VŠB TECHNICKÁ UNIVERZITA OSTRAVA
FAKULTA ELEKTROTECHNIKY A INFORMATIKY

SIP reverzní proxy s funkcí distribuce zátěže

SIP Reverse Proxy with a Loadbalancer Function

Bc. Petr Vaněk

Diplomová práce

Vedoucí práce: Ing. Jan Rozhon, Ph.D.

Ostrava, 2021

Abstrakt

Na dostupnost služeb, nehledě na typ, je kladen stále větší tlak. Nejinak je tomu i v případě služeb nabízených skrze technologii VoIP. Cílem této diplomové práce je navrhnout a realizovat za pomocí open source softwaru testovací prostředí, jehož hlavním úkolem bude automaticky distribuovat zátěž v podobě hovorů mezi skupinu aplikačních serverů, a to s ohledem na aktuální vytížení jejich procesorů. Navrhnutý systém pro vyvažování zátěže je v závěru práce testován a zhodnocen z pohledu jeho výkonosti, spolehlivosti a možného nasazení v reálném prostředí. Úvodní teoretická část práce je tvořena stručnou charakteristikou signalizačního protokolu SIP, na kterou dále navazují kapitoly zabývající se popisem principu fungování a vlastnostmi SIP serverů Kamailio a Asterisk.

Klíčová slova

VoIP, SIP, Kamailio, Asterisk, distribuce zátěže, SIPp

Abstract

Availability of services, regardless of the type, is under increasing pressure. This is also the case with services offered through VoIP technology. The aim of this master thesis is to design and implement a testing environment using open source software, whose main task will be to automatically distribute the load in the form of calls between a group of application servers, taking into account the current load of their processors. The proposed system for load balancing is tested and evaluated at the end of the thesis in terms of its performance, reliability and possible deployment in a real environment. The introductory theoretical part of the thesis consists of a brief description of the SIP signaling protocol, which is followed by chapters dealing with the description of the principle of operation and features of SIP servers Kamailio and Asterisk.

Keywords

VoIP, SIP, Kamailio, Asterisk, load balancing, SIPp

Poděkování

Na tomto místě bych rád poděkoval Ing. Janu Rozhonovi, Ph.D za jeho vřelý přístup, odbornou pomoc a zejména čas, který věnoval konzultacím během vypracovávání této diplomové práce.

Obsah

Seznam použitých zkratek a symbolů

AEL	–	Asterisk Extension Language
AGI	–	Asterisk Gateway Interface
AMI	–	Asterisk Manager Interface
API	–	Application Programming Interface
ARI	–	Asterisk REST Interface
B2BUA	–	Back to Back User Agent
CDR	–	Call Detail Records
CEL	–	Call Event Logging
CPU	–	Central Processing Unit
CRUD	–	Create, Read, Update, Delete
CSV	–	Comma Separated Values
DAHDI	–	Digium Asterisk Hardware Device Interface
DNS	–	Domain Name System
GPG	–	GNU Privacy Guard
GPL	–	General Public License
HTTP	–	Hypertext Transfer Protoco
IAX	–	Inter-Asterisk eXchange
IETF	–	Internet Engineering Task Force
IP	–	Internet Protocol
JSON	–	JavaScript Object Notation
LSS	–	Linux Support Services
MI	–	Management Interface
NAPTR	–	Name Authority Pointer
PBX	–	Private Branch Exchange
PCAP	–	Packet Capture
PSTN	–	Public Switched Telephone Network
RFC	–	Request For Comments
RPC	–	Remote Procedure Call

RTCP	–	RTP Control Protocol
RTP	–	Real-time Transport Protocol
SCCP	–	Signalling Connection Control Part
SDP	–	Session Description Protocol
SER	–	SIP Express Router
SIP	–	Session Initiation Protocol
SMTP	–	Simple Mail Transfer Protocol
SOAP	–	Simple Object Access Protocol
SQL	–	Structured Query Language
SRTP	–	Secure Real-time Transport Protocol
SRV	–	Service record
TCP	–	Transmission Control Protocol
TCP/IP	–	Transmission Control Protocol/Internet Protocol
TLS	–	Transport Layer Security
UA	–	User Agent
UFW	–	Uncomplicated Firewall
URI	–	Uniform Resource Identifier
URL	–	Uniform Resource Locator
uWSGI	–	Web Server Gateway Interface
WAV	–	Waveform Audio File Format
WebRTC	–	Web Real-Time Communication
WSGI	–	Web Server Gateway Interface
XAVP	–	extended Attribute-Value Pairs

Seznam obrázků

Seznam tabulek

Kapitola 1

Úvod

Nachazíme se v době, ve které je kladen velký důraz na dostupnost služeb, a to všeho typu. V oblasti VoIP se může jednat například o call-centra, jejichž dostupnost může být v některých případech klíčová. Možným řešením jak vysokou dostupnost nabízené služby zajistit je distribuovat zátěž v podobě hovorů mezi soustavu aplikačních serverů pomocí takzvaného reverzního SIP Proxy serveru. Tím je zajištěno rovnoměrné rozložení zátěže a zároveň v případě výpadku některého z řady aplikačních serverů i automatické přesměrování provozu na servery zbylé. Další výhodou takto organizované topologoie je zvýšení zabezpečení celého systému, jelikož identita aplikačních serverů a topologie vnitřní sítě mohou být skryty právě za reverzním SIP Proxy serverem, který tak do jisté míry plní roli bezpečnostního prvku.

Tato práce si klade za cíl navrhnout a nakonfiguravat s pomocí open source softwaru testovací prostředí, ve kterém bude roli reverzního SIP Proxy serveru zastávat SIP server Kamailio a roli aplikačních serverů budou plnit Asterisk servery. Kamailio disponuje modulem s názvem Dispatcher, který je pro účely distribuce zátěže přímo určen. Modul nabízí řadu algoritmů, které určují na základě jakých parametrů bude zátěž přerozdělována. Prvním úkolem praktické části je proto tento modul nakonfigurovat a ověřit funkčnost a možnosti tohoto řešení. Hlavní úskalí, které je však s tímto modulem a jeho algoritmy spojeno je to, že jen "slepě" přerozděluje zátěž bez toho, aniž by věděl něco o skutečném vytížení jednotlivých aplikačních serverů. Mým cílem je proto vytvořit takový systém, ve kterém bude mít Kamailio jakožto distribuční prvek přístup k informacím týkajících se aktuálního vytížení aplikačních serverů a na jejich základě rozhodne, kam v danou chvíli zátěž v podobě hovoru směrovat. Takové řešení může být vhodné například pro infrastrukturu, ve které jednotlivé aplikační servery nedisponují stejnou výpočetní kapacitou a nejslabší z nich by tak za normálních okolností tvořil takzvané úzké hrdlo celého systému.

Teoretická sekce práce se v úvodu zabývá stručným popisem signalizačního protokolu SIP, který je v současnosti nejpoužívanějším protokolem tohoto typu v technologii VoIP a z pohledu praktické části práce se jedná o stěžejní protokol. Následně je uveden podrobný popis dvou zástupců SIP

serverů, a to Kamailia a Asterisku. V obou případech bude uveden náhled do jejich historie, význam z pohledu VoIP, princip jejich činnosti a popis architektury obou serverů. Na tuto část navazuje kapitola obsahující samotný návrh testovací topologie, a to včetně popisu klíčových komponent, ze kterých bude navrhnutý systém tvořen. Z teoretické části se následně přesuneme k samotné instalaci a konfiguraci systému pro distribuci zátěže. Ukážeme si konfiguraci s využitím Dispatcher modulu, ale primárně se zaměříme na implementaci vlastního řešení, kterému bude věnována i kapitola zabývající se jeho testováním.

Kapitola 2

SIP

SIP (Session Initiation Protocol) je signalizační protokol určený k sestavení, modifikaci či ukončení multimediálního spojení v sítích založených na IP protokolu. Tento protokol, vyvinutý skupinou IETF (Internet Engineering Task Force), byl v roce 2002 standardizován v aktuální verzi 2 pod RFC (Request For Comments) s označením 3261.

SIP představuje neproprietární textově orientovaný protokol (podobně jako SMTP a HTTP), který z pohledu TCP/IP pracuje na aplikační vrstvě. Samotné komunikující strany využívající SIP nevyžadují nasazení centrálního prvku, jelikož veškerá potřebná logika pro sestavení relace je obsažená v koncových prvcích. Z hlediska terminologie se tedy jedná o takzvaný *end to end* protokol.

Pro značení uživatelů využívá SIP takzvané SIP URI (Uniform Resource Identifier) neboli jmené identifikátory. Ty lze najít v několika polích SIP hlavičky, viz kapitola 2.3, ve kterých plní důležitou roli identifikace volajícího a volaného. SIP URI se skládá z části definující uživatele označované jako *username* a z části *host*, která definuje uživatelovu doménu. Ukázka podoby a struktury SIP URI v její základní i rozšířené podobě je uvedena na následujících příkladech:

- sip:bob@mydomain.com
- sip:123456789@192.168.0.1:5060
- sip:alice@herdomain.com;transport=udp;method=INVITE?Subject=example [1, 2, 3]

2.1 SIP prvky

V SIP infrastruktuře se můžeme setkat se dvěmi základními prvky, kterými jsou koncová zařízení označovaná jako UA (User Agents) a SIP servery. Z hlediska funkcí existuje celá řada SIP Serverů. V reálném nasazení se však většinou jedná o jeden prvek, který kombinuje jejich funkcionalitu do jednoho komplexního celku. Pro snažší popis budou v této části jednotlivé typy SIP serverů popsány jako samostatné entity. Výčet a stručná charakteristika SIP prvků vypadá takto:

- **UA** - představuje softwarový/hardwarový koncový prvek, na kterém je inicializováno spojení. Každý UA musí být schopen plnit roli UAC (User Agent Client) a UAS (User Agent Server). Pokud UA inicializuje spojení, tzn. posílá žádost na jeho sestavení, funguje v režimu UAC. Naopak pokud UA přijímá žádost a generuje odpověď, pracuje jako UAS. Samotné role klient/server se mezi účastníky mohou v průběhu signalizace měnit.

- **B2BUA** - jedná se o speciální případ UA, který přijímá žádost ze strany klienta, následně ji upraví a posílá jako žádost novou. Hovor je tímto prvkem rozdělen na dva samostatné hovory, kde B2BUA hraje pro inicializátora spojení funkci serveru, zatímco pro adresáta žádosti funkci klienta. Výchozí vlastností B2BUA je rovněž to, že skrze něho prochází i samotný mediální tok, viz kapitola 2.5. Tato vlastnost mu umožňuje provádět transkódování mezi účastníky, kteří nepodporují stejný kodek. Daní za tuto funkci jsou však vyšší nároky na výpočetní kapacitu.

- **SIP Registrar server** - je server, který přijímá registrační žádosti od UA. Potřebné informace obsažené v hlavičce žádosti jsou následně uloženy do lokační databáze, kde jsou v případě potřeby k dispozici pro SIP Proxy server. Součástí lokační databáze mohou být rovněž záznamy, obsahující umístění ostatních SIP Proxy serverů spravující jiné domény.

- **SIP Proxy server** - přijímá SIP zprávy od klientů či jiných SIP Proxy a směruje je dál k cíli. V praxi obvykle jeden SIP Proxy spravuje jednu doménu. Pro sestavení spojení mezi klienty nacházejícími se v různých doménách využívá SIP Proxy volajícího pro nalezení SIP Proxy serveru spravující doménu volaného své záznamy, či DNS (Domain Name System).

- **SIP Redirect server** - stejně jako SIP Proxy server přijímá žádosti na sestavení spojení a provádí vyhledávání adresáta v lokační databázi, avšak žádosti narozdíl od SIP Proxy nepřeposílá dál, ale pouze informuje jejich strůjce o aktuální adrese volaného. [1, 3, 4]

2.2 SIP zprávy

Samotný průběh signalizace je tvořen SIP zprávami. Ty se dělí do dvou skupin na SIP žádosti a SIP odpovědi. Zprávy jsou obvykle přenášeny pomocí transportního protokolu UDP (User Datagram Protocol) na portu 5060. Pro přenos však může být použit i spolehlivý spojově orientovaný protokol TCP (Transmision Control Protocol) anebo šifrovaný protokol TLS (Transport Layer Security).

Základní SIP žádosti definované v RFC 3261 jsou:

- **INVITE** - žádost o sestavení spojení.

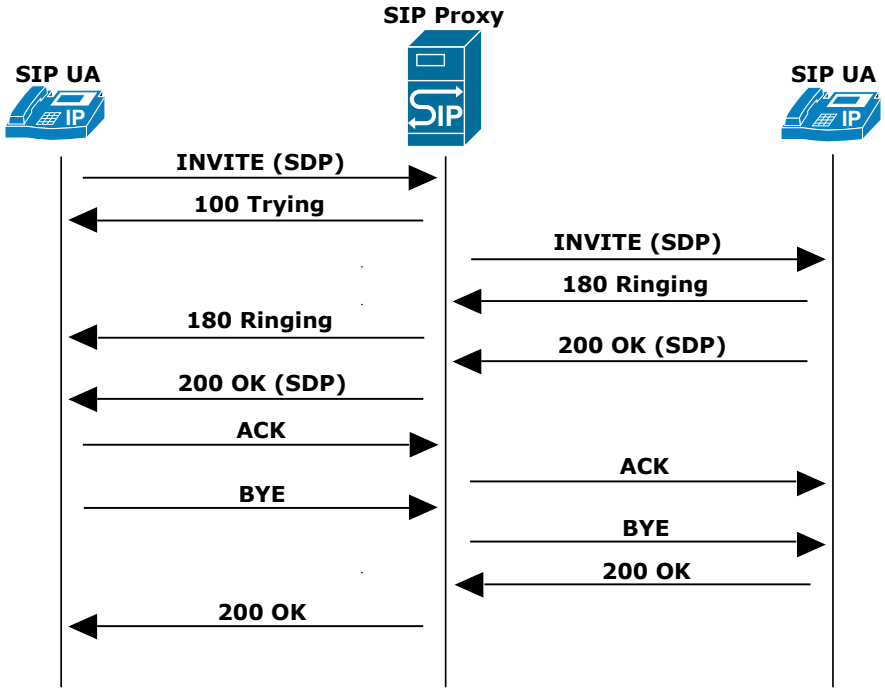
- **REGISTER** - žádost o registraci uživatele.

- **ACK** - žádost potvrzující přijetí konečné odpovědi na žádost INVITE.

- **OPTIONS** - žádost o zaslání informací týkajících se možností serveru nebo klienta.
- **BYE** - žádost o ukončení spojení, které bylo úspěšně sestaveno.
- **CANCEL** - žádost o ukončení ještě nesestaveného spojení. [1, 4, 5]

SIP odpovědi jsou rozděleny do celkem šesti tříd podle jejich kódu. Počáteční číslice kódu určuje konkrétní třídu a tím i obecný význam odpovědi. Krátká charakteristika a příklady odpovědí spadajících do jednotlivých tříd jsou uvedeny v následujícím seznamu:

- **1xx** - dočasné informativní odpovědi. Mezi zástupce této třídy patří odpověď *180 Ringing*.
- **2xx** - pozitivní finální odpovědi. Typickým zástupcem je odpověď *200 OK*.
- **3xx** - odpovědi posílané serverem, které se týkají přesměrování. Příklad: *305 Use Proxy*.
- **4xx** - negativní finální odpovědi, jejichž asi nejznámnějším zástupcem je *404 Not Found*.
- **5xx** - odpovědi oznamující chybu na straně serveru, například *502 Bad Gateway*.
- **6xx** - odpovědi spojované s globální chybou, například *600 Busy Everywhere*. [4, 5]



Obrázek 2.1: Ukázka SIP komunikace dvou UA skrze SIP Proxy server

2.3 Obsah SIP zprávy

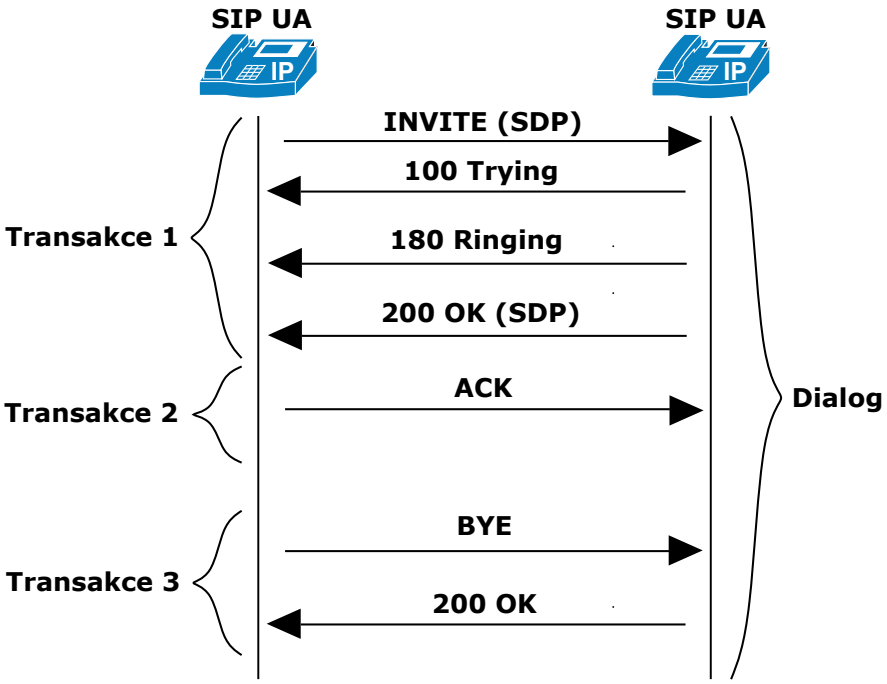
Součástí každé SIP žádosti či odpovědi je řada polí, které plní jistou informační roli. V případě SIP žádostí se na prvním řádku vždy nachází pole obsahující název žádosti, Request-URI a verzi SIP protokolu. Na tento úvodní řádek dále navazuje obsah samotné SIP hlavičky, která dle RFC 3261 musí obsahovat minimálně pole: *To*, *From*, *CSeq*, *Call-ID*, *Max-Forwards* a *Via*. Některé SIP žádosti/odpovědi obsahují mimo SIP hlavičku i takzvané tělo zprávy, jehož obsah a význam bude uveden v kapitole 2.5. Seznam a stručný popis základních polí objevujících se v SIP hlavičce doplněný o pole *Request-URI* má tuto podobu:

- **Request-URI** - označuje cíl, na který je žádost směrována.
- **From** - logická adresa volajícího.
- **To** - logická adresa volaného.
- **CSeq** - obsahuje číslo s náhodnou počáteční hodnotou, které se obvykle zvyšuje o 1 s každou novou žádostí. Výjimkou jsou žádosti CANCEL a ACK, které používají stejnou hodnotu CSeq jako žádost INVITE. Hlavním smyslem této položky je přiřazení odpovědi k dané žádosti.
- **Call-ID** - jedná se o náhodné číslo identifikující hovor. Toto číslo je vždy generováno uživatelem.
- **Max-Forwards** - udává maximální počet průchodu skrze SIP Proxy servery, které žádost může absolvovat. Na každém SIP Proxy serveru je číslo sníženo o 1. Jakmile se hodnota tohoto pole dostane na hodnotu 0, je strůjci žádosti poslána odpověď *483 Too Many Hops*.
- **Via** - pole sloužící k zaznamenání trasy, skrze kterou žádost putovala. Každý prvek, a to včetně odesílatele žádosti, přes který žádost prošla přidá do tohoto pole záznam se svou fyzickou adresou. Získané informace jsou následně použity pro směrování SIP odpovědi. Toto pole dále obsahuje důležitý atribut nesoucí název *branch*, který identifikuje takzvanou transakci, jejíž význam bude zmíněn v kapitole 2.4.
- **Contact** - fyzická adresa, na které odesílatel očekává příjem dalších SIP žádostí. Žádost o ukončení hovoru BYE tedy nemusí putovat skrze SIP Proxy server (pokud to SIP Proxy nevyžaduje), ale může být odeslána přímo adresátovi.
- **Record-route** - pole záhlaví využívané SIP Proxy servery k vynucení jejich setrvaní v signalizační trase na celou dobu jejího trvání. [1, 4, 5]

2.4 Transakce a dialog

Při pohledu na obrázek 2.1 zmíněný v kapitole 2.2 se SIP signalizace jeví jako prostá výměna určité množiny žádostí/odpovědí bez jakéhokoliv dalšího členění. Opak je ale pravdou. SIP přiřazuje určitou sekvenci zpráv do takzvaných transakcí. Transakce představuje žádost a všechny odpovědi, které se k ní vztahují. Nutno dodat, že odpověď nemusí být žádná. Z obrázku číslo 2.1 je patrné, že první transakce je tvořená žádostí INVITE a odpověďmi 100 Trying, 180 Ringing a 200 OK. Výjimkou by bylo, pokud by namísto finální zprávy 200 OK následovala finální zpráva z třídy 3xx, 4xx, 5xx nebo 6xx. V tom případě by do transakce patřila i následující žádost ACK. To ale není náš případ, a proto tvoří ACK samostatnou transakci. Jak bylo zmíněno v předchozí kapitole, transakce lze identifikovat pomocí hodnoty atributu *branch*, který se nachází v poli *Via*. Proto všechny SIP zprávy patřící do jedné transakce musí mít hodnotu tohoto atributu stejnou.

Další pojem, který se pojí s členěním zpráv, je dialog. Dialog lze popsat jako souhrn všech SIP zpráv, které spojuje určitá stejná vlastnost. Touto vlastností se myslí stejná hodnota pole *Call-ID* nacházejícího se v SIP hlavičce a dále také hodnota takzvaných tagů, jež jsou součástí polí *To* a *From*. Jeden hovor charakterizovaný hodnou pole *Call-ID* tedy tvoří dialog.



Obrázek 2.2: Transakce a dialog v SIP signalizaci

Členění zpráv do transakcí a dialogů umožňuje SIP Proxy serverům provádět dodatečné funkce, jako například větvení hovorů neboli takzvaný Forking. To platí ale pouze v případě, kdy se jedná o takzvaný stateful SIP Proxy server. Tento typ SIP Proxy serveru je schopen přiřazovat jednotlivé

zprávy k transakcím a dialogu. Rovněž si pamatuje stav transakce, což mu umožňuje detekovat výskyt opakujících se zpráv. Opakem statefull serverů jsou stateless SIP Proxy servery, které pouze přeposílají zprávy bez jejich přiřazování k transakcím či dialogu. [1, 3]

2.5 Protokoly spolupracující se SIP

Jak bylo uvedeno na začátku této kapitoly, SIP slouží pro sestavení, modifikaci či ukončení multimediálního spojení v sítích využívajících IP protokol. SIP však nepopisuje vlastnosti multimediální relace jako takové a nestará se ani o samotný přenos multimediálního obsahu. Pro tyto účely se využívájí protokoly SDP (Session Description Protocol) a RTP (Real-Time Transport Protocol), jejichž základní vlastnosti si nyní ve stručnosti popíšeme .

SDP je protokol sloužící k popisu parametrů multimediální relace. Obsahuje například pole informující o typu mediální relace, použitém kodeku apod.. Obsah SDP je posílán jako součást určitých SIP zpráv. Ve většině případů se jedná o žádost INVITE a na ni navazující finální odpověď 200 OK. Tím je zajištěna výměna informací o mediálním spojení mezi oběma stranami. Jak jsme si již uvedli, SIP zpráva obsahuje i takzvané tělo zprávy, které v případě potřeby může sloužit právě pro přenos dat SDP protokolu. Obsah SDP je tvořen položkami, jež jsou identifikovány písmeny. Jejich základní výčet a stručný popis je následující:

- **v (protocol version number)** - verze protokolu. Je vždy nastavena na hodnotu 0.
- **o (owner/creator and session identifier)** - definuje zakladatele relace.
- **s (session name)** - název relace.
- **c (connection information)** - informace o připojení. Například: IN IP4 192.168.0.5.
- **i (session information)** - informace o relaci v textové podobě.
- **t (timer session starts and stops)** - obsahuje časovou značku začátku a konce relace.
- **m (media information)** - udává typ mediálního toku např. audio. Dále číslo portu, typ transportního protokolu a typ užitečné zátěže.
- **a (media attributes)** - rozšiřující vlastnosti. Bývají zde popsány podporované kodeky, přičemž každý kodek je definován na samostatném řádku. [1, 3, 4]

RTP je protokol určený k přenosu real-time obsahu (audio/video) v sítích využívajících IP protokol. Z hlediska jeho funkce lze říci, že se jedná o transportní protokol pracující nad nespolehlivým nespojově orientovaným protokolem UDP, jehož vlastnosti vylepšuje právě pro nasazení v real-time aplikacích. Zabezpečenou alternativou protokolu RTP je protokol SRTP (Secure RTP). Hlavička

RTP protokolu má velikost 96 bitů a obsahuje celkem deset položek, z nichž lze z hlediska významnosti vyzdvihnout zejména tyto:

- **Payload Type** - 7-bitové pole definující formát užitečné zátěže.
- **Sequence Number** - 16-bitové pole s náhodnou počáteční hodnotou, která se zvyšuje o jedna s každým dalším odeslaným datagramem. Díky tomuto poli je příjemce schopen sestavit datagramy do správného pořadí či detekovat ztrátu některého z nich, což u samotného UDP protokolu není možné.
- **Timestamp** - 32-bitové pole obsahující časovou značku získanou z lineárního časovače během vzorkování prvního oktetu užitečné zátěže. Příjemce je schopen na základě této položky určit velikost jitteru neboli rozptylu zpoždění mezi pakety.

Spolu s RTP je definován i dohledový protokol RTCP (RTP Control Protocol), jehož úkolem je sběr statistických informací týkajících se přenosu a jeho kvality. [1, 4, 6]

Kapitola 3

Kamailio

Oblast páteřní SIP komunikace je charakteristická svými vysokými nároky na rychlost, spolehlivost, škálovatelnost a bezpečnost. Mezi volně dostupné implementace SIP serveru, které zmíněné nároky páteřní oblasti bezesporu plní, patří i Kamailio.

Kamailio představuje vysoce výkonný open-source SIP server vydaný pod licencí GPL (General Public License), který je určen výhradně pro Unix/Linux systémy. Kořeny Kamailia sahají do roku 2001, kdy byl institutem FhG Fokus Research vytvořen projekt SIP Express Router (SER). V roce 2005 se od projektu SER odtrhla skupina, která svůj projekt nazvala OpenSER, ale z důvodu porušování ochranných známek byl v roce 2008 tento projekt přejmenován na Kamailio. Ještě v témže roce se projekty SER a Kamailio (OpenSER) opět sloučily, aby sjednotily veškerou práci a výhody vycházející z obou dočasně odtržených projektů. V roce 2012 byla integrace dokončena, přičemž bylo rozhodnuto, že sloučený projekt dále ponese jméno Kamailio. [4, 7]

Za svůj výkon vděčí Kamailio zejména jednoduchosti, hardwarové nenáročnosti a struktuře své architektury. Tyto vlastnosti umožňují jeho nasazení i v takzvaných embedded systémech, což jsou systémy s omezenou výpočetní kapacitou. Primárně se však Kamailio uplatňuje jako páteřní SIP server, který je schopen v závislosti na použitém hardwaru zpracovávat tisíce až desetitisíce požadavků každou sekundu. Takový výkon lze v případě potřeby dále škálovat přidáním dalšího Kamailia.

Z pohledu jednotlivých zástupců SIP entit je Kamailio typickým představitelem SIP Proxy serveru. Může ale plnit i roli Registrar či Location serveru, což se velmi často děje právě spolu se SIP Proxy režimem. Aby bylo možné nasadit SIP server i v oblastech se specifickými nároky na funkcionalitu a chování, disponuje Kamailio velkým množstvím rozšiřujících modulů. Jako příklady moderních funkcí, které jsou zajišťovány právě skrze moduly, lze zmínit:

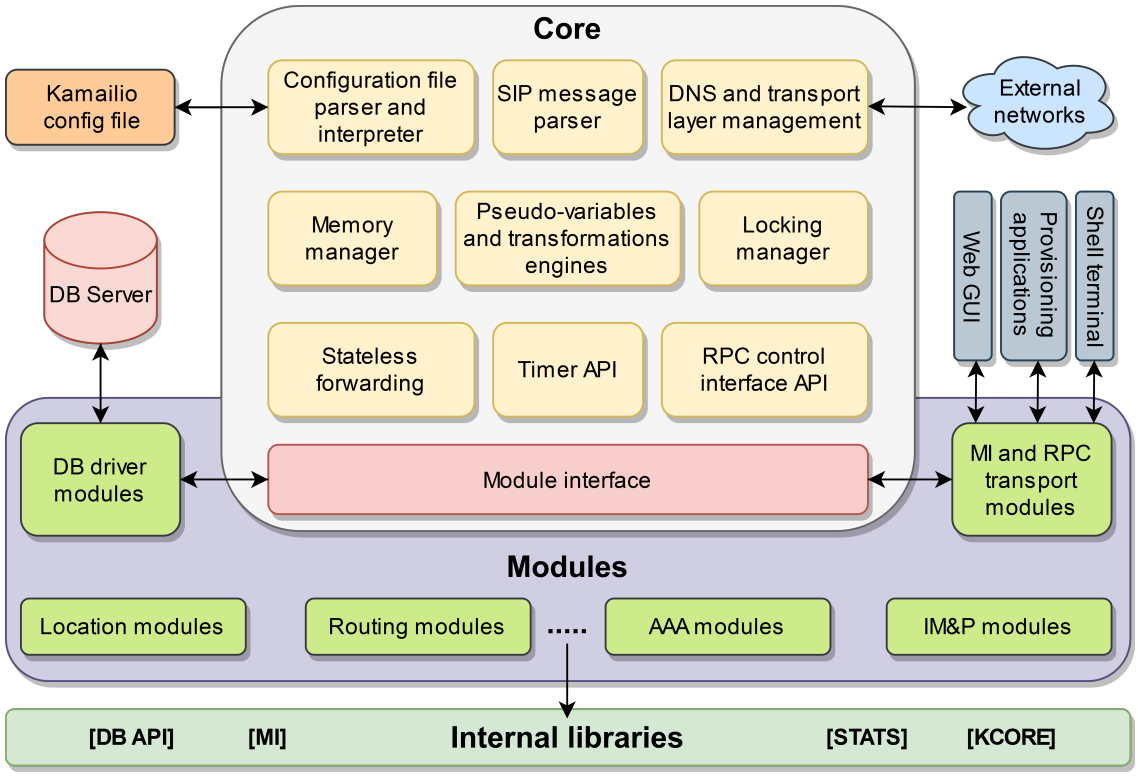
- Load-balancing (vyvažování zátěže),
- WebSocket pro WebRTC. [7, 8]

3.1 Architektura

Kamailio disponuje modulární snadno rozšiřitelnou architekturou, jejíž struktura se dělí na:

- Jádro (Core),
- Moduly (Modules).

Podrobnější představu o struktuře a obsahu jednotlivých částí architektury, jejíž aktuální podoba je platná od verze 3.0.x, si lze udělat při pohledu na níže uvedený obrázek č. 3.1. Hlavní části architektury, které jsme si již uvedli, budou oblastí našeho zájmu v následující části této kapitoly, kde se s nimi blíže seznámíme. [9]



Obrázek 3.1: Struktura architektury Kamailia [9, 10]

3.1.1 Jádro

Obsah jádra prošel od verze 3.0.x určitou restruktualizací. Nově jsou v něm obsaženy takzvané interní knihovny, které obsahují sbírku funkcí, jež jsou využívaný řadou modulů. Tyto funkce však nemají natolik velký význam, aby byly samostatnou části jádra. Do interních knihoven byly dále zahrnuty některé méně používané komponenty původní verze jádra.

Popis prvků obsažených v jádru Kamailia je uveden na následujícím seznamu:

- **Memory manager** - Jelikož je Kamailio multi-procesní aplikace, je v mnohých případech vyžadován přístup ke sdílené paměti. Memory Manager se pomocí jednoduchého rozhraní snaží práci se sdílenou pamětí zjednodušit a tím i optimalizovat rychlost všech operací, které k ní vyžadují přístup.
- **SIP message parser** - Kamailio disponuje vlastní implementací SIP parseru nesoucího název "inkrementující". Tento název vychází z principu jeho fungování, kdy každou SIP zprávu prochází tak dlouho, dokud nenalezne hledaný prvek nebo dokud nedojde na konec SIP zprávy.
- **Locking system** - Kamailio obsahuje svůj zamykací systém, který obsahuje jednoduché rozhraní pro jeho případný vývoj. Klíčovým prvkem systému je takzvaný mutex semafor, jenž se může nacházet ve stavu "odemknut", "zamknut". Zámky mohu být použity například jako jednoduché proměnné.
- **DNS and transport layer management** - Implementuje podporu pro UDP, TCP, TLS a SCTP. Z hlediska požadavků specifikovaných v RFC 3263 (rozšiřuje RFC 3261) podporuje Kamailio DNS záznamy NAPTR a SRV, které jsou důležité při lokaci serveru a služby.
- **Configuration file parser and interpreter** - Pro parsování konfiguračního souboru jsou využívány prvky Flex a Bison. Úkolem těchto prvků je sestavit strom akcí, které jsou následně za běhu vykonávany pro každou SIP zprávu.
- **Stateless forwarding** - Část jádra zajišťující bezestavové směrování SIP žádostí. Pro bezestavové zpracování SIP odpovědí disponuje Kamailio modulem s názvem *sl*.
- **Pseudo-variables and transformations engines** - Pseudo-proměnné představují speciální tokeny umožnující řízení a přístup k částem SIP zprávy. Získané hodnoty mohou být předávány jako parametry ruzným skriptovacím funkcím, kde jsou před volaním dané funkce nahrazeny hodnotou. Transformace lze zařadit mezi funkce úzce spjaté právě s pseudo-proměnnými, na které jsou aplikovány primárně za účelem parsování nebo úpravy jejich hodnot.
- **RPC control interface API** - RPC (Remote Procedure Call) je rozhraní určené pro komunikaci s externími aplikacemi. Externí aplikace mohou skrze toto rozhraní volat funkce nebo procedury, které budou spuštěny uvnitř Kamailia. RPC API je zkonstruováno tak, aby podporovalo XML-RPC (Extensible Markup Language - Remote procedure call).
- **Timer API** - Kamailio poskytuje interní časovač s milisekundovou přesností. Časovač nabízí vývojářské API umožnující registrovat funkce, které mají být spuštěny každou sekundu/milisekundu nebo v násobcích sekund/milisekund. API rovněž umožňuje zahájit nový proces časovače. [9, 11, 12]

Mezi prvky obsažené v interních knihovnách patří:

- **Database abstraction layers** - Databázové API (v1, v2), které bylo původně součástí jádra.
- **Management interface (MI) API** - API řídícího rozhraní, které je zastaralou alternativou řídícího rozhraní RPC.
- **Statistics engine** - statistický prvek poskytující zpětnou vazbu v reálném čase týkající se zátěže a stavu jednotlivých instancí Kamailia. Informace jsou reprezentovány proměnou typu integer nebo funkcí, která vrací integer.
- **Některé vybrané komponenty ze staré verze jádra** - ostatní funkce, jejichž příme zařazení do jádra již postrádalo uplatnění. [9]

3.1.2 Moduly

Moduly představují dodatečný software rozšiřující základní nabídku funkcí SIP serveru. Pro zajištění větší stability jádra jsou moduly připojovány skrze modulové rozhraní. Nedochází tak k přímé interakci komponent jádra a samotných modulů. Počet modulů se s každou další verzí Kamailia zvětšuje, v případě nejnovější verze (5.5.x) je jich k dispozici více než 200. [9]

Jako konkrétní zástupce modulů lze zmínit například tyto:

- **AUTH** - modul zajišťující autentizaci,
- **DISPATCHER** - modul pro distribuci zátěže,
- **RTPENGINE** - modul pro řízení mediálních toků. [13]

3.2 Hlavní konfigurační soubor

Hlavní konfigurace SIP serveru probíhá skrze konfigurační soubor s názvem *kamailio.cfg*. Uvnitř tohoto souboru je možné definovat velké množství parametrů, načítat potřebné moduly a zejména ovlivňovat a upravovat chování SIP serveru jako takového. Struktura konfiguračního souboru je rozdělena na tyto tři části:

- sekce globálních parametrů,
- sekce nastavení modulů,
- sekce směrovacích bloků.

Konfigurace obsažená v sekci globálních parametrů a nastavení modulů je vykonána pouze jednou při inicializaci Kamailia, zatímco konfigurace nacházející se v sekci směrovacích bloků je po inicializaci vykonávána opakovaně. [9, 14]

3.2.1 Sekce globálních parametrů

Jedná se o úvodní sekci konfiguračního souboru, ve které jsou definovány parametry jádra a parametry nadefinované uživatelem. Hodnoty, kterých mohou dané parametry nabývat, jsou obvykle datového typu integer, string nebo boolean. Je možné se ale setkat s parametry, viz následující ukázka, které mohou být tvořeny kombinací již zmíněných datových typů. Ukázka definování a přiřazení hodnoty parametrům může mít tuto podobu: [14]

```
enable_tls=yes
listen=udp:192.168.0.1:5060
alias="sip.domain.com"
```

3.2.2 Sekce nastavení modulů

Tato sekce slouží k načtení modulů a nastavení jejich parametrů. V úvodu sekce se obvykle nachází příkaz *mpath*, který udává cestu ke složce s moduly. Následuje samotné načtení modulů pomocí klíčového slova *loadmodule* a nastavení parametrů modulů pomocí příkazu *modparam*. Nutno dodat, že příkaz *modparam* obsahuje vždy právě tři argumenty, a to konkrétně název modulu, parametr modulu a hodnotu parametru. Obsah sekce může mít následující podobu: [14]

```
mpath="/usr/local/lib/kamailio/modules/"
loadmodule "dispatcher.so"
modparam("dispatcher", "list_file", "/etc/kamailio/dispatcher.list")
```

3.2.3 Sekce směrovacích bloků

Nejdůležitější sekce obsahující veškerou potřebnou logiku pro zpracování každé SIP zprávy. Mezi hlavní směrovací bloky, se kterými se můžeme v této sekci setkat, patří:

- **request_route** - směrovací blok, který je volán pro každou příchozí SIP žádost. Provádí kroky jako kontrolu či samotné směrování SIP žádosti. Jedná se o jediný povinný blok.
- **reply_route** - představuje hlavní blok pro zpracování všech SIP odpovědí.
- **onreply_route** - sekundární blok pro zpracování SIP odpovědí v souvislosti s jejich zařazením do aktivních transakcí.
- **branch_route** - definuje úkony pro zpracování každé SIP žádosti v rámci SIP transakce. Je používán zejména pro větvení hovorů, a to jak sériového, tak i paralelního.
- **failure_route** - definuje sadu možných akcí, jež mají být vykonány pro každou transakci, u které byla na SIP žádost přijata negativní finální odpověď třídy 4xx, 5xx nebo 6xx. [14, 15]

3.3 Princip činnosti Kamailia

Jak jsme již uvedli, Kamailio primárně reprezentuje zástupce SIP Proxy serverů, a to jak stateless, tak i transakčních a dialog stateful. Stateless režim je charakteristický svou rychlostí, protože není nutné udržovat jakékoliv informace o stavu transakce a přiřazovat SIP odpovědi ke konkrétním žádostem. Na zprávu je jednoduše zapomenuto, jakmile je zpracována a odeslána. Daní za rychlost a menší nároky jsou ale omezené možnosti z pohledu nabízených funkcí.

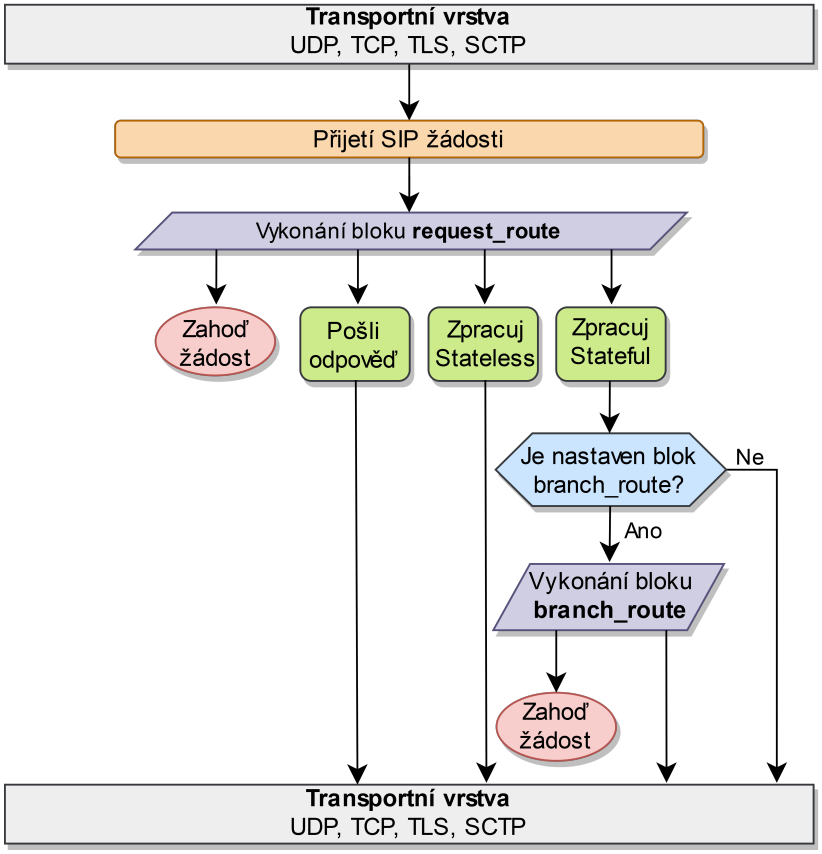
Kamailio jako transakční stateful SIP Proxy (výchozí stav) má naopak přehled o jednotlivých žádostech a k nim patřících odpovědí, protože jsou stavy transakce ukládány do paměti, a to do té doby, dokud není transakce ukončena finální odpovědí nebo dokud nevyprší časový limit. Právě režie týkající se správy transakcí a prací s pamětí tvoří rozdíl v rychlostí, s jakou jsou zpracovávány zprávy ve stateless a stateful módu. Tento handicap je však vynahrazen níže uvedenou funkcionalitou:

- **Paralelní a sériové větvení hovorů** - V případě, kdy má volaný k dispozici více SIP zařízení registrovaných pod stejným SIP URI, může Kamailio směrovat žádosti volajících na všechna tato zařízení. Samotný hovor je pak sestaven pouze s tím zařízením, skrze které byl hovor přijat. Rozdíl mezi paralelním a seriovým větvením je v tom, že zatímco paralelní větvení směruje žádost INVITE na všechna zařízení současně, seriové větvení pošle INVITE na jedno ze zařízení a teprve v případě neúspěchu posílá INVITE na další.
- **Detekce retransmisí** - Jelikož Kamailio ze stavu transakce ví, které odpovědi na žádost již přijalo, je schopno snadno detekovat a odstranit vicenásobný výskyt stejné zprávy.
- **Hlídaní časových limitů** - Kamailio může být nastaveno tak, aby hlídalo maximální dobu trvání transakce. Jakmile časový limit vyprší, je volajícímu zaslaná zpráva *408 - Request Timeout*. [3, 15]

To, zda bude Kamailio fungovat v režimu steless nebo stateful, je závislé pouze na použitých funkcích, modulech a směrovacích blocích. Funkce jsou buďto přímou součástí jádra, nebo konkrétního modulu. Aby bylo možné přiřazovat žádosti/odpovědi do transakcí a poskytovat funkce charakteristikcé pro stateful režim, je nezbytné využít funkce, nacházející se v modulu s názvem TM *Transaction modulu*. Ve stateless režimu je situace odlišná, jelikož jak jsme si uvedli v kapitole 3.1.1, pro stateless zpracování žádostí je využíváno jádro Kamailia, avšak u odpovědí je potřeba využít funkce SL modulu *StateLess*. Rozdíl mezi jednotlivými režimy není skryt pouze v používání odlišných modulů a jejich funkcí, ale i v tom, jaké směrovací bloky daný režim využívá. Všechny žádosti/odpovědi nehledě na režim využívají bloky *request_route* a *reply_route*, zatímco bloky *onreply_route*, *branch_route* a *failure_route* jsou určeny výhradně pro stateful režim. [14, 15]

V následujích částech této kapitoly si pomocí stavových diagramů a jejich popisu ukážeme proces zpracování SIP žádosti/odpovědi uvnitř Kamailia.

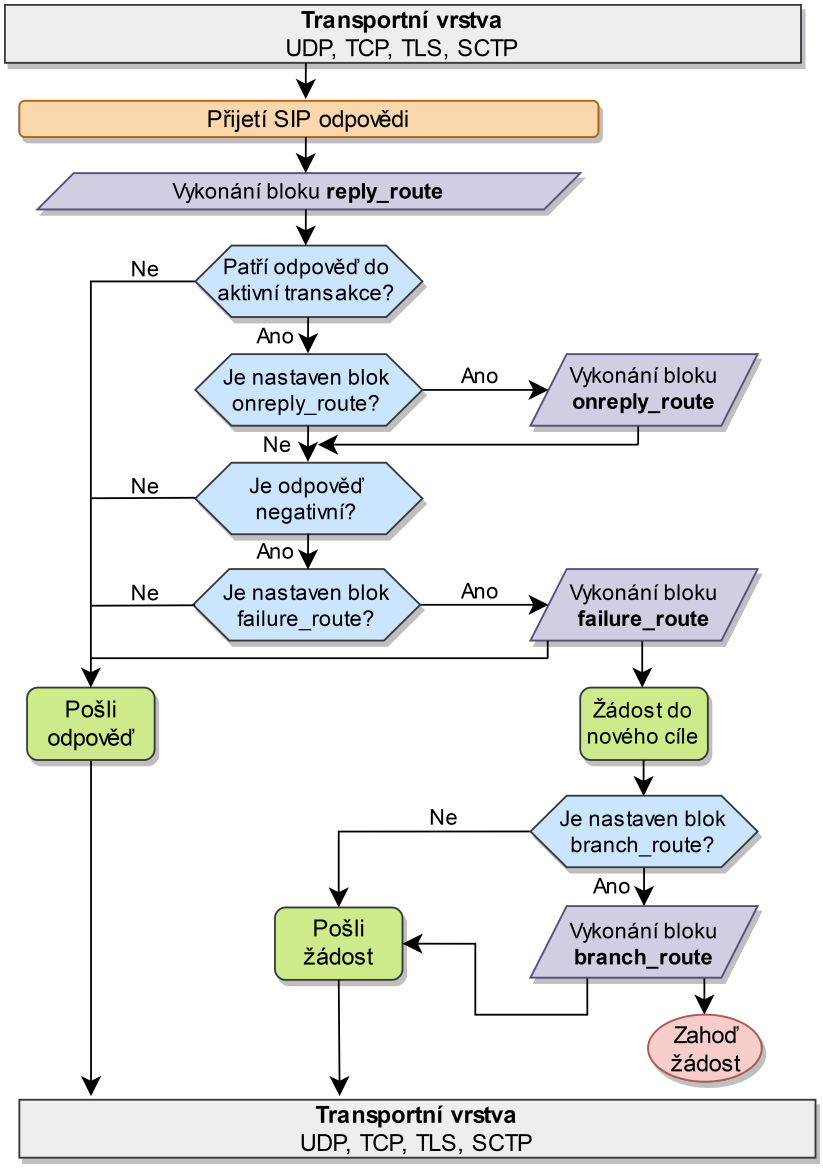
3.3.1 Proces zpracování SIP žádosti uvnitř Kamailia



Obrázek 3.2: Proces zpracování SIP žádosti uvnitř Kamailia [9]

Z výše uvedeného obrázku je viditelné, že proces zpracování SIP žádosti začíná na transportní vrstvě, skrze kterou byla žádost doručena. Nehledě na použitý transportní protokol putují všechny SIP žádosti do směrovacího bloku *request_route*. V tomto bloku se rozhodne, která z akcí bude provedena. V případě, kdy je žádost například špatně formulována nebo postráda nějaký parametr, je obvykle provedeno její zahození, načež je strůjci žádosti poslána zpráva třídy 4xx. Akce v podobě zaslání odpovědi je zvolena například i po přijetí žádosti OPTIONS, která má za úkol zjistit schopnosti serveru nebo plní funkci tzv. *keepalive* zprávy. Poslední dvě akce, které může blok *request_route* s žádostí vykonat, se týkají samotného režimu zpracování. Jak jsme si uvedli v předchozích kapitolách, režim může být buďto stateless nebo stateful. Nutno dodat, že zvolený režim se může měnit s každou přijatou žádostí. Ve stateless režimu je žádost jednoduše přeposlána pomocí funkcí obsažených v jádře Kamailia. Pro případ zvolení stateful módu je neprve zjištěno, zda je nastavená funkce pro větvení či speciální nastavení žádosti pomocí bloku *branch_route*. Pokud taková možnost není, je žádost rovnou odeslána. V případě, kdy je blok *branch_route* nastaven, dojde k provedení definovaných akcí, avšak i v tomto bloku může dojít k zahození žádostí. [9, 14, 15]

3.3.2 Proces zpracování SIP odpovědi uvnitř Kamailia



Obrázek 3.3: Proces zpracování SIP odpovědi uvnitř Kamailia [9]

Každá SIP odpověď putuje v rámci jejího zpracování nejprve do bloku *reply_route*. Zde se určí, zda odpověď přísluší do některé z aktivních transakcí. Pokud nepřísluší, je zpracována a odeslána ve stateless režimu. V opačném případě proces zpracování pokračuje a je ověřováno, zdali je k dispozici blok *onreply_route*, který má na starost zpracování odpovědí z pohledu transakcí. Jestliže je blok přítomen, je v něm odpověď patřičně zpracována. Následně je bez ohledu na přítomnost/nepřítomnost bloku *onreply_route* určeno, zda se jedná o negativní finální odpověď patřící do třídy 4xx, 5xx, 6xx. Pokud ne, může být odpověď odeslána. Pokud ano, je ověřováno, zda je pro tuto situaci

nastaven blok *failure_route*. Přítomnost tohoto bloku umožňuje poslat tutéž žádost, na kterou byla v prvním případě přijata negativní finální odpověď do jiného cíle. Tento postup je využíván zejména u sériového větvení hovorů. Mezi další akce, které je blok *failure_route* po přijetí negativní finální odpovědi schopen vyvolat, patří například spuštění hlasové schránky. [9, 14, 15]

Další kroky následující po vykonání bloku *failure_route*, již plně korespondují s procesem pro zpracování SIP žádosti, který byl zmíněn v kapitole 3.3.1.

Kapitola 4

Asterisk

Asterisk jakožto současná ikona a symbol IP telefonie patří mezi nejoblíbenější a nejnasazovanější rešení v oblasti komunikačních serverů. Jedná se o open source software určený výhradně pro Unix/-Linux operační systémy, který je schopen učinit z běžného počítače například pobočkovou ústřednu (PBX) se širokou škálou nabízených funkcí a služeb. [16, 17]

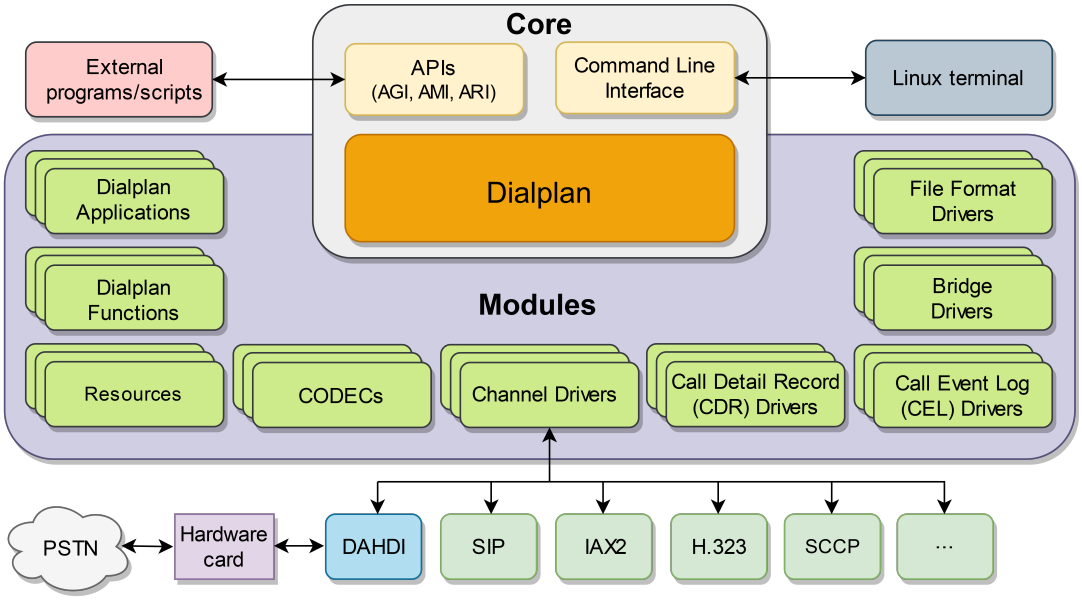
Strůjcem celé myšlenky, která byla u vzniku Asterisku je Mark Spencer. Ten se v roce 1999 jako čerstvý absolvent Alabamské Auburn University rozhodl vyplnit mezeru na firemním trhu a založit tak společnost LSS (Linux Support Services) nabízející linku podpory pro IT pracovníky využívající operační systém Linux. Růst firmy dosáhl v řádech několika málo měsíců takové meze, že bylo nezbytné nasadit nový telefonní systém, který by byl schopen distribuovat hovory mezi jednotlivé zaměstnance. Mark Spancer se proto obrátil na místní prodejce telefonních systémů s požadavkem na návrh systému pro jeho firmu, avšak k jeho zklámání byla cena všech nabídek daleko za možnostmi firmy LSS. Tento fakt přiměl Marka Spencera k myšlence naprogramovat vlastní řešení plnící roli pobočkové ústředny, která by splňovala nároky jeho firmy. V řádech měsíců bylo hotovo jádro Asterisku a zanedlouho poté i funkční prototyp, jehož zdrojový kód byl zveřejněn pod licencí GPL. Původní firma LSS dnes nese název Digium a spolu s komunitou se stále stará o rozvoj a vývoj Asterisku. [18]

Asterisk je velmi často zmiňován s přívlastkem "telefonní pobočková ústředna", avšak toto označení již dávno není přesné. Od původní role Asterisku jakožto pobočkové ústředny ušel celý projekt dlouhou cestu, v jejímž průběhu se rozvinul v komplexní komunikační nástroj, který je kromě oné zmíněné role schopen fungovat například i jako:

- VoIP brána,
- call centrum,
- voicemail server a další. [17, 19]

4.1 Architektura

Stejně jako v případě Kamailia je architektura Asterisku tvořena ze dvou základních prvků, a to jádra a modulů. Jádro obsahuje pouze prvky nezbytné pro chod samotné aplikace nicméně veškerá funkcionalita, kterou Asterisk disponuje, je k dispozici až v podobě přídavných modulů. Na níže uvedeném obrázku č.4.1 je uveden nástin struktury architektury Asterisku. Klíčové prvky, které jsou v architektuře obsaženy, budou předmětem popisu v následujících kapitolách.



Obrázek 4.1: Struktura architektury Asterisku [19, 20, 21]

4.1.1 Dialplan

Veškerá komunikace, ať už příchozí, nebo oddchozí se nehledě na typ použitého protokolu (SIP, IAX2 ...) soustřeďuje v Asterisku do jednoho klíčového místa, kterým je Dialplan. Dialplan obsahuje sadu instrukcí, které říkají jak příchozí/odchozí hovor zpracovat a jaké úkony nad ním vykonat. Obsah Dialplanu je definován uvnitř souboru s názvem *extensions*. Přípona tohoto souboru nás informuje o tom, jaký způsob byl z pohledu použitého programovacího/skriptovací jazyka použit pro jeho konfiguraci. Asterisk nabízí následující způsoby konfigurace:

- konfigurace pomocí skriptovacího jazyka Asterisku (*extensions.conf*).
- konfigurce pomocí AEL (Asterisk Extension Language) jazyka (*extensions.ael*).
- konfigurace pomocí programovacího jazyka Lua (*extensions.lua*). [22]

V této práci se zaměříme na první ze zmíněných způsobů, přičemž popis tohoto hlavního konfiguračního souboru bude obsažen v kapitole 4.2.

4.1.2 Moduly

Stejně jako u Kamailia je i v Asterisku drtivá většina funkcí dostupná v podobě přídavných modulů. Moduly jsou načítány prostřednictvím souboru, který nese název *modules.conf*. Z hlediska logického členění dělíme moduly Asterisku do těchto devíti skupin:

- **Dialplan Applications** - jsou aplikace používané uvnitř již zmíněného souboru *extensions.conf*, kde zprostředkovávají určité akce, které mohou být nad hovorem vykonány (vytočení čísla, zavěšení apod.).
- **Dialplan Functions** - jsou funkce, které umožňují pracovat či manipulovat s nastavením hovoru. Jako příklad funkcí lze uvést změnu ID volajícího u odchozího hovoru či manipulaci s textovými řetězci.
- **Resources** - představují moduly poskytující Asterisku funkcionalitu, která může být použita v průběhu hovoru. Lze zde zařadit takzvané parkování hovorů nebo službu music on hold. Tento blok dále obsahuje řadu dalších modulů, jejichž význam nezapadal mezi ostatní kategorie.
- **CODECs** - moduly určené pro kódování/dekódování mediálních toků. Jednou z funkcí Asterisku, které jsme si doposud neuvedli, je transkódování mediálních toků. Právě při této činnosti najdou tyto moduly největší uplatnění.
- **Channel Drivers** - představují jakési rozhraní mezi jádrem Asterisku a externím zařízením, skrze které procházejí všechny hovory. Každý Asteriskem podporovaný protokol má svůj takzvaný Channel driver, který je charakterizován samostatným modulem. Například Channel driver protokolu SIP nese název *chan_sip.so*, případně *res_pjsip.so*.
- **Call Detail Record (CDR) Drivers** - jsou moduly určené pro ukládání parametrů každého hovoru, a to primárně pro proces účtování. Data mohou být ukládána do CSV souborů nebo do relačních databází.
- **Call Event Logging (CEL) Drivers** - podobně jako CDR Drivers představují sadu modulů, jež mají za úkol zaznamenávat určité akce vyskytující se v průběhu hovoru. Oproti CDR Drivers jsou však záznamy podrobnější a komplexnější.
- **Bridge Drivers** - sada modulů určená k propojení většího množství hovorů například za účelem sestavení konference. Část modulů patřících do této skupiny je vyhrazena pro činnost spojenou s mixováním mediálních toků.
- **File Format Drivers** - je sada modulů, která je v Asterisku používána pro uložení mediálních toků (audio/video) na disk. V případě potřeby mohou být uložené záznamy převedeny zpět na mediální tok. [21, 23]

4.2 Hlavní konfigurační soubor

Jak již bylo uvedeno, soubor *extensions* je hlavním konfiguračním souborem, pomocí kterého je možno definovat Dialplan a tím i celé chování Asterisku jako takového. Z pohledu členění tohoto souboru jsou klíčovým prvkem takzvané *kontexty*. Kontexty rozdělují Dialplan do sekcí, které jsou na sobě zcela nezávislé, avšak za určitých podmínek může docházet k jejich interakci. Uvnitř kontextu jsou následně definovány jednotlivé operace, které jsou přístupné pouze tomu uživateli, jemuž byl daný kontext přidělen. Tím je umožněno přidělit určitou funkcionalitu Dialpalnu pouze konkrétním uživatelům nebo jednotlivým protokolům. Kontexty jsou definovány uvnitř hranatých závorek přičemž platí, že všechny instrukce nacházející se za názvem kontextu jsou jeho součástí. [23]

Instrukce/pravidla uvnitř kontextu nesou název *extensions*. Těchto pravidel se uvnitř kontextu může nacházet libovolné množství. Jakmile je vytvořeno volání pro konkrétní extension, například za pomocí vytočení čísla, jsou Asteriskem postupně vykonána veškerá pravidla definovaná pro tuto extension. Každá extension se skládá z následujících částí:

- Název nebo číslo dané extension.
- Priorita, která udává pořadí, v jakém budou jednotlivá pravidla pro konkrétní extension vykonána.
- Aplikace nebo akce, kterou má dané pravidlo vykonat. [19, 23]

Struktura a ukázka příkazů pro definování jednotlivých extensions uvnitř souboru extensions.conf má následující podobu:

```
exten => název, priorita, aplikace()
exten => 123, 1, Answer()
exten => 123, 2, Playback(hello-world)
exten => 123, 3, Hangup()
```

Z pohledu výše uvedených příkazů nabízí Asterisk možnosti, jak zápis zjednodušit, což je vhodné zejména v případě rozsáhlých konfigurací. Mezi tato ulehčení patří například příkaz *same*, který umožňuje vyhnout se nestálemu opakovaní názvu u pravidel patřících pod tutéž extension. Dalším usnadněním je vložení písmena *n* (od slova next) na pozici priority. Jakmile Asterisk pracuje s pravidlem mající prioritu n automaticku mu přiřadí prioritu o 1 větší, než mělo předchozí pravidlo. Z toho rovněž vyplývá, že je nezbytné definovat úvodní pravidlo s prioritou 1. Ukázka aplikace těchto ulehčení na dříve zmíněné příklady extensions vypadá následovně: [23, 24]

```
exten => 123, 1, Answer()
same => n, Playback(hello-world)
same => n, Hangup()
```

Velmi důležitým a často používaným nástrojem Dialplanu je vytváření vzorů - *Pattern matching*. Díky Pattern matchingu je možné předejít vytváření velkého množství extensions pro jednotlivá čísla tím, že dovoluje vytvořit jeden extension vzor, který bude odpovídat více než jednomu číslu. Extension vytvářející vzor poznáme podle podtržítka na začátku jeho názvu. V rámci vytváření vzorů se používá sada písmen/znaků. Výpis a význam těch základních je uveden na následujícím seznamu:

- **X nebo x** - reprezentuje číslo 0-9
- **Z nebo z** - reprezentuje číslo 1-9
- **N nebo n** - reprezentuje číslo 2-9

Ukázka definování jednoduchého vzoru, který bude vykonán, pokud někdo vytočí číslo 100-199, má následující podobu:

```
exten => _1XX, 1, Answer()
exten => _1XX, 2, Playback(hello-world)
exten => _1XX, 3, Hangup()
```

Soubor veškerých funkcí, které Dialplan nabízí, je velice rozsáhlý a přesahuje rámec této práce. Ve zkratce lze zmínit, že Dialpan umožňuje vytvářet cykly while(), přecházet mezi jednotlivými pravidly pomocí příkazu Goto() případně Gotoif() či práci s proměnnými. Pro základní účely této práce je však uvedený nástin funkcí a vlastností konfiguračního souboru extensions.conf plně dostačující.

Kromě souborů *extensions.conf* a *modules.conf* patří mezi důležité konfigurační soubory rovněž soubor *asterisk.conf*, který slouží k definování umístění jednotlivých adresářů a souborů a dále také ke konfiguraci některých parametrů jádra Asterisku. Pro účely vytvoření a nastavení SIP účtů a jejich zařazení do určitého kontextu se v práci dále setkáme s neméně důležitým konfiguračním souborem *pjsip.conf*. [19, 23, 25]

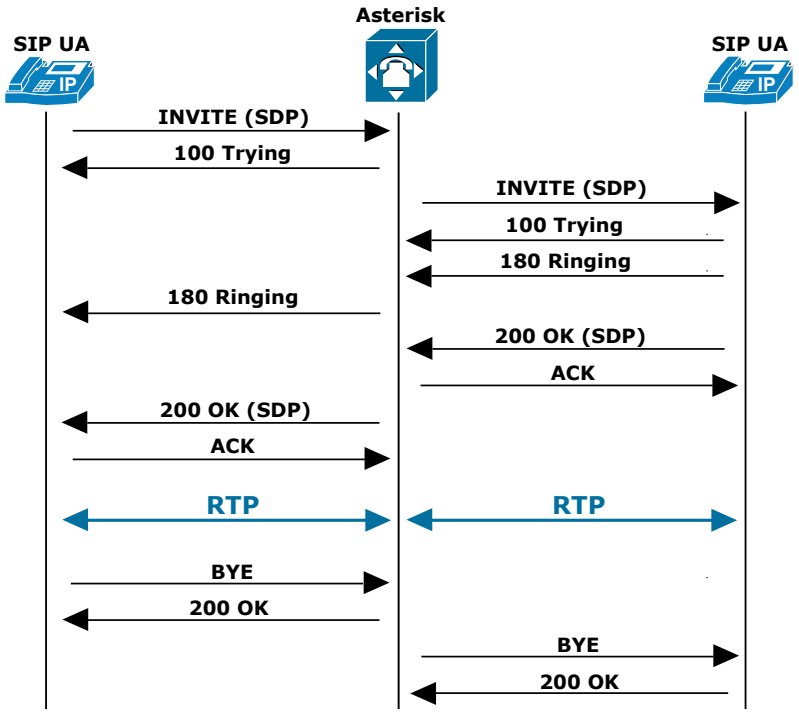
4.3 Princip činnosti Asterisku

V předchozím výčtu vlastností a funkcí jsme si uvedli, že Asterisk představuje univerzální komunikační server nabízející podporu pro protokoly SIP, IAX2, H.323, Skinny(SCCP) a další. Každý z těchto protokolů má svůj takzvaný ovladač kanálu, který je reprezentován modulem a odpovídajícím konfiguračním souborem. Primárním protokolem této práce je SIP, z tohoto důvodu bude i Asterisk plnit výhradně roli SIP prvku.

Z pohledu vnitřní logiky a samotného chování je Asterisk typickým představitelem B2BUA prvku. Základní charakteristikou B2BUA je to, že rozděluje jeden fyzický hovor na dva logické. Jeden probíhá mezi volajícím a Asteriskem a druhý mezi Asteriskem a volaným. Při pohledu na

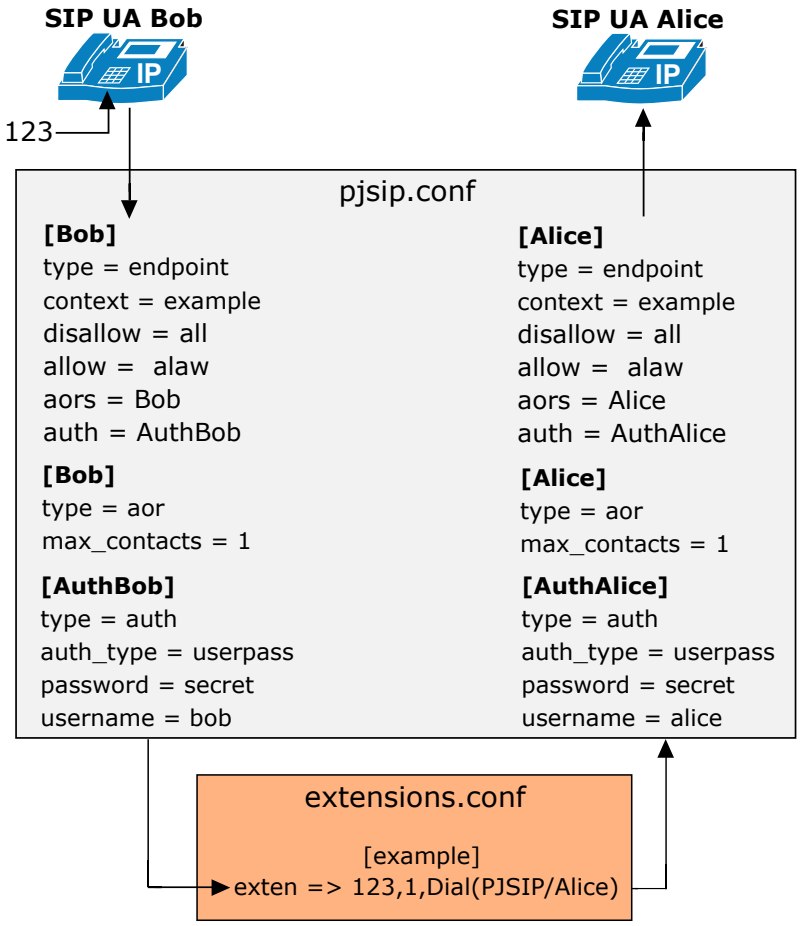
SIP hlavičku je tento fakt nejlépe patrný na poli Call-ID, které bude nabývat různých hodnot pro obě logické části hovoru. To, že Asterisk přepisuje určitá pole každé SIP hlavičky zvyšuje nároky na výpočetní kapacitu, což je také jeden z hlavních důvodů, proč není schopen zpracovávat zdaleka tak velké množství hovorů jako SIP Proxy server. Mezi ony přepisované hodnoty patří i ty, které jsou obsažené uvnitř SDP v polích *o (owner/creator)* a *c (connection information)*, čímž je zajištěno, že přes Asterisk bude procházet i samotný mediální tok. [23]

Vlastnosti, které charakterizují B2BUA režim umožňují nabídnout uživateli pokročilé funkce, které není možné skrze klasický SIP Proxy server nabízet. Daní za ony pokročilé funkce je ale nemalý vliv B2BUA režimu na nárust zpoždění, a to jak u SIP zpráv, tak i u samotného multimediálního toku, u kterého se tím navíc zvyšuje riziko ztrátovosti, což může dále vést k poklesu kvality multimediální relace. [1]



Obrázek 4.2: Ukázka SIP komunikace dvou UA skrze B2BUA (Asterisk)

Na výše uvedeném obrázku je znázorněn průběh SIP signalizace mezi dvěmi UA a Asterisk serverem jakožto B2BUA. Pokud srovnáme tento průběh s průběhem uvedeným na obrázku č. 2.1, jsou na první pohled patrné jisté rozdíly. Jedním z nich je, že Asterisk posílá ACK na odpověď 200 OK okamžitě po jejím přijetí, zatímco ke skutečnému adresátovi se odpověď 200 OK dostane až posléze. Stejný případ nastává i poté, co klient odešle žádost BYE, na kterou Asterisk odpoví pomocí 200 OK ještě dříve, než je žádost doručena druhému účastníkovi hovoru. Posledním na první pohled patrným rozdílem je průchod mediálního toku skrze Asterisk. Již jsme si uvedli, že tato vlastnost je pro B2BUA režim typická.



Obrázek 4.3: Vztah mezi konfiguračními soubory pjsip.conf a extensions.conf [23]

Obrázek č.4.3 ukazuje vzájemnou provázanost mezi konfiguračním souborem pjsip.conf a Dialplanem reprezentovaného souborem extensions.conf. Je logické, že u každého hovoru je nejdříve ověřena jeho identita, tzn. jaký protokol byl použit (SIP, H.323..) a tedy jakému konfiguračnímu souboru Asterisku má být hovor předán. V ukázce se jedná o hovor využívající signalizační protokol SIP, jehož konfigurace je obsažena v souboru pjsip.conf. SIP účtu volajícího je uvnitř tohoto souboru přidělen kontext, který určuje, jaká sekce konfiguračního souboru extensions.conf má být pro zpracování hovoru použita. V našem případě se jedná o kontext example, který říká, že bylo-li vytočeno číslo 123, má se použít aplikace Dial() pro SIP účet Alice. Následně je znovu použit konfigurační soubor pjsip.conf, jehož úkolem je v tuto chvilí vykonat potřebné kroky pro to, aby mohla být žádost o sestavení hovoru předána Alici. [23]

Kapitola 5

Distribuce zátěže

Distribuce zátěže představuje důležitou funkci sítě, s jejíž pomocí jsme schopnosti zajistit vysokou dostupnost a kvalitu nabízených služeb. Tuto funkci obvykle plní takzvaný reverzní proxy server. Rozdíl mezi klasickým a reverzním proxy serverem je zejména v jeho umístění v síti. Zatímco klasický proxy server je umístěn co nejblíže uživateli a zajišťuje služby jako například autentizaci, revezní proxy server je umístěn bezprostředně před aplikační servery. Jelikož v této pozici tvoří reverzní proxy server rozhraní mezi veřejnou sítí a sítí poskytovatele, přes které prochází všechny požadavky, je nezbytné, aby byl výpočetní výkon tohoto prvku dostatečně dimenzován. [26]

Mezi další fukce, které může reverzní proxy server plnit lze zařadit například skrytí vnitřní topologie sítě. Toho je docíleno již samotným principem fungování, protože jsou veškeré příchozí požadavky zákazníků směrovány na reverzní proxy server, který až posléze předá žádost jednomu z aplikačních serverů. Pro uživatele se tedy reverzní proxy jeví jako server zajišťující onu službu, čímž je docíleno skrytí identit aplikačních serverů. Samotný distribuční prvek proto do jisté míry plní roli firewallu a je tedy důležitým bezpečnostním elementem sítě poskytovatele. [26]

V této diplomové práci bude roli reverzního SIP Proxy serveru s funkcí distribuce zátěže zajišťovat Kamailio. V dalších částech kapitoly č.5 se seznámíme se dvěma způsoby, jak spolu s Kamailiem distribuovat zátěž, přičemž jeden z nich si navrhneme sami.

5.1 Kamailio modul pro distribuci zátěže

Pro funkci distribuce zátěže je pro Kamailio přítomen modul s názvem *Dispatcher*. Tento výpočetně nenáročný modul umožňuje distribuovat vstupní požadavky mezi skupinu aplikačních serverů pomocí některého z nabízených algoritmů. Typ použitého algoritmu určuje jak, a na základě jakých parametrů budou požadavky přerozdělovány. V této části si popíšeme základní vlastnosti a parametry, které jsou důležité pro chod tohoto modulu.

5.1.1 Definování aplikačních serverů

Dispatcher modul umožňuje definovat množinu aplikačních serverů dvěma způsoby, a to za pomocí textového souboru dispatcher.list nebo s využitím databáze. Všechny definice parametrů modulu probíhají uvnitř konfiguračního souboru *kamailio.cfg*. Definování cesty k souboru dispatcher.list uvnitř sekce pro nastavení modulů vypadá následovně:

```
modparam("dispatcher", "list_file", "/etc/kamailio/dispatcher.list");
```

Každý řádek v souboru dispatcher.list odpovídá jednomu záznamu v našem případě jednomu konkrétnímu aplikačnímu serveru. Struktura zápisu a jeho příklad vypadá takto:

```
setid        destination      flags       priority            attrs
  1     sip:192.168.0.1:5060    0             0      duid=asterisk1;maxload=500
```

Pole *setid* udává příslušnost daného záznamu k určité skupině, ze které jsou nasledně aplikační servery vybírány dle použitého distribučního algoritmu. Další pole obsahuje SIP URI destinace. Toto pole spolu se *setid* tvoří povinná pole záznamu. Následuje bitové pole *flags*, které určuje mód použití daného záznamu a dále způsob zasílání takzvaných *keepalive* zpráv. Toto pole může nabývat hodnot 0, 1, 2, 4, 8 nebo 16. Celočíselné pole *priority* slouží pro úvodní seřazení cílů uvnitř listu. Závěrečné pole s dodatečnými atributy obsahuje doplňkové informace jako například atribut *maxload*, který udává maximální počet souběžných hovorů pro daný cíl, dále atribut *weight* udávající váhu/významnost cíle, čehož se využívá primárně při nasazení distribučního algoritmu založeného právě na váze jednotlivých cílů, tzv. *weight-based* algoritmus. Existují ještě další atributy, jejichž význam ale není pro účely této práce podstatný, a proto zde nebudou zmíněny. [27]

Pokud chceme pro definování aplikačních serverů zvolit variantu s využitím databáze, je nutné v prvním kroku nastavit URL databáze, název databáze, a pokud pro definovaní jednotlivých polí používáme jiné názvy než ty uvedené v ukázce souboru *dispatcher.list*, je potřeba vlastní názvy k odpovídajícím polím přiřadit. [27]

5.1.2 Prověřování dostupnosti cílů

Jakmile máme definováno umístění cílů a nastaveny potřebné parametry je důležité průběžně zjišťovat, zda jsou dané cílé dostupné. Pokud bychom tak neučinili, mohlo by se stát, že by řada hovorů směřovala do destinace, která v tu chvíli není k dispozici, ať už z důvodu selhaní sítě, či samotné aplikace. Pro účely zjišťování dostupnosti cílů se používají keepalive zprávy, což jsou jednoduše periodicky zasíláne SIP žádosti vybraného typu (obvykle SIP OPTIONS). Definování, která ze SIP žádostí bude použita pro zasílání keepalive zpráv vypadá následovně:

```
modparam("dispatcher", "ds_ping_method", "OPTIONS");
```

Lze definovat i interval mezi jednotlivými keepalive zprávami:

```
modparam("dispatcher", "ds_ping_interval", 30);
```

Pokud cíl na dané zprávy neodpovídá, je vhodné nastavit počet po sobě jdoucích nezodpovězených zpráv, při jehož překročení je cíl označen jako neaktivní.

```
modparam("dispatcher", "ds_probing_threshold", 10);
```

Pokud je cíl označen jako neaktivní a je nastaven mód, u něhož jsou testovány i neaktivní cíle, je vhodné uvést počet zodpovězených po sobě jdoucích zpráv, po kterém je stav změněn na aktivní:

```
modparam("dispatcher", "ds_inactive_threshold", 5);
```

Každý cíl je charakterizován určitým příznakem, který popisuje jeho stav a dále také to, zda je prověřována jeho dostupnost či ne. Stavy mohou být následující:

- **A (Active)** - cíl je aktivní, tzn. odpovídá na keepalive zprávy.
- **I (Inactive)** - cíl je neaktivní, tzn. neodpovídá na keepalive zprávy.
- **D (Disabled)** - cíl je zablokován administrátorem.
- **T (Trying)** - dočasný stav, ve kterém se cíl nachází pokud neodpovídá na keepalive zprávy, ale doposud nepřekročil mez definovanou parametrem *ds_probing_threshold*.

Způsob značení módu pro testování dostupnosti má následující charakter:

- **P (Probing)** - cíl je průběžně testován zasíláním keepalive zpráv.
- **X (Not Probing)** - dostupnost cíle není prověřována keepalive zprávami. Očekáva se proto, že stav cíle je neměnný. [27]

Výsledná charakteristika některého z cílů může mít tedy podobu například AP (Active - Probing) nebo DX (Disabled - Not Probing). Dispatcher modul dále umožňuje nastavit, na které cíle mají být keepalive zprávy zasílány. Hodnota parametru může nabývat hodnot 0-3 a má tuto podobu:

```
modparam("dispatcher", "ds_probing_mode", 1);
```

- **Hodnota 0** - Pouze cíle v módu Probing jsou otestovány. (Hodnota pole *flags* v souboru dispatcher.list musí být nastavena na hodnotu 8). Poté, co je daný cíl otestován, je stav Probing smazán. Cíl je tedy otestován pouze jednou, a to při spuštění služby nebo po restartování Dispatcher modulu.
- **Hodnota 1** - Všechny cíle jsou testovány. Pokud není keepalive zpráva zodpovězena změní se stav daného cíle na Trying.

- **Hodnota 2** - Pouze cíle ve stavu Inactive a v módu Probing jsou testovány.
- **Hodnota 3** - Všechny cíle v módu Probing jsou testovány, ale stav cíle je neměnný.

Jako dodatečný parametr, který lze u jednotlivých cílu sledovat je doba odezvy na keepalive zprávy. Administrátor tak má přehled o průměrné/maximální odezvě či množství nezodpovězených keepalive zpráv. Spuštění této funkce se provede následovně: [27]

```
modparam("dispatcher", "ds_ping_latency_stats", 1);
```

5.1.3 Funkce modulu

Každý modul Kamailia nabízí sadu funkcí, jejichž činnost se odvíjí od zaměření konkrétního modulu. Dané funkce se poté volají v odpovídajících směrovacích blocích uvnitř hlavního konfiguračního souboru *kamailio.cfg*. Nejinak je tomu i u modulu *dispatcher.so*, jehož nabídka funkcí přesahuje rámec této práce, a proto zde budou zmíněny jen ty subjektivně nejvýznamnější.

První funkcí, kterou je dobré zmínit je funkce *ds_select_dst()*. Úkolem této funkce je vybrat cíl, na který bude žádost následně zaslána. Funkce vrátí hodnotu TRUE pokud je cíl vybrán a nastaven, v opačném případě vrátí hodnotu FALSE. Vybraný cíl je vložen do pole *dst_uri*, na základě kterého je žádost směrována. Vybraná hodnota není viditelná v žádném poli SIP hlavičky. Funkce obsahuje celkem tři parametry, přičemž první dva jsou povinné. Výsledná struktura funkce má nasledující podobu: [27]

```
ds_select_dst(set, alg[, limit]);
```

Parametr *set* odpovídá hodnotě pole *setid* ze souboru *dispatcher.list*. Tato hodnota reprezentuje skupinu cílů, ze kterých bude daný cíl vybrán na základě použitého algoritmu. Použitý algoritmus je definován ve druhém parametru pomocí jeho celočíselného identifikátoru. Algoritmů je k dispozici celkem 13, přičemž výčet těch základních včetně krátkého popisu je uveden na následujícím seznamu.

- **Algoritmus č. 0** [**hash over call-id**] - na základě vypočítané hash hodnoty pole call-id je určen cíl.
- **Algoritmus č. 4** [**round robin**] - algoritmus cyklicky přerozděluje jednotlivé hovory mezi dostupné cíle.
- **Algoritmus č. 6** [**random**] - cíl je vybrán náhodně na základě funce rand().
- **Algoritmus č. 12** [**dispatch to all destinations**] - algoritmus, který pošle danou žádost všem cílům se stejnou hodnotou pole *setid* zároveň. Vytváří se tak paralelní větvení hovorů.

Poslední parametr této funkce nese název *limit* a slouží pro definování maximálního počtu položek, které budou uloženy do seznamu XAVP (extended Atribute Value Pairs), což je struktura pro ukládání skupiny hodnot. XAVP v tomto případě obsahují informace o dalších cílech pro případ selhání nebo nedostupnosti některého/některých z nich. [11, 27]

Na prvně zmíněnou funkci volně navazuje funkce *ds_next_dst()*, jejíž náplní je vybrat další cíl z odpovídajícího XAVP seznamu a nastavit jeho cílovou adresu do pole *dst_uri*. Tato bezparametrová funkce se používá především ve směrovacích blocích *failure_route* v situaci, kdy se po volání funkce *ds_select_dst* nepodařilo sestavit hovor prostřednictvím vybraného cíle, a je proto zapotřebí vybrat jiný cíl. [27]

Dalšími funkcemi jsou funkce *ds_select_domain(set, alg[, limit])* a *ds_next_domain()*, které jsou velmi podobné výše uvedeným funkcím, ale s tím rozdílem, že u nich dochází k přepisu request-URI, a to konkrétně částí *host* a *port*. [27]

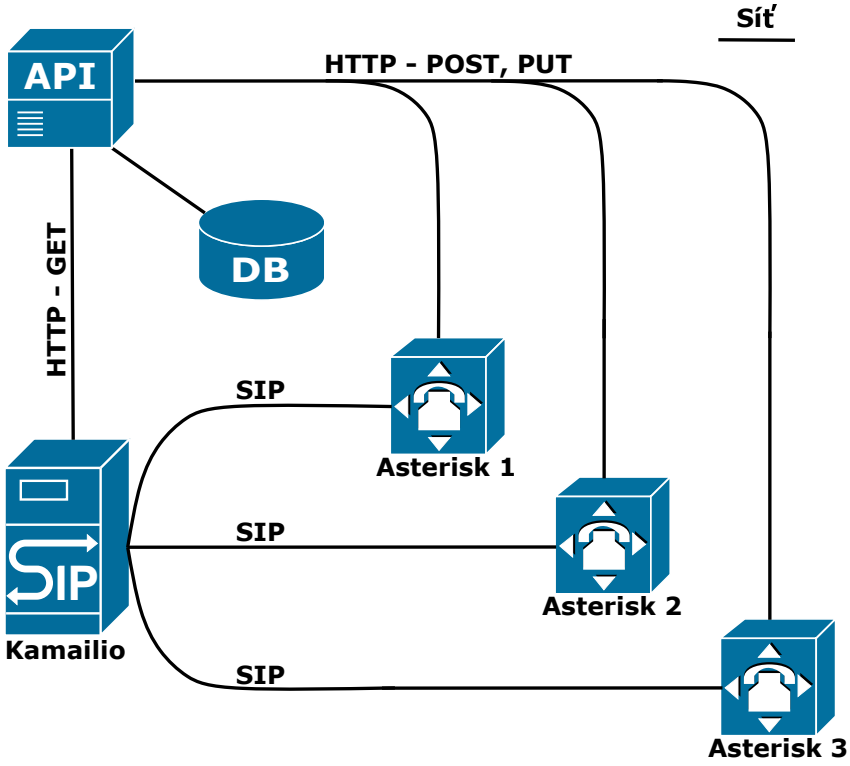
Závěrem lze říci, že modul *dispatcher.so* je komplexní nástroj vhodný pro většinu aplikací, ve kterých je nezbytné distribuovat zátěž v podobě hovorů. Jeho hlavní nevýhodou je však neznalost skutečného stavu vytížení jednotlivých cílu v našem případě aplikačních serverů, na základě kterého by se rozhodoval, na jaký cíl v danou chvíli hovor poslat, aby byl systém maximálně efektivní. V další části této kapitoly se proto pokusíme navrhnout alternativu Dispatcher modulu.

5.2 Návrh vlastního řešení distribuce zátěže

V této části práce se seznámíme s vlastním návrhem distribučního systému, ve kterém bude proces přerozdělování zátěže v podobě hovorů postavený na aktuálním vytížení CPU jednotlivých aplikačních serverů, jejichž role bude v našem případě plněna Asterisk servery. Námi požadovaná funkcionalita však vyžaduje vhodné propojení řady komponent do jednoho funkčního celku. S prvky, ze kterých bude náš systém tvořen, se proto v této části kapitoly seznámíme.

V úvodu je dobré si uvědomit, co je vlastně klíčovým aspektem našeho systému. Shodneme se, že jim je nepochybně ono informování distribučního prvku o aktuálním vytížení procesoru jednotlivými aplikačními servery. Je proto nezbytné položit si řadu otázek. Jak budou data sdílená? Kam se budou data ukládat? Jak k nim bude distribuční prvek přistupovat? Jako poměrně moderní a pro tuto situaci vhodné řešení se jeví vytvoření jednoduchého webové API (Application Programming Interface), v našem případě API typu REST (Representational State Transfer), které bude hrát roli prostředníka při výměně dat mezi aplikačními servery a serverem distribučním. REST API bude dále propojeno s databází, kde budou data nesoucí informaci o aktuálním vytížení procesoru uložena. Ve chvíli, kdy do systému vstoupí nový požadavek na sestavení spojení budou aktuální informace připraveny pro distribuční prvek, který na jejich základě určí, na jaký aplikační server hovor směrovat.

Na níže uvedeném obrázku je znázorněn základní návrh systému pro distribuci zátěže. Hlavní komponenty návrhu, které dohromady zajišťují požadovanou funkci systému, budou blíže popsány v následujících částech této kapitoly.



Obrázek 5.1: Schéma základního návrhu systému pro distribuci zátěže

5.2.1 REST API

Na úvod si ve stručnosti popišme, co je to API. API představuje rozhraní charakterizované sadou pravidel, funkcí, protokolů a knihoven, jenž dovoluje vzájemnou interakci mezi dvěmi platformami, mezi kterými díky tomu může probíhat výměna dat. [28]

REST je architektura rozhraní, která říká, jak má API vypadat a jak se má chovat. Jedná se o rozhraní využívané pro pohodlný přístup ke zdrojům (resources). Zdroji máme na mysli data nebo stavy aplikace (pokud lze stav aplikace vyjádřit daty). REST je tedy na rozdíl od XML-RPC či SOAP (Simple Object Access Protocol) orientován datově a nikoli procedurálně. Termín REST je úzce spojen s webovým protokolem HTTP, (Hypertext Transfer Protocol) což není nic překvapivého, protože jeho autorem je Roy Fielding, tedy jeden z autorů právě onoho HTTP protokolu. Nutno však říct, že REST není striktně vázán pouze na HTTP, ale díky obrovské rozšířenosti tohoto protokolu tomu tak v drtivé většině případů je. [29, 30]

Způsob, jak přistupovat ke zdrojům (datům) je definován pomocí takzvaných CRUD operací. Jedna se o čtyři základní úkony, které lze nad zdroji vykonat. Pro mapování těchto operací se využívají HTTP žádosti. Jejich přiřazení k jednotlivým CRUD operacím je uvedeno níže.

Operace	**HTTP žádost**	**Popis operace**
Create (vytvořit)	POST	Vytvoření nového unikátního zdroje
Read (číst)	GET	Čtení zdroje nebo skupiny zdrojů
Update (aktualizovat)	PUT	Aktualizace zdroje
Delete (smazat)	DELETE	Smazání zdroje

Tabulka 5.1: Mapování CRUD operací na HTTP žádosti [29]

Zdroje, ke kterým pomocí výše zmíněnch HTTP žádostí přistupujeme jsou identifikovány pomocí URL (Uniform Resource Locator). Představme si, že máme k dispozici doménu s názvem *http://server.com*, na které provozujeme REST API se skupinou zdrojů *asterisk*. URL, která tuto skupinu identifikuje, má podobu *http://server.com/api/asterisk*. V níže uvedené tabulce je pro názornost uveden vliv jednotlivých HTTP žádostí aplikovaných na naši ukázkovou skupinu zdrojů. [31]

Žádost	**Výsledek žádosti**
POST */api/asterisk*	Vytvoří nový unikátní zdroj (asterisk) ve skupině zdrojů asterisk
GET */api/asterisk/1*	Umožňuje čtení konkrétního zdroje (asterisku) s ID 1
GET */api/asterisk*	Dovoluje čtení celé skupiny zdrojů, tzn. vrátí seznam asterisků
PUT */api/asterisk/1*	Umožňuje aktualizovat konkrétní zdroj (asterisk) s ID 1
DELETE */api/asterisk/1*	Smaže konkrétní zdroj (asterisk) s ID 1

Tabulka 5.2: Aplikování určitých HTTP žádostí na skupinu zdrojů [29]

Pro implementaci vlastního REST API jsem se na základě konzultace s vedoucím této práce rozhodl použít framework pro programovací jazyk Python nesoucí název Flask. Jedná se o nenáročný framework pro rychlou a snadnou tvorbu webových aplikací bez nutnosti využívat jakékoliv další nástroje. Komplexnější alternativou Flasku je framework Django, který však nevyniká takovou jednoduchostí nasazení a z tohoto důvodu bude v této práci upřednostněn právě Flask. Tvorba REST API bude součástí další kapitoly, která se bude zabývat výhradně implementací a konfigurací celého systému. [32, 33]

5.2.2 Databáze

Pro ukládání a přístup k datům v našem systému jsem se rozhodl použít relační databázi SQLite. Tato databáze je velmi nenáročná a na rozdíl od plnohodných databází jako MySQL nebo PostgreSQL, které jsou spuštěny jako samostatné procesy je SQLite pouze malá knihovna připojená k určité aplikaci. Samotná databáze je poté uložena ve formě souboru, s nimž je možné komunikovat

prostřednictvím jazyka SQL (Structured Query Language). V souhrnu pak tyto vlastnosti umožňují snadnou přenositelnost databáze spolu s aplikací, se kterou je svázaná. [34, 35]

SQLite nepředstavuje náhradu velkých robustních databází, avšak pro malé a střední aplikace, mezi které se řadí i ta naše, je více než dostačující. Hlavní omezení SQLite spočívá v celkovém uzamčení databáze po dobu vykonávání operace zápisu. Ostatní procesy mohou v daný okamžik z databáze pouze číst, ale jen původní data. K novým datům lze přistupovat až po ukončení transakce zabývající se zápisem do databáze. Z tohoto důvodu není SQLite databáze vhodná pro případ, kdy do ní bude zapisovat velké množství uživatelů souběžně. Dále nenabízí například řízení přístupových práv či provozování databáze na sdíleném úložišti. Omezení, která se týkají SQLITE databáze je samozřejmě více. Pro nás je však důležité, že žádná z těchto omezení nejsou pro naši aplikaci nikterak limitující, jelikož do databáze bude zapisovat pouze nevelký počet aplikačních serverů (Asterisky) a jeden prvek (Kamailio) z ní bude číst. [34, 36]

5.2.3 Způsob odesílání dat ze strany aplikačních serverů

Abychom mohli Kamailio informovat o aktuálním vytížení jednotlivých aplikačních serverů, je nezbytné zasílat potřebné informace v podobě odpovídajících HTTP žádostí na naše REST API. Jedním z oblíbených a dostupných nástrojů pro posílání HTTP žádostí je například Python knihovna *requests*, kterou jsem se pro účely této práce rozhodl použít.

V našem systému se na straně Asterisků spokojíme se dvěma typy HTTP žádostí, a to konkrétně žádostí POST (vytvoření zdroje) a žádostí PUT (aktualizování stávajícího zdroje). Knihovna umožňuje definovat různé parametry HTTP hlaviček či definování vlastního obsahu, který bude v žádosti obsažen. Jako formát posílaných dat se velice často využívá JSON (JavaScript Object Notation). JSON je textový formát založený nad podmnožině programovacího jazyka JavaScript. Pro nás je ve spojitosti s tímto formátem důležitý způsob zápisu objektů. Objekt je v JSON definován jako neuspořádaná množina párů *název:hodnota*. Na počátku objektu je vždy levá složená závorka a na konci pravá složená závorka. Jednotlivé páry jsou odděleny čárkou a v rámci páru se název od hodnoty odděluje dvojtečkou. Způsob zápisu objektu pomocí JSON může vypadat následovně: [37, 38]

```
{
"cpu_load" : 47,
"id" : 1
}
```

Tvorba samotné HTTP žádosti pomocí knihovny *requests* je velmi jednoduchá. Jediný povinný parametr, který musí být v žádosti obsažen je URL zdroje. Ukázka jednoduché POST žádosti, která bude pro účely našeho REST API plně dostačující, může vypadat například takto: [39]

```
requests.post('http://server.com/api/asterisk', json={'cpu_load':load})
```

Obdobně by vypadala i žádost PUT s tím rozdílem, že by URL obsahovalo i konkrétní ID, které bylo přiřazeno zdroji (Asterisku) na základě žádosti POST, viz následující ukázka.

```
requests.put('http://server.com/api/asterisk/1', json={'cpu_load':load})
```

Jako zdroj informací o aktuálním vytížení procesoru nám poslouží knihovna *psutil*, jejíž funkce *psutil.cpu_percent()* nám vrátí aktuální vytížení procesoru v procentech.

5.2.4 Způsob získávání dat ze strany Kamailia

Pro přístup k datům z Kamailio serveru je nezbytné zvolit odlišný způsob v porovnání s odesíláním dat z Asterisk serverů. V tomto případě potřebujeme zajistit, aby byla získaná data k dispozici uvnitř hlavního konfiguračního souboru a mohlo tak být na jejich základě rozhodnuto o směrování jednotlivých požadavků. Pro řešení tohoto problému jsou k dispozici celkem tři varianty. V první variantě by Kamailio přistupovalo přímo k datům uloženým v databázi a obcházelo by tak samotné REST API. Tento způsob je však neefektivní a degraduje základní význam našeho REST API.

Druhá možnost spočívá ve využití Kamailio modulu s názvem *app_python* případně modulu *app_python3* (pro Python 3), které umožňují spouštění Python skriptů přímo z konfiguračního souboru Kamailia. Tím bychom mohli definovat skript využívající knihovnu *requests*, skrze který bychom pomocí žádostí HTTP GET přistupovali k datům uloženým v databázi. Výstup skriptu by nám následně určil aplikační server, na který žádost směrovat.

Poslední a z mého pohledu nejefektivnější varianta, kterou jsem se v rámci této práce rozhodl aplikovat, je postavena na Kamailio modulu *http_client*. Tento modul implementuje funkce HTTP protokolu s využitím *libcurl* knihovny. Umožňuje tak získat data ze vzdálených HTTP serverů (žádost GET) či odeslat data na některý HTTP server (žádost POST). [40]

Nyní si ukážeme pár základních parametrů tohoto modulu, pomocí kterých lze ovlivnit jeho chování. První z nich je například nastavení proxy serveru, přes který bude všechen HTTP provoz procházet:

```
modparam("http_client", "httpproxy", "http://proxyserver.example.com");
```

V některých případech je vhodné definovat maximální dobu čekání (v sekundách) na odpověď ze serveru. Ta lze nastavit pomocí následujícího parametru modulu (hodnota je typu *int*):

```
modparam("http_client", "connection_timeout", 1);
```

Další velká skupina parametrů tohoto modulu se zabývá zabezpečenou formou komunikace. Lze definovat umístění klientského klíče a certifikátu či certifikátu certifikační autority. Dále můžeme nastavit seznam povolených typů šifrování nebo preferovanou verzi TLS. Pro naše účely je ale nejdůležitejší definovat spojení na naše REST API. Modul *http_client* nabízí celkem dva způsoby jak

formulovat spojení a jeho vlastnosti. První z nich je pomocí externího souboru. Tato možnost je vhodná zejména pokud spojení vyžaduje nastavení velkého množství parametrů, protože uvnitř hlavního konfiguračního souboru by se takové nastavení jevilo jako nepřehledné. Pro definici vlastností našeho spojení si bohatě vystačíme s parametrem modulu *httpcon* (druhý způsob). Struktura tohoto parametru a jeho ukázka vypadá takto: [40]

```
modparam("http_client", "httpcon",  <connection-name>=><schema>:
//[<username>:<password>@]<hostname/address>[;param=value]);
modparam("http_client", "httpcon", "restapi=>http://server.com/api;timeout=1");
```

Výše uvedená struktura zápisu nevyžaduje definování všech polí. Pokud spojení nepotřebuje žádné rozšiřující nastavení, stačí definovat pouze pole *connection-name* a *schema*. Tímto jsme se v krátkosti seznámili se základními parametry *http_client* modulu. Z pohledu nabízených funkcí je pro náš systém vhodná funkce s názvem *http_connect*. Tato funkce umožňuje posílat GET a POST žádosti a ukládat data obsažená v odpovědi do pseudo proměnných. Struktura zápisu této funkce a příklad použití má tuto podobu: [40]

```
http_connect(connection, url, [content_type, data,] result);
http_connect("restapi", "/asterisks", "$var(result)");
```

Pole *connection* obsahuje název HTTP spojení nadefinovaného v parametru modulu *httpcon*, viz předchozí ukázka. Pole *URL* obsahuje část URL, která má být přidána k URL definované v poli *schema* v parametru modulu *httpcon*. Pole *content_type, data* se definují pouze v případě zasílání žádosti POST, proto je v našem případě nepotřebujeme. Poslední pole s názvem *result* obsahuje název pseudo proměnné, do které budou uložena data obsažená v odpovědi na danou žádost. [40]

Z mého úhlu pohledu se nasazení modulu *http_client* jeví jako nejvhodnější volba, a to z toho důvodu, že se využívá pouze funkcionalita Kamailia bez nutnosti volat externí aplikace, jako je například Python. Navíc se jedná o nenáročný modul, jehož nastavení je přehledné, snadné a pro naše účely zcela dostačující.

5.2.5 Způsob fungování navrhnutého systému a jeho možné alternativy

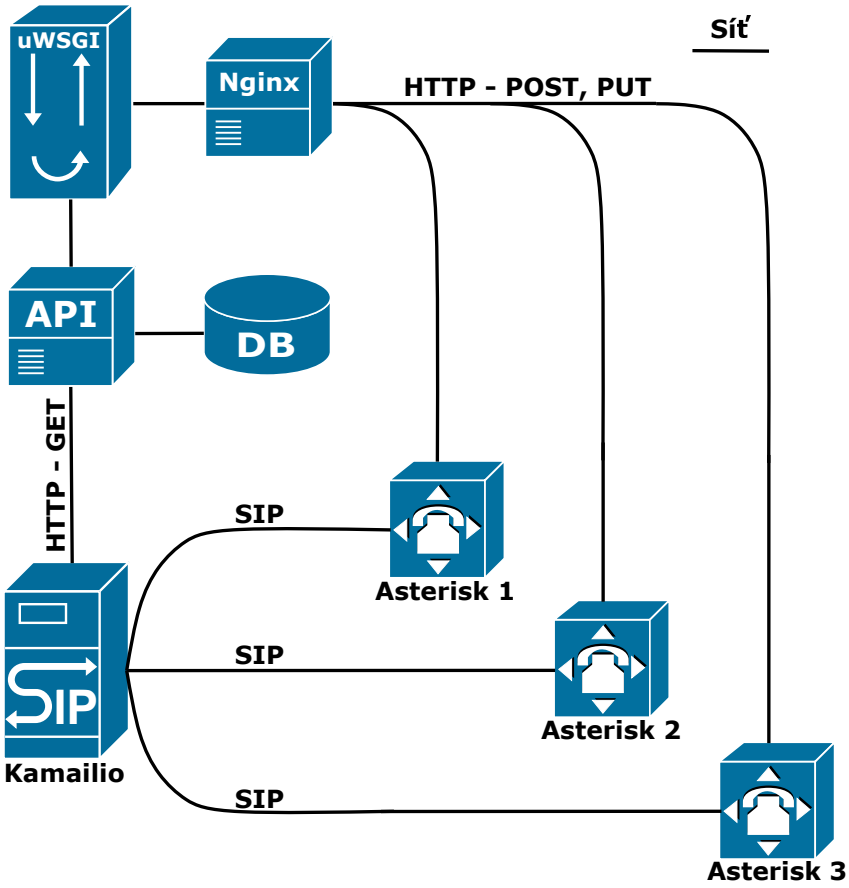
Nyní máme popsány všechny důležité komponenty našeho systému, a tak můžeme přejít na samotný popis toho, jak bude základní návrh systému pro distribuci zátěže fungovat. Informování o aktuálním stavu vytížení CPU ze strany Asterisků může probíhat periodicky. Jelikož chceme minimalizovat množství přenášených informací, stačí nám k popsání jednotlivých aplikačních serverů pouze jejich ID a vytížení CPU. Tyto informace budou uloženy skrze REST API do databáze. Jakmile na Kamailio dorazí žádost o sestavení spojení v podobě SIP INVITE, dotáže se Kamailio za pomoci *http_client* modulu našeho REST API, které mu vrátí aktuální informace o všech Asterisk serverech.

Kamailio si z příchozí zprávy vyparsuje potřebné informace, na jejichž základě následně určí cíl, který je v době příchodu žádosti do systému nejméně vytížen.

Takto navržený systém je schopen reagovat na nerovnoměrnosti ve vytížení jednotlivých aplikačních serverů, a to nehledě na poskytovanou službu (IVR, voicemail, konference, call centrum). Jednotný přístup pro sdílení/čtení informací pomocí aplikačních serverů a Kamailia dělá z tohoto systému poměrně elegantní a snadno distribuovatelné řešení. Je zde však řada vlastností, které dovolují kvalitu a chování systému značně vylepšit. Jedna z nich se týká periodicity s jakou jsou data uvnitř databáze aktualizována. Pokud bude mít perioda pro zasílání informačních zpráv délku například jednu sekundu, může se tento interval zdát v době, kdy jsou aplikační servery nevytížené jako zbytečně krátký. Naopak v případě velkého provozu je tato doba příliš dlouhá. To může způsobit shlukovité nárusty ve vytížení jednotlivých aplikačních serverů, protože během jedné sekundy může do systému dorazit klidně několik desítek žádostí, které budou v tomto případě všechny zaslány na totožný v tu dobu nejméně vytížený cíl. Interval zásílaní informačních zpráv měnící se podle aktuálního stavu vytížení by tento problém pomohl částečně eliminovat.

Nastíněný problém by šlo vyřešit i celkovou změnou principu fungování našeho systému a to tak, že by aplikační servery obsahovaly naprosto jednoduché REST API, skrze které by Kamailio přistupovalo k informacím o stavu vytížení až ve chvíli, kdy by do systému vstoupila nová SIP žádost. Pro každou SIP žádost bychom tak měli naprosto aktuální informace. V případě nulového vstupního provozu by dále nedocházelo k zasílání žádných, v tu chvíli zbytečných informačních HTTP zpráv. Daní za tuto funkcionalitu by však byl zvýšený síťový provoz v době s velkým vstupním SIP provozem a zejména nárust zpoždění při navazování SIP dialogu. Proto se pro tuto práci ukazuje jako vhodnější dříve zmíněný způsob s proměnným intervalem pro zasílání informačních zpráv.

Pokud bychom chtěli skutečně věrohodně napodobit produkční nasazení takového systému, bylo by nezbytné nasadit k provozování naší Flask aplikace výkonný webový server, jakým je například Nginx či Apache. Nginx/Apache však není schopen komunikovat přímo s Pythonem. K tomuto účelu slouží uWSGI (Web Server Gateway Interface), což je tzv. kontejner aplikačního serveru, který se chová jako prostředník mezi žádostmi přicházejícími z webového serveru a samotnou aplikací/frameworkem. Podoba komunikace mezi uWSGI a Flask frameworkem je dána Python specifikací WSGI, což je jednoduše sada definic, jejichž významem je standardizace a zjednodušení této komunikace. Pro komunikaci mezi uWSGI a webovým serverem (např. Nginx) se velmi často používá binární protokol s názvem uwsgi, který je implementován uWSGI serverem. Hlavním cílem práce proto bude provozovat naše RESP API společně s uWSGI a výkonným webovým serverem, abychom tak věrohodně imitovali nasazení naší aplikace v reálném prostředí. Pro zasílání informačních zpráv z aplikačních serverů se dále pokusíme nasadit variantu s proměnným intervalem, čímž bude zajištěno efektivnější využití přenosové kapacity. Cílená podoba systému pro distribuci zátěže by tedy v závěru práce měla odpovídat schématu na obrázku 5.2. [41, 42]



Obrázek 5.2: Schéma finálního návrhu systému pro distribuci zátěže

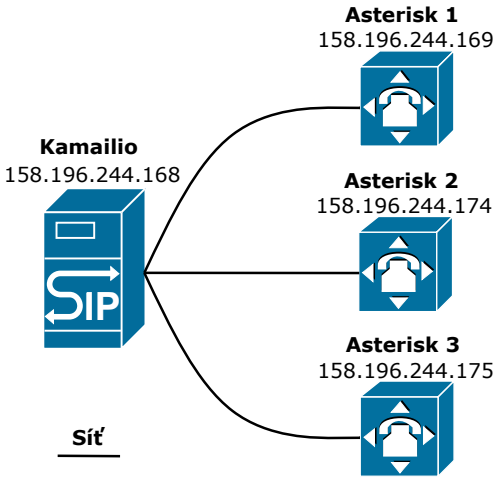
V další části této práce se přesuneme k praktické implementaci systému pro distribuci zátěže s cílem upravit podobu a chování našeho systému do takové formy, ve které by byl maximálně stabilní a rozprostíral zátěž vhodným způsobem.

Kapitola 6

Instalace a konfigurace

V dosavadních teoretických kapitolách jsme si popsali základní prvky a technologie, které budou nezbytné pro chod našeho systému. V této části práce se je budeme snažit prakticky propojit do jednoho funkčního celku. Kapitola bude rozdělena celkem na tři sekce. V první sekci provedeme základní konfiguraci klíčových prvků našeho systému, to znamená Kamailia a Asterisk serverů. V druhé sekci si ukážeme základní konfiguraci pro distribuci zátěže s využitím *Dispatcher* modulu. Hlavní závěrečná pasáž bude věnována implementaci a konfiguraci vlastního dynamického systému pro distribuci zátěže.

Na úvod je ale vhodné seznámit se s naší testovací topologií, ve které se bude vše důležité odehrávat. Pro účely této diplomové práce mi byly vedoucím práce vyhrazeny celkem čtyři virtuální servery. První z nich je určen výhradně pro Kamailio a následně i chod našeho REST API, uWSGI a webového serveru Nginx. Zbývající tři servery budou využity pro provozování Asterisk serverů. Podoba testovací topologie včetně IPv4 adresace a základní technické parametry virtuálních strojů jsou následující:



Obrázek 6.1: Schéma testovací topologie

- **Operační systém** - Ubuntu 20.04.1 LTS 64-bit (Focal Fossa)
- **Procesor** - Intel Xeon Silver 4116 2.10 GHz (Kamailio - 2 jádra, Asterisk servery - 1 jádro)
- **Operační paměť** - 1 GB DDR4 2400 MHz
- **Kapacita uložiště** - 20 GB
- **Síťové rozhraní** - VMXNET3 Ethernet Controller 10 Gbit/s

6.1 Instalace a úvodní konfigurace Kamailia a Asterisk serverů

Instalaci Kamailia i Asterisku lze uskutečnit celkem dvěmi způsoby. U prvního z nich je nutné provést kompilaci ze zdrojových kódů. Druhý a v této práci aplikovaný způsob využívá repozitáře, ze kterých je možné daný software stáhnout a nainstalovat ve formě balíčku.

6.1.1 Kamailio

Jelikož chceme z repozitářů získat jednu z posledních verzí Kamailia, je nezbytné informovat operační systém o umístění tohoto balíčku, a to přidáním záznamu do souboru nacházejícího se v */etc/apt/source.list*. Tento soubor otevřeme v libovolném textovém editoru a přidáme do něho následující položky:

```
deb     http://deb.kamailio.org/kamailio54 focal main
deb-src http://deb.kamailio.org/kamailio54 focal main
```

Abychom mohli z tohoto zdroje balíček stáhnout a nainstalovat, je dále nezbytné přidat takzvaný GPG (GNU Privacy Guard) klíč, který rovněž slouží pro zvyšení důvěryhodnosti onoho zdroje. Přidání klíče zajistíme následujícím příkazem:

```
wget -O- http://deb.kamailio.org/kamailiodebkey.gpg | sudo apt-key add -
```

Nyní můžeme přistoupit k samotné instalaci. V prvním kroku aktualizujeme seznam balčků, což je nezbytné po změnách, které jsme provedli v souboru */etc/apt/source.list*. Následně provedeme samotnou instalaci Kamailia.

```
sudo apt update
sudo apt install kamailio
```

V běžných případech se spolu s Kamailiem instalují i MySQL moduly a MySQL server, které slouží zejména pro ukládání vytvořených uživatelů a jejich hesel. V naší práci však tyto funkce nepotřebujeme, jelikož naším cílem je nakonfigurovat reverzní SIP Proxy, která obvykle autentizaci uživatelů neprovádí. Výše uvedenými příkazy jsme nainstalovali Kamailio ve verzi 5.4.4 a můžeme proto přejít k jeho základní konfiguraci. V prvním kroku nastavíme pole *SIP_DOMAIN* nacházející se v souboru */etc/kamailio/kamctrlc* a to tak, aby obsahovalo IP adresu našeho serveru (158.196.244.168). Dále se přesuneme do hlavního konfiguračního souboru Kamailia uloženého v */etc/kamailio/kamailio.cfg*, ve kterém nastavíme následující položky:

```
listen = udp:158.196.244.168:5060
auto_aliases = no
```

Položka *listen* udává transportní protokol, IP adresu a port, na kterém bude Kamailio naslouchat příchozí SIP komunikaci. Pole *auto_aliases* definuje, zda má Kamailio po spuštění provést reverzní DNS překlad, při kterém se snaží k adrese definované v poli *listen* přiřadit doménové jméno. Tuto funkci ale nechceme, a proto jí přiřadíme hodnotu *no*. [14, 43, 44]

V posledním kroku odkomentujeme pole *RUN_KAMAILIO=yes* nacházející se v souboru */etc/default/kamailio*. Tím umožníme spouštěcímu skriptu spoustit Kamailio. Následujícím příkazem celou službu spustíme:

```
sudo systemctl start kamailio
```

V tuto chvíli jsme s úvodní konfigurací Kamailia hotoví, můžeme se tak přesunout k další části této sekce, která se bude zabývat instalací a konfigurací Asterisku.

6.1.2 Asterisk

V případě instalace Asterisk serverů použijeme podobný postup jako v případě Kamailia s tím rozdílem, že balíček nainstalujeme přímo z oficiálních repozitářu pro naši distribuci operačního systému:

```
sudo apt update
sudo apt install asterisk
```

Tímto je instalace Asterisku ve verzi 16.2.1 dokončena. Ve složce */etc/asterisk* se po instalaci vytvořilo značné množství konfiguračních souborů, které jsou určeny jednotlivým Asterisk modulům. V našem případě si však vystačíme pouze s konfiguračními soubory *asterisk.conf*, *modules.conf*, *extensions.conf* a *pjsip.conf*. Přebytečné konfigurační soubory můžeme smazat příkazem:

```
rm  !("asterisk.conf"|"modules.conf"|"pjsip.conf"|"extensions.conf")
```

Soubory *pjsip.conf* a *extensions.conf* obsahují ve výchozím stavu ukázkovou konfiguraci, která je ale pro naše účely nevhodná, proto obsah obou souborů smažeme příkazy:

```
> pjsip.conf
> extensions.conf
```

V Asterisku jsou pro konfiguraci SIP kanálu k dispozici celkem dva moduly, *chan_sip.so* a *res_pjsip.so*. V dnešní době je převážně používán druhý z nich a v této práci tomu nebude jinak. Abychom po spuštění služby zabránili načtení modulu *chan_sip.so*, přidáme do sekce *[modules]* konfiguračního souboru *modules.conf* následující položku:

```
noload => chan_sip.so
```

Nyní můžeme přejít k samotné konfiguraci SIP kanálu pomocí souboru *pjsip.conf*. V prvním kroku vytvoříme sekci *transport*, a jak název napovídá, jedná se o sekci pro definování parametrů síťové komunikace (konfigurace jednotlivých Asterisk serverů je totožná, s výjimkou pole *bind*).

```
[transport-udp]
type = transport
protocol = udp
bind = 158.196.244.X  ;[v našem případě dosadíme za X hodnoty 169, 174 a 175]
```

Veškerá SIP komunikace směřující na jednotlivé Asterisk servery bude vždy přicházet z Kamailia. Do konfiguračního souboru tedy přidáme sekci typu *endpoint* a *identify* a nastavíme veškeré potřebné parametry pro komunikaci s Kamailiem. [45]

```
[kamailio]
type = endpoint
context = incoming
direct_media = no
from_domain = 158.196.244.168
disallow = all
allow = alaw
allow = g729

[kamailio]
type = identify
match = 158.196.244.168
endpoint = kamailio
```

Kamailiu v první řadě přiřadíme kontext, na který se bude následně odkazovat v souboru *extensions.conf*. Položka *direct_media* udává, zda chceme vytvořit přímé mediální spojení mezi uživateli. Pomocí pole *from_domain* můžeme nastavit část *domain* v poli *from* pro odchozí SIP žádosti. Dále definujeme povolené kodeky pro mediální tok (G.711 A-law, G.729). Poslední částí konfigurace je sekce *identify*. Zde nastavíme způsob, jak má Asterisk rozpoznat, že se jedná právě o endpoint s názvem Kamailio. Jelikož má Kamailio statickou IP adresu, je nejpohodlnější ho identifikovat pomocí pole *match*, které porovnává zdrojovou IP adresu s adresou definovanou v tomto poli.

Konfiguraci SIP kanálu máme úspěšně dokončenou. Aktuálně se můžeme přesunout k definici Dialplanu. Do souboru */etc/asterisk/extensions.conf* vložíme následující řádky: [46]

```
[incoming]
exten => _[0-9a-zA-Z]., 1, Answer()
    same => n, Echo()
```

V prvním kroku definujeme název kontextu, ten musí odpovídat kontextu, který jsme přiřadili Kamailiu v souboru *pjsip.conf*. Dále vytvoříme *pattern-matching*, který byl zmíněn v kapitole 4.2 a přiřadíme mu aplikaci *Answer* neboli vyzvednutí hovoru. Abychom během testování simulovali chod libovolné služby, která pracuje s mediálním tokem, použijeme v dalším pravidlu aplikaci *Echo()*. Ta zajistí, že věškerý příchozí mediální tok bude poslán zpět k jeho zdroji. Budeme tak mít zaručenou obousměrnou výměnu RTP toku bez nutnosti připojit k jednotlivým Asterisk serverům klientské zařízení, které by tuto funkci zastávalo. Na závěr můžeme službu Asterisk spustit příkazem:

```
sudo systemctl start asterisk
```

6.2 Distribuce zátěže s využitím Dispatcher modulu

V tuto chvíli máme dokončenou úvodní konfiguraci Kamailia a Asterisk serverů. V případě Asterisků již nebude nutné provádět další úpravy konfiguračních souborů *pjsip.conf* a *extensions.conf*, jelikož provedené nastavení je pro účely této práce dostačující. Veškerá konfigurace bude nyní probíhat výhradně na serveru s IP adresou 158.196.244.168 (Kamailio).

Tato sekce má za cíl ukázat základní konfiguraci Kamailia jakožto prvku provádějícího distribuci zátěže, a to s pomocí Dispatcher modulu.

6.2.1 Soubor dispatcher.list

Jak jsme si uvedli v kapitole 5.1.1, jednotlivé aplikační servery, mezi které si přejeme rozprostírat vstupní zátěž, je možné definovat buďto pomocí databáze anebo s využitím souboru *dispatcher.list*. Jelikož jsme při instalaci Kamailia neinstalovali dopňující databázové moduly, použijeme druhou ze zmíněných možností. Soubor vytvoříme například pomocí editoru *nano* níže uvedeným příkazem:

```
sudo nano /etc/kamailio/dispatcher.list
```

Do souboru vložíme následující záznamy:

```
1 sip:158.196.244.169:5060 0 3 duid=asterisk1
1 sip:158.196.244.174:5060 0 2 duid=asterisk2
1 sip:158.196.244.175:5060 0 1 duid=asterisk3
```

První sloupec udává skupinu, do které si přejeme Asterisky zařadit. Z této skupiny budeme následně v hlavním konfiguračním souboru jednotlivé aplikační servery vybírat. Druhým sloupcem definujeme SIP URI konkrétního Asterisk serveru. Následuje sloupec *flags*, který necháme na výchozí hodnotě 0. Předposlední sloupec nese údaje o prioritě jednotlivých Asterisk serverů. Zadané číslo slouží pro uvodní seřazení uvnitř této skupiny, a to nehledě na pořadí, v jakém jsou servery do tohoto souboru zapsány. Poslední sloupec obsahuje volitelné atributy, ze kterých nastavíme pouze pole *duid* používané distribučním algoritmem č. 10 (call load distribution). Nutno dodat, že hodnota tohoto pole musí být unikátní. [27]

6.2.2 Úprava hlavního konfiguračního souboru Kamailia

První věcí, kterou je nutné provést v hlavních konfiguračním souboru *kamailio.cfg*, je načtení modulu *dispatcher.so*. Tento modul je výchozí součástí instalačního balíčku Kamailia, proto ho v sekci modulů pouze načteme příkazem:

```
loadmodule "dispatcher.so"
```

Dále budeme pokračovat v sekci pro nastavení parametrů modulů. Dispatcher modul nabízí širokou škálu různých nastavení, pro naše účely však bude zcela dostačující, pokud do zmíněné sekce vložíme níže uvedené záznamy. [27]

```
modparam("dispatcher", "list_file", "/etc/kamailio/dispatcher.list")
modparam("dispatcher", "xavp_dst", "_dsdst_")
modparam("dispatcher", "flags", 2)
modparam("dispatcher", "ds_ping_interval", 60)
modparam("dispatcher", "ds_ping_latency_stats", 1)
modparam("dispatcher", "ds_probing_mode", 1)
modparam("dispatcher", "ds_probing_threshold", 3)
modparam("dispatcher", "ds_inactive_threshold", 3)
```

První záznamem informujeme Kamailio o umístění souboru *dispatcher.list*. Druhý záznam definuje název proměnné xAVP, do které budou uloženy potřebné informace ze souboru *dispatcher.list*. Tyto informace mohou být použity při volání záložní cesty v případě výpadku některého z aplikačních serverů. Abychom však umožnili *Dispatcher* modulu volaní záložní cesty v případě selhání, je nezbytné nastavit parametr *flags* na hodnotu 2. Na dalším řádku definujeme interval, s jakým budou zasílány *keepalive* (OPTIONS) zprávy, ověřující dostupnost jednotlivých aplikačních serverů. Nastavení parametru *ds_ping_latency_stats* na hodnotu 1 umožňuje měřit odezvu na *keepalive* zprávy. Parametr s názvem *ds_probing_mode* udává, které cíle mají být testovány *keepalive* zprávami. Tento parametr nastavíme na hodnotu 1, což znamená, že testovány budou všechny cíle. Poslední dva parametry se rovněž zabývají dostupností jednotlivých cílů. Parametrem *ds_probing_threshold* definujeme množství po sobě jdoucích nezodpovězených testovacích zpráv, po kterém je cíl označen jako INACTIVE. Naopak parametr *ds_inactive_threshold* udává počet zodpovězených, po sobě jdoucích *keepalive* zpráv, po kterém je cíl nacházející se ve stavu INACTIVE označen jako ACTIVE. Tím je nastavení modulu hotovo, můžeme se tak přesunout do sekce směrovacích bloků.

V naší práci si vytvoříme vlastní směrovací blok s názvem *DISPATCH*, který bude volán skrze hlavní směrovací blok *request_route*, přes který prochází všechny příchozí SIP žádosti. Volání bloku *DISPATCH* bude vypadá následovně:

```
request_route {
    ...

    # Volání směrovacího bloku pro distribuci zátěže
    route(DISPATCH);

    ...
}
```

Definici a vlastnosti bloku *DISPATCH* včetně bloku *DISPATCH_FAILURE* volaného v případě selhání některého z námi definovaných cílů provedme v libovolné části sekce směrovacích bloků. Výsledná podoba kódu vypadá takto:

```
route[DISPATCH] {
  if(method=="INVITE"){
    ds_select_dst("1", "4");
    t_on_failure("DISPATCH_FAILURE");
    route(RELAY);
  }
}

failure_route[DISPATCH_FAILURE] {
  xlog("L_WARN", "Trying next destination");
  ds_next_dst();
  route(RELAY);
}
```

V Bloku *DISPATCH* na úvod ověřujeme, zda se jedná o žádost INVITE, protože tuto funkci nám stačí volat pouze jednou, a to při sestavování spojení. Další žádosti v rámci dialogu jsou směrovány pomocí výchozího směrovacího bloku s název *[WITHINDLG]*. Dále voláme funkci *ds_select_dst*, kterou jsme si popsali v kapitole 5.1.3. První parametr udává skupinu, ze které budou cíle vybírány. Nastavíme ho proto na hodnotu 1, viz tvorba souboru *dispatcher.list*. Druhý parametr určuje typ použitého algoritmu. V této práci použijeme algoritmus *round robin*, z toho důvodu druhý parametr nastavíme na hodnotu 4. Pokud by transakce vytvořená žádostí INVITE byla ukončena negativní finální zprávou informujeme Kamailio, aby v takové chvíli použilo záložní cestu definovanou ve funkci *t_on_failure()*. Na závěr provedeme volání bloku *RELAY*, který je součástí výchozího konfiguračního souboru a má na starost směrování žádostí.

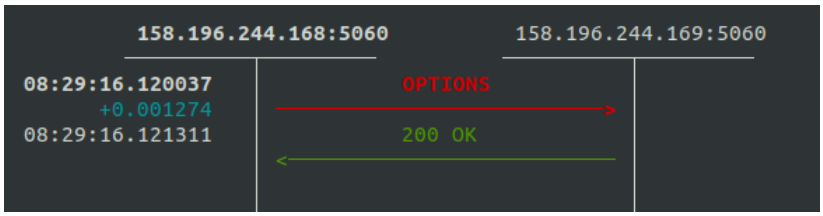
V bloku *DISPATCH_FAILURE* si necháme do logu vypsat informační zprávu, abychom věděli, že došlo k selhání některého z cílů. Následným voláním funkce *ds_next_dst()* vybereme z proměnné xAVP, kterou jsme si definovali v parametrech *Dispatcher* modulu další cíl. Na závěr opět provedeme volání bloku *RELAY*, který se postará o odeslání žádosti. [14, 27]

Oproti výchozí podobě konfiguračního souboru a výše uvedeným změnám byl v hlavním konfiguračním souboru dále proveden přesun, úprava, či smazání některých částí směrovacích bloků, které byly pro účely této práce nepotřebné. Uvedení všech těchto drobných změn by tuto část kapitoly činilo pro čtenáře nepřehlednou, proto je celý konfigurarční soubor pro tuto sekci součástí elektronické přílohy.

Abychom aplikovali do praxe změny provedené v hlavním konfiguračním souboru *kamailio.cfg*, restartujeme službu Kamailio.

```
sudo systemctl restart kamailio
```

V tuto chvíli jsou provedené změny připraveny k praktickému ověření. Na úvod můžeme zkontrolovat, zda jsou zasílány *keepalive* zprávy. Použijeme k tomu například nástroj *sngrep*, který slouží k odchytu SIP komunikace. Podoba testování dostupnosti aplikačního serveru Asterisk 1 (.169) je uvedena na následujícím obrázku.



Obrázek 6.2: Testování dostupnosti aplikačního serveru pomocí SIP OPTIONS

Pokud chceme zobrazit statistiky týkající se odezvy na *keepalive* zprávy, které jsme si povolili v sekci parametrů modulů, zadáme do příkazové řádky příkaz:

```
kamcmd dispatcher.list
```

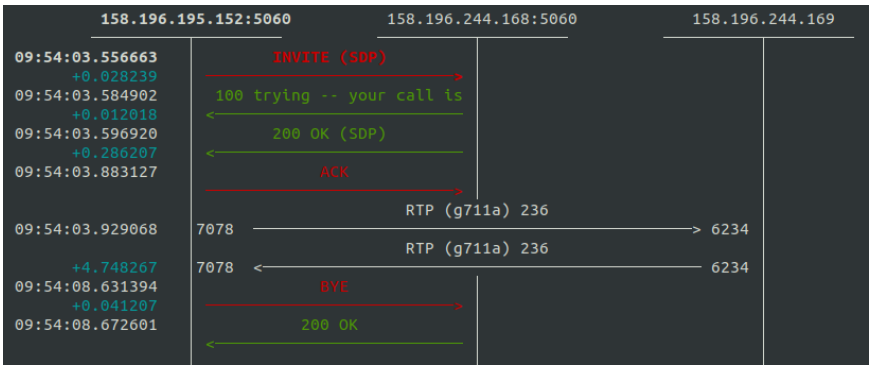
Příkaz vypíše kromě statistik jednotlivých cílů také hodnoty, které jsme nastavili v souboru *dispatcher.list*. Podoba výpisu pro server Asterisk 1 (.169) je uvedena níže.

```
DEST: {
        URI: sip:158.196.244.169:5060
        FLAGS: AP
        PRIORITY: 3
        ATTRS: {
                BODY: duid=asterisk1
                DUID: asterisk1
                MAXLOAD: 0
                WEIGHT: 0
                RWEIGHT: 0
                SOCKET:
                SOCKNAME:
                OBPROXY:
        }
        LATENCY: {
                AVG: 2.333000
                STD: 0.577000
                EST: 2.333000
                MAX: 3
                TIMEOUT: 0
        }
}
```

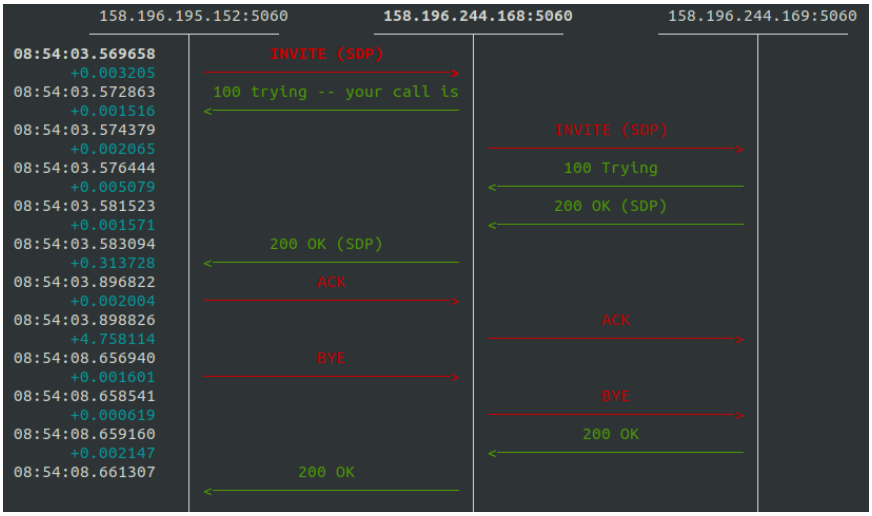
Obrázek 6.3: Zobrazení statistik aplikačního serveru pomocí nástroje kamcmd

Důležitou hodnotu obsahuje zejména pole *TIMEOUT*. To nese informaci o tom, kolikrát došlo k nezodpovězení *keepalive* zprávy. Pokud v našem případě toto pole nabyde hodnoty 3, je cíl označen jako *INACTIVE*.

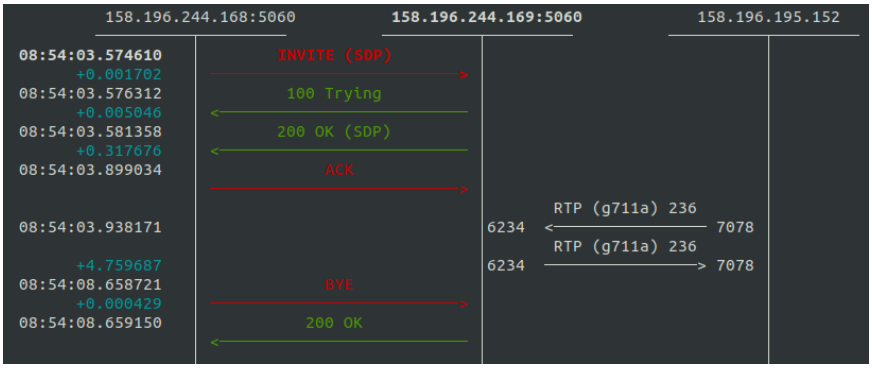
Nyní si dokažme funkčnost celého systému pomocí testovacího hovoru. Průběh SIP signalizace z pohledu všech stran zapojených do hovoru je uveden na třech níže uvedených obrázcích.



Obrázek 6.4: Průběh SIP signalizace z pohledu SIP klienta



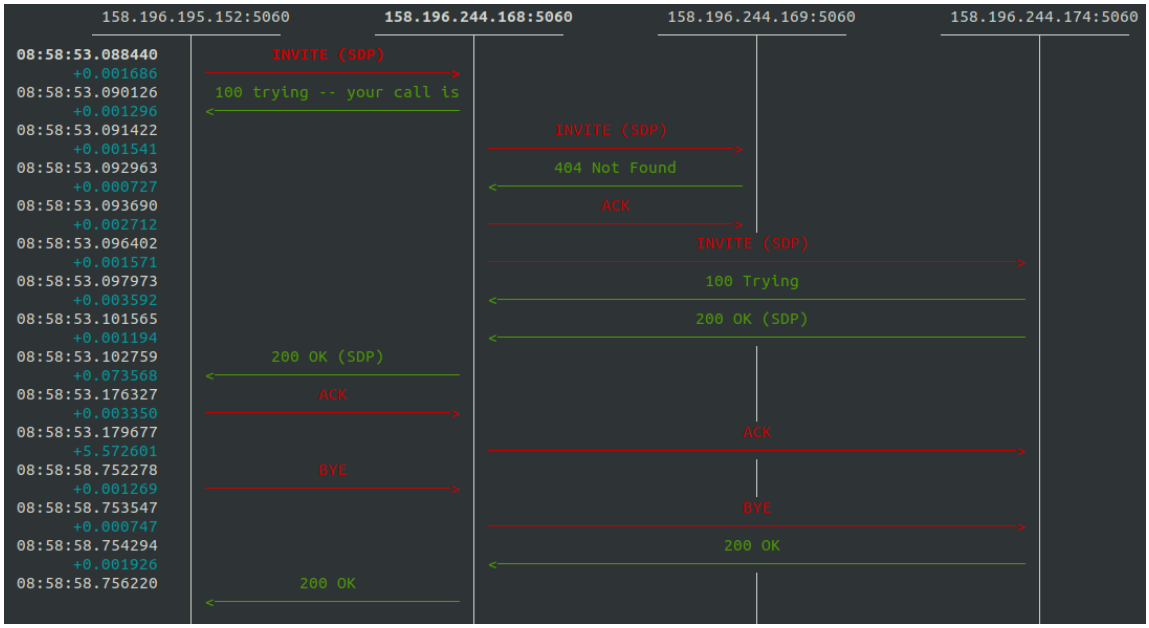
Obrázek 6.5: Průběh SIP signalizace z pohledu SIP Proxy Kamailio



Obrázek 6.6: Průběh SIP signalizace z pohledu aplikačního serveru Asterisk 1

Je zřejmé, že hovor proběhl úspěšně. Pokud bychom nyní provedli další hovor, viděli bychom obdobný průběh s tím rozdílem, že by byl namísto Asterisku 1 (.169) vybrán za cíl Asterisk 2 (.174), což je dáno principem *round robin* algoritmu. Lze si také všimnout, že mediální tok prochází přímo mezi klientem a Asteriskem. Kamailio funguje pouze jako SIP Proxy, a proto se ho mediální tok vůbec netýká.

Nyní už zbývá pouze ověřit, zda funguje automatické spuštění záložní cesty v případě selhání některého z cílů. Demonstrace takzvané *failover* funkce je uvedena na následujícím obrázku.



Obrázek 6.7: Demonstrace failover funkce pro konfiguraci využívající Dispatcher modul

Vidíme, že námi provedená konfigurace funguje dle očekávání, a to včetně *failover* funkce. Drobnou úpravou bychom mohli použít i ostatní distribuční algoritmy s výjimkou těch, které pro svůj chod vyžadují nastavení dalších parametrů *Dispatcher* modulu. Pro základní ukázku konfigurace a chodu tohoto modulu však byla tato sekce více než dostačující. V tuto chvíli se proto přesuneme k hlavní části této kapitoly, ve které se pokusíme nakonfigurovat a zprovoznit podobný systém, který však jeden z cílů vybere na základě aktuálního stavu vytížení jejich procesorů.

6.3 Distribuce zátěže pomocí vlastního řešení

Tuto poměrně rozsáhlou sekci si rozdělíme na celkem čtyři části. První dvě se budou zabývat úvodní konfigurací REST API aplikace a jejím následným rozšířením o uWSGI a Nginx. V dalším kroku vytvoříme uživatelský skript pro automatické odesílání informací o aktuální zátěži. V závěru provedeme úpravu hlavního konfiguračního souboru Kamailia, abychom zajistili, že na základě dat získaných z REST API provede správné určení cíle.

6.3.1 Vytvoření REST API

Jak jsme si již uvedli v předchozí kapitole, naše REST API budeme vytvářet pomocí Python knihovny Flask. Při vývoji a vytváření Python aplikací je vhodné pracovat ve virtuálním prostředí, které nám zajistí, že balíčky nainstalované v tomto prostředí nebudou v žádném případě ovlivněny jinými Python projekty či aplikacemi nacházejícími se na stejném zařízení. My pro vytvoření takového izolovaného prostředí použijeme nástroj *pipenv*, který nainstalujeme příkazem:

```
pip3 install pipenv
```

Dále si pod běžným uživatelem vytvoříme složku, do které bude umístěn veškerý obsah našeho projektu. Pro vytvoření této složky a přechod do ní použijeme příkazy:

```
$ mkdir /home/van0201/rest_api
$ cd /home/van0201/rest_api
```

Z výše uvedeného adresáře poté zadáme příkaz, s jehož pomocí dojde k vytvoření a aktivaci virtuálního prostředí. Aktivní stav je snadno detekovatelný pomocí části příkazové řádky s označením *prompt*, která bude ve tvaru *(rest_api) user@host:~/rest_api$*.

```
pipenv shell
```

Níže uvedeným příkazem nainstalujeme sadu potřebných Python knihoven, které budou nezbytné pro správnou funkčnost naší aplikace.

```
(rest_api)$ pipenv install flask flask-sqlalchemy flask-marshmallow
marshmallow-sqlalchemy datetime
```

Nyní už můžeme přejít k vytváření samotné aplikace. Pod běžným uživatele proto vytvoříme soubor s koncovkou .py, v našem případě:

```
(rest_api)$ nano api.py
```

Na začátek souboru vložíme příkazy sloužící k importování potřebných knihoven či jejich částí.

```
from flask import Flask, request, jsonify
from flask_sqlalchemy import SQLAlchemy
from flask_marshmallow import Marshmallow
import datetime
import os
```

V dalším kroku vytvoříme instanci třídy a následně definujeme proměnou *homedir* nesoucí umístění naší aplikace a proměnou *record_max_age*, pomocí které budeme v dalších krocích určovat maximální tolerované stáří databázového záznamu (hodnota určuje počet sekund).

```
app = Flask(__name__)
homedir = os.path.abspath(os.path.dirname(__file__))
record_max_age = 6
```

Uvedli jsme si, že pro ukládání dat budeme používat SQLite databázi. Pro informování aplikace o umístění databáze použijeme dříve vytvořenou proměnou *basedir*, která bude sloužit jako jeden z parametrů pro funkci *os.path.join()*. Lokaci databáze provedeme následovně: [47]

```
app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///' + os.path.join(homedir, 'db.sqlite')
app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False
```

Druhým z výše uvedených příkazů poté deaktivujeme výpis varovných hlášení týkajících se databáze do konzole. V další části úvodní konfigurace provedeme inicializaci knihovny *SQLAlchemy* a knihovny *Marshmallow*. Druhou ze zmíněných knihoven budeme používat k převodu objektů na nativní datové typy pro Python a dále také k zpětnému převodu do standardního typu, kterým je například JSON. Inicializaci obou knihoven provedeme takto:

```
db = SQLAlchemy(app)
ma = Marshmallow(app)
```

V kapitole č. 5.2.1 jsme si uvedli, že v REST API přistupujeme k takzvaným zdrojům. Zdrojem, ke kterému budeme v našem případě přistupovat je Asterisk. Proto vytvoříme třídu s tímto názvem a definujeme atributy, jež budou náš zdroj charakterizovat. [48]

```
class Asterisk(db.Model):
  id = db.Column(db.Integer, primary_key = True)
  load = db.Column(db.Integer)
  date = db.Column(db.DateTime, default = datetime.datetime.utcnow)

  def __init__(self, load, date):
    self.load = load
    self.date = date
```

Každý zdroj bude popsán pomocí jeho *id*, které bude zastávat funkci primárního klíče, dále položkou *load* obsahující hodnotu vytížení procesoru. Poslední atribut nese název *date*, a jak již název napovídá, jeho obsahem bude datum. Tento atribut bude plnit důležitou roli při určování stáří záznamu. Částí začínající příkazem *def ___init___():* vytváříme konstruktor. Ten se volá v případě, kdy vytvaříme objekt uvnitř třídy a chceme inicializovat atributy této třídy. [29, 48]

Nyní vytvoříme další třídu s názvem *AsteriskSchema*, ve které definujeme schéma Asterisku. Jinými slovy specifikujeme atributy zdroje Asterisk, ke kterým bude mít uživatel zasílající konkrétní HTTP žádost přístup. Podoba této třídy včetně inicializace schématu je následovná:

```
class AsteriskSchema(ma.Schema):
  class Meta:
    fields = ('id', 'load')

asterisk_schema = AsteriskSchema()
```

V tuto chvíli máme provedeno veškeré potřebné nastavení, můžeme se tak pustit do části, ve které budeme definovat funkce pro vytvoření, čtení, aktualizování, či mazání jednotlivých zdrojů. Na úvod logicky začneme s funkcí, ve které definujeme vytvoření zdroje:

```
@app.route('/asterisk', methods = ['POST'])
def add_asterisk():
  load = request.json['load']
  date = datetime.datetime.utcnow()
  new_asterisk = Asterisk(load, date)
  db.session.add(new_asterisk)
  db.session.commit()
  return asterisk_schema.jsonify(new_asterisk)
```

Na úvodním řádku definujeme pro jakou URL a typ HTTP žádosti má být funkce *add_asterisk()* spuštěna. Již dříve jsme si uvedli, že pro vytváření nového zdroje se využívá žádost POST. V žádosti musí být obsaženo pouze pole *load*, a to v JSON formátu. Pole *id* je vytvořeno automaticky, stejně jako pole *date*, které bude obsahovat čas příchodu žádosti. Následně zapíšeme data do databáze a potvrdíme zápis příkazem *db.session.commit()*. Uživateli zašleme schéma zdroje, to znamená jeho *id* a hodnotu atributu *load*. V dalším kroku provedeme definování funkce pro získání skupiny zdrojů s názvem *get_asterisks*. [48, 49]

```
@app.route('/asterisk', methods = ['GET'])
def get_asterisks():
  now = datetime.datetime.utcnow()
  max_age = now - datetime.timedelta(seconds=record_max_age)
  result = Asterisk.query.filter(Asterisk.date > max_age).order_by(Asterisk.load).first()

  if not result:
    return jsonify('empty')
  else:
    return asterisk_schema.jsonify(result)
```

Za normálních okolností by tato funkce měla za úkol vrátit celou skupinu zdrojů, tedy všechny záznamy vytvořené žádostmi POST. My však chceme Kamailio informovat pouze o jednom konkrétním nejméně vytíženém zdroji ze skupiny zdrojů charakterizované URL */asterisk*. Nejprve ale provedeme prohledávání dátabáze s cílem vyfiltrovat všechny záznamy, jejichž stáří je menší než hodnota proměnné *record_max_age*. Následně výsledek tohoto prohledávání seřadíme vzestupně podle hodnoty atributu *load* a do proměnné *result* zapíšeme údaje o zdroji, který je po seřazení na první pozici (je nejméně vytížený). Touto filtrací máme ošetřeno, že Kamailiu nevrátíme informace

o zdroji, který je sice na první pohled nejméně vytížený, ale stáří jeho záznamu napovídá tomu, že s REST API nekomunikuje, a proto lze předpokládat, že došlo k jeho selhání. Pokud tedy Asterisk posílající data přestane komunikovat, nebudou na něho v žádném případě hovory směrovány. Na závěr pouze ověřujeme, zda databáze při prohledávání našla nějaký záznam, pokud ano, zašleme v JSON formátu informace o zdroji, který splnil nastavená kritéria. Pokud není nalezen žádný záznam, zašleme zprávu *"empty"*, kterou oznamujeme, že v databázi není žádný záznam splňující nastavené podmínky. V tuto chvíli nadefinujeme další funkci pro žádost GET, která ale bude směrována na konkrétní zdroj, charakterizovaný hodnotou pole *id*, viz níže. [48, 49]

```
@app.route('/asterisk/<id>', methods = ['GET'])
def get_asterisk(id):
  now = datetime.datetime.utcnow()
  max_age = now - datetime.timedelta(seconds=record_max_age)
  asterisk = Asterisk.query.get(id)

  if asterisk.date < max_age:
    return jsonify('empty')
  else:
    return asterisk_schema.jsonify(asterisk)
```

Vidíme, že nyní je zdroj charakterizován URL ve tvaru */asterisk/<id>*. Mírně odlišným způsobem (z důvodu přistupování ke konkrétnímu zdroji) opět ověřujeme stáří záznamu. Pokud je starší než hodnota proměnné *record_max_age*, pošleme uživateli přistupujícímu ke zdroji zprávu *"empty"*. V opačném případě zašleme informace o zdroji v JSON formátu.

Další funkcí budeme provádět aktualizaci zdroje. Funkce bude volána v případě, kdy budeme přistupovat k URL */asterisk/<id>* pomocí žádosti PUT.

```
@app.route('/asterisk/<id>', methods = ['PUT'])
def update_asterisk(id):
  asterisk = Asterisk.query.get(id)
  load = request.json['load']
  date = datetime.datetime.utcnow()
  asterisk.load = load
  asterisk.date = date
  db.session.commit()
  return jsonify('saved')
```

Ve funkci *update_asterisk(id)* provedeme vyhledání již existujícího zdroje, jehož informace chceme aktualizovat. Nová hodnota atributu *load* bude získaná z příchozí žádosti, zatímco atributu *date*

přiřadíme hodnotu odpovídající době příchodu PUT žádosti do systému. Po provedení změn následně potvrdíme zápis do databáze funkcí *db.session.commit()* a autorovi žádosti zašleme informační zprávu potvrzující uložení dat.

Poslední z takzvaných CRUD operací, kterou naše aplikace postrádá, je funkce pro smazání zdroje. Totožně jako v případě PUT žádosti provedeme nejdříve vyhledání konkrétního zdroje podle hodnoty jeho atributu *id*. Poté záznam charakterizující daný zdroj z databáze odstraníme a celou operaci potvrdíme. Uživatel je o úspěšném odstranění informován zprávou *"deleted"*. [48, 49]

```
@app.route('/asterisk/<id>', methods=['DELETE'])
def delete_asterisk(id):
  asterisk = Asterisk.query.get(id)
  db.session.delete(asterisk)
  db.session.commit()
  return jsonify('deleted')
```

Do závěru našeho kódu umístíme dva níže uvedené řádky. V podmínce *if* ověřujeme, zda speciální vestavěná proměnná *___name___* nabývá hodnoty *___main___*. Pokud ano, dojde ke spuštění naší aplikace pod IP adresou 158.196.244.168 port 5000. K této situaci dojde, pokud je aplikace spuštěna jako hlavní program. Pokud tomu tak není, například z důvodu importování jedné aplikace do druhé, odpovídá hodnota proměnné *___name___* názvu aplikace bez přípony *.py*. V tom případě dojde k vykonání pouze kódu takzvané nejvyšší úrovně. Soubor *api.py* je tímto dokončen, proto můžeme přejít k jeho uložení.

```
if __name__ == '__main__':
  app.run(host='0.0.0.0')
```

Poslední prvek, který nám aktuálně chybí, je databáze. Ve virtuálním prostředí proto zadámé příkaz *python*, s jehož pomocí přejdeme do příkazové řádky Pythonu. Do ní následně zadáme tyto příkazy:

```
from api import db
db.create_all()
exit()
```

Tím dojde k importu objektu *db*, který jsme si uvnitř souboru *api.py* nadefinovali a následně i k vytvoření samotné databáze s názvem *db.sqlite*. Databáze je vytvořena včetně námi specifikovaného obsahu, tedy sloupců odpovídajícího názvu a datového typu. Virtuální prostředí můžeme v tuto chvíli opustit příkazem *exit*. [48]

6.3.2 Nasazení uWSGI a webového serveru Nginx

Námi vytvořená aplikace je v tuto chvíli připravena ke spuštění. Dříve než se tak stane, provedeme konfigurovaci rozšiřujících služeb uWSGI a Nginx, které chceme použít pro chod naší aplikace. V úvodu se pustíme do nastavení a tvorby uWSGI. Celý proces započneme doinstalováním potřebných balíčků, které v našem systému aktuálně chybí.

```
sudo apt update
sudo apt install libffi-dev build-essential
```

Dále se přesuneme zpět do složky, kterou jsme vytvořili pro naše RESP API a provedeme aktivaci virtuálního prostředí. To vše následujícími příkazy:

```
cd /home/van0201/rest_api
pipenv shell
```

Z virtuálního prostředí následně nainstalujeme *uwsgi* balíček.

```
(rest_api)& pipenv install uwsgi
```

Nyní máme nachystané veškeré potřebné doplňky. Ve složce */rest_api* proto vytvoříme další soubor s koncovkou .py, který bude sloužit jako brána do námi vytvořené aplikace. Soubor opět vytvoříme například s pomocí textového editoru *nano*.

```
(rest_api)& nano wsgi.py
```

Do souboru vložíme níže uvedené řádky, které importují instanci naší aplikace a spustí ji. [50]

```
from myproject import app

if __name__ == "__main__":
  app.run()
```

Soubor s tímto obsahem můžeme uložit. Pokud chceme v tuto chvíli ověřit, zda je uWSGI schopno provozovat naší aplikaci, zadáme do příkazové řádky virtuálního prostředí příkaz:

```
uwsgi --socket 0.0.0.0:5000 --protocol=http -w wsgi:app
```

Následně skrze prohlížeč nebo libovolný testovací nástroj (např. Postman) navštívíme stránku s následující adresou:

```
http://158.196.244.168:5000/asterisk
```

Tím provedeme zaslání žádosti GET na výše uvedenou adresu. Výstupem je v tuto chvíli hláška *"empty"*, což je očekávaný výsledek, jelikož je databáze naší aplikace aktuálně prázdná. Takto spouštět naši aplikaci ovšem není příliš pohodlné a robustní. Z tohoto důvodu nyní opustíme virtuální prostředí a ve složce */rest_api* vytvoříme další soubor s koncovkou *.ini*. [50]

```
$ nano /home/van0201/rest_api/rest_api.ini
```

Do tohoto souboru umístíme řádky definované níže. Názvem *uwsgi* v záhlaví informuje uWSGI, že má použít níže definované nastavení. Parametrem *module* definujeme název aplikace, která se má spustit. Dále nastavujeme spuštění uWSGI v hlavním režimu s celkem pěti procesy. Pokračujeme definováním socketu, skrze který bude probíhat komunikace mezi webovým serverem Nginx a uWSGI. Tomuto socketu dále nastavíme přístupová práva a parametrem *vacuum=true* říkáme, aby byl socket po ukončení procesu smazán. Poslední parametrem umožňujeme ukončení procesu při obdržení SIGTERM signálu (např. při použití příkazu *kill*). [50]

```
[uwsgi]
module = wsgi:app
master = true
processes = 5
socket = rest_api.sock
chmod-socket = 660
vacuum = true
die-on-term = true
```

V dalším kroku vytvoříme soubor pro spuštění naší aplikace jakožto jednotky systemd. Tím zajistíme automatické zapnutí uWSGI při spuštění serveru. Soubor vytvoříme následovně:

```
sudo nano /etc/systemd/system/rest_api.service
```

Do souboru poté vložíme příkazy, které budou členěny do sekcí *[Unit]*, *[Service]* *a* [Install]:

```
[Unit]
Description=uWSGI instance for REST API
After=network.target

[Service]
WorkingDirectory=/home/van0201/rest_api
User=van0201
Group=www-data
Type=simple
ExecStart=/usr/local/bin/pipenv run uwsgi --ini rest_api.ini

[Install]
WantedBy=multi-user.target
```

Sekce *[Unit]* slouží pro specifikaci metadat a závislostí. My v této části oznamujeme systému, aby aktivoval jednotku až ve chvíli, kdy je spuštěna síťová služba našeho systému. V sekci *[Service]* definujeme uživatele a skupinu, pod kterou chceme proces spustit. Zde záměrně nastavujeme skupinu na *www-data*, abychom zajistili přístup pro Nginx. Nastavíme rovněž cestu ke složce obsahující veškeré potřebné soubory a informujeme systém o umístění spustitelného souboru, který

bude spouštěn skrze naše virtuální prostředí. Poslední sekcí s názvem *[Install]* oznamujeme, s čím má *systemd* naši službu propojit, pokud povolíme její spuštení při startu systému. Nastavíme zde, že chceme, aby byla služba spuštěna, jakmile je načten a spuštěn běžný víceuživatelský systém. [50]

Soubor uložíme a následně službu spustíme a povolíme její automatické spuštení při startu systému.

```
sudo systemctl start rest_api
sudo systemctl enable rest_api
```

Stav služby ověříme příkazem:

```
sudo systemctl status rest_api
```

```
● rest_api.service - uWSGI instance for REST API
     Loaded: loaded (/etc/systemd/system/rest_api.service; enabled; vendor preset: enabled)
     Active: active (running) since Tue 2021-03-16 13:54:19 UTC; 1s ago
   Main PID: 1112871 (uwsgi)
      Tasks: 6 (limit: 1074)
     Memory: 60.3M
     CGroup: /system.slice/rest_api.service
             ├─1112871 /home/van0201/.local/share/virtualenvs/rest_api-SxRq8ldF/bin/uwsgi --ini rest_api.ini
             ├─1112883 /home/van0201/.local/share/virtualenvs/rest_api-SxRq8ldF/bin/uwsgi --ini rest_api.ini
             ├─1112884 /home/van0201/.local/share/virtualenvs/rest_api-SxRq8ldF/bin/uwsgi --ini rest_api.ini
             ├─1112885 /home/van0201/.local/share/virtualenvs/rest_api-SxRq8ldF/bin/uwsgi --ini rest_api.ini
             ├─1112886 /home/van0201/.local/share/virtualenvs/rest_api-SxRq8ldF/bin/uwsgi --ini rest_api.ini
             └─1112887 /home/van0201/.local/share/virtualenvs/rest_api-SxRq8ldF/bin/uwsgi --ini rest_api.ini
```

Obrázek 6.8: Zobrazení stavu služby rest_api

Služba je spuštěna a aktivní. Nyní ke službě připojíme webový server Nginx, který bude se službou *rest_api* komunikovat skrze námi vytvořený socket. Instalaci webového serveru Nginx provedeme příkazy:

```
sudo apt update
sudo apt install nginx
```

Nyní nastavíme firewall, a to tak, aby umožňoval přístup k webovému serveru Nginx. Pro tento účel použijeme jednoduchý systémový nástroj pro správu firewallu s názvem UFW (Uncomplicated Firewall). Po instalaci nabízí Nginx celkem tři profily, které specifikují, jaké porty mají být otevřeny pro příchozí síťový provoz. Následujícím příkazem proto nastavíme profil s otevřeným portem číslo 80 (HTTP). [50]

```
sudo ufw allow 'Nginx HTTP'
```

V tuto chvíli zbývá pouze nastavit, kam má Ngnix směrovat příchozí žádosti. Přesuneme se proto do složky */etc/nginx/sites-available* a vytvoříme zde soubor s názvem *rest_api*.

```
sudo nano /etc/nginx/sites-available/rest_api
```

Do souboru vložíme následující řádky určující IP adresu a port, na které bude Nginx naslouchat příchozím žádostem a dále také umístění socketu určeného pro komunikaci mezi webovým serverem a uWSGI. [50]

```
server {
    listen 80;
    server_name 158.196.244.168;

    location / {
        include uwsgi_params;
        uwsgi_pass unix:/home/van0201/rest_api/rest_api.sock;
    }
}
```

Pro povelení výše uvedené konfigurace vytvoříme propojení se složkou */etc/nging/sites-enabled*, se kterou Nginx pracuje při jeho spuštění.

```
sudo ln -s /etc/nginx/sites-available/rest_api /etc/nginx/sites-enabled
```

Posledními dvěma příkazy provedeme ověření správnosti konfigurace a restartování služby Nginx.

```
sudo nginx -t
sudo systemctl restart nginx
```

Nyní je naše aplikace pro výměnu informací mezi Kamailiem a aplikačními servery hotová. Aplikace je dostupná pod URL:

```
http://158.196.244.168/asterisk
```

6.3.3 Vytvoření skriptu pro odesílání informací o aktuální zátěži

V této sekci vytvoříme klientský skript/skripty, kterými budeme skrze REST API zapisovat do databáze aktuální stav vytížení jednotlivých aplikačních serverů. Již dříve jsme si uvedli, že pro tyto účely budou primární Python knihovny *requests* a *psutil*. Na úvod si však musíme říct, co od klientského skriptu očekáváme a jak by měl fungovat. Určitě se shodneme, že je nezbytné zasílat dva typy HTTP žádostí. Při vytváření zdroje žádost POST a při aktualizaci stavu zdroje žádost PUT. Žádost POST nám však stačí v případě každého zdroje zaslat pouze jednou v samém úvodu, čímž dojde k vytvoření záznamu uvnitř databáze. Náš skript ale budeme chtít podobně jako REST API spouštět pomocí *systemctl*, aby se například při selhání serveru a jeho znovu spuštění automaticky zapnul. V tom případě však není vhodné, aby jeden skript obsahoval oba typy žádostí, protože by při znovu spuštění skriptu došlo ke zbytečnému zaslání nové žádosti POST a tím i k vytvoření

nového zdroje v databázi. Takový zdroj by pak po dobu, kdy je považován za aktuální (šest sekund) mohl být falešně vyhodnocen jako nejvhodnější cíl.

Na každém aplikačním serveru proto vytvoříme dva skripty. Jedním vytvoříme záznam v databázi (POST) a pomocí druhého (PUT) budeme tentýž záznam aktualizovat. Jako službu poté budeme spouštět pouze skript pro aktualizaci záznamu. Stejně jako v případě REST API si pod běžným uživatelem nejdříve vytvoříme složku, ve které bude uložena veškerá konfigurace.

```
$ mkdir /home/van0201/rest_client
$ cd /home/van0201/rest_client
```

Dále vytvořeníme a aktivujeme virtuální prostředí a druhým příkazem nainstalajume dvojici potřebných knihoven.

```
pipenv shell
(rest_client)$ pipenv install requests psutil
```

Nyní vytvoříme soubor s název post.py, který bude sloužit jako skript pro úvodní vytvoření zdroje prostřednictvím POST žádosti.

```
(rest_client)$ nano post.py
```

V úvodu souboru provedeme importování již zmíněných knihoven. Následně definujeme proměnou *cpu_load*, které přiřadíme hodnotu získanou z funkce *psutil.cpu_percent()* zaokrouhlenou na celé číslo. Do této funkce vstupuje parametr *interval*, který definuje dobu blokování, po kterou je provedeno průměrování vytížení CPU. Bez tohoto parametru by funkce mohla vrátit značně zkreslené hodnoty. Posledním příkazem zašleme na adresu REST API žádost POST s obsahem ve formátu JSON. Jedinou položku, jež naše REST API v příchozí žádosti vyžaduje, je pole *load*, které ponese hodnotu proměnné *cpu_load*. Celý obsah skriptu je k nahlédnutí níže. [51, 52]

```
import requests
import psutil


cpu_load = round(psutil.cpu_percent(interval = 0.5))
requests.post('http://158.196.244.168/asterisk', json = {'load': cpu_load})
```

Soubor uložíme a z virtuálního prostředí provedeme jeho vykonání příkazem:

```
(rest_client)$ python post.py
```

Po spuštění výše uvedeného skriptu z jednotlivých aplikačních serverů se v databázi vytvořily celkem tři záznamy. (Pro náhled do databáze lze použít například aplikaci DB Browser (SQLite)). Záznamy mají v poli ID hodnoty 1, 2 a 3. Ve druhém skriptu bude mít každý z aplikačních serverů za úkol aktualizovat jeden konkrétní databázový záznam charakterizovaný hodnotou pole ID.

Druhý skript pro aktualizaci databázových záznamů skrze PUT zprávy s názvem *client.py* vytvoříme rovněž ve virtuálním prostředí příkazem:

```
(rest_client)$ nano client.py
```

Obsah souboru bude tvořit níže uvedný kód:

```
import requests
import psutil
import time

while True:
  cpu_load = round(psutil.cpu_percent(interval = None))
  requests.put('http://158.196.244.168/asterisk/1', json = {'load': cpu_load})

  if cpu_load <= 15:
    wait = 3
  elif cpu_load > 15 and cpu_load <= 30:
    wait = 2.5
  elif cpu_load > 30 and cpu_load <= 45:
    wait = 2
  elif cpu_load > 45 and cpu_load <= 55:
    wait = 1.5
  elif cpu_load > 55 and cpu_load <= 65:
    wait = 1
  elif cpu_load > 65 and cpu_load <= 75:
    wait = 0.5
  else:
    wait = 0.25

  time.sleep(wait)
```

Na úvod stejně jako v případě skriptu *post.py* importujeme potřebné knihovny, ale tentokrát navíc i knihovnu *time*. Dále vytvoříme nekonečnou smyčku *while True*, ve které bude délku cyklu určovat aktuální stav vytížení CPU. S rostucím vytížením se tedy úměrně zvýší frekvence zasílání informačních dat a naopak v případě klesajícího vytížení se adekvátně sníží. Takto zamezíme zbytečně častému zasílání dat v případě nízkého vytížení procesoru a na druhou stranu dostatečně krátký interval v případě velké zátěže. Uvedené intervaly a meze byly zvoleny na základě praktických testů. Platí rovněž, že nejdelší interval (tři sekundy) je dvojnásobně kratší, než maximální přípustné stáří záznamu nastavené v REST API. Co se týče samotné POST žádosti, jsou zde v

porovnání s předchozím skriptem patrné jisté změny. Parametr *interval* máme nyní nastavený na hodnotu *None*, a to z toho důvodu, že dochází k automatickému průměrovaní mezi voláním funkce *psutil.cpu_percent()* v jednotlivých cyklech, k čemuž ve skriptu *post.py* nedocházelo, jelikož v něm byla funkce volána pouze jednou. Kromě volání funkce *request.put()* namísto *request.post()* je patrný rozdíl rovněž v URL. Nyní aktualizujeme konkrétní zdroj, tudíž musíme specifikovat ID onoho zdroje. Při přiřazování jednotlivých ID k aplikačním serverům (Asteriskům) jsem vycházel ze schématu na obrázku č. 6.1. To znamená, že Asterisk 1 (IP .169) aktualizuje zdroj s ID 1, Asterisk 2 (IP .174) aktualizuje zdroj s ID 2 a Asterisk 3 (IP .175) obnovuje informace zdroje s ID 3. [51, 52]

Na závěr této sekce vytvoříme soubor, který podobně jako v případě uWSGI zajistí, že budeme schopni tento skript spouštět jako službu skrze *systemd*.

```
sudo nano /etc/systemd/system/rest_client.service
```

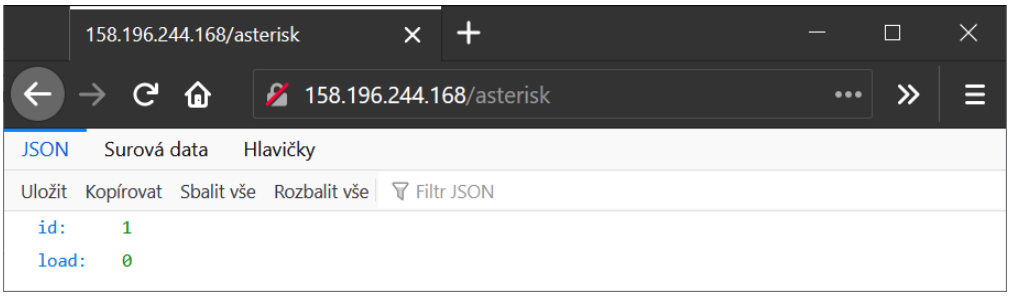
Obsah souboru bude téměř obdobný jako v případě souboru pro definování uWSGI služby. Rozdíl bude pouze v poli *Description*, ve kterém přidáváme popisek naší služby. Další změnou je jiný název pracovní složky a mírně odlišné spouštení skriptu skrze virtuální prostředí. Obsah tohoto souboru proto bude následovný:

```
[Unit]
Description=REST client service
After=multi.user.target

[Service]
WorkingDirectory=/home/van0201/rest_client
User=van0201
Type=simple
ExecStart=/usr/local/bin/pipenv run python /home/van0201/rest_client/client.py

[Install]
WantedBy=multi-user.target
```

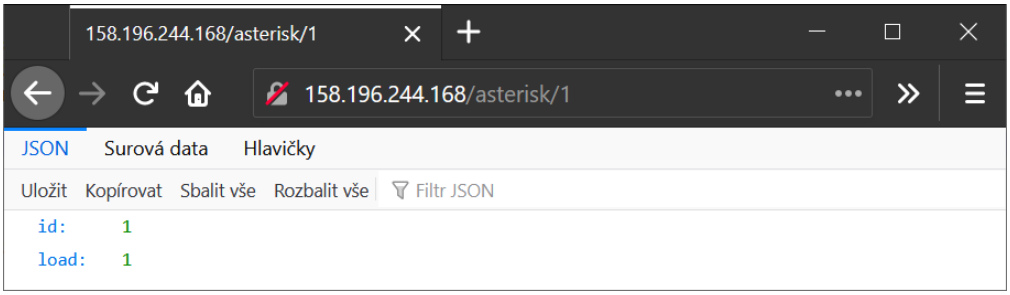
Soubor uložíme a celou službu spustíme. Na závěr povolíme automatické spuštění služby v případě restartování serveru a ověříme její aktuální stav.

```
sudo systemctl start rest_client
sudo systemctl enable rest_client
sudo systemctl status rest_client
```

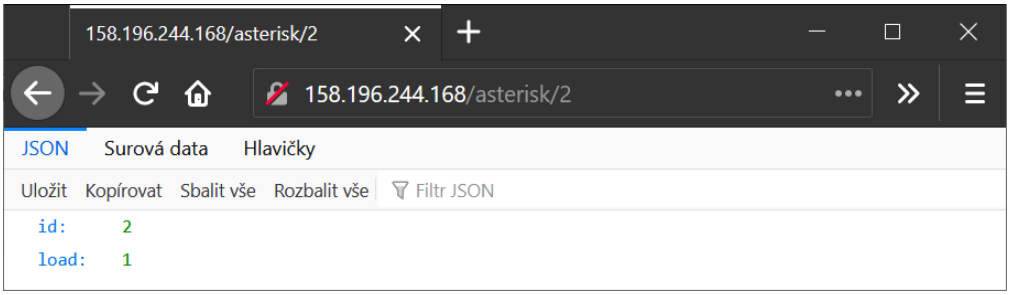
Nyní již můžeme otestovat, zda jsme schopni zobrazit data dostupná pod specifickými URL. Pro čtení dat budeme vždy používat žádost GET. Nejprve ji aplikujeme na skupinu zdrojů */asterisk*, přičemž výstupem by měl být aktuální (ne starší než 6 sekund) a v tu dobu nejméně vytížený Asterisk. Dále pak žádost aplikujeme na URL konkrétních zdrojů charakterizovaných pomocí ID.



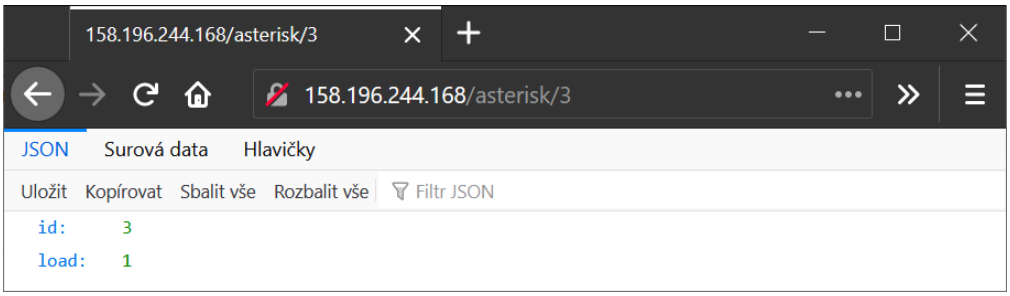
Obrázek 6.9: Výstup po zaslání žádosti HTTP GET na skupinu zdrojů Asterisk



Obrázek 6.10: Výstup po zaslání žádosti HTTP GET na zdroj Asterisk s ID 1



Obrázek 6.11: Výstup po zaslání žádosti HTTP GET na zdroj Asterisk s ID 2



Obrázek 6.12: Výstup po zaslání žádosti HTTP GET na zdroj Asterisk s ID 3

Výměna a aktualizace dat mezi aplikačními servery a REST API očividně funguje a webový prohlížeč rovněž správně rozpoznal, že jsou data zapsána v JSON formátu. Funkce GET aplikovaná na skupinu zdrojů Asterisk byla podrobena delšímu testování, aby bylo ověřeno, zda vrací skutečně aktuální, nejméně vytížený zdroj. Pro tyto účely jsem využíval testovací nástroj Postman, který je určen právě pro vývoj a testování API.

V tuto chvíli máme zprovozněno REST API spolu s uWSGI a Ngnix a také službu pro automatické zasílání informačních HTTP PUT zpráv z aplikačních serverů. Zbývá nám tedy nakonfigurovat Kamailio, a to tak, aby při příchodu SIP žádosti do systému zaslalo HTTP GET dotaz a na základě obsahu odpovědi provedlo směrování SIP žádosti na konkrétní aplikační server. Tato konfigurace bude předmětem poslední sekce této kapitoly.

6.3.4 Úprava hlavního konfiguračního souboru Kamailia

V této sekci budeme navazovat na základní konfiguraci provedenou v kapitole 6.1. Dříve než však s konfigurací začneme, musíme doinstalovat potřebné balíčky, jelikož Kamailio ve výchozím stavu neobsahuje *http_client* ani *JSON* modul, přičemž druhý ze zmíněných modulů budeme používat ke zpracování HTTP odpovědí. Potřebné balíčky proto nainstalujeme příkazy:

```
sudo apt update
sudo apt install kamailio-utils-modules kamailio-json-modules libcurl-dev libjson-c-dev
```

Přesuneme se do hlavního konfiguračního souboru Kamailia a provedeme načtení potřebných modulů níže uvedeným způsobem.

```
loadmodule "http_client.so"
loadmodule "json.so"
```

V sekci parametrů modulu potřebujeme pouze nadefinovat spojení na naše RESP API. Vytvoříme proto spojení s názvem *api*, nastavíme jeho základní URL a maximální dobu čekání na odpověď nastavíme na tři sekundy:

```
modparam("http_client", "httpcon", "api=>http://158.196.244.168;timeout=3")
```

Podobně jako v případě konfigurace *Dispatcher* modulu si vytvoříme vlastní směrovací blok, který budeme volat z hlavního směrovacího bloku *request_route*:

```
request_route {
    ...

    # Volání směrovacího bloku pro dynamickou distribuci zátěže
    route(DYNAMIC_DISPATCH);
}
```

Směrovací blok *DYNAMIC_DISPATCH* je příliš rozsáhlý, než abychom ho zde mohli umístit celý a následně si ho pospat. Z tohoto důvodu si popíšeme pouze jeho klíčové části, přičemž celý konfigurační soubor zahrnující i některé další dodatečné úpravy jako přesun, či smazání určitých částí výchozího konfiguračního souboru bude součástí elektronické přílohy.

Na začátku směrovacího bloku *DYNAMIC_DISPATCH* nejprve ověřujeme, zda se jedná o žádost INVITE, jelikož pro jiný typ žádosti nechceme funkce našeho bloku vykonávat. Pokud se o žádost INVITE nejedná, opustíme směrovací blok.

```
route[DYNAMIC_DISPATCH] {
    if(method=="INVITE") {
        ...
    } else {
        exit;
    }
}
```

Pokud je podmínka splněná a jedná se o INVITE, pokračujeme uvnitř bloku *if* těmito úkony:

```
http_connect("api", "/asterisk", "$var(result)");
$var(test) = $(var(result){s.substr,1,5});

if ($var(test) == "empty") {
    xlog("L_WARN", "All database records are old, Asterisk servers are probably down");
    sl_send_reply("503", "Service Unavailable");
    exit;
} else {
    ...
}
```

Prvním příkazem provedeme zaslání žádosti HTTP GET na naše REST API a odpověď na žádost uložíme do proměnné *$var(result)*. Žádost je v tomto případě určena pro skupinu zdrojů chrakterizovanou URL */asterisk*. Příkazy na druhém řádku vyvoláme parsování proměnné *$var(result)*, a to konkrétně tak, že vyjmeme obsah počínaje indexem 1 s délkou 5 znaků. Výsledek parsování uložíme do proměnné *$var(test)*. V podmínce *if* následně ověřujeme, zda obsah proměnné *$var(test)* odpovídá hodnotě *"empty"*. Pokud ano, znamená to, že při prohledávaní databáze nebyl nalezen žádný aktuální záznam, který by splňoval podmínku, že je maximálně šest sekund starý. To jinými slovy znamená, že žádný aplikační server (Asterisk) nejméně po dobu šesti sekund neaktualizoval svůj stav a je proto pravděpodobné, že došlo k selhání všech Asterisk serverů nebo služby pro zasílání informačních zpráv. Do logu si proto vypíšeme varovné hlášení, že došlo k této situaci a odesílateli žádosti INVITE zašleme bezestavově (jelikož ještě nedošlo k vytvoření transakce) zprávu *503* -

Service Unavailable. Pokud obsah proměnné *$var(test)* neodpovídá řetězci *empty*, dojde k vykonání části *else*, jejíž obsah si definujeme za chvíli. Pojďme se ještě vrátit k onomu parsování HTTP odpovědi. Bylo by jistě vhodnější z REST API zasílát informační zprávu *"empty"* v klasickém JSON formátu jako pár *"key": "value"* a následně ji v Kamailiu zpracovávat pomocí funkce určené pro práci s daty v JSON formátu. V praktických testech se však tento postup ukázal jako problémový, protože bychom v každé odpovědi hledali klíč ("key"), který je ale zasílán pouze v případě, kdy je výsledek prohledávaní databáze prázdný. Při nepřítomnosti tohoto klíče v HTTP odpovědi docházelo k selhání Kamailia a jeho chybnému chování. Z tohoto důvodu jsem výhradně pro toto ověřování musel použít parsování na základě obsahu na konkrétním indexu.

Vraťme se ale zpět ke konfiguraci. V tuto chvíli máme ověřeno, že se jedná o zprávu INVITE a že obsah odpovědi není prázdný ("empty"). V části *else* můžeme nyní přistoupit k samotnému určení cíle, na který má být žádost INVITE směrována. Do části *else* umístíme následující konfiguraci:

```
json_get_string("$var(result)", "id", "$var(id)");
json_get_string("$var(result)", "load", "$var(load)");

if ($var(id) == 1) {
    if ($var(load) > 94) {
        xlog("L_WARN", "Asterisk 1 is overloaded");
        sl_send_reply("503", "Service Unavailable");
        exit;
    } else {
        $du = "sip:158.196.244.169:5060;transport=udp";
        t_on_failure("AS1_FAILURE");
        route(RELAY);
    }
} else if ($var(id) == 2) {
    ...
} else if ($var(id) == 3) {
    ...
} else {
    xlog("L_WARN", "Unknown ID in GET response");
    sl_send_reply("503", "Service Unavailable");
    exit;
}
```

Z HTTP odpovědi, kterou jsme si již dříve uložili do proměnné *$var(result)*, vyjmeme pomocí funkce *json_get_string* hodnotu atributů *id* a *load* zapsaných v JSON formátu a uložíme je do proměnné *$var(id)* respektive *$var(load)*. V dalším kroku už pouze ověřujeme, jaké hodnoty proměnná

id nabývá. Pokud hodnota odpovídá 1 víme, že REST API na základě prohledání databáze určilo jako aktuální, nejméně vytížený cíl Asterisk 1, viz přidělení ID jednotlivým aplikačním serverům sekce 6.3.3. Databáze však neověřuje hodnotu atributu *load*. Proto v dalším kroku porovnáváme, zda je hodnota tohoto atributu větší než 94, pokud ano, blížíme se k přetížení serveru. Z tohoto důvodu vypíšeme do logu informační zprávu administrátorovi a odesílateli žádosti INVITE zašleme bezestavovou odpověď s kódem 503. Stanovení meze, při které je vybraný cíl označen jako přetížený, vychází z praktických zkoušek systému. Musíme totiž počítat s tím, že v průběhu intervalu s jakým aplikační servery v danou chvíli obnovují svůj stav pomocí HTTP PUT zpráv, může do systému přijít relativně velké množství SIP žádostí INVITE, které budou poslány na totožný, v tu chvíli nejméně vytížený aplikační server Množství SIP žádostí zaslaných v jednu chvíli na jeden konkrétní cíl se odvíjí od aktuálního vytížení aplikačních serverů a tedy i intervalu s jakým aktualizují svůj stav. Je tedy nezbytné mít určitou rezervu pro případ, kdy je aplikační server blížící se mezi maximálního vytížení vybrán za cíl. V našem případě je rezerva cca. 5% výkonu CPU. Pokud k překročení meze vytížení procesoru nedojde, nastavíme proměnou *$du (destination uri)* na hodnotu obsahující IP adresu příslušného aplikačního serveru, viz adresace na obrázku 6.1. Následně definujeme *failure route* blok pro případ, že by z pohledu SIP komunikace došlo k selhání. V závěru provedeme volání směrovacího bloku *RELAY*, který zajistí odeslání odpovědi obdobně, jako tomu bylo v případě konfigurace s využitím *Dispatcher* modulu.

Původním záměřem při vypracovávání této konfigurace bylo vyhodnocovat vytížení CPU i v rámci desetiných míst. Zde jsem však narazil na problém, jelikož Kamailio neumožňuje práci s hodnotami typu *float*. Byl jsem tedy odkázán pouze na vyhodnocování v rámci celých čísel. Někdo by mohl namítnout, že překročení maximální meze vytížení CPU bychom mohli detekovat již v rámci prohledávaní databáze, to jistě ano, ale při stávajícím nastavení REST API bychom tak přišli o informaci, která ze dvou možných situací nastala. Na straně Kamailia bychom proto nevěděli zda jsou databázové záznamy staré anebo jsou aplikační servery přetíženy.

Vraťme se ale zpět k poslednímu výpisu kódu. Proces zpracování a ověření, který je ve výpisu kódu demonstrován na případě, kdy dojde ke splnění podmínky, kde je hodnota pole ID rovna 1 je obdobný i pro zbývající aplikační servery charakterizované ID 2 a 3, a proto není ve výpisu uveden. Odlišnost logicky najdeme pouze v proměné *$du*, změně ve výpisu do logu a v názvu *failure route* bloku, který je definován zvlašť pro každý aplikačních server. Pro přehlednost jsou názvy *failure route* bloků pro jednotlivé aplikační servery uvedny na následujícím seznamu.

- **AS1_FAILURE** - failure route blok pro aplikační server Asterisk 1
- **AS2_FAILURE** - failure route blok pro aplikační server Asterisk 2
- **AS3_FAILURE** - failure route blok pro aplikační server Asterisk 3

Podobu záložní cesty budeme demonstrovat na *failure route* bloku pro aplikační server Asterisk 1. Definování záložní cesty s názvem *AS1_FAILURE* tedy vypadá takto:

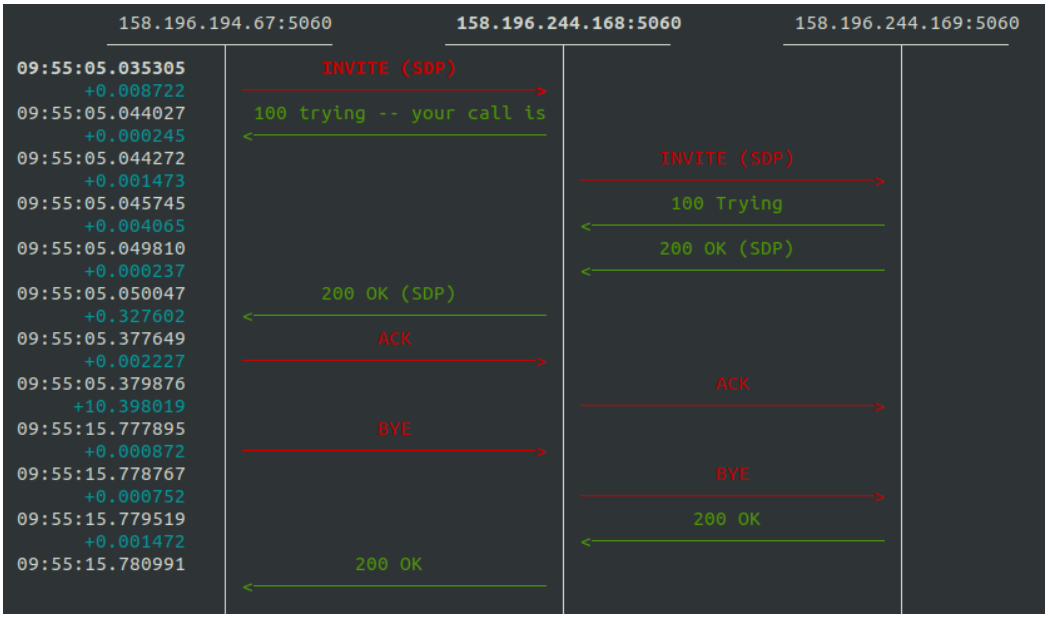
```
failure_route[AS1_FAILURE] {
    xlog("L_WARN", "Failover for Asterisk 1, trying Asterisk 2");
    http_connect("api", "/asterisk/2", "$var(as2)");
    $var(test_as2) = $(var(as2){s.substr,1,5});

    if ($var(test_as2) == "empty") {
        xlog("L_WARN", "Database record for backup Asterisk 2 is old");
        t_reply("503", "Service Unavailable");
        exit;
    } else {
        json_get_string("$var(as2)", "load", "$var(as2_load)");

        if ($var(as2_load) > 94) {
            xlog("L_WARN", "Backup Asterisk 2 is overloaded");
            t_reply("503", "Service Unavailable");
            exit;
        } else {
            $du = "sip:158.196.244.174:5060;transport=udp";
            route(RELAY);
        }
    }
}
```

Záložní cesty jednotlivých aplikačních serverů fungují tak, že v případě, kdy cíl vybraný v bloku *DYNAMIC_DISPATCH* odpoví na INVITE zprávou s kódem 4xx, 5xx nebo 6XX, dojde k volání *failure route* bloku. Voláním tohoto bloku se pokusíme přesměrovat hovor na následující aplikační server. V případě bloku *AS1_FAILURE* se pokusíme hovor přesměrovat na Asterisk 2. Pokud dojde k volání bloku *AS2_FAILURE* je hovor přesměrován na Asterisk 3 a v případě bloku *AS3_FAILURE* se logicky pokoušíme hovor směrovat na Asterisk 1. V rámci jednoho hovoru může být volán maximálně jeden *failure route* blok. Pokud se tedy hovor nepovede spojit skrze hlavní blok *DYNAMIC_DISPATCH*, má poslední možnost v podobě záložní cesty.

Co se obsahu bloku *AS1_FAILURE* týče, je proces podobný jako v případě bloku *DYNAMIC_DISPATCH*. Hlavním rozdílem je zasílání HTTP GET žádostí na konkrétní zdroj chrakterizovaný jeho ID a nikoliv na skupinu zdrojů. V případě bloku *AS1_FAILURE* tak zasíláme GET žádost na URL */asterisk/2*, čímž získáme informace o aplikačním serveru Asterisk 2, který je v tuto

chvíli naší zálohou. Obdobně jako v bloku *DYNAMIC_DISPATCH* ověřujeme zda zpráva obsahuje hodnotu *"empty"* a jestli není založní Asterisk přetížený. Pokud by byl přetížen, je uživateli zaslána zpráva s kódem 503 a hovor je tím nadobro ukončen bez jakéhokoliv dalšího pokusu o sestavení spojení. Pozorný čtenář jistě postřehl, že nyní na rozdíl od bloku *DYNAMIC_DISPATCH* zasíláme zprávy pomocí funkce *t_reply()*, a to z toho důvodu, že při prvním pokusu o sestavení spojení došlo k vytvoření transakce. Podoba konfigurace jednotlivých *failure route* bloků je, co se týče použitých funkcí a operací, obdobná. Rozdíl je pouze v názvu proměných, IP adrese obsažené v proměné *$du* a příkazech obsahujících informace o ID aplikačního serveru.

Tímto je konfigurace pro dynamickou distribuci zátěže hotová. Můžeme se proto přesunout k praktickému ověření funkčnosti celého systému. Aby se však změny provedené v konfiguračním souboru *kamailio.cfg* projevily, je nezbytné službu restartovat příkazem *sudo systemctl restart kamailio*. SIP dialog testovacího hovoru z pohledu Kamailia je uveden na následujícím obrázku.



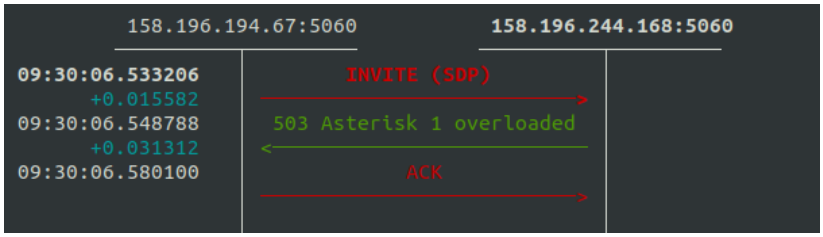
Obrázek 6.13: Test vlastního rešení pro distribuci zátěže z pohledu SIP Proxy Kamailio

Vidíme, že průběh komunikace je obdobný jako v případě distribuce zátěže s využitím *Dispatcher* modulu. Nyní si však ověříme chování systému i v některých krajních situacich, ve kterých by měla být uživateli zaslána zpráva *503 - Service Unavailable*. Jelikož je text zprávy pro různé situace vždy stejný, může být obtížné rozpoznat, která ze situací nastala. Proto pro účely ověření funkčnosti přepíšeme text zprávy *"Service Unavailable"* na řetězec, na základě kterého budeme schopni jasně určit vzniklou situaci. Například pokud je výsledek prohledávaní databáze prázdný a HTTP odpoveď obsahuje řetězec *"empty"*, zašleme uživateli SIP zprávu ve tvaru *"503 All DB records are old"*. Dále si ověříme i fungování záložní cesty a správné vyhodnocování podmínek, které se v ní nacházejí.

Podoba SIP komunikace v situaci, kdy jsou všechny databázové záznamy příliš staré:

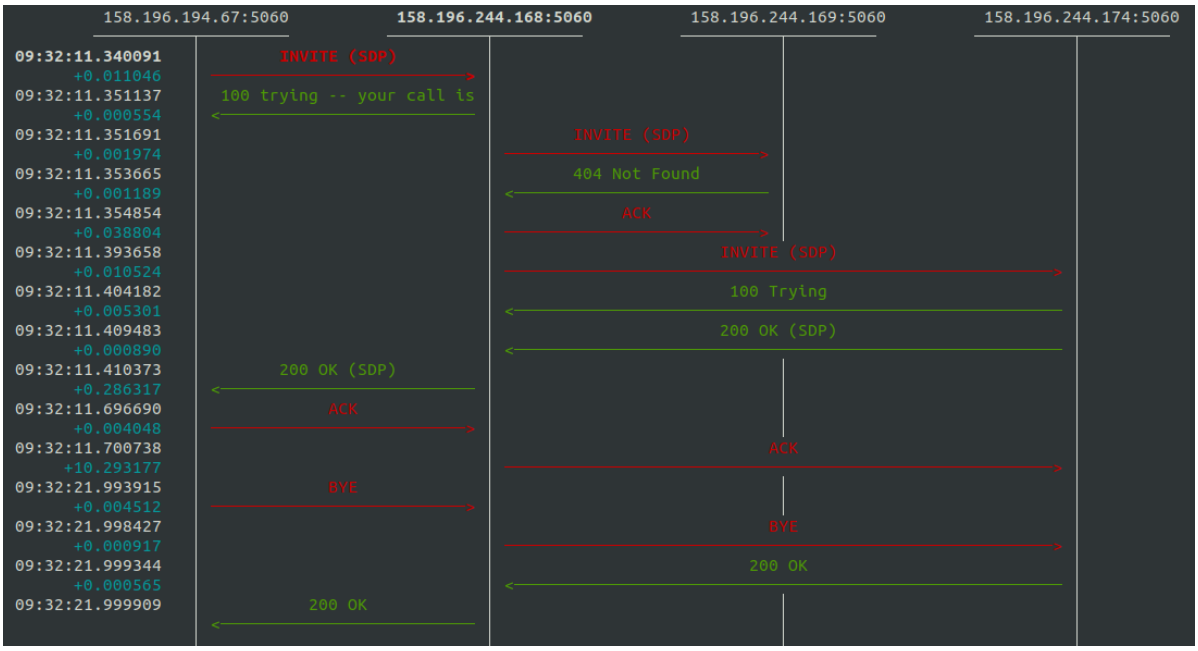
Obrázek 6.14: Průběh SIP komunikace v případě, kdy jsou všechny databázové záznamy staré

Podoba SIP komunikace v situaci, kdy se cíl vybraný pomocí REST API blíží stavu přetížení:



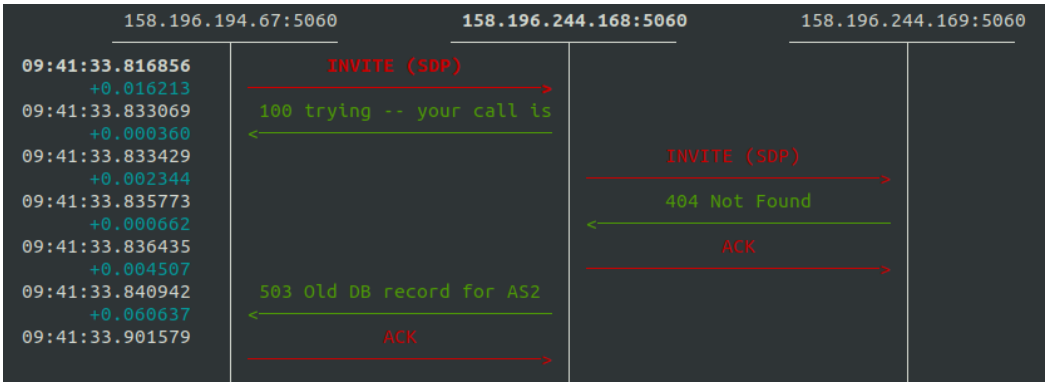
Obrázek 6.15: Průběh SIP komunikace v případě, kdy je vybraný Asterisk označen jako přetížený

Podoba SIP komunikace a demonstrace fungování *failover* funkce při volání záložní cesty pro Asterisk 1, jestliže záložní Asterisk 2 splnil veškeré podmínky:



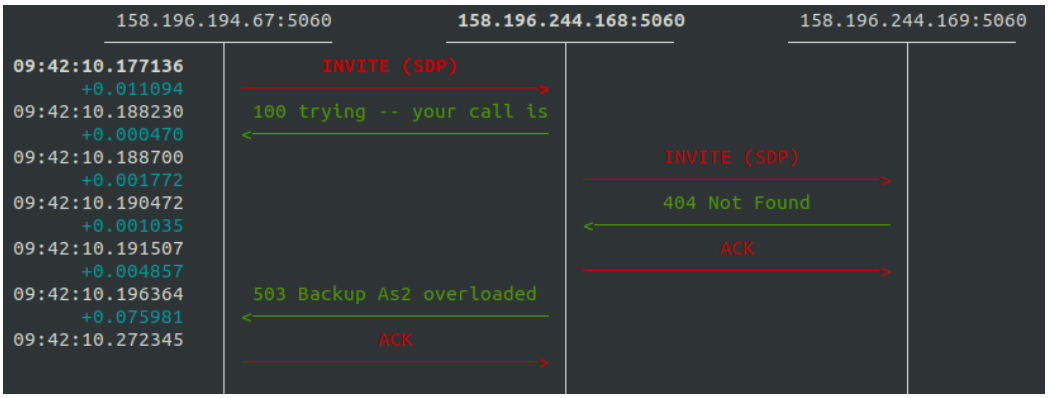
Obrázek 6.16: Demonstrace failover funkce pro konfiguraci využívající vlastní distribuční řešení

Podoba SIP komunikace v případě, kdy byl jako cíl určen Asterisk 1, avšak z pohledu SIP došlo při sestavování spojení k selhání, následkem čehož proběhlo uvnitř Kamailia volání záložní cesty pro Asterisk 1, ale bylo zjištěno, že databázový záznam záložního Asterisku 2 je příliš starý:



Obrázek 6.17: Průběh SIP komunikace, jestliže je databázový záznam záložního Asterisku starý

Podoba SIP komunikace v případě, kdy byl jako cíl určen Asterisk 1, avšak z pohledu SIP došlo při sestavování spojení k selhání, následkem čehož proběhlo uvnitř Kamailia volání záložní cesty pro Asterisk 1, ale bylo určeno, že se záložní Asterisk 2 blíží stavu přetížení:



Obrázek 6.18: Průběh SIP komunikace, jestliže je záložní Asterisk označen jako přetížený

Nyní máme ověřeno, že je systém schopen správně vyhodnotit vzniklou situaci a informovat o ní jak uživatele pomocí SIP zprávy, tak i administrátora systému prostřednictvím logu. Konfigurace je v tuto chvíli zcela dokončena a připravena k testování. Na závěr si však shrňme hlavní rysy námi navrhnutého systému. Povedlo se nám navrhnout a nakonfigurovat RESTful aplikaci a úspěšně ji propojit s uWSGI a webovým serverem Nginx. S pomocí těchto prvků jsme schopni provádět výměnu informací o aktuálním stavu vytížení procesoru mezi Asterisk servery a Kamailiem, přičemž informace jsou vyměňovány prostřednictvím HTTP PUT žádostí v proměnném intervalu, který se odvíjí právě od úrovně vytížení.

S každou novou SIP žádostí INVITE vstupující do systému se Kamailio pomocí HTTP GET dotáže naší RESTful aplikace, která mu v odpovědi zašle informace o vybraném Asterisku, který splnil podmínky, že je jeho databázový záznam aktuální a ze všech ostatních Asterisk serverů jejichž databázový záznam je aktuální je právě on nejméně vytížený. Pokud databáze neobsahuje žádný aktuální záznam, informujeme o tom Kamailio skrze textovou informaci obsaženou v odpovědi. Způsobem, jakým ověřujeme aktuálnost záznamů jsme v podstatě nahradili funkci *Dispatcher modulu*, který pomocí SIP OPTIONS zjišťuje dostupnost aplikačních serverů. V našem případě je tato funkcionalita zakomponovaná přímo do HTTP PUT žádostí. Pokud naše aplikace příjme z daného Asterisk serveru HTTP PUT žádost alespoň jednou za 6 sekund, považuje onen cíl za aktivní a v tom případě na něho mohou být hovory směrovány. Ještě než se tak stane, ověří Kamailio, že se vybraný Asterisk neblíží stavu přetížení a pro případ selhání komunikace z pohledu SIP protokolu nastaví pro daný cíl jeho záložní cestu.

Jednou z dalších funkcí, pomocí které bylo v plánu navrhnutý systém vylepšit, byla zabezpečená forma komunikace prostřednictvím TLS. Aplikace měla být dostupná přes protokol HTTPS, čímž by bylo zajištěno šifrování všech HTTP zpráv. Bohužel zde jsem narazil na problém s TLS modulem Kamailia, při jehož nasazení a pokusech o spuštění SIP serveru docházelo k řadě kritických chyb týkajících se operační paměti. Jelikož tato funkcionalita není nezbytným předmětem obsahu této práce, rozhodl jsem ji po řadě pokusů vynechat.

V závěrečné části této práce se přesuneme k testování námi vytvořeného systému pro distribuci zátěže. Naše pozornost bude primárně směrována na schopnost systému rovnoměrně rozprostírat zátěž v ohledu na stav vytížení jednotlivých aplikačních serverů.

Kapitola 7

Testování a vyhodnocení výsledků

V této kapitole budeme demonstrovat schopnost námi navrženého systému distribuovat zátěž mezi trojici aplikačních serverů. Provedeme sérii testů, jejichž úkolem bude jasně prokázat, že je systém schopen automaticky určit v danou chvíli nejméně vytížený aplikační server a tím poskytnout maximální efektivitu a stabilitu systému jako takového.

Abychom však mohli schopnosti naší konfigurace ukázat v praxi, potřebujeme mít v první řadě dostatečné množství požadavků, které budou pro náš systém představovat vstupní zátěž. Nástroj, který je osvědčený a pro tuto situaci naprosto vhodný nese název SIPp. V další části této kapitoly si proto tento nástroj nainstalujeme a uvedeme do chodu.

7.1 Nástroj SIPp

SIPp je open source nástroj určený pro generování SIP provozu a testování SIP infrastruktury. Zprávy a obsah jednotlivých SIP zpráv, které jsou tímto nástrojem posílány, jsou definovány uvnitř XML (eXtensible Markup Language) souboru. Z pohledu SIP prvků je pomocí SIPp simulováno chování UAC a UAS. My si v této práci vystačíme pouze s UAC, jelikož role UAS bude zajišťována skrze námi nadefinované chování aplikačních serverů. [53]

Nástroj SIPp je možné stáhnout z repozitářů ve formě balíčku s názvem *sip-tester*, avšak pouze v jeho zastaralé verzi. Pokud chceme nasadit některou z aktuálních verzí, nezbývá nám nic jiného než provést kompilaci ze zdrojových kódu. V naší testovací topologii budeme SIPp spouštět ze stejného serveru, na kterém provozujeme Kamailio, uWSGI a NGINX. Z tohoto důvodu disponuje výhradně tento server (.168) jedním jádrem procesoru navíc. Dříve než se pustíme do kompilace nástroje SIPp, nainstalujeme potřebné prerekvizity.

```
sudo apt update
sudo apt install libncurses-dev libpcap-dev libnet-dev build-essential
```

Balíček *build-essential* jsme na serveru 158.196.244.168 již instalovali, viz kapitola 6.3.2. Je zde však uveden, aby byl výčet všech potřebných doplňků kompletní. V dalším kroku provedeme stažení a extrahování souborů nástroje SIPp ve verzi 3.6.0 pomocí příkazů:

```
wget https://github.com/SIPp/sipp/releases/download/v3.6.0/sipp-3.6.0.tar.gz
tar -xvf sipp-3.6.0.tar.gz
cd sipp-3.6.0
```

V našem testování budeme chtít využívat schopnost SIPp zasílat mediální tok v podobě RTP paketů např. skrze PCAP nahrávku. Ověříme proto, zda systém obsahuje všechny potřebné doplňky pro chod této funkce a následně provedeme samotnou kompilaci a instalaci nástroje SIPp. [54]

```
./configure --with-pcap
make
```

Nyní je SIPp nainstalován a připraven. Po nainstalování je k dispozici sada ukázkových scénářů a PCAP nahrávek. My si však v hlavní složce s názvem */sipp-3.6.0* vytvoříme vlastní scénář.

```
nano /sipp-3.6.0/testing_scenario.xml
```

Celý scénář včetně vlastní PCAP nahrávky bude součástí elektronické přílohy. My si v této kapitole představíme pouze stěžejní vlastnosti a části scénáře. Při sestavování spojení budeme chtít logicky jako první zaslat SIP žádost INVITE. Její definice uvnitř XML souboru vypadá takto: [54]

```
<send retrans="500">
<![CDATA[
    INVITE sip:asterisk@[remote_ip]:[remote_port] SIP/2.0
    Via: SIP/2.0/[transport] [local_ip]:[local_port];branch=[branch]
    From: <sip:sipp@[remote_ip]>;tag=[call_number]
    To: <sip:asterisk@[remote_ip]>
    Contact: sip:sipp@[local_ip]:[local_port]
    Call-ID: [call_id]
    CSeq: [cseq] INVITE
    Max-Forwards: 70
    User-Agent: SIPp
    Content-Type: application/sdp
    Content-Length: [len]

    ...
]]>
</send>
```

Vidíme, že XML podobně jako HTML používá párové značky (tagy). Například pro definování nové zprávy použijeme *<send> </send>*. Do jednotlivých polí SIP záhlaví jsou dosazovány buďto statické hodnoty, nebo hodnoty získané z přepínačů při spouštění scénáře. Ke zprávě INVITE, kterou jsme si v párovém tagu *<send> </send>* nadefinovali, přidáme ještě obsah SDP protokolu.

```
v=0
o=user1 53655765 2353687637 IN IP[local_ip_type] [local_ip]
s=-
c=IN IP[local_ip_type] [local_ip]
t=0 0
m=audio [auto_media_port] RTP/AVP 8
a=rtpmap:8 PCMA/8000
```

Náš UAC v podobě SIPp bude podporovat pouze kodek G.711 (A-law). Na námi zaslanou žádost INVITE budeme následně očekávat určité odpovědi, jejichž výčet specifikujeme následovně: [54]

```
<recv optional="true" response="100"></recv>
<recv response="200" rtd="true" rrs="true"></recv>
```

Zprávu *100 Trying* označíme jako volitelnou (optional), což znamená, že pokud táto zpráva dorazí, není nástrojem SIPp označena jako neočekávaná. Pro klasickou komunikaci bychom zde definovali i zprávu *180 Ringing*, ale jelikož roli UAS hraje Asterisk, který tuto zprávu nezasílá, není toto nastavení nutné. Hlavní očekávanou odpovědí je zpráva *200 OK*, u které musíme nastavit parametr *rrs* na hodnotu *true*, čímž dojde k uložení hodnoty z pole *Record-route* obsaženého v přijaté odpovědi. Nastavení tohoto parametru je nezbytné, jelikož Kamailio vyžaduje své setrvání v signalizaci až do konce SIP dialogu. Následně můžeme potvrdit přijetí finální odpovědi pomocí ACK.

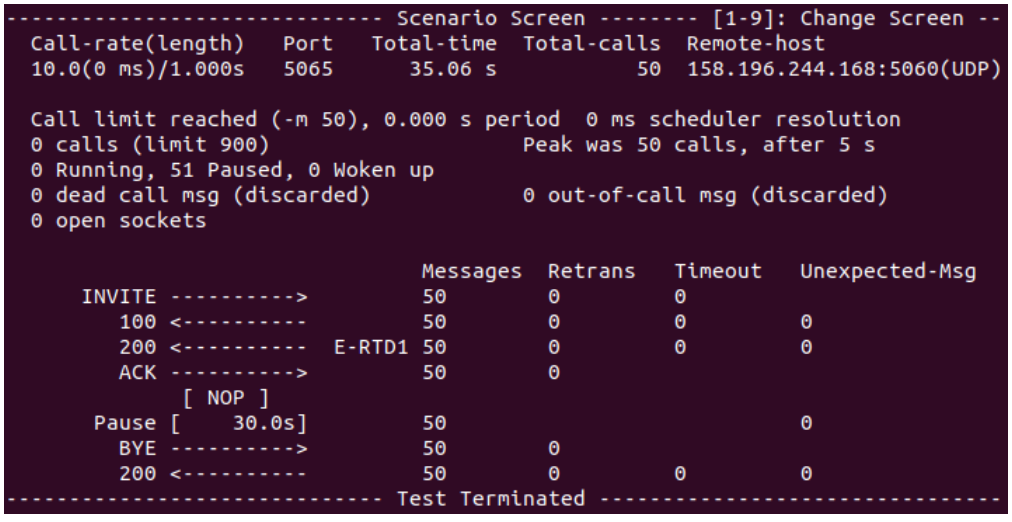
```
<send>
<![CDATA[
    ACK [next_url] SIP/2.0
    [routes]
    Via: SIP/2.0/[transport] [local_ip]:[local_port];rport;branch=[branch]
    From: <sip:sipp@[remote_ip]>;tag=[call_number]
    To: <sip:asterisk@[remote_ip]>[peer_tag_param]
    Contact: sip:sipp@[local_ip]:[local_port]
    Call-ID: [call_id]
    CSeq: [cseq] ACK
    Max-Forwards: 70
    Content-Length: 0
]]>
</send>
```

Pro zasílanou žádost ACK je klíčové použití bloku *[routes]*, čímž dojde k načtení hodnot, které jsme si ve zprávě 200 OK uložili pomocí parametru *rrs = "true"*. V našem případě do tohoto pole načteme adresu Kamailia. Nástroj SIPp nabízí řadu doplňkových funkcí, jako například podporu pro IPv6 či TLS, autentizaci uživatelů a mnoho dalších. Co je pro nás však důležité je to, že SIPp umožňuje generovat RTP tok prostřednictvím PCAP nebo WAV nahrávky. Pro tyto účely jsem si vytvořil vlastní nahrávku ve formátu G.711 A-law. Spuštění nahrávky provedeme takto: [53, 54]

```
<nop>
    <action>
        <exec rtp_stream="pcap/g711a.pcap"/>
    </action>
</nop>
<pause milliseconds="30000"/>
```

Tím máme zajištěn zdroj mediálního toku v podobě vlastní nahrávky, jejíž délku jsme nastavili na 30 sekund. Jakmile přehrávání skončí, můžeme hovor ukončit pomocí žádosti BYE. Struktura této žádosti je obdobná sjako u žádosti ACK, pochopitelně s rozdílem v názvu žádosti. Na zaslanou žádost BYE očekáváme potvrzení v podobě zprávy 200 OK, u které ale nyní nemusíme použít parametr *rrs = "true"*, jelikož je to odpověď ukončující celý dialog. V tuto chvíli můžeme ověřit funkčnost scénáře pomocí níže uvedeného příkazu, ve kterém v první řadě specifikujeme IP adresu a port našeho SIP Proxy serveru a následně pomocí přepínačů -i a -p nastavíme IP adresu a port UAC. Na závěr pomocí přepínače -m definujeme maximální počet sestavených hovorů, po kterém se testování ukončí. [54]

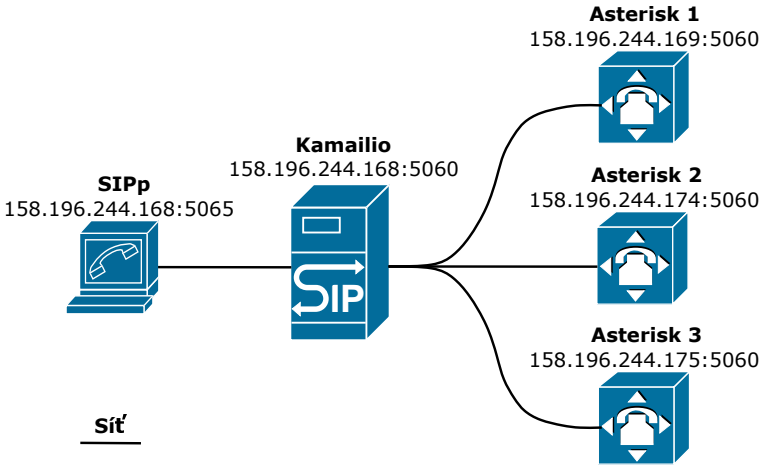
```
./sipp -sf testing_scenario.xml 158.196.244.168:5060 -i 158.196.244.168 -p 5065 -m 50
```



Obrázek 7.1: Ověření funkčnosti vytvořeného scénáře pro nástroj SIPp

7.2 Testovací metodika

V tuto chvíli máme k dispozici veškeré potřebné prvky včetně testovacího nástroje SIPp. Můžeme se proto uchýlit k popisu a realizaci závěrečného testování. Podoba topologie pro závěrečné testování z pohledu SIP protokolu je ukázána na obrázku č. 7.2.



Obrázek 7.2: Schéma topologie pro závěrečné testování

V průběhu testování budeme chtít sestavit relativně velkého množství souběžných hovorů. Toho však není možné dosáhnout s výchozím nastavením operačního systému, které dovoluje mít v jednu chvíli otevřeno maximálně 1024 souborů. Jelikož je v Linuxových operačních systémech vše považováno za soubor, musíme pro účely testování navýšit hodnotu tohoto limitu, a to na všech serverech. Limit lze navýšit buďto dočasně skrze příkaz *ulimit -n*, anebo trvale prostřednictvím souboru */etc/security/limits.conf*. Abychom nemuseli příkaz zadávat opakovaně, zvolíme druhou variantu, přičemž do již zmíněného souboru přidáme následující záznamy:

```
* hard nofile 150000
* soft nofile 150000
```

Na toto nastavení ale nereagovaly Asterisk servery. Důvodem je s velkou pravděpodobností spouštěcí skript, skrze který je za normálních okolností Asterisk spouštěn. Nejsnazší způsob jak tento fakt obejít, je spustit Asterisk skrze jeho binární soubor příkazem:

```
/usr/sbin/asterisk
```

Další věcí, kterou je pro účely testovaní nutné nastavit, je blokování příchozího provozu na UDP porty definované nástrojem SIPp uvnitř pole *m* nacházejícího se v SDP. V naší konfiguraci je generování mediálního toku zajištěno skrze vlastní nahrávku. Na jednotlivých aplikačních serverech je následně nastaveno, aby byl veškerý příchozí mediální tok zaslán jeho zdroji zpět. SIPp však zpětně zaslané RTP pakety nikterak nezpracovává, v důsledku čehož dochází ke generování nemalého

množství ICMP paketů a tedy i k nárůstu vytížení procesoru na Kamailio serveru (.168). Abychom tomuto jevu zabránili, přidáme skrze nástroj *iptables* do systémového firewallu níže uvedené pravidlo.

```
iptables -A INPUT -d 158.196.244.168 -p udp --dport 6000:65535 -j DROP
```

Tímto jsou dokončeny všechny předtestové procedury. Můžeme se proto pustit do popisu jednotlivých testů. Hlavní test, který budeme v této práci realizovat, bude mít za úkol sledovat vytížení CPU a počet v danou chvíli probíhajících hovorů na jednotlivých aplikačních serverech. V první řadě tedy potřebujeme nástroj na sledování vytížení CPU. Tyto informace bychom mohli čerpat skrze HTTP PUT zprávy, které jsou zasílány z jednotlivých aplikačních serverů. Interval v jakém jsou tyto informace zasílány je ale nejednotný, což by značně znesnadnilo následné zpracování a vyhodnocení naměřených hodnot. Pro tyto učely je vhodnější nasazení nástroje SAR, který je určený pro monitorování systémové aktivity. Tento nástroj je dostupný skrze balíček *sysstat*, který nainstalujeme následovně:

```
sudo apt update
sudo apt install sysstat
```

Nástrojem SAR budeme sledovat výhradně vytížení procesoru. Náměřené hodnoty sice mohou být mírně odlišné od hodnot zasílaných skrze HTTP PUT žádosti, avšak to co je pro nás hlavní, je průběh výsledků, který bude zachován. SAR umožňuje definovat periodu, s jakou má být sledovaná veličina odečítána a taky celkový počet měření. Pokud tedy budeme chtít sledovat vytížení CPU každou sekundu po dobu tří minut, použijeme příkaz:

```
sar -u 1 180
```

Spolu s každou naměřenou hodnotou je k dispozici i časová značka. To je pro nás velice důležité, jelikož potřebujeme zajistit synchronizaci mezi hodnotami vytížení CPU a počtem aktuálně probíhajících hovorů. Pro sledování aktuálního počtu probíhajících hovorů použijeme konzoli Asterisku, skrze kterou je možné tuto hodnotu zjistit příkazem:

```
core show calls
```

Z výstupu tohoto příkazu nás bude zajímat hodnota pole *active calls*. Zde však nastává problém spojený se synchronizací. Výpis tohoto příkazu neobsahuje informaci o času, kterému tato hodnota odpovídá. Značku si tedy musíme připojit sami skrze příkaz *date*. Abychom zajistili cyklické odečítání aktivních hovorů, je nezbytné tyto příkazy vykonávat skrze skript. Stejně jako v případě nástroje SAR i zde chceme hodnotu zjišťovat každou sekundu. SAR má vnitřní časovač, který hlídá spouštění daného příkazu v sekundovém intervalu. V případě zjišťování počtu aktivních hovorů ale dochází při velkém vytížení procesoru k prodlevě v odečítání hodnot. Je proto nutné zkrátit interval pro odečítání aktivních hovorů na méně než sekundu, abychom tak kompenzovali zpoždění způsobené vytížením procesoru. S vhodně zvolenou peridou je pak možné zajistit synchronizaci mezi hodnotami pro vytížení CPU a počtem aktivních hovorů.

V případě tohoto testu tedy budou na každém Asterisk serveru k dispozici dva skripty, které jsou rovněž součástí elektronické přílohy. První z nich bude sledovat vytížení CPU, přičemž hodnoty budou zapisovány v sekundovém intervalu do CSV souboru. Druhý skript bude v totožném intervalu provádět odečítání aktuálního počtu aktivních hovorů a i v tomto případě budou výsledky zapisovány do CSV souboru. Mezi spuštěním obou skriptů napříč Asterisky může dojít ke zpoždění, to ale nepředstavuje problém, jelikož všechny hodnoty obsahují časový údaj, který je synchronní napříč jednotlivými servery a pomocí kterého tak bude možné naměřené hodnoty synchronizovat.

V úvodním pozorování bylo zjištěno, že tak jak je systém nastaven je možné provozovat přibližně 1800-1850 souběžných hovorů při délce hovorů 30 sekund. Délka hovorů má na tuto hodnotu významný vliv, jelikož se snižováním délky úměrně roste množství zasílaných SIP zpráv a tedy i vytížení procesoru. Abychom nástrojem SIPp vygenerovali 1800 souběžných hovorů při délce hovorů 30 sekund, je nutné nastavit přepínač *-r* (počet vygenerovaných hovorů za sekundu) na hodnotu 60. S klesající délkou hovorů jsme pro zachování stejného množství souběžných hovorů nuceni hodnotu tohoho přepínače úměrně zvyšovat. V tomto testu budeme pozorovat vliv dvou rozdílných typů vstupního provozu. U prvního z nich budeme generovat krátké hovory (20 s) a u druhého dlouhé hovory (50 s). Maximum souběžných hovorů bude nastaveno na hodnotu 1700. Délka jednotlivých testů bude trvat celkem 150 sekund, přičemž do výsledků bude zahrnuta i doba, ve které systém ještě nedosáhl maxima souběžných hovorů a rovněž doba, kdy počet hovorů klesá k nule. Náběžná a sestupná hrana grafů bude hlavním ukazatelem, jak se systém chová pro různou vstupní zátěž. Příkaz, kterým budeme skrze nástroj SIPp generovat zmíněný vstupní provoz vypadá následovně:

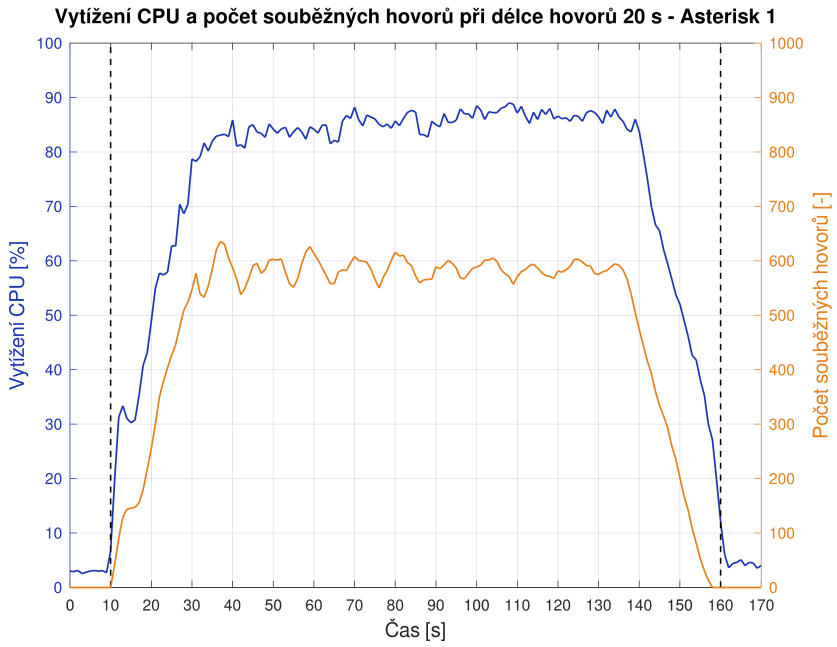
```
./sipp -sf testing_scenario.xml 158.196.244.168:5060 -i 158.196.244.168 -p 5065
       -r 85 -m 11050 -skip_rlimit -mp 6500
```

Hodnota přepínače *-r* se bude odvíjet od délky hovorů a to tak, abychom dosáhli požadovaného množství souběžných hovorů. To stejné platí pro přepínač *-m*, pomocí kterého zajistíme požadovanou délku testu (150 s). Přepínač *-skip_rlimit* dovoluje překročit vnitřní limit nástroje SIPp pro počet otevřených souborů. Přepínač *-mp* udává číslo portu, které bude nastaveno do SDP uvnitř INVITE. Hodnota je záměrně nastavená tak, aby se nacházela uvnitř intervalu blokovaných UDP portů. Výsledky tohoto testu, které jsou uvedené v kapitolách 7.3.1 a 7.3.2, jsou dány průměrem hodnot získaných z celkem deseti testů. Po každém testu proběhlo restartování všech hlavních služeb.

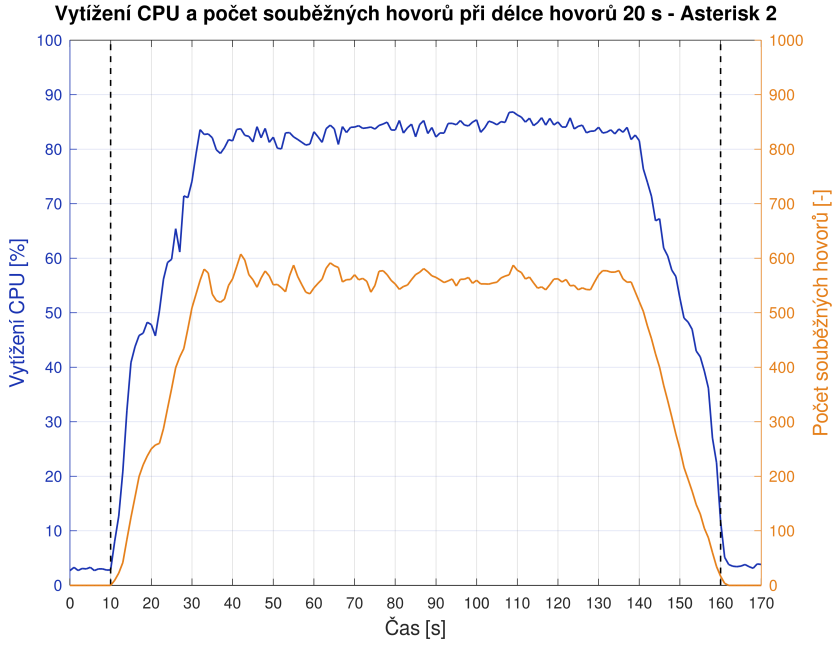
V druhém typu testu, jehož výsledky jsou součástí kapitoly 7.3.3, budeme vyhodnocovat schopnost systému distribuovat zátěž při rozdílné intenzitě vstupního provozu. Budou testovány celkem tři úrovně vstupní intenzity, a to při délce hovorů 30 s. Provoz bude generován obdobným způsobem jako v případě předchozího typu testu. Maximum sestavených hovorů ale bude nastaveno na hodnotu 6000, tudíž v ideálním případě by mělo být na každém Asterisk serveru sestaveno přesně 2000 hovorů. Hodnota celkového počtu sestavených hovorů bude opět odečítána skrze příkaz *core show calls*, ale tentokrát nás bude zajímat hodnota pole *calls processed*.

7.3 Výsledky testování

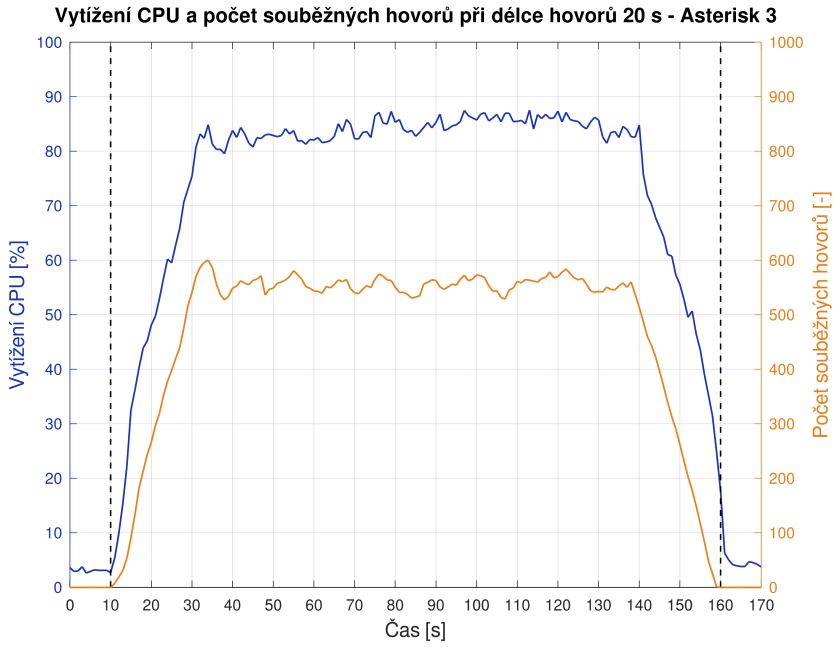
7.3.1 Vytížení CPU a počet souběžných hovorů - délka hovorů 20 sekund



Obrázek 7.3: Výsledky testování pro server Asterisk 1 při délce hovorů 20 s



Obrázek 7.4: Výsledky testování pro server Asterisk 2 při délce hovorů 20 s



Obrázek 7.5: Výsledky testování pro server Asterisk 3 při délce hovorů 20 s

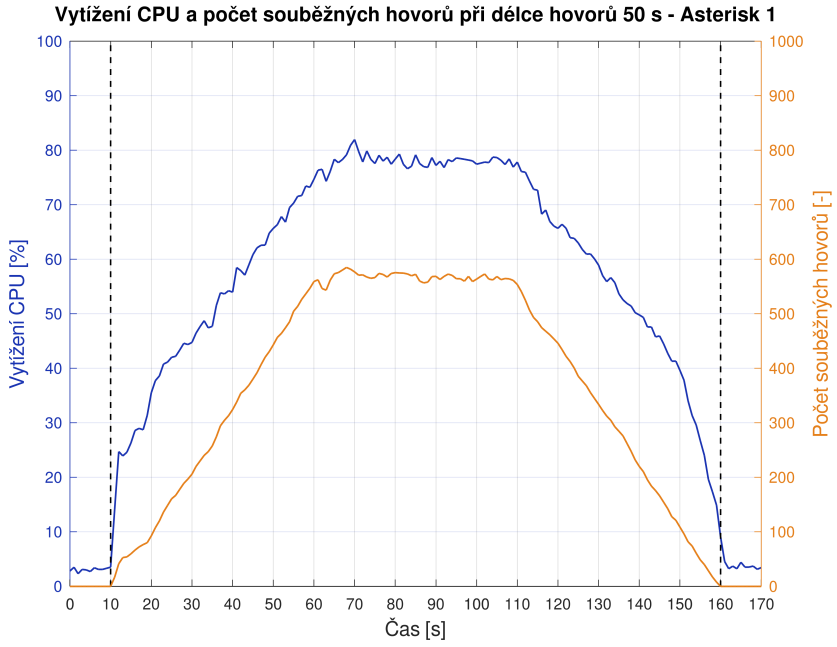
Z výsledku testů znázorněných na výše uvedených grafech, kde je skutečný začátek a konec testu označen svislými čárkovanými čarami je jasně patrný rovnoměrný náběh systému napříč aplikačními servery, viz náběžné hrany grafů. Doba trvání náběžných hran rovněž odpovídá intenzitě vstupního provozu, u kterého dojde po přibližně 20 sekundách od začátku testu k dosažení maximálního počtu souběžných hovorů (1700). Hladina vytížení CPU atakovala v jednotlivých testech hranici 95 %. Výsledným průměrováním hodnot získaných z deseti testů se však průměrné vytížení v době s maximálním počtem souběžných hovorů (31-140 s) pohybovalo okolo hodnoty 85 %. Hodnoty průměrného vytížení CPU a počtu souběžných hovorů získané z časového intervalu 31-140 sekund jsou uvedeny v následující tabulce.

Aplikační server	**Prům. vytížení CPU [%]**	**Prům. souběžných hovorů [-]**
Asterisk 1	85.23	583
Asterisk 2	83.48	560
Asterisk 3	84.18	556

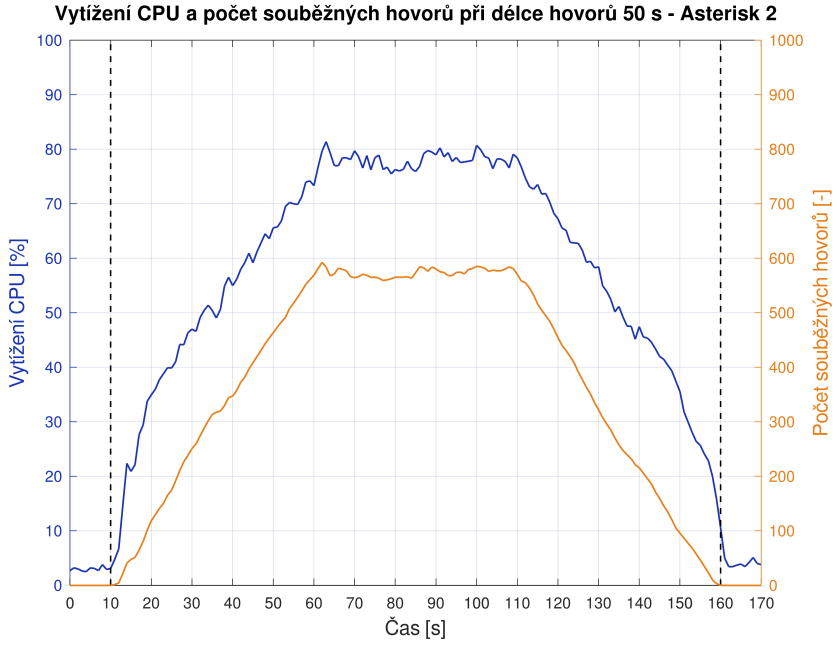
Tabulka 7.1: Průměrné hodnoty získané z výsledků testů při délce hovorů 20 s

Maximální počet souběžných hovorů dosahoval v měřeném úseku hodnoty 1700. Z tohoto důvodu by se měl v ideálním případě průměrný počet souběžných hovorů na každém Asterisk serveru pohybovat kolem hodnoty 567. Z vypočítaného průmeru je patrné, že se této ideální hodnotě všechny výsledky velmi blíží.

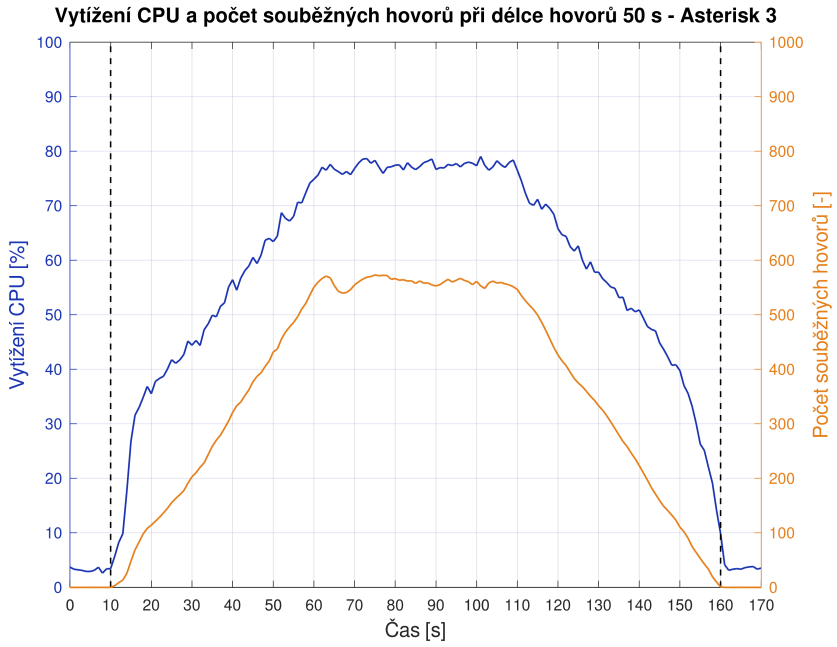
7.3.2 Vytížení CPU a počet souběžných hovorů - délka hovorů 50 sekund



Obrázek 7.6: Výsledky testování pro server Asterisk 1 při délce hovorů 50 s



Obrázek 7.7: Výsledky testování pro server Asterisk 2 při délce hovorů 50 s



Obrázek 7.8: Výsledky testování pro server Asterisk 3 při délce hovorů 50 s

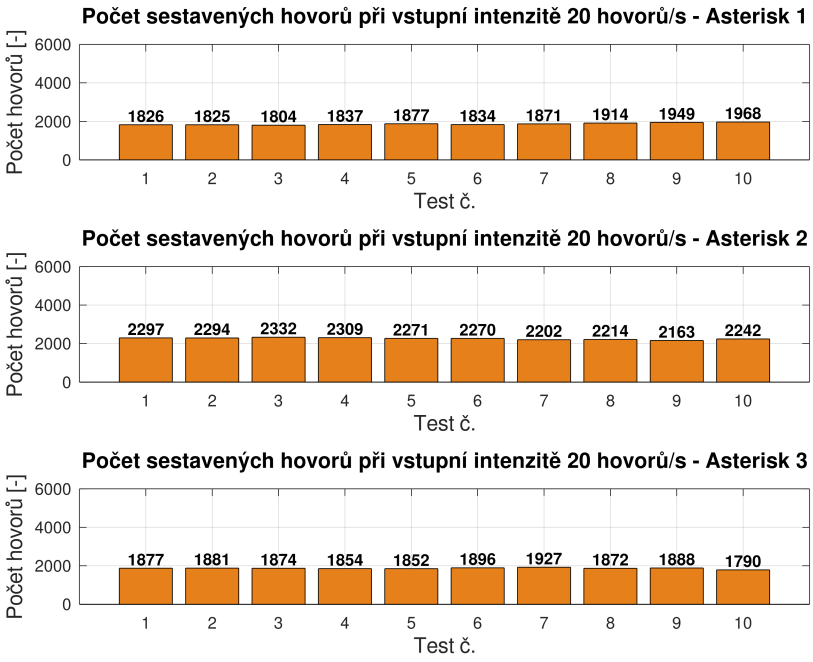
Z výsledků znázorněných na třech výše uvedených grafech je opět zřejmý rovnoměrný náběh systému z klidové fáze až do fáze s maximem souběžných hovorů. Jelikož se jedná o test s hovory o délce 50 sekund, je strmost náběžné hrany značně menší, něž tomu bylo v případě testů s hovory o délce 20 sekund. Systém v tomto případě dosáhne maxima až 50 sekund po začátku testu. Jakmile se počet souběžných hovorů ustálil (61-110 s), pohybovalo se průměrné vytížení CPU okolo hodnoty 80 %. To je přibližně o 5 % méně než u testů s délkou hovorů 20 sekund. Tento rozdíl je dán množstvím hovorů a tedy i SIP zpráv, které do systému vstupovaly každou sekundu. V případě testů s hovory o délce 20 sekund byl přepínač *-r* nastaven na hodnotu 85, zatímco u testů s hovory o délce 20 sekund nabýval hodnoty 34. Intenzita vstupního provozu tedy byla v tomto případě 2.5x menší. Průměrné vytížení CPU a množství souběžných hovorů na jednotlivých aplikačních serverech, které bylo vypočítáno z hodnot v intervalu 61-110 sekund, je uvedeno v následující tabulce.

Aplikační server	**Prům. vytížení CPU [%]**	**Prům. souběžných hovorů [-]**
Asterisk 1	78.01	567
Asterisk 2	78.05	574
Asterisk 3	77.29	560

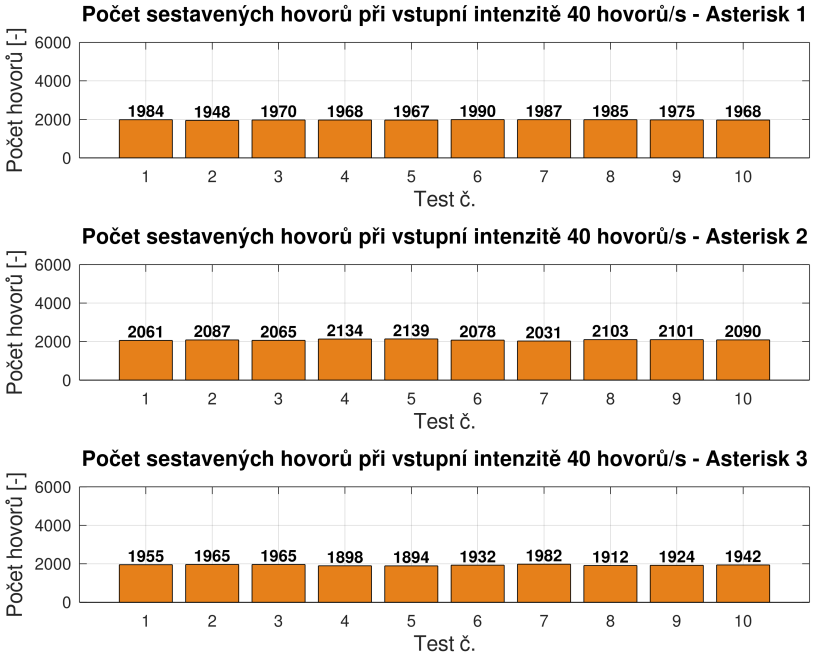
Tabulka 7.2: Průměrné hodnoty získané z výsledků testů při délce hovorů 50 s

Maximální počet souběžných hovorů byl stejný jako v případě testů z kapitoly 7.3.1. Ideálně by tedy měl být počet souběžných hovorů roven 567. Naměřené hodnoty ukazují, že se nejvyšší odchylka od této ideální hodnoty liší o pouhých 7 hovorů, což lze považovat za zanedbatelný rozdíl.

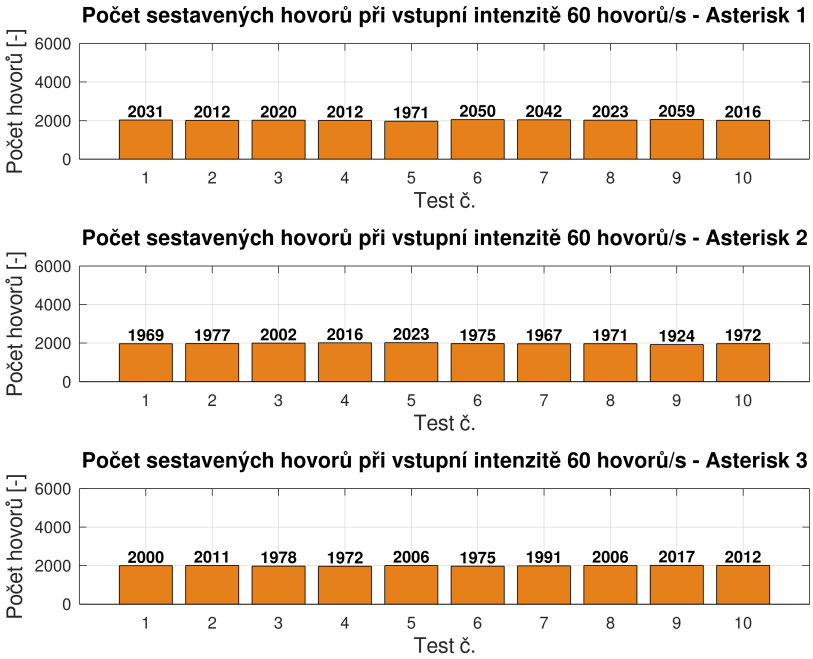
7.3.3 Rozložení hovorů při rozdílné intenzitě vstupního provozu



Obrázek 7.9: Rozložení hovorů při vstupní intenzitě 20 hovorů/s



Obrázek 7.10: Rozložení hovorů při vstupní intenzitě 40 hovorů/s



Obrázek 7.11: Rozložení hovorů při vstupní intenzitě 60 hovorů/s

Smyslem tohoto testu bylo ukázat rozložení a poměr sestavených hovorů napříč aplikačními servery při třech úrovních vstupní intenzity. V testech bylo sestaveno vždy 6000 hovorů, za ideálních okolností by tedy měla být zátěž mezi jednotlivé aplikační servery rozložena v poměru 2000:2000:2000 hovorů. Délka hovorů byla nastavena na hodnotu 30 s. Hodnoty přepínačů *-r* a tomu odpovídající maximální počet souběžných hovorů je uveden v následující tabulce.

Hodnota -r [hovorů/s]	**Max. souběžných hovorů [-]**
20	600
40	1200
60	1800

Tabulka 7.3: Nastavení přepínače -r a tomu odpovídající počet souběžných hovorů

Zejména při nízké intenzitě vstupní zátěže je viditelný nepoměr v rozložení sestavených hovorů. Tento fakt lze přisuzovat nízkému vytížení CPU, od kterého se odvíjí délka intervalu mezi zasíláním HTTP PUT žádostí a tedy i více shlukovité přerozdělování hovorů. V nízkých a středních intenzitách tedy může docházet k jisté prioritizaci některého z aplikačních serverů (v testu server Asterisk 2). Toto chování však nepředstavuje zásadní problém, jelikož k němu dochází v nízkých a středních intenzitách, při vysoké zátěži již systém díky velmi krátkému intervalu pro zasílání informačních zpráv rozděluje zátěž rovnoměrně. To se mimo jiné potvrdilo i posledním testem se vstupní intenzitou 60 hovorů/s, kde bylo rozložení zátěže téměř ideální.

Kapitola 8

Závěr

Hlavním cílem této práce bylo navhrnout a realizovat systém pro distribuci zátěže. Roli hlavního prvku zajišťujícího rozprostírání zátěže plnil SIP server Kamailio. Vstupní zátěž byla v testovací topologii distribuována mezi trojici Asterisk serverů ve funkci aplikačních serverů.

V úvodu praktické části byl věnován prostor současnému řešení pro distribuci zátěže v podobě Dispatcher modulu. Vlastnosti a jednoduchost nasazení dělají z tohoto modulu efektivní a rychlé řešení pro většinu běžných aplikací. K těmto vlastnostem je nutno připočíst i možnost vybrat si způsob distribuce zátěže pomocí některého z celkem 13 algoritmů. To dává administrátorovi systému možnost použít takový způsob distribuce, který bude z pohledu efektivity a výkonnosti pro danou oblast nasazení nejvhodnější.

Dispatcher modul však nezohledňuje jeden velmi důležitý aspekt, a to vytížení jednotlivých aplikačních serverů, mezi kterými je zátěž distribuována. Dochází tedy k pouhému předávání požadávků bez hlubší znalosti skutečného stavu vybrané destinace. V případě nerovnoměrného výpočetního výkonu nebo vytížení napříč aplikačními servery tak může snadno dojít k přetížení a následně i selhání některého/některých z nich. Na tento fakt jsem se proto v práci zaměřil. Výsledkem je vlastní distribuční systém, ve kterém dochází k výměně informací o aktuálním vytížení aplikačních serverů prostřednictvím REST API implementovaného pomocí Python frameworku s názvem Flask. Data jsou ukládána do databáze, kde k nim rovněž skrze REST API přistupuje Kamailio, a to ve chvíli, kdy do systému dorazí žádost na sestavení SIP spojení v podobě SIP INVITE. Kamailio na základě získaných dat odešle SIP žádost na v danou chvíli nejméně vytížený aplikační server. Aby se navrhnutý systém blížil podobě pro produkčního nasazení, byl do systému zakomponován webový server Nginx, skrze který prochází veškerý síťový provoz určený pro komunikaci s REST API.

Vytvořený systém je rovněž schopen ověřovat dostupnost jednotlivých aplikačních serverů. Tato funkce je zakomponovaná do HTTP PUT žádostí, pomocí kterých aplikační servery aktualizují záznam svého stavu, jenž se nachází v databázi. Pokud konkrétní aplikační server neprovede aktua-

lizaci svého stavu alepoň jednou za šest sekund, je systémem při procesu výběru nejvhodnějšího cíle automaticky opomenut. Tím je zajišťeno, že příchozí SIP požadavky nebudou zasílány na server, u kterého pravděpodobně došlo k výpadku či selhání. Z důvodů lepšího využití přenosové kapacity dochází k aktualizaci stavu prostřednictvím HTTP PUT žádostí v proměnném intervalu, který se odvíjí od aktuálního vytížení daného aplikačního serveru. Interval se pohybuje v rozmezí 250-3000 ms, přičemž nejkratší interval je dostatečně krátký i v situaci, kdy je vytížení procesoru velmi vysoké. Pokud se však hladina zátěže aplikačního serveru, který byl označen jako nejméně vytížený dostane přes 94 %, je systém automaticky označen jako přetížený. Tento stav, během kterého systém odmítne všechny požadavky na sestavení hovorů, je platný pouze do chvíle, kdy zátěž některého z aplikačních serverů neklesne pod tuto mez.

Výsledky závěrečného testování prokázaly schopnost systému vhodně rozprostírat zátěž na základě úrovně vytížení aplikačních serverů. Toto tvrzení bylo demonstrováno na dvou testech při 1700 souběžných hovorech a délce hovorů 20 respektive 50 sekund. Hodnoty vykazují srovnatelnou línii náběžné a sestupné hrany napříč aplikačními servery, což jinými slovy říká, že systém distribuuje vstupní provoz správným (rovnoměrným) způsobem. K částečně nerovnoměrné distribuci a shlukovitému přerozdělování zátěže dochází pouze při nízké a z části i střední intenzitě provozu. Tato vlastnost je dána zejména delšími intervaly pro aktualizaci stavu aplikačních serverů, které jsou pro dva zmíněné typy provozu typické. Toto chování ale nepředstavuje pro chod systému žádný problém, protože jak bylo řečeno, dochází k němu primárně při nízkém provozu. V daném nastavení a hardwarové konfiguraci byl systém schopen provozovat 1800-1850 souběžných hovorů o délce 30 sekund, a to s méně než 1 % nesestavených hovorů.

Vytvořený distribuční systém lze považovat za obstojnou alternativu Dispatcher modulu, jehož již zmíněnou negativní vlastnost napravuje. Výsledné řešení sice vyžaduje nasazení většího množství rozdílných služeb a aplikací, ale benefitem za složitější úvodní konfiguraci je schopnost distribuovat zátěž s respektem ke stavu, ve kterém se aplikační servery v danou chvíli nachází.

Literatura

1. JOHNSTON, Alan B. *SIP: understanding the session initiation protocol.* 3rd ed. Boston: Artech House, 2009. ISBN 978-1-60783-995-8

2. WALLINGFORD, Ted. *Switching to VoIP.* Sebastopol, CA: O'Reilly, 2005. ISBN 978-0-596-00868-0.

3. VOZŇÁK, Miroslav. *Technologie a protokoly multimediálních komunikací pro integrovanou výuku VUT a VŠB-TUO.* Ostrava: Vysoká škola báňská - Technická univerzita Ostrava, 2014. ISBN 978-80-248-3326-2.

4. VANĚK, Petr. *Předávání RTP toků v páteřní komunikaci na SIP* [online]. Ostrava, 2019 [cit. 2020-08-12]. Dostupné z: `http://hdl.handle.net/10084/136310`. Bakalářská práce. Vysoká škola báňská - Technická univerzita Ostrava.

5. *RFC 3261 - SIP: Session Initiation Protocol* [online]. [cit. 2020-08-12]. Dostupné z: `https://tools.ietf.org/html/rfc3261`

6. *RFC 3550 - RTP: A Transport Protocol for Real-Time Applications* [online]. [cit. 2020-08-16]. Dostupné z: `https://tools.ietf.org/html/rfc3550`

7. *Kamailio SIP Server* [online]. [cit. 2020-08-18]. Dostupné z: `https://www.kamailio.org/w/`

8. *Features - The Kamailio SIP Server Project* [online]. [cit. 2020-08-18]. Dostupné z: `https://www.kamailio.org/w/features/`

9. MIERLA, Daniel-Constantin a Elena-Ramona MODROIU. *Kamailio SIP Server v3.2.0 Development Guide* [online]. [cit. 2020-08-19]. Dostupné z: `http://www.asipto.com/pub/kamailio-devel-guide/`

10. *Kamailio Architecture, Core and Modules - Telecom R & D* [online]. [cit. 2020-08-20]. Dostupné z: `https://telecom.altanai.com/2014/11/18/kamailio-modules/`

11. *cookbook:5.4.x:pseudovariables [Kamailio SIP Server Wiki]* [online]. [cit. 2020-08-22]. Dostupné z: `https://www.kamailio.org/wiki/cookbooks/5.4.x/pseudovariables`

12. *cookbook:5.4.x:transformations [Kamailio SIP Server Wiki]* [online]. [cit. 2020-08-22]. Dostupné z: https://www.kamailio.org/wiki/cookbooks/5.4.x/transformations

13. *Kamailio Modules* [online]. [cit. 2020-08-23]. Dostupné z: https://kamailio.org/docs/modules/5.4.x/

14. *cookbook:5.4.x:core [Kamailio SIP Server Wiki]* [online]. [cit. 2020-08-24]. Dostupné z: https://www.kamailio.org/wiki/cookbooks/5.4.x/core

15. E. GONCALVES, Flavio a Bogdan-Andrei IANCU. *Building Telephony Systems with OpenSIPS Second Edition.* 2nd ed. Birmingham: Packt Publishing, 2016. ISBN 978-1-78528-061-0.

16. SOKOL, Steven. *Beginning Asterisk - Asterisk Project - Asterisk Project Wiki* [online]. [cit. 2020-08-30]. Dostupné z: https://wiki.asterisk.org/wiki/display/AST/Beginning+Asterisk

17. DAVENPORT, Malcolm. *Asterisk as a Swiss Army Knife of Telephony - Asterisk Project - Asterisk Project Wiki* [online]. [cit. 2020-08-30]. Dostupné z: https://wiki.asterisk.org/wiki/display/AST/Asterisk+as+a+Swiss+Army+Knife+of+Telephony

18. DAVENPORT, Malcolm. *A Brief History of the Asterisk Project - Asterisk Project - Asterisk Project Wiki* [online]. [cit. 2020-08-30]. Dostupné z: https://wiki.asterisk.org/wiki/display/AST/A+Brief+History+of+the+Asterisk+Project

19. VOZŇÁK, Miroslav a Filip ŘEZÁČ. *ASTERISK teorie a praxe.* Ostrava 2011. Vysoká škola báňská - Technická univerzita Ostrava. Fakulta elektrotechniky a informatiky.

20. DAVENPORT, Malcolm. *Asterisk Architecture, The Big Picure - Asterisk Project - Asterisk Project Wiki* [online]. [cit. 2020-09-01]. Dostupné z: https://wiki.asterisk.org/wiki/display/AST/Asterisk+Architecture%2C+The+Big+Picture

21. DAVENPORT, Malcolm. *Types of Asterisk Modules - Asterisk Project - Asterisk Project Wiki* [online]. [cit. 2020-09-01]. Dostupné z: https://wiki.asterisk.org/wiki/display/AST/Types+of+Asterisk+Modules

22. DAVENPORT, Malcolm. *Dialplan - Asterisk Project - Asterisk Project Wiki* [online]. [cit. 2020-09-02]. Dostupné z: https://wiki.asterisk.org/wiki/display/AST/Dialplan

23. BRYANT, Stephen. *Asterisk: the definitive guide.* 4th ed. Sebastopol: O´Reilly, 2013. ISBN 978-1-449-33242-6.

24. DAVENPORT, Malcolm. *Contexts, Extensions and Priorities - Asterisk Project - Asterisk Project Wiki* [online]. [cit. 2020-09-04]. Dostupné z: https://wiki.asterisk.org/wiki/display/AST/Contexts%2C+Extensions%2C+and+Priorities

25. DAVENPORT, Malcolm. *Pattern Matching - Asterisk Project - Asterisk Project Wiki* [online]. [cit. 2020-09-04]. Dostupné z: https://wiki.asterisk.org/wiki/display/AST/Pattern+Matching

26. SUCHÝ, Miroslav. *Reverzní proxy - Root.cz* [online]. [cit. 2021-01-12]. Dostupné z: https://www.root.cz/clanky/reverzni-proxy/

27. MIERLA, Daniel-Constantin. *DISPATCHER Module* [online]. [cit. 2021-01-14]. Dostupné z: https://kamailio.org/docs/modules/5.4.x/modules/dispatcher.html

28. KOĎOUSKOVÁ, Barbora. *Co je to API a jaké jsou možnosti jeho využití?* [online]. [cit. 2021-02-11]. Dostupné z: https://www.rascasone.com/cs/blog/co-je-api

29. FARRELL, Doug. *Python REST APIs With Flask, Connexion, and SQLAlechemy - Real Python* [online]. [cit. 2021-02-11]. Dostupné z: https://realpython.com/flask-connexion-rest-api/

30. MALÝ, Martin. *REST: Architektura pro webové API - Zdroják* [online]. [cit. 2021-02-11]. Dostupné z: https://www.zdrojak.cz/clanky/rest-architektura-pro-webove-api/

31. LIEW, Zell. *Understanding And Using REST APIs - Smashing Magazine* [online]. [cit. 2021-02-11]. Dostupné z: https://www.smashingmagazine.com/2018/01/understanding-using-rest-api/

32. *Flask | The Pallets Projects* [online]. [cit. 2021-02-11]. Dostupné z: https://palletsprojects.com/p/flask/

33. *Flask Intro - Python Begginers documentation* [online]. [cit. 2021-02-11]. Dostupné z: https://python-adv-web-apps.readthedocs.io/en/latest/flask.html

34. POSPÍCHAL, Vlastimil. *SQLite: Databáze pro váš web - Zdroják* [online]. [cit. 2021-02-12]. Dostupné z: https://zdrojak.cz/clanky/sqlite-databaze-pro-vas-web/

35. MARTINEK, Michal. *Lekce 1 - Úvod do SQLite a příprava prostředí* [online]. [cit. 2021-02-12]. Dostupné z: https://www.itnetwork.cz/sqlite/sqlite-tutorial-uvod-a-priprava-prostredi

36. *Appropriate Uses For SQLite* [online]. [cit. 2021-02-12]. Dostupné z: https://www.sqlite.org/whentouse.html

37. *Requests: HTTP for Humans^{TM} - Requests v2.25.1. documentation* [online]. [cit. 2021-02-12]. Dostupné z: https://2.python-requests.org/en/master/

38. *JSON* [online]. [cit. 2021-02-12]. Dostupné z: https://www.json.org/json-cz.html

39. RONQUILLO, Alex. *Python's Requests Library (Guide) - Real Python* [online]. [cit. 2021-02-12]. Dostupné z: https://realpython.com/python-requests/

40. JOHANSSON, Olle E., Juha HEINANEN, Carsten BOCK a Hugh WAITE. *http_client* [online]. [cit. 2021-02-12]. Dostupné z: https://kamailio.org/docs/modules/5.4.x/modules/http_client.html

41. ELLINGWOOD, Justin. *How To Set Up uWSGI and Nginx to Serve Python Apps on Ubuntu 14.04 | DigitalOcean* [online]. [cit. 2021-02-13]. Dostupné z: https://www.digitalocean.com/community/tutorials/how-to-set-up-uwsgi-and-nginx-to-serve-python-apps-on-ubuntu-14-04#definitions-and-concepts

42. YEKABOTE, Pavan Kumar. *Deploy Flask app in Nginx using uWSGI — with architectural explanation. | by Pavan Kumar Yekabote | Medium* [online]. [cit. 2021-02-13]. Dostupné z: https://medium.com/@yekabotep/deploy-flask-app-in-nginx-using-uwsgi-with-architectural-explanation-2e24a41c030a

43. *deb.kamailio.org/#stable54* [online]. [cit. 2021-02-24]. Dostupné z: https://deb.kamailio.org/#stable54

44. *Install Kamailio On Debian* [online]. [cit. 2021-02-24]. Dostupné z: https://kamailio.org/docs/tutorials/devel/kamailio-install-guide-deb/

45. NEWTON, Rusty. *res_pjsip Configuration Examples - Asterisk Project - Asterisk Project Wiki* [online]. [cit. 2021-02-25]. Dostupné z: https://wiki.asterisk.org/wiki/display/AST/res_pjsip+Configuration+Examples

46. DAVENPORT, Malcolm. *Dialplan Configuration Examples - Asterisk Project - Asterisk Project Wiki* [online]. [cit. 2021-02-25]. Dostupné z: https://wiki.asterisk.org/wiki/display/AST/Dialplan

47. *Quickstart - Flask-SQLAlchemy Documentation (2.x)* [online]. [cit. 2021-02-25]. Dostupné z: https://flask-sqlalchemy.palletsprojects.com/en/2.x/quickstart/#a-minimal-application

48. TRAVERSY, Brad. *GitHub - bradtraversy/flask_sqlalchemy_rest: Products REST API using Flask, SQL Alchemy & Marshmallow* [online]. [cit. 2021-02-25]. Dostupné z: https://github.com/bradtraversy/flask_sqlalchemy_rest

49. *Quickstart - Flask Documentation (1.1.x)* [online]. [cit. 2021-02-25]. Dostupné z: https://flask.palletsprojects.com/en/1.1.x/quickstart/#a-minimal-application

50. JUELL, Kathleen a Mark DRAKE. *How To Serve Flask Applications with uWSGI and Nginx on Ubuntu 20.04 | DigitalOcean* [online]. [cit. 2021-02-27]. Dostupné z: https://www.digitalocean.com/community/tutorials/how-to-serve-flask-applications-with-uwsgi-and-nginx-on-ubuntu-20-04

51. RONQUILLO, Alex. *Python's Requests Library (Guide) - Real Python* [online]. [cit. 2021-03-01]. Dostupné z: https://realpython.com/python-requests/

52. *psutil documentation - psutil 5.8.1 documentation* [online]. [cit. 2021-03-01]. Dostupné z: https://psutil.readthedocs.io/en/latest/

53. *Welcome to SIPp* [online]. [cit. 2021-03-31]. Dostupné z: http://sipp.sourceforge.net/

54. *SIPp Documentation - https://sipp.readthedocs.io/_/downloads/en/v3.6.0/pdf/* [online]. [cit. 2021-03-31]. Dostupné z: https://sipp.readthedocs.io/_/downloads/en/v3.6.0/pdf/

Příloha A

Obsah elektronické přílohy

1. **Konfigurační soubory pro Asterisk:**
   - pjsip.conf
   - extensions.conf
2. **Konfigurační soubory pro Kamailio:**
   - dispatcher.list
   - kamailio_dispatcher.cfg (distribuce zátěže pomocí Dispatcher modulu)
   - kamailio.cfg (distribuce zátěže pomocí vlastního řešení)
3. **Python skripty a dodatečné soubory:**
   - api.py
   - wsgi.py
   - rest_api.ini
   - client.py
   - post.py
4. **Soubory použité při závěrečném testování a naměřené hodnoty:**
   - testing_scenario.xml
   - g711a.pcap
   - test1.sh
   - test2.sh
   - results.xlsx