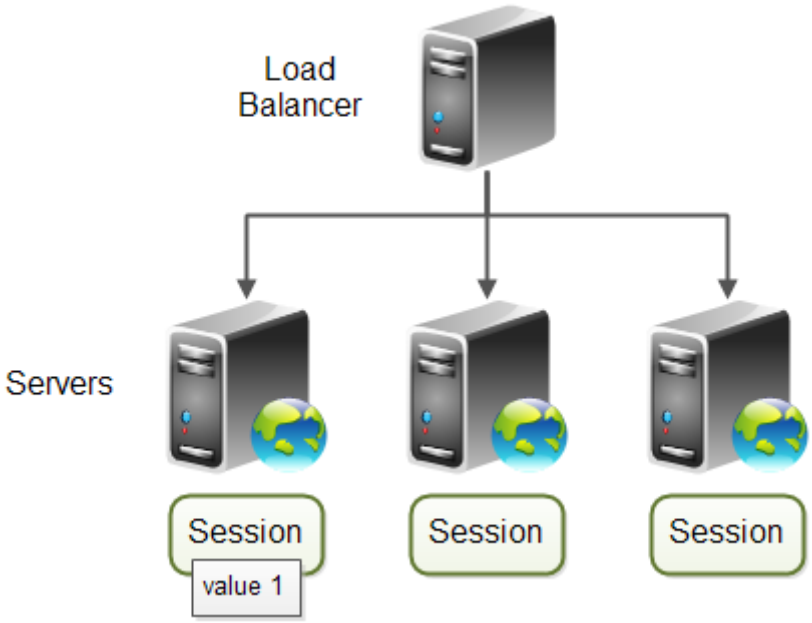
3.5.5 Hashing

Požadavky jsou distribuovány na základě uživatelem definovaných klíčů. Klíčem může být například URL, zdrojová IP adresa klienta, hodnota v hlavičce HTTP požadavku, atd. Extrahovaný klíč je většinou zahashován a uložen do mapovací tabulky. (What Is Layer 4 Load Balancing?, 2016)

3.5.6 Sticky Session

Všechny předchozí metody jsou založeny na předpokladu, že každý z příchozích požadavků může být zpracován nezávisle na již zpracovaných požadavcích. V případě, že je potřeba udržovat některá data na backend serverech (např. session ID), nastává u výše uvedených metod problém. Jako příklad lze uvést situaci, kdy se klient přihlásí do webové stránky. Jeho požadavek na přihlášení by byl odeslán load balancerem na server1. Ten by ověřil přihlašovací údaje a vygeneroval session ID které

identifikuje daného uživatele. Pokud klient pošle další požadavek identifikovaný pomocí jeho session ID, a tento požadavek bude load balancerem odeslán na server2, dojde k situaci, kdy server2 zamítne klientův požadavek, jelikož nemá uložena žádná data patřící k dané session uživatele, neboť veškerá data jsou uložena na serveru1.



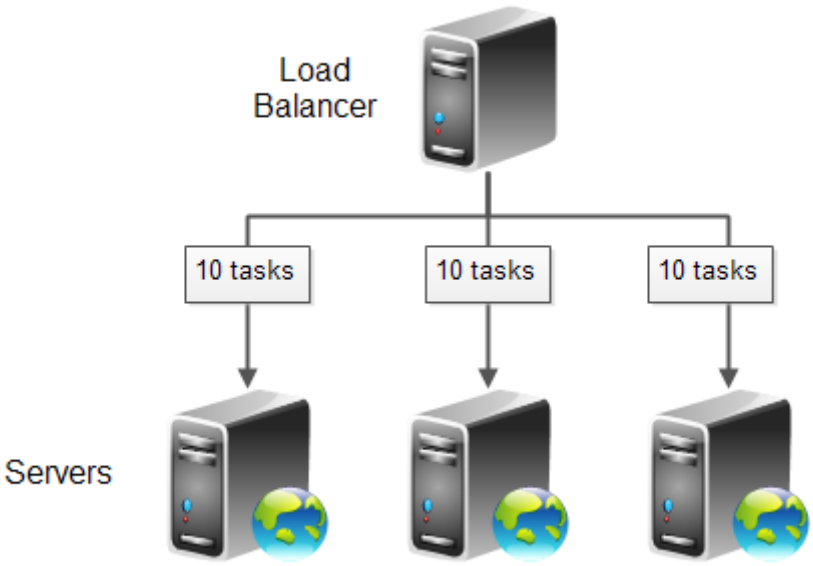
Obrázek 14: Sticky Session (Jenkov, 2016)

Řešením daného problému je metoda nazývaná jako Sticky Session Load Balancing. Load balancer si mapuje session ID k serverům v clusteru, a tak všechny příchozí požadavky (např. HTTP požadavek) patřící do stejné session (stejnému uživateli) jsou distribuovány vždy na stejný server. Tak je zajištěno, že veškerá uložená data v session potřebná následujícími požadavky (HTTP požadavek), jsou dostupná.

Další možností jak vyřešit problém s ukládáním dat patřící do session uživatele, je vůbec session nepoužívat. Možné je také řešení, které ukládá data pařící do session uživatele do databáze nebo na nějaký cache server. Jak databáze, tak cache server musí být dostupné všem serverům v clusteru. (Jenkov, 2016)

3.5.7 Even Size Queue

Load balancer si udržuje údaje o tom, kolik požadavků má každý server ve frontě. Fronta s požadavky obsahuje všechny požadavky, které jsou daným serverem aktuálně zpracovávány spolu s těmi, které čekají na zpracování.



Obrázek 15: Even Size Queue (Jenkov, 2016)

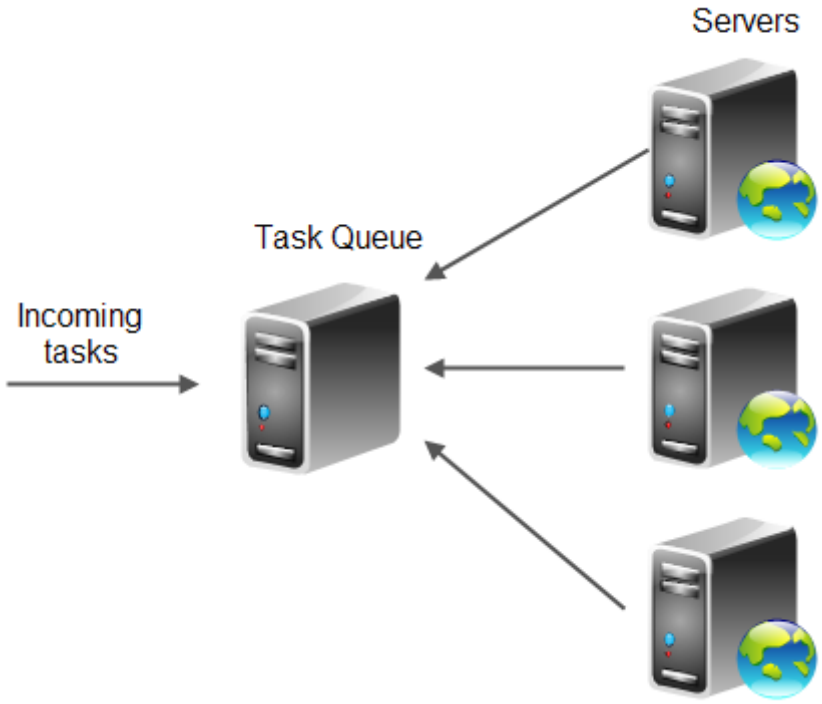
Když server dokončí zpracování požadavku (např. odešle klientovi HTTP odpověď), je požadavek pro daný server odstraněn z fronty. Tato metoda funguje tak, že se snaží v každém okamžiku vyrovnávat velikost fronty pro každý server. Servery s větší výpočetní kapacitou dokončí požadavky rychleji než servery s nižší kapacitou, z čehož plyne, že se fronta požadavků pro tyto servery odbavuje rychleji a tak jsou tyto servery schopny odbavit více požadavků

Tato metoda také implicitně zohledňuje náročnost na výpočetní čas pro jednotlivé požadavky. Nové požadavky jsou posílány na servery s nejmenším počtem požadavků ve frontě. Požadavky z fronty jsou odstraňovány až v okamžiku, kdy byly zpracovány, což značí, že výpočetní čas pro jednotlivé požadavky přímo ovlivňuje velikost fronty. Pokud je tedy jeden server dočasně přetížen, velikost jeho fronty se zvýší natolik, že jsou všechny nové požadavky distribuovány ostatním serverům, dokud se velikost všech front nevyrovná.

Toto schéma přináší také větší zátěž na load balancer, jelikož ten musí jednak sledovat počet požadavků ve frontě, ale také musí sledovat, které z požadavků jsou již hotovy, aby mohly být z fronty odstraněny. (Jenkov, 2016)

3.5.8 Autonomní fronta požadavků

Veškeré příchozí požadavky jsou uloženy ve frontě. Servery v clusteru se připojují do této fronty a berou si takový počet požadavků, který jsou schopny zpracovat.



Obrázek 16: Autonomní fronta požadavků (Jenkov, 2016)

V případě, kdy některý ze serverů v clusteru selže, zůstanou jeho potenciální požadavky nezpracovány ve frontě a jsou později zpracovány některými z dalších serverů v clusteru.

Výhodou tohoto řešení je to, že zde není žádný load balancer, který musí znát jaké servery se nacházejí v clusteru. Také fronta úloh nemusí znát, jaké servery se v clusteru nacházejí. Jediné, co je potřeba zabezpečit je to, aby servery v clusteru věděly, kde se nachází fronta s úlohami.

Toto schéma implicitně bere v potaz vytíženost serveru a jeho výpočetní kapacitu, neboť si server vždy vezme jen tolik, kolik je v daném okamžiku schopen zpracovat. (Jenkov, 2016)

3.6 Server Clustering

Ve většině definic je server cluster skupina minimálně dvou nezávislých serverů (často i osobních počítačů), které jsou logicky a často i fyzicky propojeny, avšak navenek jsou prezentovány jako jeden systém. Server v clusteru bývá nejčastěji nazýván jako „node“, někdy i jako „backend server“. Každý server disponuje svým vlastním hardwarem a vlastní instancí operačního systému. V řadě případů je však hardware i operační systém stejný. (Warner, 2005)

Hlavní výhodou organizování serverů do clusteru je dle (Warner, 2005) zajištění vysoké dostupnosti, spolehlivosti a škálovatelnosti pro služby, běžící na těchto serverech:

- vysoká dostupnost – reprezentuje schopnost poskytovat služby uživateli bez jakýchkoliv omezení, co se do výkonu a použitelnosti služby týče
- vysoká spolehlivost – znamená, že cluster disponuje odolností proti chybám díky redundanci serverů. Eliminuje se tak možnost nedostupnosti služby z důvodu selhání určitého subsystému (ať už se jedná o pevný disk, CPU, napájení, atd.) v rámci jednoho stroje.
- vysoká škálovatelnost – značí kapacitu síťového prostředí pro budoucí růst s myšlenkou na zvyšování výkonu. Výkon celého clusteru se může tedy zvyšovat přidáváním hardwareových komponent (např. procesor, RAM, pevný disk), ale také přidáváním nových nodeů do clusteru.

Veškeré aktivity serverů v clusteru jsou řízeny společnou vrstvou, díky které se všechny servery jeví jako jeden systém. Je-li touto vrstvou load balancer, mluví se často o tzv. „Load-Balanced Clusteru“.

4 Síťové programování

Cílem této kapitoly je osvětlit několik základních pojmů, na které se budu v dalších kapitolách odkazovat a souvisejících s programováním síťových aplikací. Jako programátor znám koncept síťového programování z jazyka Java. Následující příklady jsou uvedeny v tomto jazyce. Velmi analogicky je to ale v jiných jazycích.

4.1 Blocking I/O

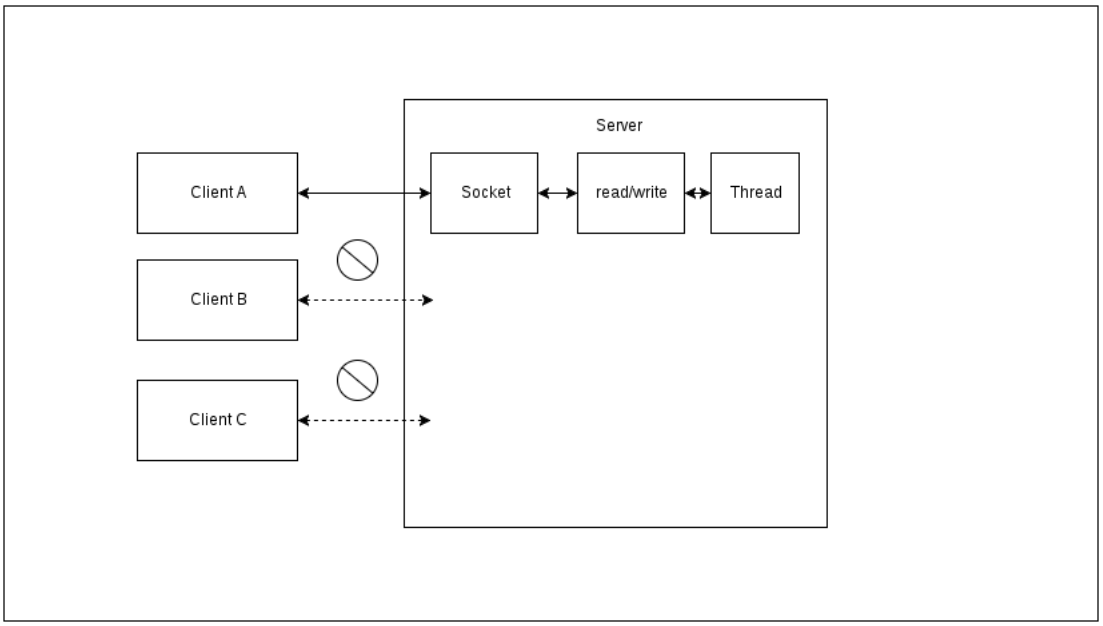
K implementaci architektury klient/server slouží v jazyce Java třída `ServerSocket`. Způsob, jak lze přijímat klientské požadavky konceptem blocking I/O popisuje (Maurer et al., 2016) následovně:

```
1     ServerSocket serverSocket = new ServerSocket(portNumber);
2     Socket clientSocket = serverSocket.accept();
3     BufferedReader in = new BufferedReader(new InputStreamReader(
          clientSocket.getInputStream()));
4     PrintWriter out = new PrintWriter(clientSocket.getOutputStream
          (), true);
5     String request, response;
6     while ((request = in.readLine()) != null) {
7         if ("Done".equals(request)) {
8             break;
9         }
10        response = processRequest(request);
11        out.println(response);
12    }
```

1. `ServerSocket` vyčkává na specifickém portu na příchozí spojení.
2. Začne-li navazování TCP spojení, metoda `accept()` blokuje vlákno, dokud není ustanoveno spojení se `ServerSocket`. Poté je vytvořen nový `Socket`, který zajišťuje komunikaci mezi klientem a server.
3. `BufferedReader` je třída odvozená z input streamu[4] třídy `Socket`. Tato třída převádí input stream do textové podoby.
4. `PrintWriter` je třída odvozená z output[5] streamů třídy `Socket`. Tato třída zapisuje textovou podobu do output streamu.
6. Metoda `readLine()` opět blokuje vlákno do té doby, dokud není předán řetězec „Done“, který by dané čtení ukončil, nebo dokud jsou v input streamu stále nějaká data, která lze přečíst.
10. Metoda `processRequest()` zpracovává klientské požadavky.

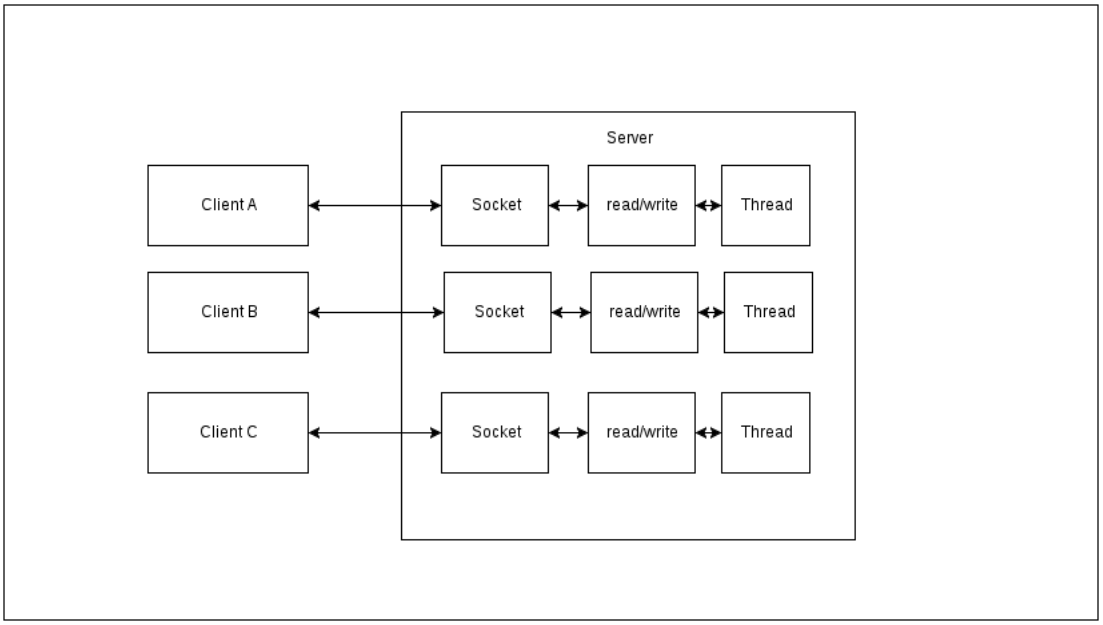
[4]Input stream – data vstupující do socketu
[5]Output stream – data vystupující ze socketu

V případě, že se klienti chtějí připojit k tomuto serveru, je zobrazeno na obrázku č. 17.



Obrázek 17: Blocking I/O – jedno vlákno (Ayedo, 2016), upraveno

Tento způsob implementace je schopen zpracovat pouze jedno připojení současně. Ostatní pokusy o připojení jsou odmítnuty. Řešením tohoto problému je přiřadit každému novému spojení jedno vlákno, tak jak je to zobrazeno na obrázku č. 18. (Ayedo, 2016)



Obrázek 18: Blocking I/O – více vláken (Ayedo, 2016), upraveno

Toto řešení má ovšem několik nevýhod. V každém okamžiku muže být mnoho vláken uspaných, pouze čekajících na to, až přijdou vstupní či výstupní data. Dost pravděpodobně se tato situace stane plýtváním výpočetních zdrojů. Další nevýhodou tohoto řešení je i to, že vlákno potřebuje pro svůj běh alokovat v paměti zásobník, jehož defaultní velikost se pohybuje od 64KB do 1MB, v závislosti na operačním systému. Poslední nevýhodou je skutečnost, že i když je JVM[6] schopno spravovat veliké množství vláken, velikost režie na jejich správu se stane problémovou dlouho před tím, než je dosažen hardwareový limit – zhruba v momentě dosažení 10000 souběžných spojení.

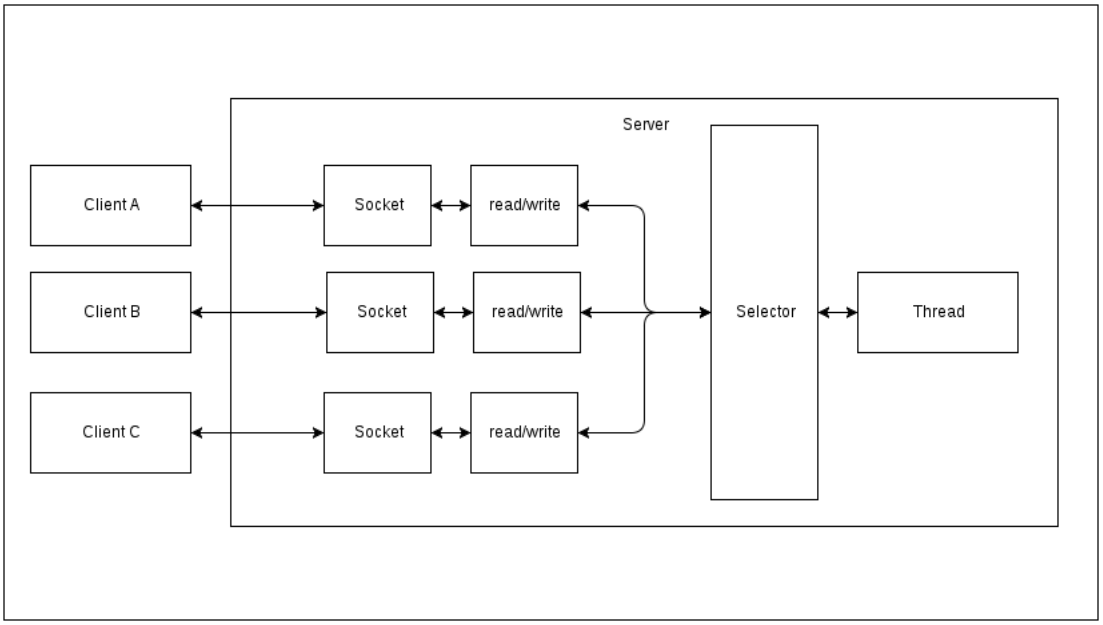
Tento způsob může být i přes všechna úskalí přijatelný pro malý až středně velký počet klientů, ne však pro podporu 100000 souběžných spojení. (Maurer et al., 2016)

4.2 Non-Blocking I/O

Opakem k blocking I/O je non-blocking I/O. Koncept je vystavěn na myšlence toho, že množina non-blocking socketů může pomocí volání systémového API dát najevo, že jsou připraveny pro čtení nebo zápis.

Na obrázku níže je zobrazen koncept non-blocking architektury, který eliminuje nedostatky předešlého návrhu. (Maurer et al., 2016)

[6]Java Virtual Machine – `http://www.theserverside.com/definition/Java-virtual-machine-JVM`



Obrázek 19: Non-Blocking I/O (Maurer et al., 2016), upraveno

Třída `Selector` je v jazyce Java základním pilířem non-blocking I/O implementace. Pomocí notifikačního API, reagujícího na události, může třída `Selector` zjistit, který socket je z množiny non-blocking socketů připraven pro I/O operace. Protože každá čtecí či zapisovací operace může být kdykoliv dotázána na její stav dokončení, může jedno vlákno obhospodařovávat mnoho souběžných spojení.

Díky výše zmíněným vlastnostem dokáže non-blocking I/O koncept lépe zacházet s výpočetními zdroji než blocking I/O. (Maurer et al., 2016)

- mnoho spojení může být obhospodařováno méně vlákny, a tedy s mnohem nižší režií na paměťový management a změnu kontextu[7]
- vlákna mohou vykonávat jiné úlohy, pokud aktuálně není určena žádná I/O operace ke zpracování

4.3 Event-driven paradigma a asynchronní zpracování

Event-driven paradigma, které je založeno na událostech, např. změnou stavu objektu nebo reakcí na uživatelský vstup, je vytvořena událost, ke které náleží nějaký naslouchač, který danou událost zachytí a může tak na ni zareagovat. (Event-driven programming, 2016)

Často se v kontextu zpracování I/O operací hovoří o asynchronním zpracováním. Jedná se o takové zpracování, kdy místo toho, aby se čekalo na dokončení zpracování úlohy (a blokovala se tak možnost vykonávat další sekvence kódu), je okamžitě navrácena reference, kde lze výsledek zpracování této úlohy někdy v budoucnu

[7]Změna kontextu – `http://www.linfo.org/context_switch.html`

vyzvednout. Sekvence kódu může tak dále pokračovat ve vykonávání ostatních úloh a někdy v budoucnu jsou výsledky vybrány a dále zpracovávány. (Urma et al., 2014)

Systém, který je synchronní a event-driven, se vyznačuje velmi užitečným chováním, tj. je schopen reagovat na příchozí události, které se objeví v jakémkoliv okamžiku, v jakémkoliv pořadí. Tato schopnost je klíčová k zajištění největší možné škálovatelnosti. Plně asynchronní I/O využívají asynchronních metod okamžitě navracejících reference, kde lze budoucí výsledek vyzvednout a také jsou schopny uvědomit uživatele o tom, že jejich zpracování již doběhlo. Díky non-blocking konceptu je pro třídu `Selektor` velmi jednoduché monitorovat desítky souběžných spojení a reagovat tak na události, které byly v rámci těchto spojení vytvořeny. (Maurer et al., 2016)

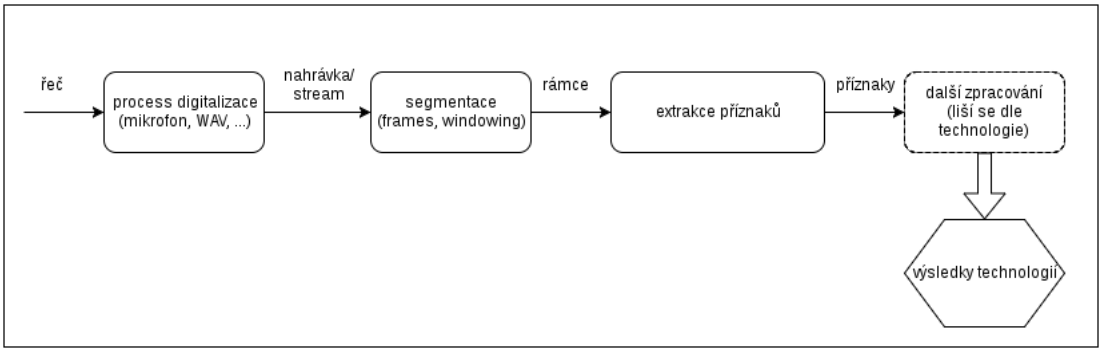
5 Analýza řeči

Analýza řeči, klasifikována dle (Han et al., 2012) jako jedna z podkategorií audio miningu, je v dnešní době velmi populární téma. Jde o dolování dat ze zvukové nahrávky lidské promluvy a hledání vzorů. Známou oblastí analýzy řeči je jistě přepis řeči do textu nebo zjišťování toho, zda v nahrávce zazněla hledaná slova. Ovšem tímto analýza řeči nekončí a může být až překvapivé, kolik se toho o řečníkovi dá z nahrávky zjistit. Zkoumat se může jeho pohlaví, věk, jakou řečí mluví a dokonce i emoční zabarvení jeho hlasu. Také oblast identifikace mluvčího má své široké uplatnění.

Mnoho společností a organizací se snaží technologiemi pro analýzu řeči zjednodušit nebo zlepšit život běžným lidem, potažmo je i chránit. Automatická analýza řeči přináší také nové způsoby jak přirozeně interagovat se zařízeními všeho druhu.

5.1 Zpracování řeči

Pro započetí fáze zpracování řeči je nejprve nutné pořídit záznam této řeči, tedy převést mluvenou řeč do digitální podoby. Častým problémem, který ovlivňuje výsledky zpracování je kvalita pořízeného signálu. Pro dosažení lepších výsledků se pro uložení záznamu řeči upřednostňují bezeztrátové formáty (např. WAV, FLAC, ...). U nahrávek horší kvality pak přichází na řadu předzpracování signálu různými filtry šumu, zkreslení, rozlišení technického signálu a jiné. (Zpracování řečových signálů, 2016; Psutka, 2006)



Obrázek 20: Zpracování řeči

Na obrázku 20 je nastíněn návrh jednoduchého workflow zpracování nahrávky. Na vstupu do procesu digitalizace je řeč. Ta je generována mluvčím a následně zachycena pomocí mikrofonu do audio záznamu (ideálně v bezeztrátovém formátu, případně v podobě real-time streamu). Tento datový tok se segmentací dělí na krátké časové intervaly, tzv. rámce (frames). K vyhlazení signálu se aplikuje tzv. windowing. Rámce jsou následně analyzovány a probíhá extrakce příznaků, kde se pro každý rámec měří, sleduje a hledá nějaká vlastnost promluvy. Tento proces je společný pro všechny řečové technologie. Jakým způsobem jsou extrahovaná data

dále zpracována, se už pak liší dle technologie. Využívána je k tomu celá řada různých algoritmů. (Zpracování řečových signálů, 2016; Psutka, 2006)

5.2 Phonexia s.r.o.

Firma Phonexia s.r.o. se zabývá vývojem řečových technologií od roku 2006 a v průběhu existence získala množství ocenění v mezinárodních soutěžích a pracuje také na projektech vypsaných Evropskou unií. Své úsilí se rozhodla investovat do vývoje následujících technologií:

1. Speech-to-Text (STT) – přepis řeči do textu
2. Keyword Spotting (KWS) – detekce klíčových slov
3. Speaker Identification (SID) – identifikace řečníka na základě hlasového otisku
4. Gender Identification (GID) – rozpoznání pohlaví řečníka
5. Age Estimation (AGE) – odhad věku řečníka
6. Language Identification (LID) – identifikace jazyka řečníka
7. Emotion Recognition (EMO) – rozpoznání emocí řečníka
8. Doplňující technologie

Tyto technologie a algoritmy pro zpracování řeči jsou zahrnuty v knihovně Brno Speech Core (BSCORE). Jako rozhraní pro přístup k BSCORE slouží Brno Speech Application Interface (BSAPI). BSCORE i BSAPI jsou napsány v jazycích C a C++. (Phonexia, 2015)

5.2.1 STT – Speech To Text

Jedná se o asi nejčastější využití v oblasti řečové analytiky, kdy se automaticky přepisují řečové nahrávky na slova, často do podoby vět. Využití této technologie je uplatněno v mnoha nejrůznějších odvětvích. Příkladem může být generování textů pro neslyšící (např. titulky pro videa) nebo se výsledný text může dále zpracovávat a analyzovat, a tak zjišťovat, zdali v řečových nahrávkách (případně i videích) zazněla některá slova, tedy je možné je v řečových nahrávkách vyhledávat. Technologie využívá akustických a jazykových modelů. (Zpracování řečových signálů, 2016; Psutka, 2006)

5.2.2 KWS – Keyword Spotting

Tato technologie slouží k rychlému hledání klíčových slov v řeči. Oproti technologii STT je rychlost hledání klíčových slov podstatně rychlejší. Obdobně jako STT i tato

technologie nejčastěji využívá akustických modelů postavených na neuronových sítích. Vhodným případem použití je real-time ovládání hlasem. (Zpracování řečových signálů, 2016)

5.2.3 SID – Speaker Identification

Jak již název napovídá, jedná se o identifikaci osoby na základě hlasu. Tato technologie nachází uplatnění zejména v oblasti bezpečnosti (hledání konkrétní osoby v neoznačených nahrávkách) a hlasové biometrie (buďto jako úplný nebo rozšířený způsob verifikace). Technologie používá pro uchování identity typ souboru nazývaný „voiceprint“. Zde bylo zvoleno podobné názvosloví jako u otisku prstu, tj. „fingerprint“. Jedná se o ztrátovou konverzi, čímž nelze voiceprint použít pro syntézu původního hlasu řečníka, je tedy anonymní. Technologie je založena na PLDA (Probabilistic Linear Discriminant Analysis). Verifikace dvou nebo více řečníků se děje procesem porovnávání voiceprintů. Technologie získává na své úspěšnosti pomocí kalibrace konkrétních dat. (Zpracování řečových signálů, 2016)

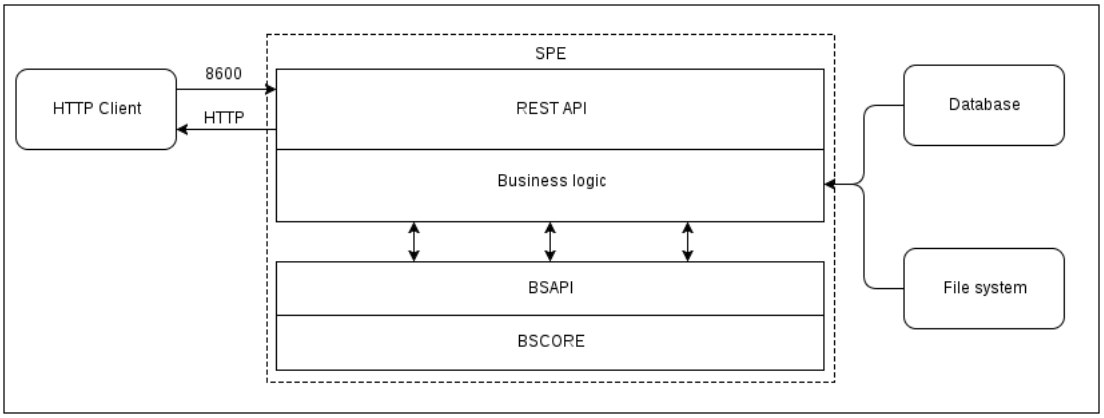
5.2.4 LID – Language Identification

Pomocí identifikace jazyka z nahrávky je možné směrovat hovory na správné telefonní linky podle toho, kterým jazykem klient mluví. Dále lze zjistit, odkud mluvčí pochází. Je-li k dispozici model i pro jednotlivé dialekty v regionu, lze celkem přesně určit místo původu dané osoby. Pro tuto technologii je využíváno dvou přístupů, kterými jsou GMM (Gaussian Mixture Model) a PRLM (Phoneme Recognition followed by Language Model). (Zpracování řečových signálů, 2016)

6 Server pro zpracování řeči

V rámci podkapitoly 5.2 byla popsána jedna z možností, jak firma Phonexia s.r.o. dodává své řečové technologie zákazníkům. Dlouhou dobu bylo BSAPI jediným možným způsobem, jak řečové technologie distribuovat. Postupem času se toto řešení ukazovalo jako nepříliš vhodné, převážně z důvodu veliké časové investice do zákaznické podpory pro integraci této knihovny. Ne všechny produkty, do kterých mělo být BSAPI integrováno, byly napsány v jazyce C++, proto vznikaly nejrůznější verze wrapperů pro jiné programovací jazyky[8]. Také externí ovládání této knihovny přes konzoli nebylo příliš praktické pro začlenění do zákaznického softwareu. Proto se hledal nějaký univerzální způsob, kterým by šlo zpřístupnit BSAPI, avšak za zachování rychlost, kterým jazyk C++ disponuje.

Nakonec se ukázalo, že velmi vhodné řešení je zapouzdřit BSAPI knihovnu do „lightweight“ HTTP nadstavby, která zpřístupňuje funkce této knihovny skrze univerzální REST API. Tímto způsobem vznikl server pro zpracování řeči – Phonexia Speech Engine (dále jen SPE server). HTTP nadstavba je stejně jako BSAPI napsána v jazyce C++, což umožňuje snadné propojení s knihovnou BSAPI. Následující obrázek ilustruje architekturu SPE serveru.



Obrázek 21: Architektura SPE serveru

Komunikace s tímto serverem probíhá přes následující HTTP metody:

- GET – pro získání výsledků nebo dat
- POST – pro nahrání dat na server
- DELETE – pro smazání dat
- PUT – pro aktualizaci stávajících dat

Předávání data lze uskutečnit jak ve formátu XML, tak i ve formátu JSON, který je výchozím formátem.

[8]Jedním z nich byl například jazyk Java nebo C#

6.1 Zpracování požadavků

Požadavky odeslané na SPE server jsou dle náročnosti operace zpracovávány dvěma následujícím způsoby:

- Synchronní zpracování – jedná se o zpracování nenáročných požadavků. Výsledek tohoto zpracování je vždy vrácen v rámci HTTP odpovědi na tento požadavek.
- Asynchronní zpracování – jedná se o zpracování náročných požadavků. Téměř všechny požadavky na zpracování řečové nahrávky nějakou technologií jsou asynchronně zpracovány. Po odeslání požadavku, který je asynchronně zpracováván, je v rámci HTTP odpovědi odesláno ID tohoto asynchronního zpracování. Klient se tak může díky tomuto ID cyklicky dotazovat na průběh tohoto zpracování. Celý proces asynchronního zpracování je popsán v podkapitole 6.1.1.

6.1.1 Asynchronní zpracování

Požadavky na asynchronní zpracování byly vytvořeny převážně z důvodu dlouhého čekání na skončení zpracování řečových nahrávek řečovými technologiemi. Postup asynchronního zpracování probíhá následujícím způsobem:

- Klient zašle požadavek na asynchronní zpracování. Například se může jednat o požadavek na zpracování nahrávky pomocí STT technologie.

```
GET /technologies/stt?path=/path/to/recording&model=ENGLISH&
    result_type=one_best"
```

- SPE server navrátí HTTP status kód 202. V hlavičce této odpovědi se nachází parametr `Location` spolu s URI `/pending/<32-bit-hash>`. Tímto server dává najevo, že se klient má na stav asynchronního zpracování dotazovat na tomto URI.

```
< HTTP/1.1 202 Accepted
< Date: Sat, 06 Aug 2016 09:11:57 GMT
< Connection: Close
< Content-Type: application/json
< Location: /pending/54048266-b694-440a-8a9e-0cef5668efea
```

- V tomto bodě se klient cyklicky dotazuje na stav asynchronního zpracování, dokud mu není navrácen HTTP status kód 303, který v hlavičce obsahuje URI `/done/<32-bit-hash>`. Tato URI souží k vyzvednutí výsledku asynchronního zpracování.

```
< HTTP/1.1 303 See Other
< Date: Sat, 06 Aug 2016 09:12:03 GMT
< Connection: Close
< Content-Type: application/json
< Location: /done/54048266-b694-440a-8a9e-0cef5668efea
```

- Dotázáním se na tuto URI je pak klientovi navrácen výsledek ve formátu XML nebo JSON.

6.1.2 WebSocket

Alternativou k cyklickému dotazování se na stav asynchronních operací je využití WebSocket protokolu. Jedná se o protokol schopný obousměrně komunikovat v rámci jednoho TCP spojení. Tento protokol využívá HTTP protokolu, pomocí kterého zasílá požadavek na povýšení spojení. (Ubl et al., 2010) Poté, co klient obdrží ID asynchronního zpracování, může dojít k navázání WebSocket spojení s SPE serverem, čímž odpadá nutnost cyklicky se dotazovat na stav, neboť spojení zůstává otevřeno tak dlouho, dokud asynchronní zpracování není dokončeno a SPE server odešle výsledky zpět klientovi.

6.1.3 RTP/HTTP streamy

Několik technologií, které firma Phonexia s.r.o. vytváří, dokáží pracovat v tzv. „online módu“. Vstupem do takovýchto technologií je sekvence data (stream). Technologie reagují na každou část tohoto streamu a provádějí její zpracování. V každém okamžiku zpracování tohoto streamu se lze dotazovat na již zpracované výsledky.

6.2 Clustering

V době psaní této práce disponoval SPE server dvěma vlastnostmi, které umožňovaly vytvořit cluster SPE serverů:

- více SPE serverů umělo sdílet společné úložiště pro přístup k řečovým nahrávkám.
- více SPE serverů umělo sdílet společnou databázi.[9]

Posledním krokem k dosažení cíle práce bylo vyřešit problematiku load balancingu.

[9]Zde se ukládali uživatelé, seznamy klíčových slov, voiceprinty atd.

7 Analýza požadavků

V současné době je již SPE nasazeno u několika zákazníků, kteří využívají buďto veškeré jeho funkcionality nebo jenom části. Aby bylo možné řešení efektivně nasadit všem novým i stávajícím zákazníkům, je třeba, aby splňovalo všechny aspekty funkcionality SPE serveru. Po diskuzi s kolegy ve firmě byly sestaveny nejdůležitější body, které jsou shrnuty v následujícím výčtu:

- sledování asynchronních požadavků
- udržování session pro uživatele
- možnost povýšení HTTP spojení na WebSocket
- podpora UDP pro RTP streamy
- kontrolování dostupnosti backend serverů
- monitoring
- upravování HTTP hlaviček
- logování
- open source
- implementace vlastní load balancing metody

7.1 Analýza existujících řešení

Již hotových řešení existuje celá řada. Probíráním se tímto množstvím jsem mezi možné kandidáty zařadil tyto produkty:

7.1.1 HAProxy

HAProxy[10] je o open source software, který se vyznačuje velikou spolehlivostí a rychlostí, umožňující load balancing TCP spojení spolu s HTTP požadavky. Mezi jeho funkcionality lze zařadit možnost fungování jako proxy pro TCP spojení nebo reverse-proxy (pracuje v proxy módu) pro HTTP protokol. Je napsaný v jazyce C, díky čemuž získal pověst velmi rychlého a efektivního (co se týče využití procesoru a paměti) softwareu.

Využívá single-threaded, non-blocking, event-driven konceptů optimalizovaných pro to, aby příchozí data forwardoval co nejrychleji s minimálním počtem operací.

[10]HAProxy – `http://haproxy.com/`

7.1.2 Apache httpd

Apache httpd[11] je světově nejpoužívanější open source webový server. Jde o velmi komplexní a modulární projekt. Využitím modulu mode_proxy [12] lze docílit toho, aby se httpd choval jako load balancer pro protokoly HTTP, FTP a AJP13. Produkt je udržován komunitou a je stále ve vývoji.

7.1.3 NGINX plus

NGINX[13] plus nabízí application-based load balancer, který dokáže balancovat HTTP požadavky, TCP i UDP spojení. Disponuje možnostmi výběru jedné z více load balancing metod, cyklické kontroly backend serverů a mimo jiné i schopností udržovat session uživatele. Jistou nevýhodou je skutečnost, že není open source.

7.1.4 Varnish

Varnish[14] je především akcelerátor pro webové aplikace, známý také jako caching HTTP reverse proxy. Přestože hlavním účelem je caching webového obsahu, lze tuto možnost vypnout a software používat čistě jako load balancer v proxy módu.

7.1.5 ExaProxy

ExaProxy[15] je high-performance non-caching load balancer pracující v proxy módu, napsaný v jazyce Python, určený k distribuci a filtraci příchozích HTTP požadavků. Poslední vydaná verze před začleněním do Exa Networks SurfProtect[16] vyšla v roce 2014.

7.1.6 Apache Traffic Server

Apache Traffic Server[17] je rychlý, škálovatelný HTTP/1.1 software, který převážně funguje jako caching proxy, nebo load balancer v proxy módu. Původně se jednalo o komerční produkt společnosti Yahoo!, která tento software darovala Apache Software Foundation.[18]

[11] Apache httpd – `https://httpd.apache.org/`

[12] mode_proxy – `https://httpd.apache.org/docs/2.2/mod/mod_proxy_balancer.html`

[13] NGINX Plus – `https://www.nginx.com/solutions/load-balancing/`

[14] Varnish – `https://www.varnish-cache.org/intro/index.html`

[15] ExaProxy – `https://github.com/Exa-Networks/exaproxy`

[16] Exa Networks SurfProtect – `http://www.exa.net.uk/business/application/surfprotect.html`

[17] Apache Traffic Server – `https://trafficserver.apache.org/`

[18] Apache Software Foundation – `https://www.apache.org/`

7.1.7 squid

Squid[19] je caching proxy pro webové aplikace podporující HTTP, HTTPS, FTP a další protokoly. Svojí činností snižuje síťový provoz a zlepšuje odezvu tím, že cacheuje stránky a výsledky často žádaných webových stránek. Disponuje také možností přepnutí do proxy módu a distribuovat příchozí požadavky mezi backend server.

7.1.8 Pound

Pound[20] je load balancer pracující v proxy módu, ale také HTTP frontend webový server. Původně byl vyvinut proto, aby umožnil jednoduchou distribuci síťového provozu mezi webové servery a poskytl jim jednoduchý SSL wrapper, pokud tuto funkcionalitu nativně nepodporují. Software je distribuován pod GPL licencí.

7.1.9 Perlbal

Perlbal[21] je load balancer pracující v proxy módu a webový server. Je napsaný v programovacím jazyce Perl. Architektonicky se jedná o single-threaded, event-driven, non-blocking server pro HTTP load balancing a byl navržen tak, aby šel konfigurovat za provozu, bez nutnosti restartu. Jeho silnou stránkou je také možnost přidání nejrůznějšího množství pluginů, které rozšiřují jeho funkcionalitu.

7.2 Výsledné srovnání

Následující dvě tabulky srovnávají výše uvedené existující řešení s předem stanovenými požadavky.

[19] squid – `http://www.squid-cache.org/`
[20] Pound – `http://www.apsis.ch/pound/`
[21] Perlbal – `https://github.com/perlbal/Perlbal`

Tabulka 1: Výsledné srovnání – požadavky 1

	Sledování asynchronních požadavků	Session	WebSocket	UDP pro RTP	Kontrola backend serverů
HAProxy	A	A	A	N	A
httpd	A	A	A	N	A
NGINX plus	A	A	A	A	A
Varnish	N	A	N	N	A
ExaProxy	N	A	N	N	N
Apache Traffic server	N	A	A	N	A
squid	N	A	N	N	A
Pound	A	A	N	N	A
Perlbal	Plugin	A	N	N	A

Tabulka 2: Výsledné srovnání – požadavky 2

	Monitoring	Open source	Upravování HTTP hlaviček	Logování	Vlastní load balancing metody
HAProxy	A	A	A	A	N
httpd	A	A	N	A	N
NGINX plus	A	N	A	A	N
Varnish	N	A	N	N	N
ExaProxy	N	A	N	N	N
Apache Traffic server	N	A	N	A	N
squid	N	A	N	N	N
Pound	A	A	N	N	N
Perlbal	Plugin	A	A	A	N

7.3 Diskuze

Analýzou obou výše zmíněných tabulek lze odvodit, že žádné s porovnávaných řešení nesplňuje všechny požadavky, které je nutné splnit. Nejblíže se k požadavkům přiblížila dvě řešení – HAProxy a NGINX plus. V obou dvou případech ale nevyhověla požadavku „implementace vlastní load balancing metody“, který je klíčový pro smysluplný load balancing SPE serveru. Z tohoto důvodu nezbývala jiná možnost, než navrhnout a implementovat řešení vlastní.

8 Návrh řešení

Vyhodnocením výsledků srovnávání existujících řešení a následné interní diskuze v rámci firmy bylo rozhodnuto, že se přistoupí k implementaci vlastního řešení postaveného na míru určeným požadavkům. Výsledné řešení by mělo být lehce konfigurovatelné, mělo by mít co nejmenší nároky na výpočetní výkon a mělo by být schopno obsloužit co největší množství souběžných připojení.

Pro implementaci tohoto řešení jsem zvažoval dva programovací jazyky – Go[22] a Java. Jelikož jsem Java vývojář a s tímto jazykem mám největší zkušenosti, rozhodl jsem se pro tento jazyk. Jazyk sám o sobě má i vhodné vlastnosti a to že je to jazyk kompilovaný, multiplatformní a poskytuje dostatečnou abstrakci nad nízkoúrovňovými operacemi.

8.1 Výběr frameworku

V poslední době je velmi vhodné se pro nejrůznější projekty poohlédnout po nějaké knihovně či frameworku, který často poskytuje další vrstvu abstrakce pro daný případ použití, ale také svým návrhem a použitím reflektuje osvědčené způsoby, jak danou problematiku řešit. Ve světě jazyka Java jsou pro síťové programovaní nejčastěji zmiňovány a používány dva open source frameworky:

- Apache MINA [23]
- Netty [24]

Porovnáváním referencí a doporučení jsem nakonec vybral framework Netty. Příznivější pro Netty bylo to, že umí „out-of-the-box“[25] pracovat s HTTP a s WebSocket protokolem. Také řada velkých firem ho ve svých produktech využívá jako základ pro síťové aplikace. Pro příklad lze uvést firmy jako je Twitter[26] nebo Facebook[27]. Lze jmenovat i několik webových frameworků, které jsou postaveny na Netty, jako je například čím dál populárnější Play[28] nebo Vert.x[29]. Posledním rozhodujícím bodem bylo objevení odborné práce[30], ve které jsou detailně a názorně vysvětleny základní principy práce s tímto frameworkem.

[22] Go – `https://golang.org/`
[23] Apache MINA – `https://mina.apache.org/`
[24] Netty – `http://netty.io/`
[25] není potřeba vlastní implementace
[26] Twitter – `https://twitter.com/`
[27] Facebook – `https://www.facebook.com/`
[28] Play – `https://www.playframework.com/`
[29] Vert.x – `http://vertx.io/`
[30] Netty in Action – `https://www.manning.com/books/netty-in-action`

8.2 Netty framework

Netty je non-blocking, event-driven, asynchronní framework, který umožňuje rychlý a jednoduchý vývoj síťových aplikací, jak klientských tak i serverových. Značně zjednodušuje a urychluje síťové programování v rovině TCP a UDP spojení. Framework vytváří lehce použitelnou abstrakci nad standardním síťovým Java API, avšak řadu existujících věcí reimplementuje dle zkušeností tvůrců pro zjednodušení jeho použití či zvýšení výkonu. (Netty, 2016) Mezi jeho hlavní přednosti dle zmíněné dokumentace patří:

- jednotné API pro blocking and non-blocking sockety
- jednoduchý, ale velmi účinný model pro správu vláken
- podpora pro UDP sockety
- řetězení logických komponent k zajištění jednoduchého znovupoužití
- žádné externí závislosti, pouze JDK[31] 1.6 a vyšší
- větší propustnost a menší odezva než při použití standardních Java API
- snížení nároků na výpočetní prostředky díky poolingu[32] vláken a jejich znovupoužití
- minimální kopírování objektů v paměti
- plná podpora SSL/TLS a StartTLS
- stále vyvíjen komunitou

8.2.1 Channels

Klíčová komponenta Channel reprezentuje navázané spojení k entitě, kterou může být například hardwareové zařízení, soubor, síťový socket nebo programová komponenta, která je schopna jedné nebo více I/O operací. Channel může být buďto otevřen nebo zavřen, připojen nebo odpojen. (Maurer et al., 2016) V rámci této práce bude `Channel` brán jako spojení, kterým buďto data přitékají nebo odtékají.

8.2.2 Callbacks

Callback je metoda, které může být předána jiná metoda. To umožňuje callback metodě provolat takto předanou metodu kdykoliv, když nastane vhodný čas. Callback metody jsou využívány v celé řadě situací a reprezentují velmi častý způsob toho, jak obeznámit zainteresované strany o tom, že operace je hotova. (Maurer et al., 2016)

[31] Java Development Kit – `https://www.java.com/en/download/faq/develop.xml`

[32] Thread pooling – `https://docs.oracle.com/javase/tutorial/essential/concurrency/pools.html`

8.2.3 ChannelFutures

Další možností, jak upozornit v jazyce Java aplikaci o tom, že operace bylo dokončena, je `Future`. Tyto objekty plní roli kontejneru pro uložení výsledku asynchronní operace, která někdy v budoucnosti skončí a její výsledek lze v tomto kontejneru vyzvednout.

Netty poskytuje vlastní implementaci – `ChannelFuture`, která plní stejnou roli jako `Future`, ovšem je rozšířena o několik metod, které dovolují registrovat jednu nebo více instancí `ChannelFutureListener`. Tyto instance disponují callback metodou, která je zavolána pokaždé, kdy je operace dokončena, tj. když asynchronní metoda uloží svůj výsledek do `ChannelFuture`. `ChannelFuture` může rozhodnout, jestli dokončená operace proběhla v pořádku nebo s nějakou chybou. (Maurer et al., 2016)

8.2.4 Events and ChannelHandlers

Netty využívá několika odlišných událostí (events), aby upozornilo na změny stavů probíhajících operací. Jakožto síťový framework lze události kategorizovat podle toho, zda patří příchozímu nebo odchozímu toku dat. Události, které mohou být vyvolány příchozími daty nebo změnou stavu jsou:

- navázání nebo ukončení spojení
- čtení příchozích dat
- události vyvolané uživatelem
- události vyvolané chybou

Události vzniklé odchozím tokem dat jsou:

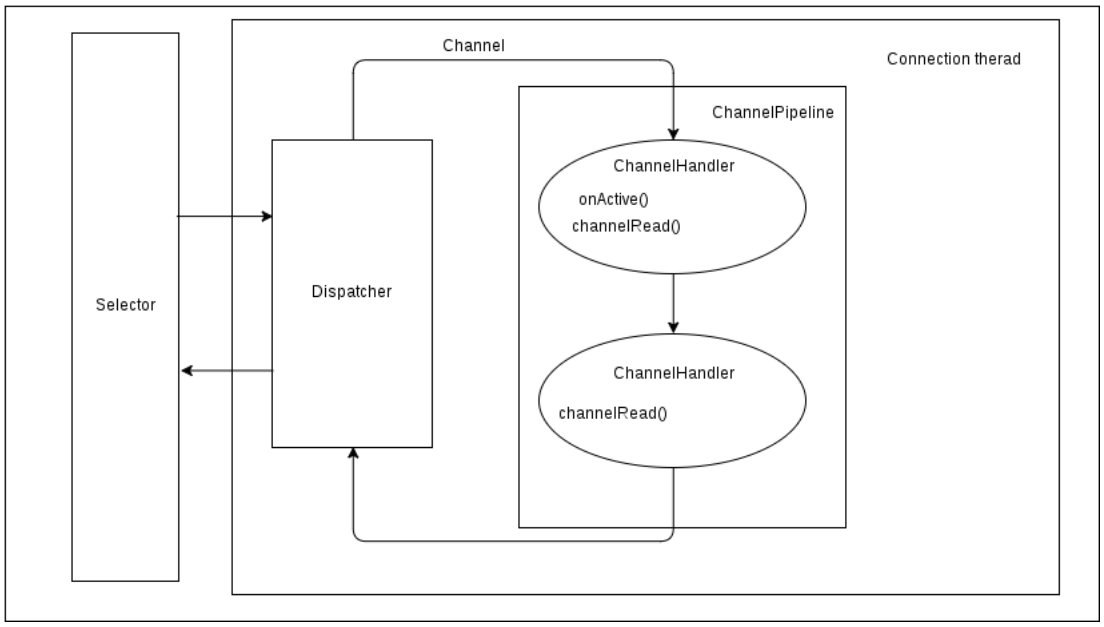
- navázání nebo ukončení spojení se vzdáleným klientem
- zapsání nebo odeslání dat do socketu

Každá z výše popsaných událostí může být odbavena překrytím metody ve třídě, která implementuje rozhraní `ChannelHandler`. `ChannelHandler` odbavuje nebo zachytává I/O operace, které se vyskytují v rámci `Channel`. Uživatel je tak schopen implementovat aplikační logiku překrytým metodám, které reagují na vzniklé události. Toto je velmi dobrý příklad event-driven paradigmatu aplikovaný na jednotlivé funkční bloky. Jako příklad možných metod, které lze překrýt, lze uvést:

- onActive() – reakce na událost navázání spojení
- channelRead() – reakce na událost, že navázané spojení posílá data, které lze číst
- channelInactive() – reakce na událost ukončení spojení

8.2.5 ChannelPipeline

Poslední důležitou komponentou je rozhraní `ChannelPipeline`. Ta slouží jako jakási „roura“, kterou prochází `Channel`. Do `ChannelPipeline` lze registrovat 1-N `ChannelHandlerů`. `ChannelHandlery` se v `ChannelPipeline` řadí jeden za druhým. Z toho plyne, že výstup předchozího `ChannelHandleru` je vstupem do nadcházejícího, dokud data nejsou odeslána nebo zahozena. Nadcházející obrázek ilustruje celý proces v návaznosti na non-blocking koncept, popsaný v podkapitole 4.2.



Obrázek 22: Netty ChannelPipeline (Ayedo, 2016), upraveno

V případě, kdy `Selektor` zjistí, že některý z připojených socketů změnil svůj stav (v případě obr. 22 je vstupní tok data připraven pro čtení), je vyvolána událost, která je předána na `Dispatcher`, který rozhoduje, jak s příchozím tokem naložit. Tok dat teče přes `Channel` do `ChannelPipeline`, ve které jsou obsaženy `ChannelHandlery`. Každý z nich většinou vykonává část aplikační logiky.

V uvedeném příkladě může vstupní tok dat (dále input stream) reprezentovat HTTP požadavek. První `ChannelHandler` může v těle metody `onActive()` zalogovat to, že došlo k navázaní spojení a v těle metody `channelRead()` převést input stream v podobě bajtů do podoby Java objektu, reprezentujícího HTTP požadavek. Takto převedený HTTP požadavek je poté poslán dále do druhého `ChannelHandleru`. Ten v metodě `channelRead()` prozkoumá podstatu tohoto požadavku, vytvoří HTTP odpověď, která je poslána zpět klientovi.

Jednotlivé `ChannelHandlery` lze dynamicky přidávat nebo odebírat, což zajišťuje maximální kontrolu nad tokem dat.

8.3 Abstraktní návrh architektury

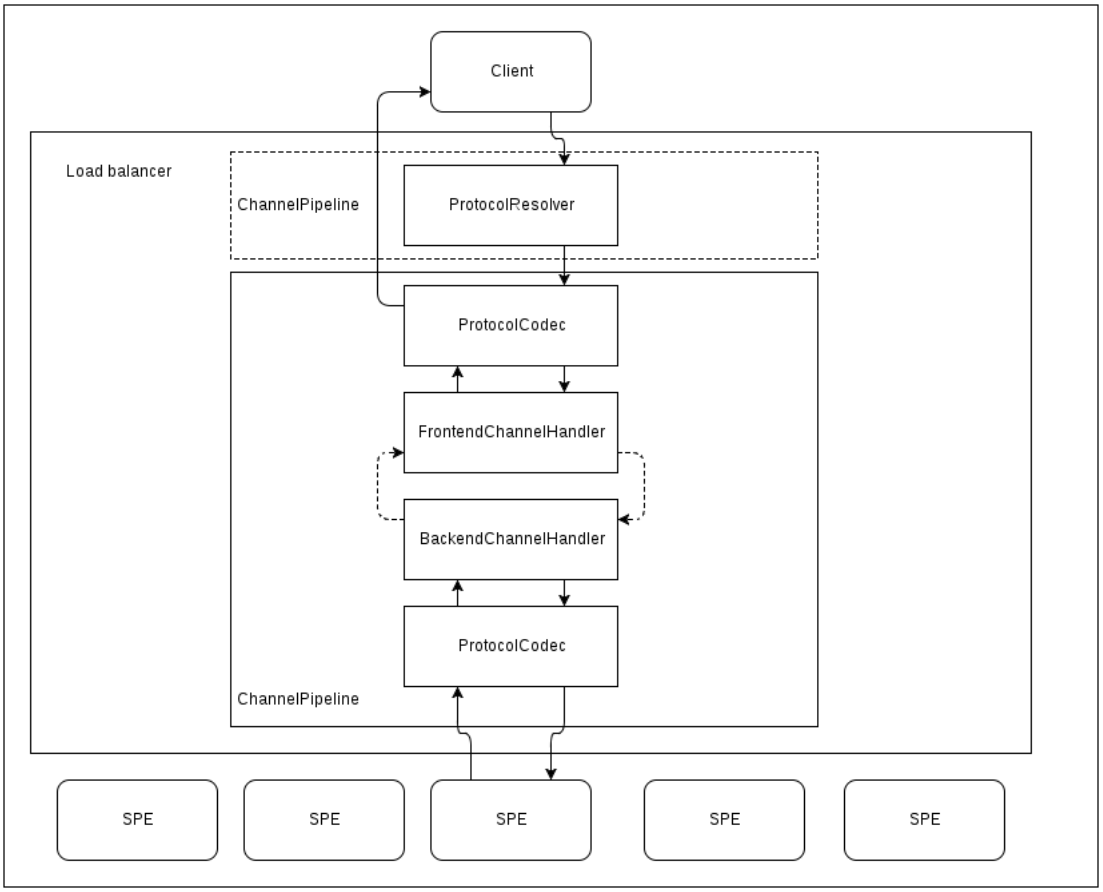
Analyzováním požadavků vyplynulo, že výsledné řešení bude muset umět pracovat jak s TCP, tak i s UDP protokolem.
V rámci TCP spojení byla v době psaní této práce požadována podpora pouze HTTP protokolu. Jelikož Netty pracuje se streamy bajtů, bylo potřeba zařadit do architektury komponentu, která by dokázala převést tento stream do podoby, se kterou půjde jednoduše pracovat – ideálně reprezentovanou pomocí POJO[33]. I když nebylo vyžadováno, aby výsledné řešení umělo pracovat s HTTPS, bylo vhodné na tuto skutečnost myslet a navrhnout takovou architekturu, která by byla v budoucnu lehce rozšířitelná. Totéž platí i pro případnou podporu jiných protokolů než je HTTP.

V rámci UDP protokolu byla vyžadována schopnost pracovat s datagramy reprezentujíc části RTP streamu.

8.3.1 TCP

Pro TCP spojení bude load balancing probíhat v proxy módu, viz. podkapitola 3.3.1. Předávání požadavků a odpovědí mezi `FrontendChannelHandlerem` a `BackendChannelHandlerem` bude probíhat pomocí vzájemného udržování referencí na instance těchto tříd. Klient tedy nebude nikdy přímo propojen s backend serverem. Jako load balancing metoda bude použita metoda popsaná v podkapitole 3.5.7, „Even Size Queue“.

[33]Plain Old Java Object – `http://www.martinfowler.com/bliki/POJO.html`



Obrázek 23: Abstraktní návrh – TCP

ProtocolResolver – úkolem této komponenty je zjistit z několika prvních bajtů příchozího streamu, o jaký protokol se jedná. V momentě příchodu streamu dat je jedinou komponentou v `ChannelPipeline`. Na obrázku č. 23 je tato `ChannelPipeline` označena přerušovaně. Jestliže se podaří určit příchozí protokol, jsou do `ChannelPipeline` předány další komponenty, které jsou schopné pracovat s tímto protokolem a tato komponenta je z `ChannelPipeline` odstraněna. Z toho vyplývá, že odpověď na klientův dotaz již znovu touto komponentou neprojde. Pokud nedojde k určení daného protokolu, je navázané spojení ukončeno.

ProtocolCodec – účelem této komponenty je buďto převést (dekódovat) vstupní stream bajtů (reprezentující požadavek detekovaného protokolu) na POJO, nebo převést (zakódovat) POJO (reprezentující odpověď) do výstupního streamu bajtů. Stejná funkcionalita platí pro `ProtocolCodec` propojený jak s `FrontendChannelHandler`, tak i s `BackendChannelHandler`.

FrontendChannelHandler – tato komponenta slouží k udržení spojení s klientem a také zde bude vykonána veškerá load balancing logika. Vstupem bude vždy POJO

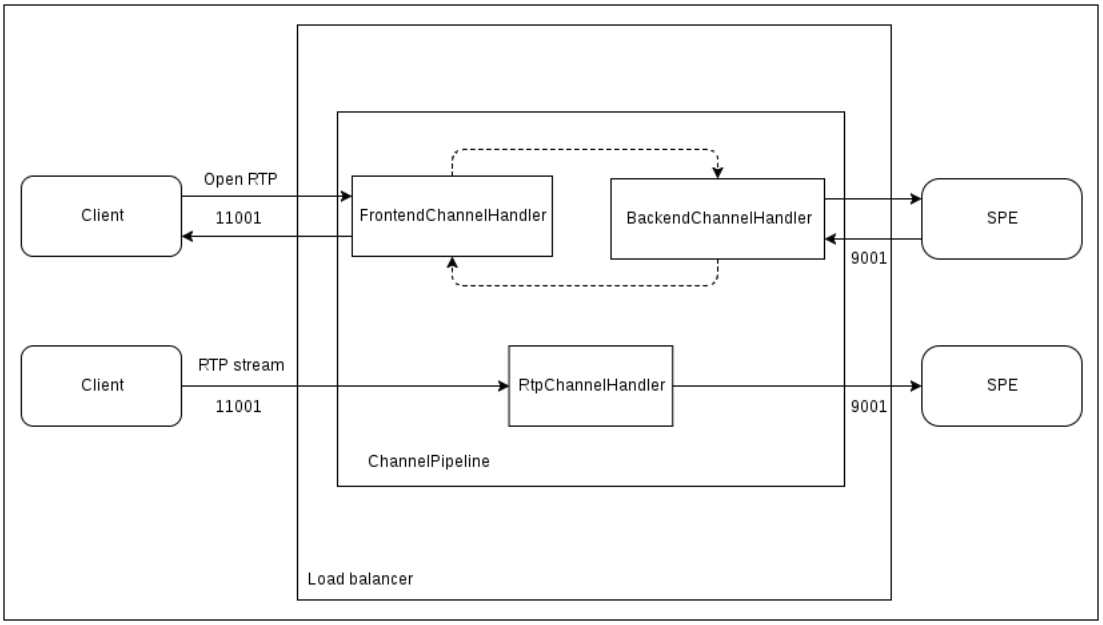
reprezentující požadavek. Tento požadavek zde bude analyzován a dle nastavené load balancing metody bude rozhodnuto, na který z množiny SPE serverů bude směrován. Jakmile bude SPE server znám, bude skrze `BackendChannelHandler` vytvořeno spojení na tento server a do tohoto spojení bude požadavek klienta předán. Spojení s klientem bude navázáno tak dlouho, dokud:

- nevyprší timeout stanovený pro toto spojení
- `FrontendChannelHandler` neobdrží odpověď od SPE serveru, kterou posléze odešle klientovi skrze skrze `BackendChannelHandler`.
- není vyhozena výjimka, reagující na nějaký problém. V takovém případě je klientovi odeslána odpovídající odpověď.

BackendChannelHandler – tato komponenta slouží k navázání a udržení spojení s SPE servery, předání klientských požadavků na tyto servery a k pozdějšímu předání odpovědí, a to vše v kooperaci s `FrontendChannelHandlerem`. Navazování a ukončování spojení s SPE servery je v plné režii `FrontendChannelHandleru`.

8.3.2 UDP

Jelikož je UDP bezstavový protokol, nejde plně docílit toho, aby byl vyhrazen pouze jeden port pro všechny příchozí UDP datagramy, tak jako tomu je v případě TCP spojení. Z tohoto důvodu když SPE server přijme požadavek na otevření portu pro RTP stream, je pro každý tento požadavek otevřen nový port, na kterém začne SPE server po předem stanovenou dobu naslouchat. Číslo nově otevřeného portu je uvedeno v odpovědi odeslané klientovi. V okamžiku, kdy load balancer zachytí tuto odpověď, analogicky otevře nový port a na tomto portu začne naslouchat. Do mapovací tabulky se poté uloží záznam o tom, že příslušný port náleží socket adrese některého SPE serveru. Číslo portu v odpovědi od SPE serveru je přepsáno za číslo portu otevřeného load balancerem. Když poté klient začne posílat části RTP streamu na vnější port (otevřený load balancerem), jsou tyto části forwardovány na správný SPE server. Následují obrázek celou situaci ilustruje.



Obrázek 24: Abstraktní návrh – UDP

9 Implementace řešení

9.1 Charakteristika řešení

Dle návrhu představeného v kapitole 8, bylo implementováno řešení – load balancer, který by se dal charakterizovat následujícími body:

- non-blocking
- event-driven
- pracující v reverse-proxy módu

Tento load balancer standardně podporuje:

- pro TCP – HTTP
- pro UDP – RTP
- povýšení HTTP spojení na WebSocket
- epoll pro Linux systémy[34]

Mezi jeho funkce patří:

- možnost implementace vlastní load balancing metody
- monitoring dostupný přes REST API
- online správa dostupných SPE serverů přes REST API bez nutnosti restartu
- health-checks SPE serverů

Implementovány byly tyto load balancing metody:

- Even Size Queue
- Even Size Queue dle technologie

9.2 Použité technologie

Níže je uveden seznam veškerých programovacích jazyků, frameworků a knihoven, použitých k implementaci tohoto load balanceru.

- Java 8 SE
- Netty (verze 4.1.6.Final)
- gson (vrze 2.7)
- log4j2 (verze 2.6.2)

[34]epoll – `https://linux.die.net/man/4/epoll`

- maven (verze 3.3.9)

9.3 ProtocolResolver

V době psaní této práce podporoval SPE server pro TCP spojení pouze HTTP protokol, proto se ProtocolResolver snaží dekódovat pouze tento protokol. Nebyl by však žádný problém v budoucnu přidat dekodér pro jakýkoliv jiný protokol. Následující kód popisuje dekódovací metodu.

```
1  @Override
2  protected void decode(ChannelHandlerContext ctx, ByteBuf in, List
       out) throws Exception {
3
4  // Using the first five bytes to detect a~protocol.
5  if (in.readableBytes() < 5) {
6      return;
7  }
8
9  final int magicInt1 = in.getUnsignedByte(in.readerIndex());
10 final int magicInt2 = in.getUnsignedByte(in.readerIndex() + 1);
11
12 if (isHttp(magicInt1, magicInt2)) {
13     switchToUriHandlerler(ctx);
14 } else {
15     // Unknown protocol, therefore throw away input stream and
           close connection.
16     in.clear();
17     ctx.close();
18 }
19 }
```

5. K začátku dekódování vstupního streamu bajtů je potřeba minimálně pět bajtů. Pokud vstupní stream nemá pět bajtů, nedojde k pokusu o dekódování.

9. Přetypování byte na integer – vznik magicInt1.

10. Přetypování byte na integer – vznik magicInt2.

12. Předání magic integerů do metody, která vrací true/false, pokud předané parametry odpovídají charakteristice HTTP protokolu. Metoda je popsána v podkapitole 9.3.1.

13. Pokud je v rámci metody `isHttp(magicInt1, magicInt2)` úspěšně dekódován HTTP protokol, je provolána metoda `switchToUriHandlerler(ctx)`. Ta je podrobněji popsána v podkapitole 9.3.2.

16. Pokud HTTP protokol nebyl úspěšně dekódován, je v tomto okamžiku vstupní stream zahozen.

17. Spojení s klientem je ukončeno.

9.3.1 Metoda isHttp()

Následující kód popisuje to, jakým způsobem je dokován HTTP protokol.

```
1  private static boolean isHttp(int magicInt1, int magicInt2) {
2      return magicInt1 == 'G' && magicInt2 == 'E' || // GET
3              magicInt1 == 'P' && magicInt2 == 'O' || // POST
4              magicInt1 == 'P' && magicInt2 == 'U' || // PUT
5              magicInt1 == 'H' && magicInt2 == 'E' || // HEAD
6              magicInt1 == 'O' && magicInt2 == 'P' || // OPTIONS
7              magicInt1 == 'P' && magicInt2 == 'A' || // PATCH
8              magicInt1 == 'D' && magicInt2 == 'E' || // DELETE
9              magicInt1 == 'T' && magicInt2 == 'R' || // TRACE
10             magicInt1 == 'C' && magicInt2 == 'O'; // CONNECT
11 }
```

Od řádku číslo 2 až do konce metody se kontroluje, zda `magicInt1` a `magicInt2` odpovídá ASCII kódu nějakého znaku. V případě druhého řádku se jedná o znaky G a E, které indikují HTTP metodu GET. Pokud by hodnoty `magicInt1` a `magicInt2` odpovídaly ASCII kódům znaků G a E, je navrácena hodnota `true` značící, že se jedná o HTTP protokol. Analogicky se testují i ostatní HTTP metody. Pokud ani v jednom případě nebyla vrácena hodnota `true`, znamená to, že se nejedná o HTTP protokol.

9.3.2 Metoda switchToUriHandlerler()

```
1  private void switchToUriHandlerler(ChannelHandlerContext ctx) {
2    ChannelPipeline p = ctx.pipeline();
3        p.addLast(new HttpRequestDecoder());
4        p.addLast("handler", new UriHandler());
5        p.remove(this);
6  }
```

2. Vytažení `ChannelPipeline` z kontextu
3. Přidání instance třídy `HttpRequestDecoder`, která má za úkol extrahovat ze streamu bajtů HTTP hlavičku a převést ji do POJO.
4. Přidání instance třídy `HttpUriHandler`. Tato třída je podrobněji popsána v podkapitole 9.6.1.
5. `ProtocolResolver` je odstraněn z `ChannelPipeline`

9.4 Perzistentní vrstva

Perzistentní vrstva je v rámci tohoto load balanceru řešena použitím instancí třídy ConcurrentHashMap z Java Collections Framework[35]. Jedná se o thread-safe[36] implementaci asociativního pole. Celá perzistentní vrstva je reprezentována třídou `Storage`.

```
1  public class Storage {
2
3      // For load balancing strategy by pending operation count (e.g.
           all servers has the same configuration)
4      public static final ConcurrentHashMap AVAILABLE_NODES_MAP = new
           ConcurrentHashMap<>();
5
6      // For load balancing strategy by technology, then by pending
           operation count
7      public static final ConcurrentHashMap>
           AVAILABLE_NODES_BY_TECHNOLOGY_MAP = new ConcurrentHashMap
           <>();
8
9      // For SPE that became unavailable during the health-checks
10     public static final Set UNAVAILABLE_NODES_MAP = new HashSet<>()
           ;
11
12     // For storing pending operation hash to SPE server address
13     public static final ConcurrentHashMap PENDING_MAP = new
           ConcurrentHashMap<>();
14
15     // For storing done operation hash to SPE server address
16     public static final ConcurrentHashMap DONE_MAP = new
           ConcurrentHashMap<>();
17
18     // For storing port number to SPE server address
19     public static final ConcurrentHashMap RTP_PORT_TO_ADDRESS_MAP =
           new ConcurrentHashMap<>();
20
21     // For storing streamID to SPE server address used with HTTP
           streams
22     public static final ConcurrentHashMap STREAM_ID_TO_ADDRESS_MAP
           = new ConcurrentHashMap<>();
23 }
```

4. Kolekce mapující socket adresu [37] SPE serveru a počet jeho operací. Využívá se v load balancing metodě popsané v podkapitole 9.5.1.

[35]Java Collections Framework – http://docs.oracle.com/javase/8/docs/technotes/guides/collections/index.html

[36]thread-safe – http://www.javaworld.com/article/2076747/core-java/design-for-thread-safety.html

[37]IP adresa + číslo portu

7. Kolekce mapující typ technologie reprezentované pomocí enumu, ke kolekci mapující socket adresu SPE serveru a počet jeho operací. Jedná se rozšířenou verzi předešle popsané kolekce. Využívá se v load balancing metodě popsané v podkapitole 9.5.2.

10. Kolekce obsahující socket adresu SPE serveru, který byl vyřazen z některé z předchozích popsaných kolekcí z důvodu nedostupnosti.

13. Tato kolekce slouží k identifikaci socket adresy SPE serveru podle hash pending operace, značící asynchronní zpracování. Asynchronní zpracování SPE serveru je popsáno v podkapitole 6.1.1. Kolekce je využívána v obou dvou load balancing metodách.

16. Tato kolekce slouží k identifikaci socket adresy SPE serveru podle hash done operace, sloužící k vyzvednutí výsledků asynchronního zpracování. Asynchronní zpracování SPE serveru je popsáno v podkapitole 6.1.1. Kolekce je využívána v obou dvou load balancing metodách.

19. Tato kolekce mapuje číslo portu otevřeného v rámci tohoto load balanceru k socket adrese SPE serveru, sloužícího pro správné forwardování RTP streamu k SPE serveru.

22. Tato kolekce mapuje id HTTP streamu k socket adrese SPE serveru, vybraného ke zpracování příchozího HTTP streamu. Kolekce je využívána v setu `HttpStreamHandlerů`, zobrazených na obrázku č. 25.

9.5 Load balancing metody

V následujících dvou podkapitolách jsou popsány load balancing metody, které byly implementovány v rámci této práce. Celý proces spočívá v nalezení nejmenšího počtu operací pro daný SPE server a k tomu odpovídající socket adresu. Výběr load balancing metody je podmíněn v konfiguračním souboru.

9.5.1 Even Size Queue

Metoda funguje stejným způsobem jako metoda popsaná v podkapitole 3.5.7. Tuto metodu lze použít tehdy, pokud SPE servery jsou stejně inicializovány, tedy mají inicializované stejné technologie. Dodržení této podmínky je nutné z toho důvodu, aby se zabránilo situaci, že by load balancer forwardoval požadavek na server, který nemá inicializovanou potřebnou technologii a zpracování by tak neproběhlo. V kódu load balanceru je tato load balancing metoda řešena následovně.

```
1 // Getting the address of SPE with the smallest count of operation
2     InetSocketAddress remoteAddress =
3             Collections.min(AVAILABLE_NODES_MAP.entrySet(),
4             Comparator.comparingInt(Entry::getValue))
5            .getKey();
```

Socket adresa SPE serveru je vybrána z kolekce voláním metody `min()` ze třídy Collections, obsažené v Java Collections Frameworku. Rychlost určení socket adresy SPE serveru dle nejmenšího počtu operací se rapidně snižuje díky JIT kompilátoru[38], který volání tohoto kódu optimalizuje. Dle výsledků testování je čas strávený při prvních výběrech socket adresy v jednotkách milisekund. Opakovaným voláním dochází k optimalizování kódu a snížení časové náročnosti na řádově stovky nanosekund.

9.5.2 Even Size Queue dle technologie

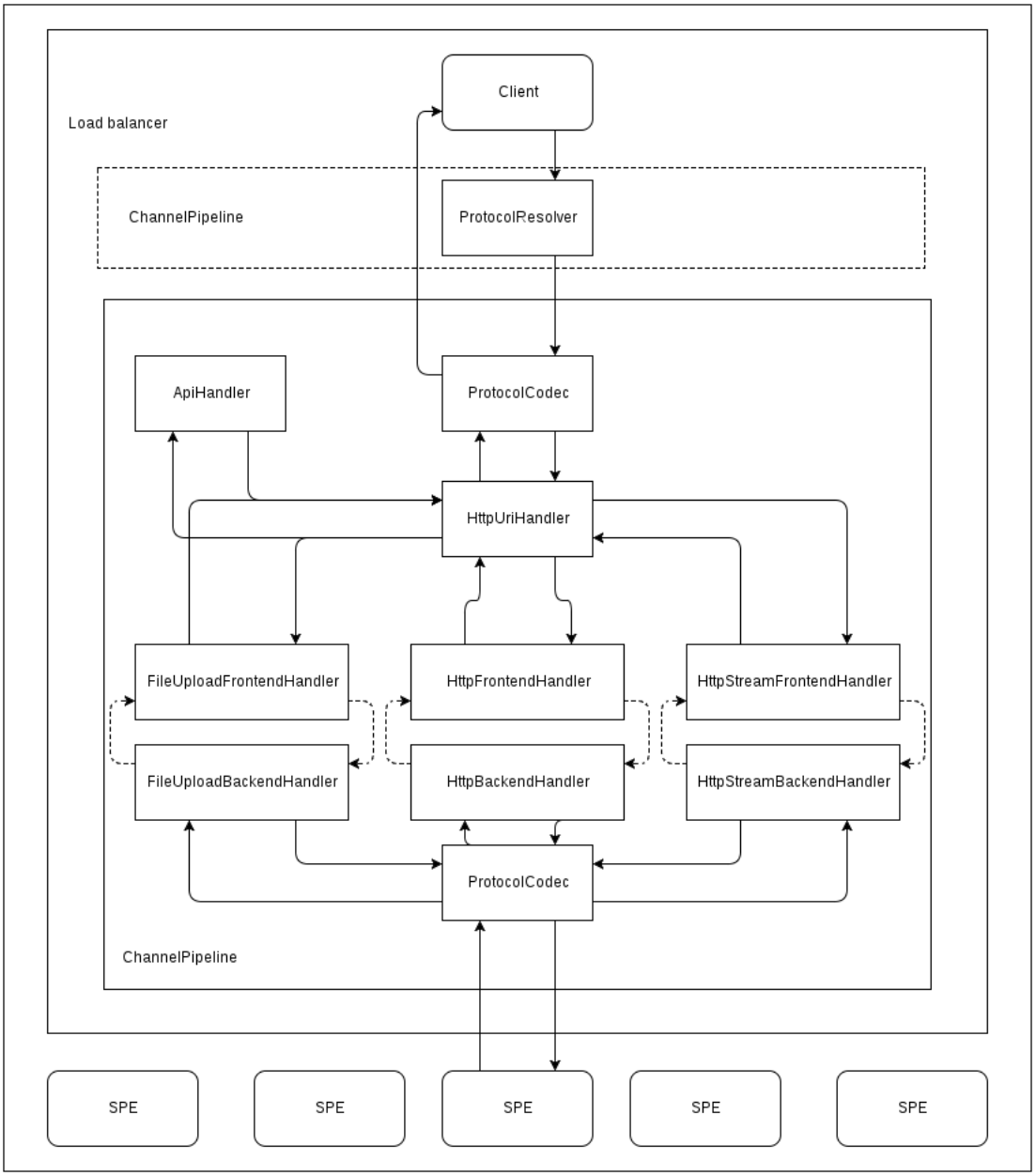
Tato možnost řeší nedostatek předchozí metody. Tedy příchozí požadavek je analyzován a jedná-li se o požadavek pro specifickou technologii, je výběr socket adresy SPE serveru zúžen nejprve na servery, které mají danou technologii inicializovanou, a poté je vybrán server s nejmenší počtem požadavků ve frontě.

```
1 // Getting the address of SPE with the smallest count of operation
      for KWS technology
2     InetSocketAddress remoteAddress =
3                 Collections.min(Storage.
                      AVAILABLE_NODES_BY_TECHNOLOGY_MAP
4               .get(TechnologyTypeEnum.KWS).entrySet(),
5               Comparator.comparingInt(Entry::getValue))
6               .getKey();
```

9.6 HTTP protokol

Jelikož pracuje SPE server v drtivé většině případů s HTTP protokolem, bylo nejvíce práce provedeno na `ChannelHandlerech` pro tento protokol. Následující obrázek popisuje to, jak je s požadavky nakládáno poté, kdy jsou v komponentě `ProtocolResolver`, detekovány jako HTTP požadavky. Stejně tak jako v případě abstraktního návrhu pro TCP spojení probraného v podkapitole 8.3.1, i zde je komponenta `ProtokolResolver` odstraněna z `ChannelPipeline` poté, co úspěšně identifikuje příchozí protokol.

[38]Just-In-Time Compilator – `https://docs.oracle.com/cd/E15289_01/doc.40/e15058/understr_jit.htm`



Obrázek 25: Zpracování HTTP protokolu

9.6.1 HttpUriHandler

Oproti abstraktnímu návrhu diskutovaném v podkapitole 8.3.1 přibyla pro konkrétní implementaci `ChannelHandlerů` pro HTTP protokol nová komponenta – `HttpUriHandler`. Ta má za úkol extrahovat URI a na základně ní zařadit do `ChannelPipeline` jeden ze čtyř různých druhů `ChannelHandlerů`:

- **ApiHandler** – Tento `ChannelHandler` slouží k odbavení veškerých HTTP požadavků na REST API load balanceru. Sám nenavazuje spojení s SPE serverem.

Do `ChannelPipeline` je tento `ChannelHandler` zařazen pro URI s prefixem `/api/v1/*`.

- **FileUploadFrontendHandler a FileUploadBackendHandler** – Tento set `ChannelHandlerů` slouží k odbavení HTTP POST požadavku pro upload nahrávky. Tento POST požadavek je forvardován na některý z SPE serverů, který jej pak uloží do podoby souboru na sdílném úložišti. Tyto `ChannelHandlery` jsou zařazeny do `ChannelPipeline` pro POST požadavky obsahující URI `/audiofile`. Důvodem vytvoření tohoto setu `ChannelHandlerů` je skutečnost, že lze uploadovat veliké soubory, pro které má Netty vytvořeno speciální `ChannelHandlery`, které minimalizují náročnost na systémové prostředky.
- **HttpStreamFrontendHandler a HttpStreamBackendHandler** – Slouží k odbavení HTTP streamů při real-time processingu. Tento set `ChannelHandlerů` je zařazen do `ChannelPipeline` pro URI `/stream/http`. Tento set byl vytvořen k minimalizování náročnosti na systémové prostředky během posílání HTTP streamů.
- **HTTP Frontend/Backend Handler** – Tento set `ChannelHandlerů` slouží k odbavení všech ostatních HTTP požadavků, které by nebyly odbaveny některým z předchozích `ChannelHandlerů`.

9.6.2 HttpFrontendHandler

Jedná se o komponentu vycházející z komponenty `FrontendChannelHandler` popsané v rámci podkapitoly 8.3.1, která byla specifikována pro HTTP protokol. Nejdůležitější metodou v rámci této třídy je metoda `channelRead0`, jejíž kód je vykonán poté, co tento `ChannelHandler` obdrží notifikaci o tom, že je klient připraven zasílat data.

```
1  @Override
2  public void channelRead0(final ChannelHandlerContext ctx,
       FullHttpRequest req) throws NoPendingOperationFoundException,
       NoBackendServerRegistredException, NoDoneOperationFoundException
       , NoBackendServerForTechnologyException {
3
4      final Channel inboundChannel = ctx.channel();
5
6      log.info("Request for: {} method: {}", req.uri(), req.method())
           ;
7
8      String uriToLower = req.uri().toLowerCase();
9
10     ConnectionParameters connParams = resolveConnectionParameters(
           uriToLower);
11
12     Bootstrap b = prepareConnectionToSpe(inboundChannel, ctx, req,
           connParams.getAsyncKey(), connParams.getAsyncKey());
13
14     connectToSpe(inboundChannel, ctx, req ,b, connParams.
           getRemoteAddress());
15
16     // NOTE: SimpleChannelInboundHandler => therefore req would be
           GC, after write and flush
17     ReferenceCountUtil.retain(req);
18 }
```

4. Vytvořena proměnná inboundChannel (příchozí spojení)

6. Zalogování příchozího HTTP požadavku

8. Extrahování URI z příchozího HTTP požadavku a převedení na malá písmena z důvodu bezpečného porovnávání

10. Vytvoření proměnné reprezentující parametry spojení. V rámci metody `resolveConnectionParameters(uriToLower)` dojde k přezkoumání předaného URI a vyhodnocení, na jaký SPE server bude HTTP požadavek forwardován. Jakým stylem je rozhodnuto o tom, kam se bude HTTP požadavek forwardovat, je probráno v rámci podkapitoly 9.5.

12. Na tomto řádku je volána metoda, pomocí které je připraveno spojení s SPE serverem. Připravené spojení je dvojího typu – pro HTTP požadavky dotazující se na asynchronního zpracování, nebo všechny ostatní dotazy. Připravené spojení je poté reprezentováno instancí třídy `Bootstrap` z Netty frameworku.

14. Voláním metody `connectToSpe(…)` dochází k navázání spojení s SPE serverem. Do `ChannelPipeline` tohoto spojení je vložen `HttpBackendHandler`, který zaštiťuje veškerou komunikaci s SPE serverem.

17. Kód na tomto řádku zvyšuje počet referencí na objekt `req`. Toto je velmi důležité, neboť by Netty po skončení práce s tímto objektem v rámci tohoto `ChannelHandleru` snížil počet referencí na nulu a objekt by tak byl v nejbližší době uvolněn Garbage Collectorem[39]. Obecně se jedná o velmi vhodnou pojistku proti tomu, aby objekty nezůstávaly v paměti déle než je potřeba. Všechny třídy, které dědí z třídy `SimpleChannelInboundHandler` mají tento mechanismus zabudován. Tedy jakmile HTTP požadavek opustí tento `ChannelHandler`, je uvolněn z paměti. To je v tomto případě ovšem nežádoucí, neboť je tento HTTP požadavek předáván dále, a to do `HttpBackendHandleru`.

9.6.3 HttpBackendHandler

Opět se jedná o komponentu vycházející z komponenty `BackendChannelHandler` popsané v rámci podkapitoly 8.3.1, která je specifikována pro HTTP protokol. Nejdůležitější metodou v rámci této třídy je metoda `channelRead0`, jejíž kód je vykonán poté, co tento `ChannelHandler` obdrží notifikaci o tom, že je SPE server připraven zasílat data.

```
1  @Override
2  public void channelRead0(final ChannelHandlerContext ctx,
       FullHttpResponse resp) {
3
4      InetSocketAddress remoteAddress = (InetSocketAddress) ctx.
           channel().remoteAddress();
5
6      checkForLocation(resp, remoteAddress);
7
8      checkForPendingHash(ctx.channel(), resp);
9
10     checkForDoneHash(ctx.channel(), resp, remoteAddress);
11
12     writeDataToClient(resp);
13 }
```

4. Vytvořena proměnná ukazující na socket adresu SPE serveru

6. V rámci této metody je kontrolováno to, jestli HTTP odpověď z SPE serveru neobsahuje v hlavičce parametr `Location`. Pokud ano, je dále v rámci této metody zkoumáno, jestli je hodnota tohoto parametru rovna `/pending/<32-bit-hash>` nebo `/done/<32-bit-hash>`, značící asynchronní zpracován podrobněji popsané v podkapitole 6.1.1. Pokud ano, je tato operace zařazena do patřičné kolekce, mapující hash asynchronní operace k SPE serveru.

[39]Garbage Collector – http://javarevisited.blogspot.cz/2011/04/garbage-collection-in-java.html

8. V rámci této metody je zkoumáno, jestli je HTTP požadavek klienta určený k dotazu na stav asynchronního zpracování. Workflow asynchronního zpracování je probírán v podkapitole 6.1.1.

10. V rámci této metody je zkoumáno, jestli je HTTP požadavek klienta určený k dotazu na vyzvednutí výsledků asynchronního zpracování. Workflow asynchronního zpracování je probíráno v podkapitole 6.1.1.

12. Data jsou zaslána zpět klientovi skrze `HttpFrontendHandler`.

9.7 REST API

Load balancer disponuje vlastním REST API, umožňujícím monitoring provozu nebo umožňujícím dynamicky měnit konfiguraci load balanceru. V době psaní této práce nebyla implementována žádná funkcionalita omezující přístup k tomuto API. Díky univerzálnosti REST rozhraní lze toto API integrovat do jakéhokoliv systému, který bude využívat tento load balancer. V následujících kapitolách jsou popsány implementované REST resources.[40]

9.7.1 Aktuální vytížení serverů

Aktuální zatížení, přesněji počet požadavků pro jednotlivé SPE servery, lze získat pomocí HTTP GET dotazu na URI `/api/v1/serverstistics`. Příkladem takového výstupu může být následující JSON:

```
{
        "server_statistics": {
                "version": 1,
                "total_requests_count": 194,
                "servers": [{
                        "address": "192.168.1.1",
                        "port": 8600,
                        "requests_count": 101
                }, {
                        "address": "192.168.1.2",
                        "port": 8601,
                        "requests_count": 93
                }]
        }
}
```

Specifikováním socket adresy lze získat i podrobnou statistiku SPE serveru náležící této adrese, např. `/api/v1/serverstistics?server=192.168.1.1:8600`.

[40] REST resources – `http://restful-api-design.readthedocs.io/en/latest/resources.html`

Load balancer má také implementovaný resource pro získání informací o samostatném běhu. Tyto informace lze získat pomocí URI `/api/v1/lobsperstatistics`. Příkladem může být následující výstup ve formátu JSON:

```
{
        "lobsper_statistics": {
                "version": 1,
                "uptime": "16:28:17 up 1 day, 3:30",
                "cpu_usage": 32,
                "memory_usage": 1.5
        }
}
```

9.7.2 Konfigurace

Pro dynamické upravování konfigurace dostupných SPE serverů slouží URI `/api/v1/availableservers`. Na toto URI se lze dotazovat následujícími HTTP metodami:

1. **GET** – je navrácena konfigurace vše SPE serverů spolu s inicializovanými technologiemi.
2. **POST** – pro přidání SPE serveru je nutno do těla toho HTTP požadavku přidat JSON, obsahující IP adresu, port a výčet technologií.
3. **DELETE** – pro odstranění serveru z konfigurace je nutné poskytnout parametr server s adresou a portem SPE serveru, např. `/api/v1/availableservers?server=192.168.1.1:8600`

Výpis konfigurace vypadá následně:

```
{
    "available_servers": {
        "version": 1,
        "servers": [{
                "address": "192.168.1.1",
                "port": 8600,
                "technologies": ["STT", "KWS, TAE"]
        }, {
                "address": "192.168.1.2",
                "port": 8601,
                "technologies": ["STT", "GID", "LID", "KWS"]
        }]
    }
}
```

Obdobný výpis lze získat i pro nedostupné SPE servery, a to pomocí URI `/api/v1/unavailableservers`. Nedostupné SPE servery lze pouze vypsat, jelikož přidávání a odebírání do/z tohoto listu je prováděno automaticky v rámci health-checků.

10 Testování

10.1 Metodika

Testování probíhalo na referenčním stroji s procesorem i5-4600 s 16GB paměti. Daný procesor disponuje čtyřmi fyzickými jádry s možností běhu čtyř současných vláken. Nejvyšší možná frekvence procesoru je 3,4 GHz. Sledované systémové prostředky byly CPU a RAM. Využití systémových prostředků je vyjádřeno pomocí procent. Maximální hodnota pro paměť RAM je 100 %. Maximální hodnota pro CPU je 100 % / jádro, v součtu pro tento procesor tedy 400 %. Testovány byly obě dvě load balancing metody. Celkové testování bylo rozděleno do několika testů, aby bylo možno sledovat dopad na zátěž jednotlivých, vzájemně odlišných operací.

Důvodem realizace tohoto testování bylo skutečnost, že dojde-li někdy v budoucnu k nějaké změně v kódu, lze díky tomuto rozdělenému testování zjistit, zda tato změna měla vliv na jednotlivé operace. (Abbott et al., 2010; Allspaw, 2008)

Veškerý testovací síťový provoz byl generován na čtyřech PC, které jej posílaly na load balancer, který ho distribuoval dále mezi 4 SPE servery.

10.1.1 HTTP požadavky

Testování probíhalo tak, že se pomocí vybraného nástroje generovaly HTTP požadavky směřující na load balancer a dále na cluster SPE serverů. Počet HTTP požadavků se navyšoval, dokud nedosáhl limitní hranice počtu požadavků za sekundu (dále jen [req/s]). Limitními hodnotami byly hodnoty přibližně odpovídající hodnotám 1000, 2000, 5000, 10000 [req/s]. Při dosažení některé z těchto hodnot se po dobu 10 minut sledovaly systémové prostředky. Při běhu testu se zaznamenávala největší [max] a nejmenší [min] naměřená hodnota. Ze všech hodnot se poté vypočítal průměr [avg].

K testování load balanceru byl využíván nástroj wkr[41]. Jedná se command-line nástroj napsaný v jazyce C, schopný generovat veliké množství HTTP požadavků pomocí jednoho či více vláken.

```
wrk -t12 -c400 -d30s~http://127.0.0.1:8080/index.html
```

Výše uvedený příkaz spuštěný v terminálu vytvoří benchmark, který poběží po dobu 30 vteřin, pro generování požadavků bude využívat 12 vláken a zachová otevřených 400 HTTP spojení. Po skončení testování poté vypíše následující statistiku:

[41] wrk – `https://github.com/wg/wrk`

```
1         Running 30s~test @ http://127.0.0.1:8080/index.html
2         12 threads and 400 connections
3         Thread Stats   Avg      Stdev     Max   +/- Stdev
4           Latency   635.91us    0.89ms  12.92ms   93.69%
5           Req/Sec   56.20k~8.07k~62.00k~86.54%
6         22464657 requests in 30.00s, 17.76GB read
7         Requests/sec: 748868.53
8         Transfer/sec:    606.33MB
```

Důležitým údajem v této statistice je hodnota „Req/Sec“, která značí počet požadavků za sekundu.

10.1.2 HTTP streamy

Aby se dalo jednoduše zjistit, jaký vliv mělo posílání HTTP streamů na load balancer, probíhal stejný test v podkapitolo výše uvedené, spolu s tím, že se na load balancer zasílaly HTTP streamy. Počet spojení pro HTTP streamy byl 100. Byl využit unit test pro HTTP streamy, kterým se SPE server testuje. Tento test byl upraven pro paralelní běh a extrahován do samostatného spustitelného souboru.

10.1.3 RTP streamy

Stejně jako v případě HTTP streamů se i v tomto případě využilo testování uvedené v podkapitole 10.1.1, pouze se na místo HTTP streamů posílaly na load balancer RTP streamy. Počet RTP streamů byl 100. Byl využit unit test pro RTP streamy, kterým se SPE server testuje. Tento test byl upraven pro paralelní běh a extrahován do samostatného spustitelného souboru.

10.1.4 Uploading nahrávek

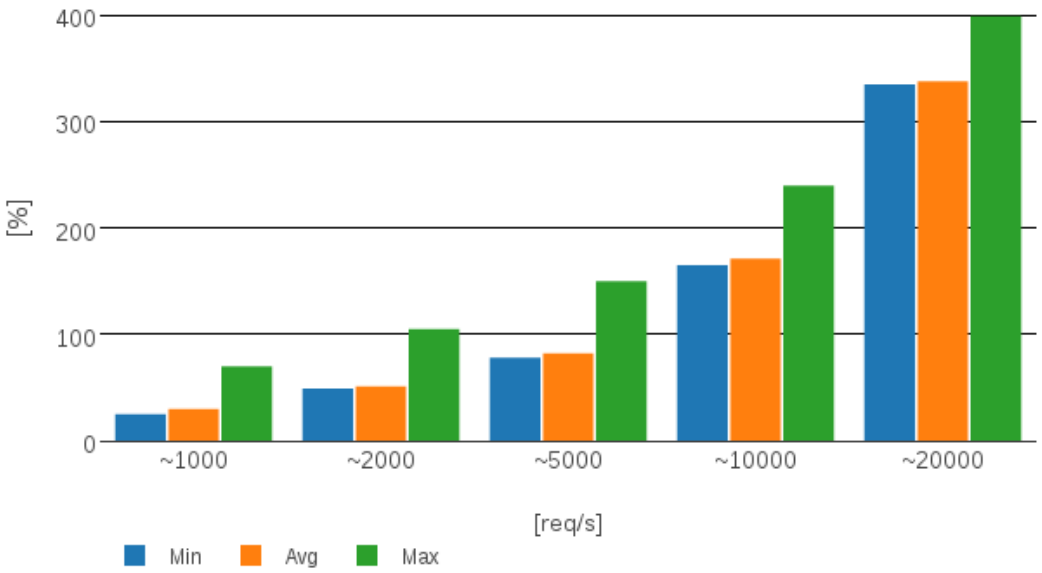
I v tomto případě bylo využito testování uvedené v podkapitole 10.1.1, kde souběžně probíhal upload nahrávek pomocí HTTP protokolu. Počet TCP spojení pro upload nahrávek byl 100. Byl využit unit test pro upload nahrávek, kterým se SPE server testuje. Tento test byl upraven pro paralelní běh a extrahován do samostatného spustitelného souboru.

10.1.5 Výše uvedené testy dohromady

Tento test vznikl spuštěním vše výše zmíněných testů.

10.2 Využití CPU

Následující graf zobrazuje vytížení CPU při přibližných hodnotách 1000, 2000, 5000, 10000 a 20000 [req/s].

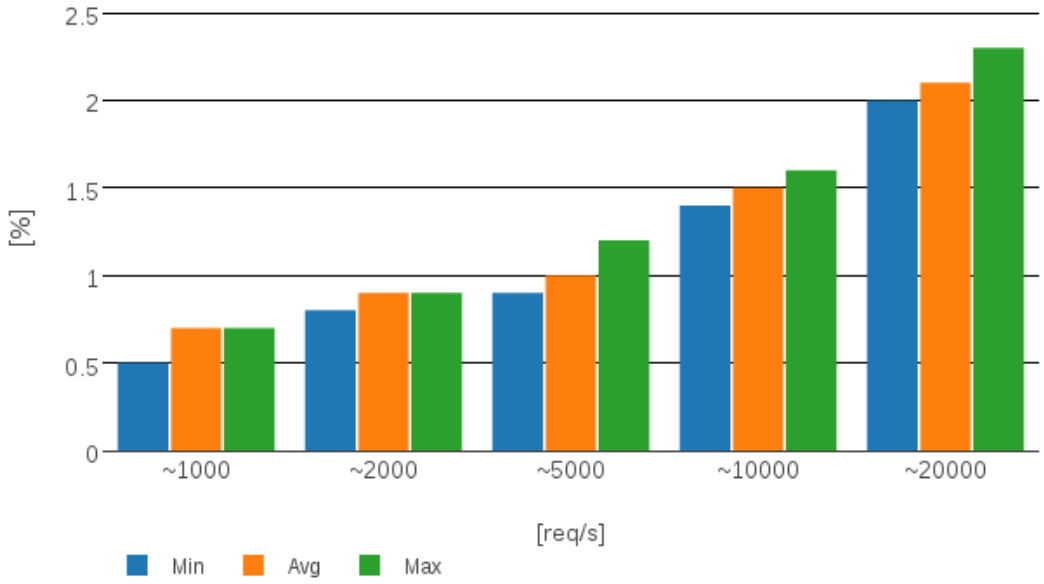


Obrázek 26: Výsledky testování pro HTTP požadavky – CPU využití

Z grafu lze pozorovat to, že se load balancer adaptuje na vzrůstající počet příchozích požadavků a zachovává si stabilní chování.

10.3 Využití RAM paměti

Následující graf zobrazuje alokaci paměti RAM při přibližných hodnotách 1000, 2000, 5000, 10000 a 20000 [req/s].

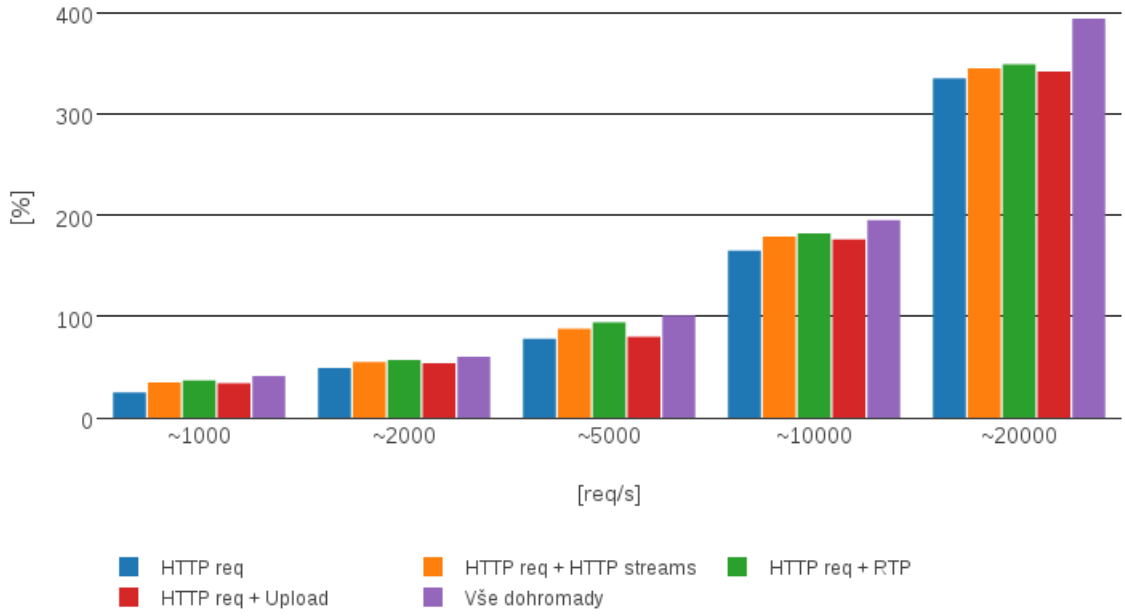


Obrázek 27: Výsledky testování pro HTTP požadavky – Alokace RAM paměti

Obdobně jako v případě CPU, i zde lze z grafu pozorovat, že se load balancer stabilně adaptuje na vzrůstající počet příchozích požadavků.

10.4 Výsledky všech testů

Následující graf zobrazuje výsledky vytížení CPU při testování všech výše popsaných testů.

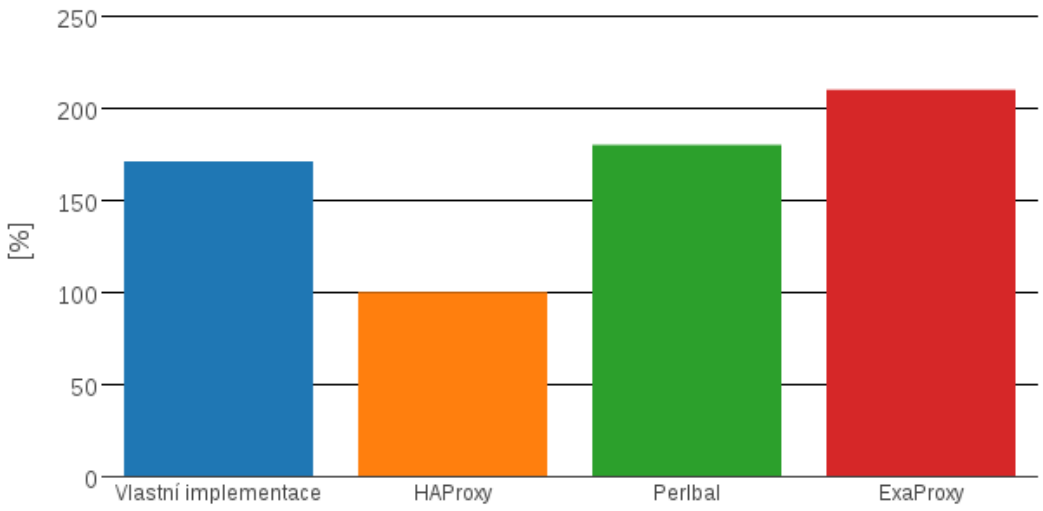


Obrázek 28: Výsledky všech testů – Využití CPU

Výsledek tohoto testu pouze potvrzuje výsledky předešlých testů. I v případě kombinace několika či všech testů lze z grafu pozorovat, že si load balancer stále drží stabilní chování s vzrůstajícím počtem příchozích požadavků.

10.5 Testování existujících řešení

Jednalo se o variantu prvního testu, při generování 10000 [req/s], kdy místo vzniklého load balancingu byla testována tyto existující řešení: HAProxy, Perlbal a ExaProxy. Pro možnost kvalifikovaného porovnání s existujícími řešeními byla implementována load balancing metoda Round Robin. Jak si vedl vzniklý load balancer oproti existujícím řešením je vidět na následujícím grafu.



Obrázek 29: Testování existujících řešení – Využití CPU

Z výsledků lze vidět, že si nejlépe vedla HAProxy. Ve finále se nejedná o velké překvapení, jelikož jde o několik let prověřené a optimalizované řešení napsané v jazyce C, který zaručuje maximální rychlost. V porovnání s Perlbalem si vedl vzniklý load balancer o něco lépe a předčil i výkon ExaProxy. Výkon vzniklého load balanceru jistě není k zahození a v porovnání s existujícími řešeními si nevedl nikterak špatně.

11 Diskuze

11.1 Zhodnocení implementace

Využití frameworku Netty pro naprogramování výsledného load balanceru se nakonec ukázalo jako velmi dobrá volba. Způsob, jak framework abstrahuje síťovou komunikaci, je velmi dobře zvládnutý a práce s tímto frameworkem byla potěšením. To, jakým způsobem framework pracuje s non-blocking I/O a event-driven paradigmatem, je taktéž velkým plusem. V rámci tvorby aplikace jsem se snažil, aby byla aplikace vhodně rozdělena do funkčních bloků a byla také lehce rozšiřitelná. Tento úkol se podařilo splnit a je tak velmi jednoduché do aplikace naprogramovat podporu pro nový protokol nebo novou load balancing metodu.

Jelikož se jedná o „high-performance“ síťovou aplikaci, snažil jsem se kód optimalizovat pro co největší výkon a v každém okamžiku dodržet principy non-blocking I/O. Analýzou výsledků testování, provedeného v kapitole 10, si dovoluji tvrdit, že se tato snaha vydařila, ovšem jistý prostor pro budoucí optimalizaci kódu zde stále je.

Rozhodnutí pro implementaci REST API sloužící pro dynamickou konfiguraci považuji taktéž za vhodné, neboť lze, teď již standardizovaným způsobem, měnit chování aplikace bez nutnosti restartu. Také poskytování informací pro monitoring a statistiky přes toto rozhraní má své výhody, neboť zpracování těchto informací lze integrovat do již existujících řešení.

Jak bývá běžné během vývoje softwareu, tak i v tomto případě se vyskytly problémy, které komplikovaly implementaci výsledného load balanceru. Nejzávažnějším problémem bylo přimět non-blocking framework, aby po přečtení HTTP hlavičky čekal na navázání spojení s SPE serverem, a nesnažil se číst další data, dokud nebylo spojení navázáno. Netty má totiž defaultně nastaveno množství dat, které má po obdržení události „read“ přečíst. Pro dekódovaný HTTP protokol to bylo vždy více dat než jen HTTP hlavička. To mělo za následek, že po přečtení HTTP hlavičky a započetí navazování spojení s SPE serverem (navázání spojení je asynchronní operace) Netty automaticky začalo číst další části HTTP požadavku, které nemohly být nikam poslány, jelikož spojení s SPE serverem v tomto okamžiku ještě nebylo navázáno. Celý problém byl vyřešen tak, že byla nastavena taková hodnota množství dat, která pro první událost „read“ přečetla pouze HTTP hlavičku. Po přečtení hlavičky a navázání spojení s SPE serverem byla nastavena znovu defaultní hodnota.

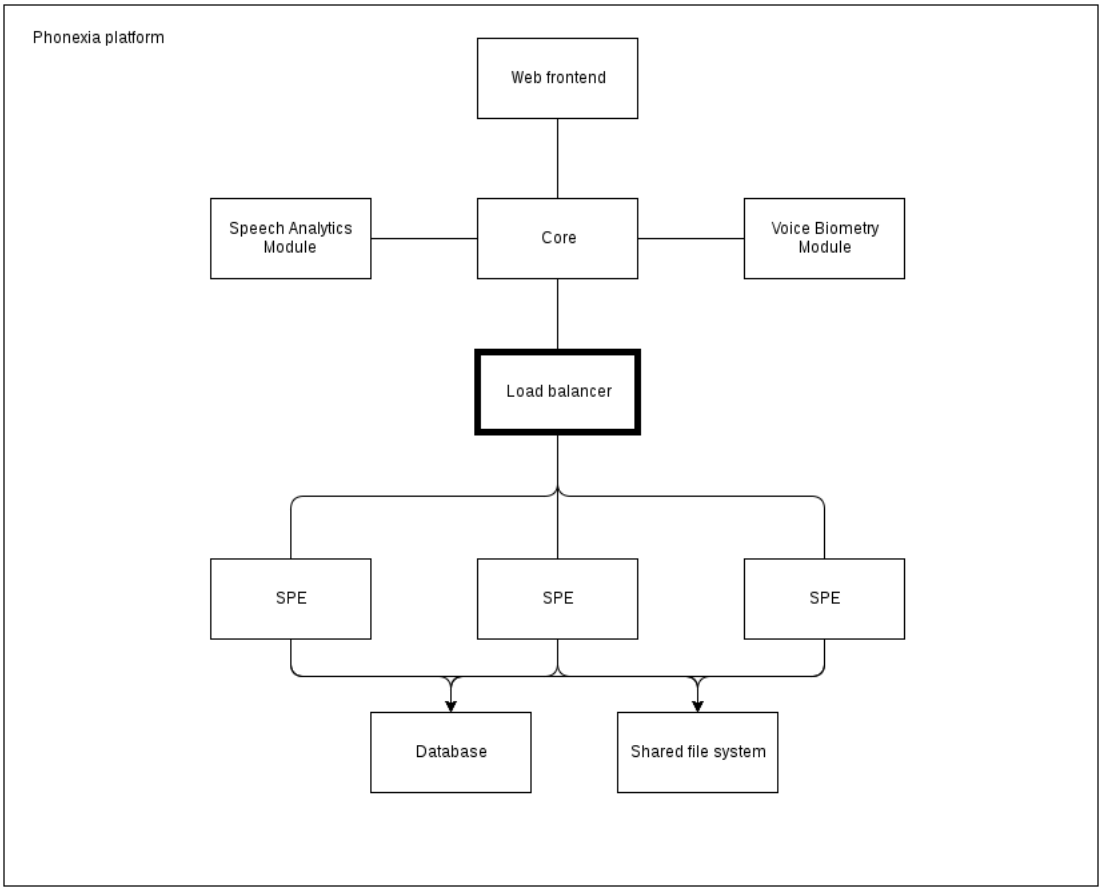
11.2 Srovnání s existujícími řešeními

V kapitole 7 jsem porovnával požadavky, které by mělo výsledné řešení splňovat, s existujícími řešeními. Z tohoto porovnání vyšlo najevo, že žádné z nich nesplňovalo všechny požadavky. Vlastní řešení tedy předčilo všechny existující, neboť všechny tyto požadavky na funkcionalitu splňuje.

I přesto, že je výsledný load balancer implementovaný svými load balancing metodami a `ChannelHandlery` přímo na míru pro SPE server, nebylo by příliš složité vytvořit verzi pro obecné případy použití, vystavěnou na stejných principech jako realizované řešení. Tato verze by tak mohla snadno konkurovat již existujícím řešením, neboť dle výsledků testování provedených v rámci podkapitoly 10.5 se ukázalo, že realizované řešení může být dostatečně výkonné pro mnoho případů užití.

11.3 Začlenění do firemní platformy

V kapitole 8 bylo zobrazeno to, jak load balancer komunikuje s SPE serverem. Šlo ale pouze o zjednodušený pohled na celou věc. Jakou roli bude load balancer hrát ve firemní infrastruktuře je zobrazeno na následujícím obrázku.



Obrázek 30: Začlenění do firemní platformy

Na výše uvedeném obrázku lze vidět, jakým způsobem jsou propojeny všechny komponenty. Tento obrázek také odráží, jak by mohlo nasazení v cloudu vypadat.

11.4 Zhodnocení a přínos

Jak možno pozorovat na předchozím obrázku, je load balancer vzniklý v rámci této práce velmi přínosným prvkem do firemní platformy. Vzniklé řešení je na míru navrženo firemním požadavkům, čímž se vhodně začleňuje mezi ostatní komponenty platformy. Jeho aplikací, ve spolupráci s SPE servery, bude možné jednoduchým způsobem zpracovávat velké množství řečových nahrávek nejen v rámci platformy, ale také u zákazníků, kteří již využívají některé firemní produkty. Z výsledků testování prováděných v rámci kapitoly 10 si lze povšimnout, že se load balancer velmi dobře adaptuje na vzrůstající počet příchozích požadavků a zachovává si velmi stabilní chování. Není překvapením jeho rychlost odbavování požadavků, jelikož je vystavěn na prověřených konceptech, které byly probírány v rámci kapitoly 4.

11.5 Budoucí rozšíření

Díky konceptu, jakým se programují síťové aplikace ve frameworku Netty, vznikl z výsledného load balanceru modulární, lehce rozšiřitelný produkt. V budoucnu tak nebude problém s rozšířením některých jeho funkcí. Může se jednat jak o podporu nových protokolů, přidání nových load balancing metod nebo rozšíření jeho REST API.

Dle mého názoru by bylo vhodné v budoucnu umožnit kooperaci se záložním load balancerem. Aktuálně je load balancer v návrhu platformy „single point-of-failure“[42]. Měla by tedy vzniknout podpora pro to, aby běžely dva load balancery v active-pasive schématu. Pasivní load balancer by periodicky kontroloval dostupnost aktivního load balanceru a pokud by došlo k jeho výpadku, automaticky by zaujal jeho místo.

Dalším z vhodných budoucích rozšíření je automatické použití protokolu WebSocket pro vyzvedávání výsledků asynchronních operací z SPE serverů. Aktuálně je load balancer schopen podpory protokolu WebSocket, ale pouze tehdy, pokud je povýšení spojení na WebSocket vyžádáno klientem. Pokud se klient cyklicky dotazuje na skutečnost, zda je asynchronní operace již hotova, jsou všechny jeho požadavky forwardovány na SPE. Dodáním logiky, že při vyzvedávání výsledků asynchronních operací by `HttpBackendHandler` navázal vždy WebSocket spojení, by mělo dojít ke snížení síťového provozu mezi load balancerem a SPE serverem. Klientovi by tato logika zůstala naprosto skryta.

Vhodným kandidátem na doplnění funkcionality je přidání autentizace a autorizace pro REST API. Tato funkcionalita by tak zabránila neautorizovanému přístupu ke konfiguraci load balanceru.

[42]Single point of failure – `http://ask-leo.com/whats_a_single_point_of_failure_and_why_do_i_need_to_know.html`

12 Závěr

Cílem této diplomové práce bylo navrhnout a realizovat řešení, jak server pro zpracování řeči clusterovat a load balancovat. Výsledné řešení by se mělo začlenit mezi komponenty, které firma Phonexia s.r.o. využívá a nabízí svým zákazníkům.

V úvodní kapitole jsem se zabýval důvodem vzniku této diplomové práce. Tato kapitola je následována kapitolami zpracovávajícími teoretickou část této práce.

Nejprve jsem detailně popsal problematiku load balancingu, jeho úskalí a principy.

V následující kapitole jsem se věnoval síťovému programování prezentovanému v jazyce Java. Zde je probráno několik koexistujících konceptů, které jsou vzájemně porovnány a také je zde popsáno, jak se mohou některé koncepty navzájem doplňovat. Důležitost této kapitoly tkví převážně v ujasnění a pochopení probíraných konceptů, neboť další části této práce se na ně často odkazují.

Následující kapitola byla věnována stručnému popisu řečových technologií a způsobům, jak získávat znalosti z mluvené řeči.

V poslední kapitole teoretické části jsem popsal server pro zpracování řeči. Rozebral jsem zde to, jak pomocí tohoto serveru lze zpracovávat nahrávky řečovými technologiemi, které firma Phonexia s.r.o. nabízí a vyvíjí. Také jsem zde rozebral to, jakým způsobem lze se serverem komunikovat, neboť pochopení této skutečnosti bylo klíčové pro budoucí realizaci load balanceru.

Praktickou část diplomové práce jsem započal analýzou a souhrnem požadavků a následným průzkumem již hotových load balancerů. Výsledkem této kapitoly bylo zjištění, že neexistuje žádné dosavadní řešení, který by splnilo všechny požadavky.

V důsledku neexistence dostupného řešení bylo potřeba navrhnout vlastní architekturu load balanceru, čemuž jsem se věnoval v 8. kapitole. V rámci ní jsem se poohlédl po vhodných nástrojích, které by usnadnily tento úkol. Zvoleným nástrojem se stal framework Netty, a inspirován konceptem tvorby síťových aplikací v tomto frameworku jsem vytvořil abstraktní návrh, pokrývající všechny analyzované požadavky. Tento návrh jsem poté implementoval a v rámci následující kapitoly jsou popsány implementační detaily.

Hotové řešení jsem podrobil testování, které mělo simulovat budoucí běh v cloudu. Výsledky byly porovnány a analyzovány, z čehož posléze vznikly charakteristiky provozu.

V poslední kapitole jsme zabýval začleněním a přínosem mého řešení a možnostmi jeho budoucího rozšíření.

Samotným výsledkem této práce je implementovaný load balancer, který splňuje veškeré požadavky a je velmi vhodný vzhledem ke své rychlosti a jednoduché možnosti budoucího rozšíření. Dovolím si tedy závěrem konstatovat, že využitím tohoto load balanceru lze server pro zpracování řeči efektivně load balancovat a clusterovat a byly tedy splněny všechny předem stanovené cíle.

13 Reference

ABBOTT, Martin L a Michael T. FISHER. *The art of scalability: scalable web architecture, processes, and organizations for the modern enterprise*. Second edition. New York: Addison-Wesley, 2015. ISBN 9780134032801.

ALLSPAW, John. *The Art of Capacity Planning: Scaling Web Resources: Being Ready for the Big Growth Spurt*. New York: Addison-Wesley, 2008. 156 s. ISBN 978-0-596-51857-8.

AYEDO, *What is Netty?*. http://ayedo.github.io/. [online]. 4.12.2016 [cit. 2016-12-04]. Dostupné z: http://ayedo.github.io/netty/2013/06/19/what-is-netty.html..

BOURKE, Tony. *Server load balancing*. Beijing ; Sebastopol, Calif: O'Reilly, c2001. ISBN 0596000502.

Event-driven programming. TechnologyUK. [online]. 4.12.2016 [cit. 2016-12-04]. Dostupné z: https://www.technologyuk.net/computing/software-development/event-driven-programming.shtml.

HAN, Jiawei, Micheline KAMBER a Jian PEI. *Data mining: concepts and techniques*. 3rd ed. Haryana, India ; Burlington, MA: Elsevier, 2012. ISBN 9789380931913.

JENKOV, Jakob. *Load Balancing*. JENKOV.COM. [online]. 4.12.2016 [cit. 2016-12-04]. Dostupné z: http://tutorials.jenkov.com/software-architecture/load-balancing.html.

Load balancing Frequently Asked Questions. http://blog.haproxy.com/. [online]. 4.12.2016 [cit. 2016-12-04]. Dostupné z: http://blog.haproxy.com/loadbalancing-faq/.

MACVITTIE, Don. *Intro to Load Balancing for Developers – The Algorithms*. F5 DevCentral. [online]. 16.6.2010 [cit. 2016-12-04]. Dostupné z: https://devcentral.f5.com/articles/intro-to-load-balancing-for-developers-ndash-the-algorithms.

MAURER, Norman a Marvin WOLFTHAL. *Netty in action*. Shelter Island, NY: Manning Publications Co., 2016. ISBN 1617291471.

MEMBREY, Peter, Eelco PLUGGE a David HOWS. *Practical load balancing ride the performance tiger*. Berkeley, CA: Apress, 2012. ISBN 9781430236818.

Netty. Netty Project. [online]. 4.12.2016 [cit. 2016-12-04]. Dostupné z: http://netty.io/.

PHONEXIA. *Brno Speech Application Interface Documentation*. Voice Biometry and Speech Recognition Technologies – Phonexia. [online]. 16.11.2015 [cit. 2016-12-19]. Dostupné z: https://www.phonexia.com/docs/bsapi/index.html.

PSUTKA, Josef. *Mluvíme s počítačem česky*. Praha: Academia, 2006. Česká matice technická (Academia). ISBN 80-200-1309-1.

ROTH, Gregor. *Server load balancing architectures, Part 1: Transport-level load balancing*. JavaWorld.com. [online]. 21.10.2008 [cit. 2016-12-04]. Dostupné z: http://www.javaworld.com/article/2077921/architecture-scalability/server-load-balancing-architectures–part-1–transport-level-load-balancing.html.

TARREAU, Willy. *Making applications scalable with Load Balancing*. Tarreaus stuff. [online]. 19.11.2006 [cit. 2016-12-04]. Dostupné z: http://wtarreau.blogspot.cz/2006/11/making-applications-scalable-with-load.html.

UBL, Malte a Eiji KITAMURA. *Introducing WebSockets: Bringing Sockets to the Web*. HTML5 Rocks – A resource for open web HTML5 developers. [online]. 20.10.2010 [cit. 2016-12-19]. Dostupné z: https://www.html5rocks.com/en/tutorials/websockets/basics/.

URMA, Eibe, Mario FUSCO a Alan MYCROFT. *Java 8 in action: lambdas, streams, and functional-style programming*. Shelter Island: Manning, 2015. ISBN 1617291994.

WARNER, Tim. *Server Clustering Basics*. Certification Magazine. [online]. 19.10.2005 [cit. 2016-12-31]. Dostupné z: http://certmag.com/server-clustering-basics/.

What Is Global server load balancing?. NGINX. [online]. 4.12.2016 [cit. 2016-12-04]. Dostupné z: https://www.nginx.com/resources/glossary/global-server-load-balancing/.

What Is Layer 4 Load Balancing?. NGINX. [online]. 4.12.2016 [cit. 2016-12-04]. Dostupné z: https://www.nginx.com/resources/glossary/global-server-load-balancing/.

Zpracování řečových signálů – ZRE 2015/16. Faculty of Information Technology, Brno University of Technology. [online]. 19.12.2016 [cit. 2016-12-19]. Dostupné z: http://www.fit.vutbr.cz/study/courses/ZRE/public/.