VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ
DEPARTMENT OF COMPUTER SYSTEMS

FLEXIBILNÍ VYVAŽOVAČ ZÁTĚŽE S VYUŽITÍM JAZYKA P4

FLEXIBLE LOAD BALANCER USING P4 LANGUAGE

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — **JINDŘICH ŠESTÁK**
AUTHOR

VEDOUCÍ PRÁCE — **Ing. TOMÁŠ MARTÍNEK, Ph.D.**
SUPERVISOR

BRNO 2020

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav počítačových systémů (UPSY) Akademický rok 2019/2020

Zadání bakalářské práce

Student: **Šesták Jindřich**
Program: Informační technologie
Název: **Flexibilní vyvažovač zátěže s využitím jazyka P4**
Flexible Load Balancer Using P4 Language
Kategorie: Počítačové sítě

Zadání:

1. Seznamte se s technikami vyvažování zátěže síťového provozu a jazykem P4.
2. Navrhněte vhodný způsob realizace vyvažování zátěže síťového provozu, proveďte jeho implementaci s využitím jazyka P4 a jeho funkčnost ověřte simulací např. s využitím behaviorálního modelu BMv2.
3. Seznamte se s kompilátorem jazyka P4 do VHDL vyvíjeným v rámci sdružení CESNET a podporovaným hardware.
4. Navržený vyvažovač zátěže implementuje s využitím kompilátoru P4 do VHDL a jeho funkčnost ověřte na dostupném hardware.
5. Zhodnoťte dosažené výsledky a diskutujte další pokračování práce.

Literatura:

- Dle pokynů vedoucího.

Pro udělení zápočtu za první semestr je požadováno:

- Splnění bodů 1 a 2 zadání.

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/

Vedoucí práce: **Martínek Tomáš, Ing., Ph.D.**
Vedoucí ústavu: Sekanina Lukáš, prof. Ing., Ph.D.
Datum zadání: 1. listopadu 2019
Datum odevzdání: 28. května 2020
Datum schválení: 25. října 2019

Abstrakt

V současnosti jsou servery internetových služeb většinou shlukovány do skupin, aby měly dostatečný výkon obsloužit dotazy klientů. Každý tento shluk potřebuje Vyvažovač zátěže, který pro každý dotaz vybere jeden ze serverů, který dotaz obslouží. Pro popis takového zařízení zpracovávající pakety lze využít jazyk P4. V rámci této práce byly prostudovány principy vyvažování, proveden návrh, implementace a testování jednoduchého Vyvažovače zátěže popsaného v jazyce P4. Program je testován pomoci Behaviorálního modelu jazyka P4 na běžném procesoru a také na kartě NFB-200G2QL díky prostředí Netcope od sdružení CESNET.

Abstract

Currently servers of internet services are usually grouped together into clusters to provide sufficient performance to serve clients' queries. Each cluster needs Load Balancer, so it can choose one server which will process query from one client. For describing such device that processes packets is convenient to use P4 language. Within this work, the principles of load balancing, design, implementation and testing of a simple Load Balancer described in P4 language were demonstrated. The program is tested using Behavioral model of P4 language on a common processor and on the NFB-200G2QL card thanks to the Netcope environment from the CESNET association.

Klíčová slova

P4, Vyvažovač zátěže, jazyk P4, Behaviorální model P4, FPGA, NDK platforma

Keywords

P4, Load Balancer, P4 language, Behavioral model P4, FPGA, NDK platform

Citace

ŠESTÁK, Jindřich. *Flexibilní vyvažovač zátěže s využitím jazyka P4*. Brno, 2020. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Tomáš Martínek, Ph.D.

Flexibilní vyvažovač zátěže s využitím jazyka P4

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Tomáše Martínka Ph.D. Další informace mi poskytli pracovníci výzkumné skupiny Liberouter ve sdruženi CESNET. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Jindřich Šesták
28. května 2020

Poděkování

Rád bych poděkoval svému vedoucímu Ing. Tomáši Martínkovi Ph.D. za jeho pomoc a podporu při studiu a psaní této bakalářské práce. Dále bych chtěl poděkovat Ing. Martinu Špinlerovi a Marku Bencovi za jejich odbornou pomoc a rady při práci s kartou NFB-200G2QL. Také bych chtěl poděkovat Ing. Tomáši Fukáčovi a Davidu Vodákovi při studiu DPDK platformy potřebné pro testování na kartě NFB.

Obsah

Kapitola 1

Úvod

S rostoucím počtem koncových zařízení využívající Internet rostou nároky na servery poskytující služby těmto zařízením. Z důvodu docílení bezproblémového provozu a dostupnosti těchto služeb často se musí koncové servery povyšovat za vysoké náklady, aby zvládly obsloužit velké množství dotazů. Dalším způsobem, jak zajistit hladký chod webových aplikací je decentralizace. Místo jednoho drahého serveru, který by se mohl selhat, se koupí několik levnějších serverů. Nemusí to nutně být fyzické stroje. Dnes se v mnoho případech používají i cloudová řešení, která poskytují vysokou škálovatelnost a které dohromady tvoří shluky. Každý počítačový shluk potřebuje Vyvažovač zátěže (Load Balancer), který umí distribuovat zátěž na koncové servery co nejefektivněji. Zároveň jsou díky tomu servery odstíněny od okolního světa, což zajišťuje značnou míru bezpečnosti.

V cloudech existuje mnoho těchto softwarových Vyvažovačů. Avšak ty mají velice omezenou rychlost. Proto se vyvíjejí i hardwarová řešení, která dosahují mnohem větší propustnosti. Ale tato řešení jsou většinou velmi drahá. A jejich vývoj je velice náročný. Neboť pro zajištění rychlosti je třeba, aby byl programovací jazyk nízkoúrovňový. Avšak v dnešní době je psaní velkého projektu pro zpracování paketů v jazycích jako VHDL nebo C zbytečné.

Zde přichází na řadu jazyk P4, což je deklarativní jazyk popisující zpracování paketů síťovými prvky, jako jsou směrovače nebo přepínače. Tento platformě nezávislý jazyk se skvěle hodí pro tvorbu síťového Vyvažovače zátěže, jelikož poskytuje vysokou úroveň abstrakce. Díky této abstrakci je tvoření programu mnohem snazší a příjemnější.

Cílem této práce je seznámení s technikami vyvažování a prostudování schopností jazyka P4. Navržený program je implementován v jazyce P4 a za pomocí překladače *p4VHDL* poskytnutého organizací CESNET otestován na kartě NFB-200G2QL.

Práce je rozdělena do šesti kapitol. Druhá kapitola přibližuje problematiku Vyvažování a popisuje vlastnosti takového zařízení. Je zde i popsán jazyk P4, pomocí něhož je řešení implementováno. Třetí kapitola se dotýká kritiky současných řešeních Vyvažovačů a obsahuje argumentaci pro vznik této práce. Ve čtvrté kapitole je podrobně popsán návrh aplikace umožňující vyvažování, a vlastnosti s tím spojené. V předposlední kapitole 5 jsou popsány způsoby testování a ověření funkčnosti této aplikace. V závěru práce 6 jsou shrnuty dosažené výsledky a tipy pro budoucí zlepšení.

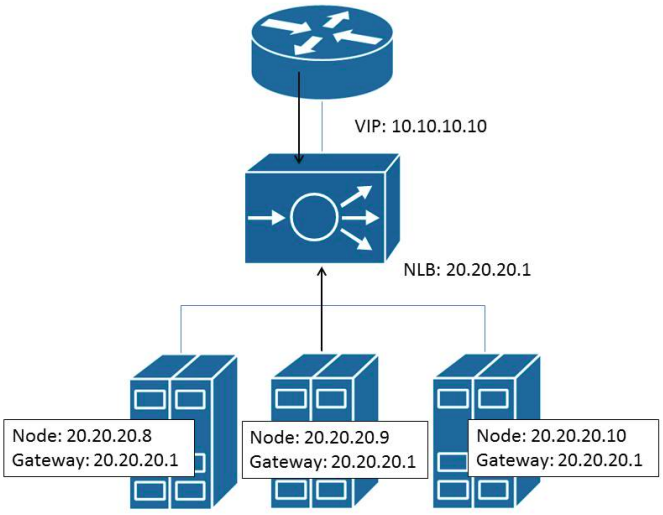
Kapitola 2

Seznámení s problematikou

2.1 Vyvažovač Zátěže

Vyvažovač zátěže využívá techniku rozložení zátěže mezi více prvků. Těmito prvky jsou například procesory, síťové linky, pevné disky i celé počítače. Dochází tak k optimalizaci a lepšímu využití zdrojů. Zamezí se přetížení jednoho prvku, protože se zátěž rozdělí mezi více objektů. Tato kapitola se zaměří na popis serverového Vyvažovač zátěže na obrázku 2.1. Tedy takového, který rozděluje dotazy klientů mezi více koncových serverů. Tato technika rozkládání dotazů, poskytuje lepší využití zdrojů, lepší čas odezvy i větší odolnost proti výpadkům služby.

Za jednoduchý Vyvažovač zátěže by se dal považovat DNS server, který pro jedno doménové jméno má více IP adres. Klient by při dotazu zpravidla použil první vrácenou adresu. Přičemž by tyto adresy DNS server nabízel v náhodném pořadí. Dotazy by se rovnoměrně rozdělily mezi koncové servery.



Obrázek 2.1: **Vyvažovač zátěže** je prvek rozkládající požadavky klientů na více serverů. On sám má Veřejnou IP adresu (VIP) a privátní. Vyvažovač zátěže přeposílá požadavky na privátní adresy serverů (DIP) (převzato z [19]).

Vyvažovač zátěže musí umět určité postupy, aby dobře rozkládal zatížení. Nejde jen o slepé rozdělování paketů. Existuje více variant, ale obecně by každý Vyvažovač zátěže měl minimálně implementovat tyto 3 základní vlastnosti:

- **Objevování služby (Service Discovery)** - Vyvažovač zátěže musí znát přesně množinu serverů, které jsou k dispozici pro danou službu. Asi nejjednodušším způsobem je zde konfigurace pomocí souboru. Vyvažovači zátěže předložíme seznam serverů, případně jejich adres, v souboru. Tím Vyvažovač zátěže zjistí, jaké servery jsou za ním zastíněné.
- **Kontrola živosti (Health Checking)** - Měl by si také být vědom, že některý ze severů už není aktivní a zastavit přeposílání na tento server. Udržuje si tak informace o dostupných serverech k přijmutí nového toku.
- **Vyvažování zátěže (Load Balancing)** - Samotné vyvažování se implementuje pomocí vyvažovacího algoritmu 2.1.2. Algoritmus rozhodne, na který koncový server bude tok přesměrován.

2.1.1 Rozdělení Vyvažovačů zátěže

Jelikož existuje tolik různých služeb, kdy každá má jiné specifikace a potřebuje něco jiného, existuje i celá řada typů a dělení Vyvažovačů. Každý z nich se k navazování spojení a ke přeposílání mezi serverem a klientem staví trochu jinak. V této kapitole bude představeno pár nejzákladnějších rozdělení Vyvažovačů zátěže. Je důležité si představit toto rozdělení, protože díky tomu získáme pohled na to, co všechno a jak může Vyvažovač zátěže vykonávat. Také nám to dá představu o tom, jaká řešení existují a jak by se dala zkombinovat.

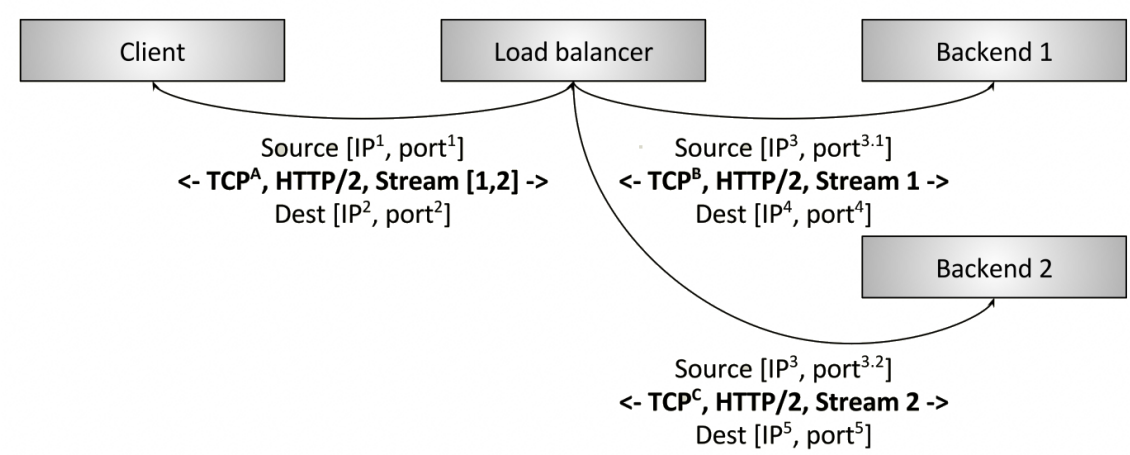
L4 vs. L7

Asi nejzákladnější dělení je právě dělení podle ISO/OSI vrstvy (viz obrázek 2.3), na které Vyvažovač zátěže operuje. Dělí se na Vyvažovač zátěže na L4 Transportní vrstvě a L7 Aplikační vrstvě. Avšak toto dělení nejde naprosto přesně definovat, jelikož ne každý Vyvažovač zátěže L7 implementuje všechny dílčí protokoly a pod-protokoly patřící do aplikační vrstvy. A naopak, některý Vyvažovač zátěže L4 vrstvy může implementovat vyšší protokoly nebo mechanismy z vyšších vrstev.

L4 Vyvažovač zátěže umí parsovat a přeposílat pouze na základě hlaviček ethernetové, IP a transportní vrstvě (TCP/UDP). Můžeme říci, že Vyvažovač zátěže vůbec neví jaká data paket nese. Nezná ani vyšší protokoly než je TCP/UDP. To má jak výhody, tak nevýhody. Značným kladem je rychlost. Jelikož Vyvažovač zátěže nemusí analyzovat celý paket, ale stačí mu nahlížet jen do úrovně transportního protokolu, je tento způsob rychlejší.

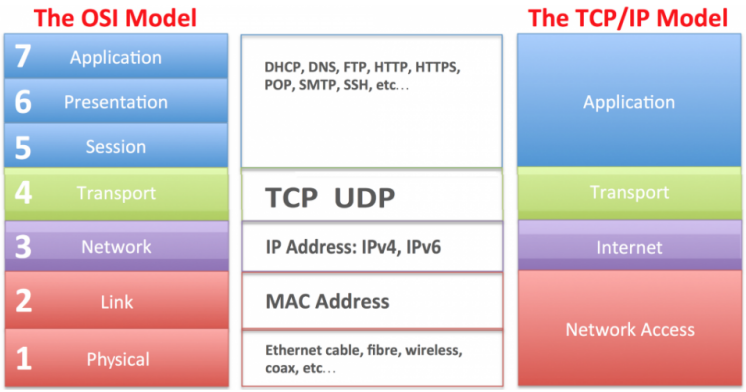
L7 Vyvažovač zátěže jsou sofistikovanější a mnohem složitější. Analyzují celý paket až po aplikační L7 vrstvu. Můžeme říct, že přeposílají jednotlivé dotazy a ne celé toky jako u L4 Vyvažovač zátěže. To má velké využití například u HTTP protokolu, kdy se v jednom toku nachází několik HTTP dotazů. Každý z těchto dotazů je rozložen mezi servery. L4 Vyvažovač zátěže by rozložil celé toky, takže například jeden tok s jedním dotazem na první server a druhý tok nesoucí 10 dotazů na druhý server. To by znamenalo, že druhý server je 10x více vytíženější než první server. Tuto problematiku právě řeší L7 Vyvažovač zátěže. Existují i Vyvažovače rozkládající dotazy podle jejich obsahu. Vyvažuje například tak, že požadavky obsahující obrázky přeposílá na speciální server k tomu určený (2.1.2 -

L7 přepínání podle obsahu). Protože analyzuje paket až do L7 vrstvy, je méně výkonný než L4 Vyvažovač. Jeho složitost také zvětšuje riziko chyby.



Obrázek 2.2: **L7 Vyvažovač zátěže** rozkládá jednotlivé dotazy (*Stream*), ne celé toky.

Zajímavou kombinací je pak síť, ve které je L4 i L7 Vyvažovač zátěže. Jeden L4 Vyvažovač zátěže je před více L7 Vyvažovači a rozesílá dotazy na ně. Tato kombinace využívá jak rychlosti vyvažování na L4 vrstvě, tak i propracovanosti a lepší optimalizace vyvažování na L7 úrovni.



Obrázek 2.3: **ISO/OSI model** ukazující rozdělení síťových protokolů.

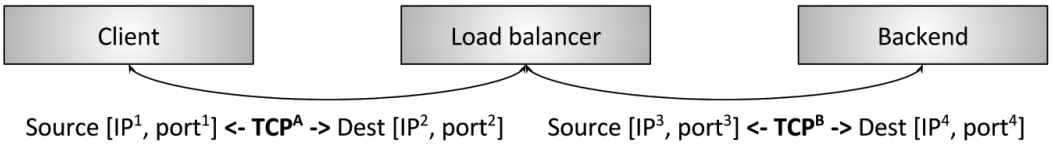
TCP/UDP ukončování vs. propouštění

V tomto dělení jde především o počet spojení vytvořeného pro jeden tok. Klient zná adresu pouze Vyvažovač zátěže, který vystupuje jako server. Ten má dvě možnosti jak bude s klienty komunikovat a udržovat spojení.

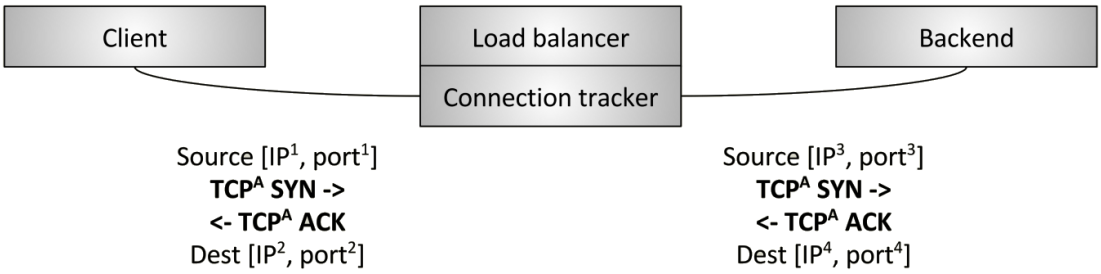
TCP/UDP ukončování je způsob komunikace, kdy Vyvažovač zátěže vystupuje přímo jako server a odpovídá klientovi on sám. Pro jeden tok jsou udržována dvě spojení. Jedno si udržuje Vyvažovač zátěže s klientem a druhé si vytvoří Vyvažovač zátěže se serverem (viz obrázek 2.4 a 2.2). Můžeme tak říci, že v tomto případě jsou nároky větší, protože se musí

udržovat dvě spojení. To platí především u TCP, kdy je třeba inicializovat spojení pomocí třícestného hanshaku (3-way hanshake) [22].

TCP/UDP propouštění naopak přeposílá pakety přímo na server. Vyvažovač zátěže nevytváří nový paket, ale pozmění originální paket od klienta a ten přepošle (viz obrázek 2.5). Tato technika se dá označit jako NAT (Network Address Translation). Tento způsob dost zjednodušuje Vyvažovač zátěže, protože nemusí implementovat mechanismy týkající se navazování a ukončování spojení, které jsou ohledně TCP dosti komplikované (Congestion Control - kontrola přetížení). Tyto mechanismy nechává na koncových serverech.



Obrázek 2.4: **Ukončovací Vyvažovač zátěže** navazuje a udržuje dvě spojení najednou. Klient<—>Vyvažovač a Vyvažovač<—>Server (Backend) (převzato z [15]).

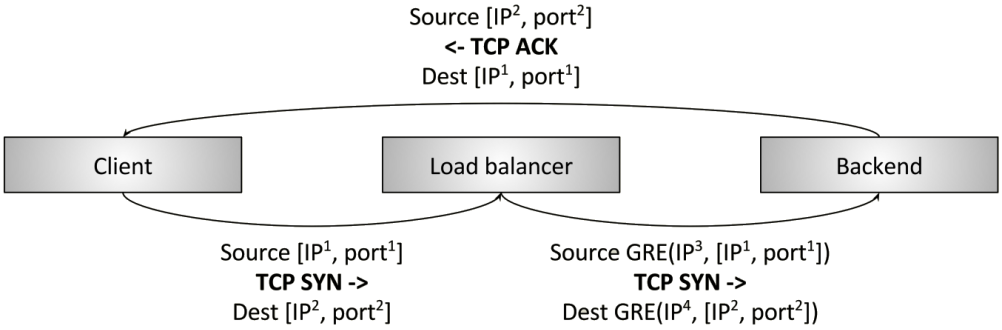


Obrázek 2.5: **Propouštěcí Vyvažovač zátěže** nevytváří nová spojení, přeposílá aktuální pakety (převzato z [15]).

Transparentní vs. netransparentní

Toto hledisko rozdělení se zabývá viditelností. Zda server zná IP adresu klienta nebo je od něj naprosto odstíněn. Vyvažovač zátěže poté co rozhodne, na který server paket pošle, může ponechat v IP hlavičce původní zdrojovou adresu klienta. Takovýto způsob vyvažování se nazývá **Transparentní** (viz obrázek 2.6).

Naopak pokud Vyvažovač zátěže pří odeslání paketu na server změní zdrojové adresy na své, úplně tím odstíní koncové servery od okolního světa (tzv. *Black Box/černá krabice* - zapouzdření a izolovanost od okolí) a takový druh Vyvažovač zátěže je označován jako **Netransparentní** (viz obrázky 2.4 a 2.5).



Obrázek 2.6: **Transparentní Vyvažovač zátěže** umožňuje, aby server znal koncovou adresu klienta. Zde je zdrojová IP adresa klienta zabalena v tagu, ale server si ji umí rozbalit a odpovědět přímo klientovi (převzato z [15]).

2.1.2 Algoritmy vyvažování

Algoritmus lze chápat jako deterministický návod, jehož provedení nám dává jednoznačný výsledek. Je to pevně daný pracovní postup. Algoritmů pro vyvažování je hned několik a mají různé verze. V této kapitole budou představeny základní algoritmy, pomocí kterých Vyvažovač zátěže rozděluje zátěž na své koncové servery.

Round Robin je postup vyvažování, kdy Vyvažovač zátěže postupně přiřazuje toky jednotlivým serverům. První dotaz je přepojen na první server, druhý na druhý, třetí opět na první a tak dále (viz obrázek 2.7). Obdobný algoritmus se používá u DNS serveru s tím rozdílem, že toky nepřiřazuje postupně serverům, ale náhodně. Existuje i modifikace zvaná Váhový Round Robin, který funguje obdobně s tím rozdílem, že každý server má přiřazenou váhu. Stanicím s vyšší váhou je přiřazováno více spojení. Váhy vytváří poměr počtu spojení serverů.

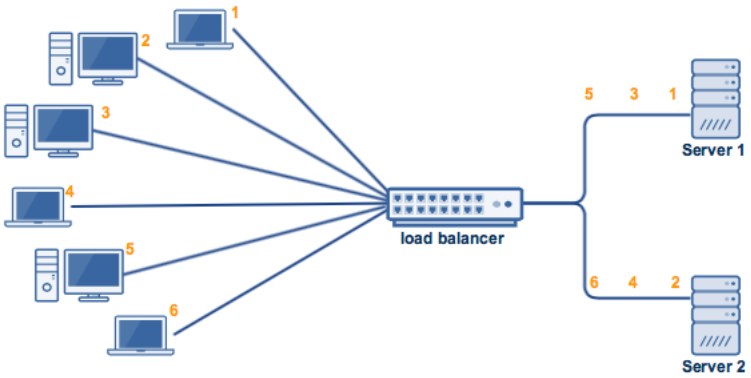
Vyvažování podle nejméně spojení (Least Connection Balancing) je metoda, při které Vyvažovač zátěže sleduje počet aktivních spojení k jednotlivým serverům. Typicky musí mít nějakou tabulku pro tyto záznamy. Nové příchozí spojení je přiřazeno serveru s nejmenším počtem aktivních toků. Zde nastává problém. Jak určit, že spojení skončilo? Například si můžeme nastavit čas, po který je spojení aktivní. Pokud po tuto dobu nepřijde paket z tohoto spojení, považuje se neaktivní a záznam se z tabulky vymaže. Nebo můžeme sledovat příznaky paketů u TCP spojeních a při ukončovacím paketu spojení z tabulky vymazat. U UDP paketů žádné příznaky nejsou. U nich se musí tedy vždy použít maximální čas, po který lze tok považovat za aktivní. Avšak u obou technik musí mít Vyvažovač zátěže tabulku, ve které není jen počet spojení, ale i informace o jednotlivých spojeních (hlavně tedy čas přijmutí posledního paketu spojení). I tento algoritmus má svou váhovou verzi.

Agentní Vyvažování (Agent-based Adaptive Load Balancing) potřebuje pro své fungování informaci o vytížení serveru. Agentní znamená, že na serverech musí běžet program, který pravidelně reportuje Vyvažovači zátěže své vytížení (Agent). Vyvažovač zátěže poté přiřazuje nová spojení serveru s nejmenším číslem. Většinou je toto číslo vypočítáno z kombinace využití několika komponent stroje, jako například procesoru, RAM paměti, ethernetové karty... Příkladem takového nástroje je protokol SNMP (System Netowrk Managemenet Protocol).

L7 přepínání podle obsahu (Layer 7 Content Switching) vyvažuje toky na sedmé aplikační vrstvě ISO/OSI modelu. Tato metoda se hodí pro specializované servery. Napří-

klad může být server optimalizovaný pro přenos velkých souborů. Nebo server pro rychlé HTTP odpovědi. Vyvažovač zátěže tedy podle dotazu klienta rozhodne, zda má nějaký speciální server pro tento dotaz. Pokud ano, pošle paket na tento server. Pokud ne musí mít implementována další pravidla, jak v tomto případě postupovat.

5-hashovaní většinou používá pětici hodnot z hlaviček paketu, ze kterých pomocí hashování získá jednu hodnotu. Vyvažovač zátěže má přiřazené rozsahy hodnot pro dané servery. Podle této hodnoty tedy přeposílá pakety. Lze říci, že spojení zůstává konstantní v tom smyslu, že daný tok paketů od klienta na Vyvažovač zátěže má vždy stejné hodnoty hlaviček (IP adresy, typ protokolu, čísla portů, ...), takže se vždy zahashuje na stejné číslo, tudíž ho Vyvažovač zátěže vždy přesměruje na stejný server po celou dobu tohoto spojení.



Obrázek 2.7: **Algoritmus Round Robin** postupně rozesílá požadavky na servery (převzato z [6]).

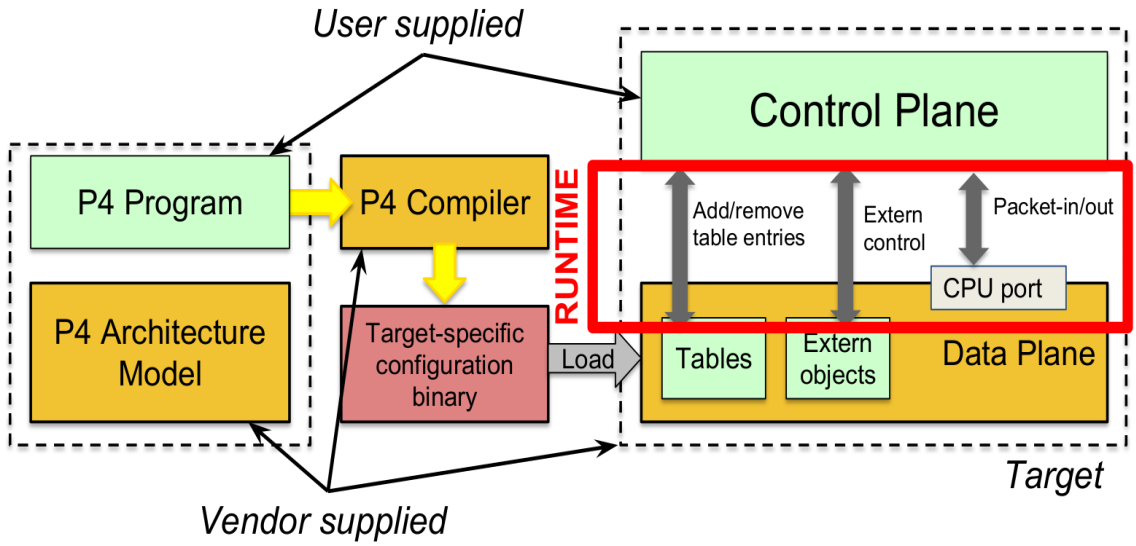
2.2 Jazyk P4

Tato kapitola se zabývá programovacím jazykem P4 a většina obsahu je převzata z oficiálního stránek tohoto jazyka.

P4 stojí pro Programming Protocol-independent paket Processors. P4 je tedy vysokoúrovňový deklarativní jazyk pro programování protokolově nezávislých síťových procesorů. Pomocí tohoto jazyka je možné přesně definovat zpracování paketů v zařízení. Skvěle se tak hodí pro popis síťových prvků, jako jsou například směrovače a přepínače. Jelikož je jazyk protokolově nezávislý, umožňuje vytvářet protokoly nové.

Aktuálně jsou k dispozici dvě verze tohoto jazyka. $P4_{14}$ pochází z roku 2014 a jedná se o první představení tohoto jazyka. Tento standard je sám o sobě dost flexibilní. Avšak větší volnost poskytne verze $P4_{16}$ z roku 2016, která nabízí možnost rozšíření o uživatelem definované funkce pomocí *extern* výrazu. Navíc oproti starší verzi se již neřídí Abstraktním modelem 2.2, ale je možno definovat vlastní architekturu P4 zařízení.

P4 jazyk nabízí široké uplatnění díky implementační nezávislosti. To znamená, že P4 program může být kompilován pro různé typy zařízení, jako jsou CPU, FPGA a další. Avšak je nutné, aby k dané platformě byl dodán i kompilátor pro toto cílové zařízení (*target* viz obrázek 2.8). Kompilátor mapuje kód jazyka P4 na zdroje cílového zařízení, proto je třeba, aby ke každému P4 targetu byl dodáván výrobcem platformy. Uživatel/Programátor pak definuje formát tabulek nezávisle na platformě (je možné vzít stejný kód, který poběží na dvou různých platformách např. CPU a FPGA). Běžící P4 program se nazývá Data plane. Control plane jsou vstupy do tabulek s hodnotami podle, kterých se následně bude vykonávat vyhledávání v tabulkách.

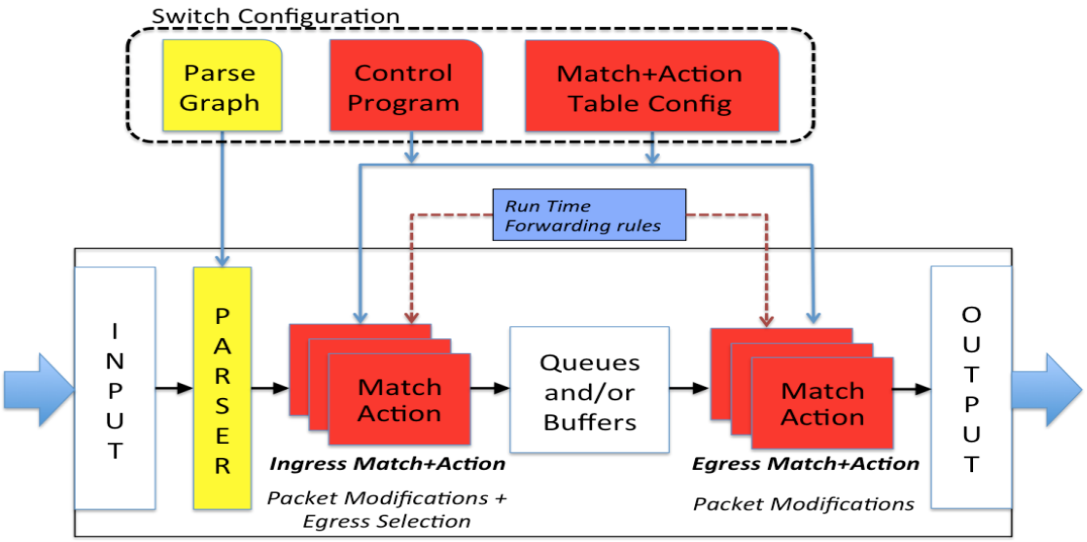


Obrázek 2.8: **Programování P4 aplikace** vyžaduje, aby výrobce cíle (target) dodal ke kartě kompilátor. Architektura pro $P4_{14}$ je Abstraktní model, pro $P4_{16}$ je k dispozici několik standardních architektur (případně musí výrobce také vytvořit). Programátor pak dodá program P4 (Data plane) a vstupy tabulek (Control plane) (převzato z [12]).

Abstraktní model jazyka $P4_{14}$

Starší verze jazyka z roku 2014 $P4_{14}$ [13] je založena na Abstraktním modelu popsaného na obrázku 2.9, který udává základní strukturu zařízení pro zpracování paketů. Popis tohoto modelu je následující:

- Parser vyextrahuje hlavičky paketu. Hlavičky následně slouží match+action tabulkám pro vyhledávání.
- Match+Action tabulky v Vstupní (Ingress) části (Ingress Match+Action) modifikují paket a nastavují síťový výstupní port paketu, kterým má být paket po skončení zpracování odeslán. Pakety poté vstupují do front.
- Mechanismus front (Queues and/or Buffers) rozděluje pakety do front podle posloupnosti match+action tabulek, kterými paket prošel. Umožňuje tak řazení paketů a například upřednostňování určitého provozu (Quality of Service).
- Match+Action tabulky v Egress části (Egress Match+Action) mohou modifikovat paket a jeho hlavičky, avšak tyto akce už nemohou změnit výstupní port paketu.
- Po zpracováním v Výstupní (Egress) části je paket a jeho hlavičky poskládán a poslán výstupním portem (Output).



Obrázek 2.9: **Abstraktní model** (převzato z [13]).

Každý program popsaný v jazyce P4 je složený z těchto částí. Každou z nich si programátor může definovat podle svých potřeb, avšak vykonávání programu bude vždy dodržovat toto schéma. Při psaní programu by měl programátor specifikovat následující prvky síťového zařízení:

- Definice hlaviček protokolů, které zařízení podporuje.
- Parsování hlaviček je proces postupného extrahování hlaviček protokolů za pomocí konečného automatu.

- Definice tabulek zahrnuje metodu porovnávání, vstupní hodnoty k porovnávání a seznam akcí, které se mohou vykonat.
- Akce se definují pomocí skládání primitivních akcí dohromady.
- Kontrolní program je dán posloupností tabulek a podmínek, za kterých se tabulky mohou na paket aplikovat.

Definice hlaviček a jejich položek

Hlavička se v jazyce P4 skládá z několika položek. Její definice je velice podobná strukturám v jazyce C. Definuje se pomocí syntaktické konstrukce *header_type název_hlavičky*. V rámci hlavičky se pak definují jednotlivé položky v části *fields* pomocí spojení *název_položky : šířka*. Šířka zde vyjadřuje velikost položky v bitech. Celková velikost hlavičky může být libovolná, avšak se doporučuje definovat hlavičky s celkovou velikostí v násobcích 8 (po celých bajtech). Stejný způsob platí jak pro definici hlaviček protokolů, tak pro definici hlaviček metadat doprovázející každý paket. Následující výpis ukazuje deklaraci Ethernetové hlavičky.

```
header_type ethernet_t {
    fields {
        dst_addr : 48; // width in bits
        src_addr : 48;
        ethertype : 16;
    }
}
```

Výpis 2.1: **Deklarace Ethernetové hlavičky** (převzato z [13]).

Definice parseru

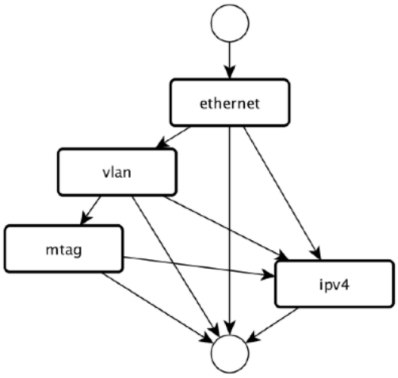
Parsování je proces extrakce jednotlivých hlaviček paketu pomocí konečného automatu. Jednotlivé stavy automatu jsou definovány pomocí konstrukce *parser* název_stavu. Přičemž platí pravidlo, že počátečním stavem automatu je námi první definovaný stav. Ve stavu je možno vyextrahovat data a uložit je do hlaviček pomocí funkce *extract()*, která jako parametr vyžaduje dříve definované proměnné parseru. Tyto proměnné se definují pomocí klíčového slova *header název_hlavičky název_proměnné* pro definici proměnných hlaviček paketu. Pokud zaměníme klíčové slova *header* za *metadata*, získáme proměnnou metadat, která vytvořena ke každému paketu, zatímco hlavičky se pro paket vytvoří jen pokud jsou vyparsované pomocí funkce *extract()*.

Ve stavu lze také modifikovat položky metadat paketu, případně jen přejít do dalšího stavu (viz parser start {...}). Přechody automatu do dalších stavů se implementují pomocí konstrukce *return název_dalšího_stavu*. Společně v kombinaci s příkazem *select*, lze dosáhnout větvení parseru, což poskytuje možnost zpracování velkého množství protokolů, které se používají v počítačových sítích. Příkaz *select* je ekvivalentní příkazu *switch* v jazyce C. Jako parametr potřebuje vyparsovanou hodnotu. Ta může být jak jedna položka z hlavičky, tak spojení několika položek dohromady tvořící jediný vektor. Hodnoty položek můžeme i vymaskovat pomocí klíčového slova *mask*. Lze rovnou přejít do vykonávání kontrolního programu (například *ingress*). Na obrázku 2.10 je ukázán jednoduchý graf parsování.

```
header ethernet_t ethernet;
metadata local_metadata_t local_metadata;
parser start {
    return ethernet;
}

parser ethernet {
    extract(ethernet); // Start with the ethernet header
    return select(latest.ethertype) {
        0x8100: vlan;
        0x800: ipv4;
        default: ingress;
    }
}
```

Výpis 2.2: **Definice Parsování ethernetové hlavičky** (převzato z [13]).



Obrázek 2.10: **Parsovací graf** ukazuje stavy automatu (převzato z [13]).

Definice Match+Action tabulek

Tabulky v jazyce P4 slouží pro mapování vyextrahovaných paketů na dané akce. Tabulka se skládá ze dvou nejdůležitějších částí. Nejdříve se v sekci *reads* definují klíčové položky, podle kterých se bude vyhledávat. Rozhodovací položky jsou hlavičky protokolů a jejich políčka (header a metadata). K těmto položkám se přiřadí i typ porovnání. Existuje 5 typů porovnávání:

- **Exact** - Hodnota v tomto políčku se musí přesně shodovat se záznamem v tabulce.
- **Ternary** - Vstupu tabulky je přiřazena maska. S hodnotou políčka se nejdříve provede logický AND s maskou a teprve pak se porovnává se samotnou hodnotou v tabulce.
- **LPM (Longest Prefix Match)** - Hodnoty se porovnávají podle nejdelšího prefixu.
- **Range** - Vstup tabulky obsahuje specifikaci nejnižší a nejvyšší možné hodnoty pro shodu a vyvolání dané akce. Porovnává se zda hodnota políčka spadá do tohoto rozsahu.

- **Valid** - Tato metoda lze použít jen u hlaviček (nelze u metadat). Tabulka pak obsahuje jen dva vstupy. Jeden pokud je políčko platné (true) a druhý pokud není platné (false). True znamená, že paket obsahuje danou hlavičku, jinak je aplikováno pravidlo pro false.

Druhou části definice tabulky je sekce *actions*, která obsahuje seznam akcí pro tuto tabulku.

```
table forward {
    reads {
        ipv4.dstAddr : lpm;
    }
    actions {
        drop;
        route_action;
    }
}
```

Výpis 2.3: **Definice Tabulky** pro směrování na L3 vrstvě.

Tabulka může obsahovat i klíčové slovo *size*. K němu přiřazena hodnota udává jak velká tabulka může být. Tedy kolik záznamů se v tabulce může vyskytovat. Výše uvedená tabulka definuje jednou políčko *reads* a to cílovou adresu IPv4 protokolu. Akce validní pro tuto tabulku jsou *drop* (což je primitivní akce na zahození paketu) a uživatelem definovaná akce *route*. Vidíme, že algoritmus pro porovnávání klíče je *lpm*. Mapování na tyto dvě akce se děje pomocí definovaných pravidel, které se vkládají do této tabulky (Control Plane). Nahrané pravidlo obsahuje vždy specifikované tyto údaje:
název_tabulky přidání/odebrání_pravidla reads_hodnoty => název_akce parametry...
zde zapsané jako pseudokód.

Definice Akcí

Akce v jazyce P4 se definují jako posloupnost primitivních akcí. Akce se v jazyce P4 volá z tabulky, kde ke každému záznamu je přiřazena právě jedna akce. Ze záznamu z tabulky se akci také předávají parametry pro vykonávání. Definice akcí nedovoluje použití cyklů nebo if-else bloků. Zde je znázorněna akce pro směrování provozu (funkce směrovače = routeru). Tato akce vyžaduje dva parametry a to port, jehož hodnota určí výstupní port zařízení, kterým bude paket poslán. A druhou hodnotou je cílová ethernetová adresa dalšího směrovače nebo cílového zařízení. Po úpravě hodnot, kdy jsou původní políčka nahrazena novými hodnotami, můžeme použít konstrukci *subtract(dest, value)*, pomocí které snížíme hodnotu políčka TTL IPv4 o jedna.

```
action route_action(port, dstMAC){
    modify_field(intrinsic_metadata.egress_spec, port);
    modify_field(ipv4.srcMAC, ipv4.dstMAC);
    modify_field(ipv4.dstMAC, dstMAC);
    subtract(ipv4.ttl, 1);
}
```

Výpis 2.4: **Definice Akce** pro směrování na L3 vrstvě.

Jazyk P4 nabízí řadu již definovaných primitivních akcí. Funkce *modify_filed* a *subtract* patří právě do těchto akcí. Pomocí nich si uživatel nadefinuje své složitější akce.

Definice Kontrolního programu

Definice Kontrolního programu se děje pomocí klíčového slova *control název_programu*. V něm je uvedeno v jakém pořadí a za jakých podmínek budou jednotlivé tabulky aplikovány. Aplikování tabulky se spouští funkcí *apply*, která jako parametr akceptuje jméno tabulky. Vykonávání v tomto *control* bloku je možno definovat jako jednoduchou posloupnost těchto *apply* příkazů. Další možností je použití klíčových slov *hit / miss* za již aplikovanou tabulkou. Tyto slova rozdělují následující tok programu, podle toho zda se našel shodující se záznam v tabulce (*hit*) nebo ne (*miss*). Poslední možností je možnost rozdělit tok programu podle konkrétně vykonaných akcí v tabulce. Navíc řízení toku programu můžeme ovlivňovat i pomocí konstrukcí *if-else* stejně jako v jazyce C.

```
control ingress{
    apply(forward) {
        route_action { //route_action was invoked
            apply(NoOp);
        }
    }
    if (ipv4.ttl < 1){
        apply(drop);
    }
}
```

Výpis 2.5: **Definice Kontrolního programu**

2.3 Behaviorální model jazyka P4

Behaviorální model jazyka P4 je framework pro P4 programy. Díky němu je možné spouštět a vyvíjet P4 aplikace přímo na CPU počítače. Je tedy zapotřebí kompilátor pro jazyk P4. Aktuálně existují dva a to :

- P4C - Je doporučováno používat tento kompilátor, protože podporuje jak $P4_{14}$ tak i $P4_{16}$. [9]
- P4C-BM je zastaralejší verze, která již není aktivně spravována a podporuje pouze $P4_{14}$. [10]

Jelikož je P4C navržen pro obě verze jazyka P4, je třeba při překladu specifikovat, jak target (zde bmv2 neboli simple_switch) tak i architekturu. Podporovanou architekturou pro behaviorální model je *v1model*, který je vhodný pro použití obou verzí jazyka. Pokud se používá novější specifikace $P4_{16}$, je třeba naimportovat soubor s touto architekturou pomocí *#include <v1model.p4>*.

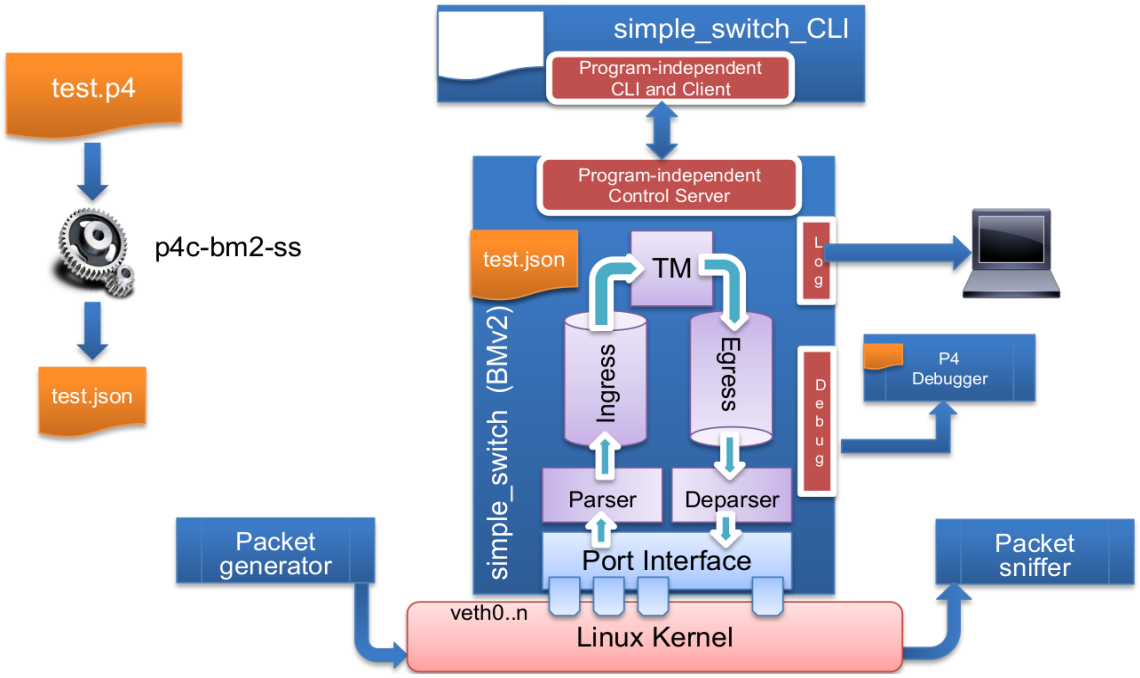
Kompilátor P4C ze zdrojového souboru vytvoří jeden soubor ve formátu JSON a druhý s příponou *.p4i*, což je výsledek preprocesoru (obsahuje všechny include soubory v sobě). Samotný behaviorální model pak na vstup přijímá soubor ve formátu JSON, který byl vygenerován překladačem P4C. Na následující ukázce je zobrazena JSON reprezentace z definované ethernetové hlavičky (viz 2.1).

```
{
      "name" : "ethernet_t",
      "id" : 2,
```

```
    "fields" : [
      ["dst_addr", 48, false],
      ["src_addr", 48, false],
      ["ether_type", 16, false]
    ]
  },
```

Výpis 2.6: **JSON reprezentace definice ETH hlavičky** v sekci header_types

Model je napsán v jazyce C++11 a implementuje chování zpracování paketu definovaného v P4 programu. BMv2 je vhodný především pro vývoj a debugování P4 programů, díky snadnému spouštění. Avšak není vhodný pro nasazení programu do přímého provozu zejména kvůli velice omezené propustnosti na úrovni softwaru.



Obrázek 2.11: **Schéma Behaviorálního modelu** zobrazuje části modelu a jejich propojení (převzato z [12]).

Na obrázku 2.11 je detailně zakresleno, jak jsou komponenty propojeny při fungování P4 programu za využití Behaviorálního modelu. V levé části obrázku máme zobrazen překlad ze zdrojového souboru na JSON reprezentaci pomocí kompilátoru p4c-bm2-ss (lze použít pouze kompilátor P4C, viz [9]. V dolní části obrázku vidíme Linuxové jádro, které je spojeno přes (virtuální) ethernetové porty s behaviorálním modelem. Z portových rozhraní putuje paket dále do modulu parseru, kde jsou vyextrahovány hlavičky protokolů. Následuje Ingresní část programu (kontrol bloku), ve které jsou klíčové hodnoty hlaviček paketů testovány v tabulkách. V případě nalezení záznamu se provede daná akce a pozmění se paket (TM část - Table Match). Poté paket prochází Egress částí programu, ve které se opět aplikují tabulky a mění paket. Část Deparser funguje reverzně k Parser části. Tedy vezme graf parseru (například 2.10) a stavy aplikuje v opačném pořadí (odspoda-nahoru). Následně pak celý paket putuje portovými rozhraními ven. Celá tato architektura je pevně daná (v1model), ale programátor definuje vnitřek těchto komponent.

Behaviorální model poskytuje velice schopný příkazový řádek *simple_switch_CLI* pomocí kterého může uživatel komunikovat s Behaviorálním modelem. Lze skrze něj například spravovat tabulky (přidávat, odebírat pravidla,...). CLI nástroj se připojuje k běžící P4 aplikaci a komunikuje s ní za běhu. Připojuje se přes Thrift RPC server, který běží pro každý spuštěný P4 program. Defaultní port je 9090, ale platí že jeden CLI nástroj se může přípojit pouze k jedné aplikaci. Tudíž při více běžících aplikacích P4 musíme specifikovat port, na kterém má server běžet, jinak nedojde k úspěšnému spuštění.

Metadata

Behaviorální model poskytuje základní metadata pod názvem *standard_metadata*. Oproti hlavičkám, metadata jsou dostupná ke každému paketu. Zato hlavička je dostupná jen k paketu, u kterého byla vyparsována. Metadata obsahují položky spjaté s průchodem paketu zařízením. Mezi nejdůležitější položky patří:

- ingress_port - Každý paket má nastavenou tuto hodnotu na číslo portu, kterým byl přijat.
- egress_port - Je číslo výstupního portu paketu. Tato položka je dostupná jen v Egress části (výstupním kontrol programu) pouze pro čtení (read-only).
- egress_spec - Do této položky lze v Ingress části (vstupní kontrolní program) přiřadit číslo výstupního portu. Tato hodnota se poté přiřadí do egress_portu. Pokud v celém programu nedojde k nastavení tohoto pole, automaticky se nastaví na hodnotu ingress_port
- packet_length - Délka paketu se nastaví pro nově přijatý paket.

Více informací o Behaviorálním modelu lze naleznout na oficiálních stránkách tohoto projektu, viz [11].

2.4 Netcope Development Kit

NDK platforma (Netcope Development Kit platform) je prostředí pro jednoduché vytváření nových aplikacích pro karty COMBO s FPGA čipem [14, 16]. Především pro vývoj síťových aplikací. Vývojem a správou této platformy se zabývá výzkumná skupina Liberouter patřící do sdružení CESNET [1]. Toto prostředí ulehčuje vývojáři práci, neboť v repozitáři jsou připravené základní komponenty, které vývojář už nemusí vyvíjet. Základními prvky NDK platformy jsou tyto složky:

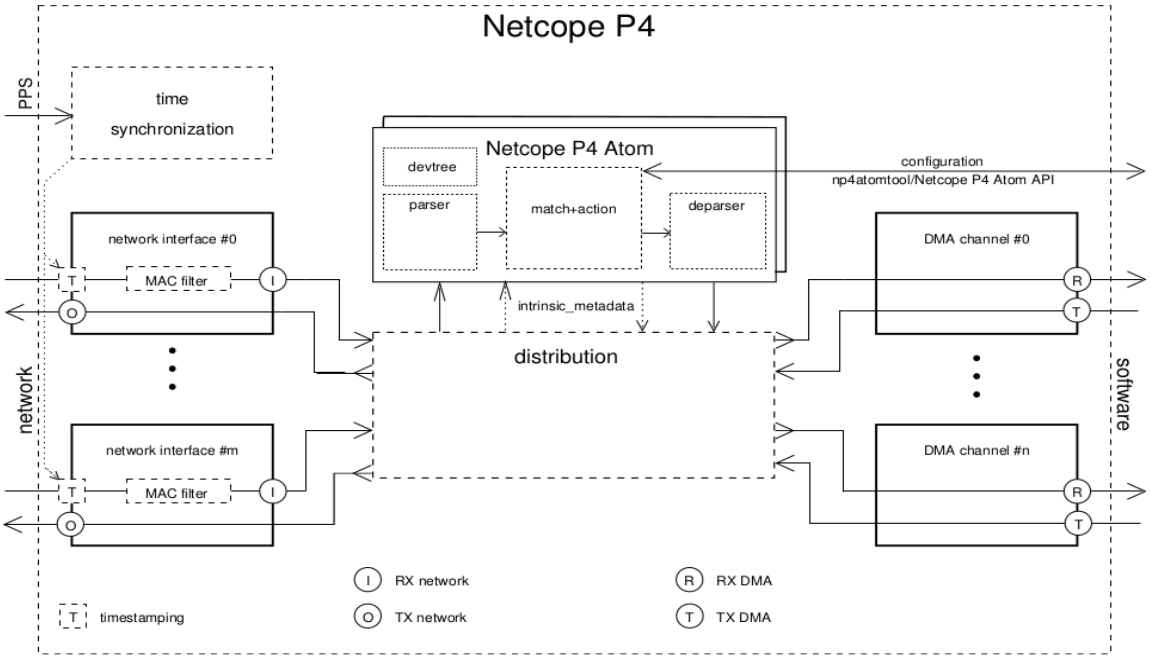
- fwbase/ndk/[karta] - poskládané prostředí pro karty
- fwbase/ndk/common/build - překladový systém pro nástroje
- fwbase/ndk/common/comp - komponenty pro použití při vývoji

NDK platforma poskytuje i jednotný Softwarový ovladač pro komunikaci s kartami nfb (Netcope FPGA Board). Pomocí něj lze nakonfigurovat karty, nahrát do nich firmware a konfigurovat komponenty karty. Nástroje ndp (Netcope Data Plane) jsou datové nástroje, které zajišťují DMA přenosy dat mezi kartou a počítačem. Tyto datové DMA přenosy kopírují data mezi počítačem a kartou s minimální režií procesoru. DMA přenosy mohou fungovat v několika režimech:

- SZE je typ proudového (stream) přenosu uskutečněný pomocí kruhových bufferů v počítači. Pakety se skládají za sebe zarovnané na 8 bajtů. Přes PCIe se nepřenášejí samostatné pakety, ale souvislý tok dat, což může být několik paketů najednou.
- DPDK režim je více podporován výrobci a slouží pro paketové přenosy. V kruhovém bufferu jsou uloženy odkazy na jednotlivé pakety v paměti počítače. Pomocí SZE se přenese proud dat a posléze software přenese přes PCIe pouze jeden paket.

Netcope P4

Netcope P4 je nástroj vyvíjený v rámci NDK platformy pro podporu P4 aplikací na COMBO kartách. na obrázku 2.12 je ukázáno schéma zapojení P4 firmware na kartě NFB-200G2QL. Netcope P4 aplikace se může skládat z několika Netcope P4 Atom (jádra P4) a jejich propojení s rozhraními karty a DMA moduly. Netcope P4 Atom je pak naprogramován pro vykonávání P4 programu. takže obsahuje komponenty naprogramované pro parser, match+action tabulky, deparser a kontrolní bloky programu.



Obrázek 2.12: **Schéma firmware Netcope P4** pro kartu NFB-200G2QL (převzato z [3]).

Pro kartu NFB-200G2QL byla vytvořena komponenta dma_tx_mapper. Umožňuje konfigurovatelnost propojení datových cest. Pomocí ní lze ze softwaru (hostitelského počítače) vysílat skrze DMA kanály data jak do obou P4 jader, tak i na ethernetová rozhraní karty. Pro firmware karty NFB-200G2QL jsou inicializovány dvě tyto komponenty, protože první polovinu DMA kanálu (0-15) spravuje první mapper a druhou polovinu (16-31) spravuje druhý. Ve výchozím nastavení jsou DMA kanály napojeny na P4 jádra (0-15 na jádro P4-0 a 16-31 kanálů na jádro P4-1).

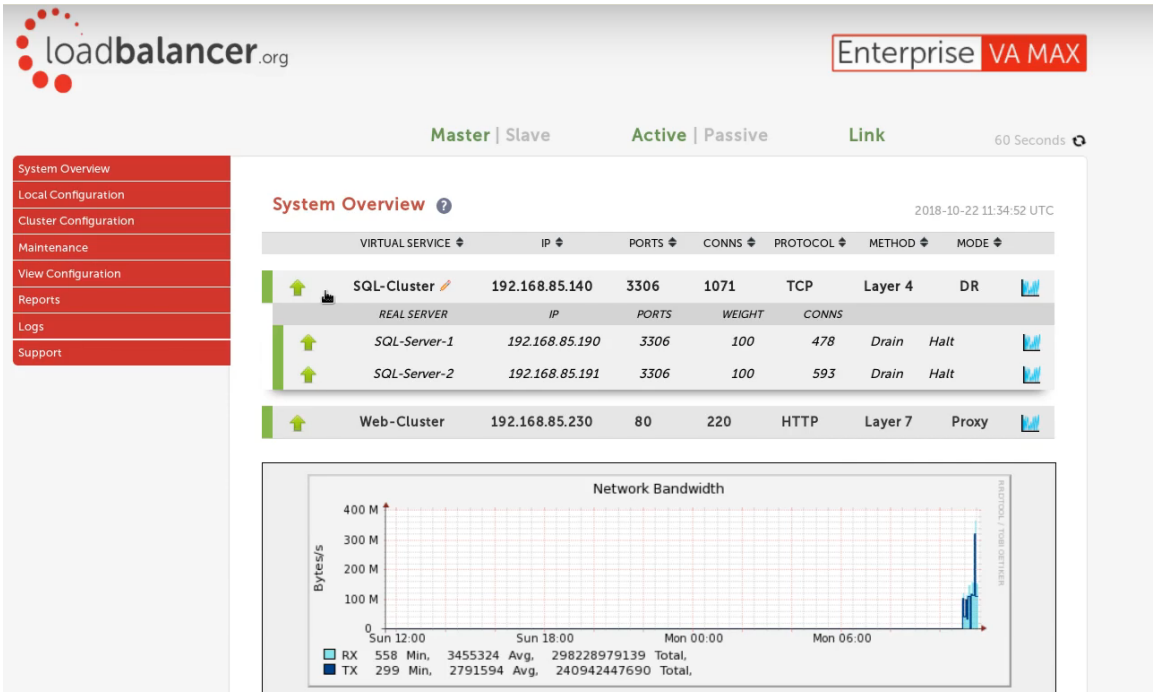
Existující kompilátor od vývojářů P4 poskytuje možnost vývoje P4 aplikací v softwaru. Avšak pro nasazení zařízení do síťového provozu je mnohem vhodnější použít technologii FPGA (Programovatelná hradlová pole - Field Programmable Gate Array) z důvodů větší

rychlosti zpracování paketů, která dosahuje i 100 Gb/s. Čipem s touto technologií disponuje i karta NFB-200G2QL. Právě sdružení CESNET vyvíjí kompilátory *P4VHDL*, které umí přenést reprezentaci z P4 jazyka do VHDL (Ve reprezentace pro dané zařízení. Následně lze VHDL kód implementovat na dané zařízení s FPGA technologií. Pro jazyk $P4_{14}$ je kompilátor založen na P4 komunitou poskytovaném P4-HLIR projektu [7], který vytvoří popis P4 programu v jazyce Python. Následně se pomocí této reprezentace vygeneruje VHDL kód pro konkrétní kartu s FPGA čipem. Pro novější verzi $P4_{16}$ kompilátor není zcela dokončen. Ten je pro změnu založen na již zmíněném P4C projektu.

2.5 Shrnutí současného stavu

V současné době se některé aplikace přemísťují do datových center a na Cloudy. Zde využívají schopnost škálovatelnosti, kdy vytvoří více instancí, aby byly schopny obsloužit všechny dotazy, zlepšili propustnost paketů a celkovou efektivitu serverů. Nebo je pro zajištění bezpečnosti rozmístěno více fyzických serverů. Je zřejmé, že takovéto služby potřebují Vyvažovač zátěže, aby mohli rozdistribuovat zátěž mezi více instancí koncových serverů.

Častá jsou softwarová řešení, která poskytují velké množství funkcí a výhod. Softwarový Vyvažovač zátěže se nazývá také Application Delivery Controller (ADC). Tyto Vyvažovače poskytují propustnosti v jednotkách Gb/s. Většinou mají příjemné a přehledné grafické uživatelské rozhraní, ve kterém lze nastavit vše potřebné pro správnou funkci Vyvažovače zátěže. Na následujícím obrázku můžeme vidět uživatelské rozhraní virtuálního (softwarového) Vyvažovače zátěže VA MAX od společnosti LoadBalancer.org. Firma se zabývá vývojem všech typů Vyvažovačů zátěže a má velice kladné reference. Na webových stránkách je k dispozici i video-návod, který podrobně popisuje toto WUI a ukazuje možnosti konfigurace. Díky softwarovému řešení jsou většinou snáze vyvíjeny a jejich ceny jsou tak nižší oproti hardwarovým Vyvažovačům zátěže.



Obrázek 2.13: **Webové uživatelské rozhraní** Vyvažovače zátěže od společnosti LoadBalancer.org (převzato z [17]).

Dalším populárním dodavatelem je společnost Kemp [24], která na svých stránkách poskytuje mnoho článků o problematice vyvažování. Obě dvě zmíněné firmy poskytují i cloudová řešení Vyvažovačů zátěže, mimo jiné i pro platformu AWS (Amazon Web Service), která patří mezi nejrozšířenější. Zároveň si lze po registraci vyzkoušet i verzi zdarma na 30 dní a nasadit ji právě na již zmíněné AWS. Tyto firmy spolu soupeří i na poli hardwarových Vyvažovačů zátěže a patří k nejlepším firmám zabývající se touto problematikou na trhu.

Obecně nelze říci, které Vyvažovače zátěže jsou lepší. Záleží na požadavcích zákazníka a specifikacích dané služby, jenž bude Vyvažovač zátěže vyvažovat.

Kapitola 3

Kritika současného stavu

Současných řešení vyvažování zátěže je velké množství. Mnoho z nich má plno skvělých nástrojů a umožňuje tak lepší nastavení systému vyvažování pro výhodnější rozložení zdrojů koncových serverů.

Softwarová řešení Vyvažování zátěže jsou velice rozšířená. Avšak jejich softwarová implementace jim braní ve snížení doby odezvy (response time / latance), protože je zpomaluje operační systém zařízení, na kterém software běží. Navíc u Vyvažovačů zátěže operujících na L7 vrstvě dochází k dalšímu zpomalování, protože paket parsují téměř celý. Potřebují tedy větší režii pro zpracování jednoho paketu a tím se doba odezvy také zvyšuje.

Další nevýhodou softwarových Vyvažovačů zátěže je propustnost. I když jsou softwarová řešení velice flexibilní a jejich propustnost se postupně zvyšuje (okolo 3Gb/s, maxima 10Gb/s viz [23]), co se týče rychlosti, nemůžou se rovnat hardwarovým Vyvažovačům. Ty jsou ve formě speciálních zařízení zapojena před koncové servery. Rychlost hardwarových Vyvažovačů zátěže se pohybuje v řádu desítek Gb/s. Jsou tedy o poznání výkonnější díky odstranění operačního systému. Využívají hardwarové zdroje přímo. Zato ale většinou neposkytují takovou flexibilitu a počet nástrojů jako ty softwarové. Navíc je vývoj hardwarových Vyvažovačů zátěže náročný a zdlouhavý, proto se tato řešení prodávají velice draze.

Na obrázku můžeme vidět hardwarové vyvažovače společnosti Kemp a jejich srovnání. Lze vidět, že nejvýkonnější Vyvažovač zátěže LM-X40 dosahuje rychlosti propustnosti 40 Gb/s, což je desetkrát větší propustnost než softwarové Vyvažovače zátěže od stejné společnosti. Tyto stroje jsou velice drahé, protože jsou implementovány v podobě výkonných serverů. Mají tedy drahé více-jádrové procesory a velkou operační paměť RAM (LM-X40 má 64GB RAM), což klade vysoké požadavky na cenu.

Model	LM-X1	LM-X3	LM-X15	LM-X25	LM-X40
USD (MSRP)	$2,500	$4,000	$9,800	$20,000	$30,000
Max Throughput (Gbps)	1	3.4	15	25	40

Obrázek 3.1: **Porovnání cen** Vyvažovačů zátěže od společnosti Kemp a jejich propustnosti (převzato z [23]).

Oproti tomu existují i síťové karty SmartNIC (Smart Network Interface Card), které implementují zpracování síťového provozu právě na dedikované NIC místo procesoru. Jádrem těchto SmartNIC může být například čip s technologií ASIC nebo FPGA, který lze snadno naprogramovat pro vykonávání konkrétní činnosti.. Nejenže tyto karty jsou k dostání za pár tisíc dolarů, ale navíc jsou schopny dosáhnout rychlosti zpracování paketů i 100Gb/s. Proto by tyto karty mohly poskytnout levnější i efektivnější řešení než drahé serverové stroje za desítky tisíc dolarů.

Kapitola 4

Návrh a implementace řešení

Tato kapitola se zabývá návrhem programu v jazyce P4 2.2, který bude schopen vyvažovat zátěž mezi více koncových serverů. Cílem je vytvořit Vyvažovač zátěže běžící na dostupné kartě a ověřit propustnost této implementace. Vývoj nejdříve probíhal v jazyce $P4_{16}$ a po sléze byl přepsán do starší verze $P4_{14}$ kvůli lepší podpoře P4VHDL překladače sdružení CESNET. Vývoj aplikace byl uskutečněn pomocí Behaviorálního modelu jazyka P4 popsáného v sekci 2.3.

Práce byla nejdříve velice obecnou, protože neexistoval konkrétní scénář (Use-case = případ užití) aplikace. Poté se však začala utvářet specifikace Vyvažovače zátěže a aplikace se přiblížila praktickému využití. Cílem se stalo vytvořit prototyp Vyvažovače zátěže pro DNS servery a nahradit tak drahé aktuální řešení. Díky rozhodnutí, o jakou službu se jedná, bylo definováno, jaké toky bude Vyvažovač zátěže vyvažovat (ty na portu 53 - DNS port).

V této kapitole je nejdříve popsána definice Vyvažovače zátěže a jeho vlastnosti, které jsou implementovány a použity. Je zde i popsáno zdůvodnění některých rozhodnutí, která se skrývala za výběrem specifikací vyvažovače. Další část kapitoly se věnuje samotnému návrhu jednotlivých komponent v jazyce P4, které jsou klíčové pro fungování tohoto zařízení.

4.1 Specifikace Vyvažovače zátěže

Prvním úkolem bylo zamyslet se nad vlastnostmi Vyvažovače zátěže a vybrat, které a jak se budou implementovat. Některá omezení byla dána i samotným Jazykem P4.

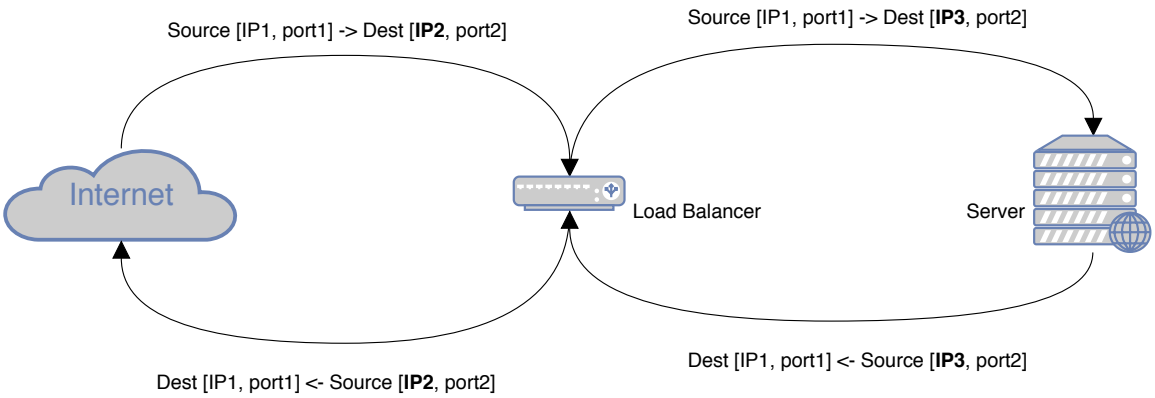
Pro správné fungování internetu je potřeba zajistit dostupnost DNS služby pro překlad domén na adresy. Aby DNS servery stihli obsloužit obrovské množství dotazů, je k dispozici více stejných DNS serverů. Vyvažovač zátěže musí umět paket rychle zpracovat a rozhodnout, na který server daný paket pošle. Následně musí Vyvažovač umět poslat paket zpět od serveru ke klientovi. Nejlépe tak, aby klient nevěděl o struktuře za Vyvažovačem. Jelikož se jedná o DNS službu, Vyvažovač bude rozkládat pouze pakety od klientů s cílovým portem 53 (DNS port) . Stejně tak bude umožňovat odpovídat serverům ze zdrojového portu 53. Pakety na ostatních transportních portech bude Vyvažovač přeposílat do softwaru, který takovéto pakety bude muset zpracovat.

Jelikož se jedná o DNS službu, která je velice používaná, hlavním požadavkem byla rychlost. Cílem bylo zkusit implementovat Vyvažovač zátěže, který by zvládal srovnatelné rychlosti produktů stejného typu na trhu (40 Gb/s). Z toho důvodu byl zvolen Vyvažovač zátěže operující pouze na L4 Transportní vrstvě ISO/OSI modelu. Pokud by Vyvažovač

zátěže měl fungovat na L7 vrstvě, znamenalo by to větší zátěž na zdroje a zpomalení zpracování paketů.

S faktorem rychlosti a využití zdrojů souvisí i volba Transparentního Vyvažovače zátěže 2.1.1. V této bakalářské práci jsem předpokládal, že Vyvažovač zátěže bude umístěn před servery, které jej budou používat jako Výchozí bránu pro DNS dotazy/odpovědi. Tudíž Vyvažovač zátěže u příchozího dotazu od klienta změní cílovou IP adresu na adresu vybraného serveru, avšak zdrojovou ponechá původní. Server zpracuje dotaz a odpoví. Paket doputuje na Vyvažovač zátěže a ten pozná, že paket přišel od jednoho ze serverů a přepíše zdrojovou adresu svou adresou, jak lze vidět na obrázku 4.1. Klientovi pak dojde odpověď z adresy Vyvažovače zátěže a tím pádem je mu struktura za ním skrytá. Tento postup zpracování zajišťuje tak i bezpečnost, protože žádný útočník nemůže vědět kolik koncových serverů je skryto za Vyvažovačem zátěže. Ale hlavně díky tomuto způsobu není třeba implementovat tabulku spojení, ve které by jinak bylo třeba udržovat záznamy o milionech spojení (podobně jako u NAT), což by bylo paměťově nákladné. Navíc vyhledávání v této tabulce by zpomalovalo zpracování paketů.

Z předchozího popisu komunikace vyplývá i další definice Vyvažovače zátěže. Vyvažovač zátěže neimplementuje mechanismy navazování či ukončování spojení s klienty. Lze ho tedy nazvat UDP/TCP propouštěcí Vyvažovač zátěže 2.1.1, protože jen pozměňuje hlavičky paketů a nevytváří nové. Tento způsob je velice příznivý, protože implementace mechanismů na ustanovení nového spojení a ukončování spojení, které jsou potřeba především u TCP komunikace, by byla velice komplikovaná především s využitím Jazyka P4. Zato takto jsou tyto mechanismy ponechány serverům.



Obrázek 4.1: Ukázka **výměny IP hlaviček** při přeposílání Vyvažovačem zátěže.

Ze základních metod vyvažování 2.1.2 jsem hned na začátku vyřadil metody Vyvažování podle nejmenšího počtu spojení, protože by bylo třeba udržovat tabulku spojení (náročné a zbytečné) a metodu Agentního vyvažování z důvodu nutnosti implementace agentů na koncové servery a softwaru pro vyhodnocování zpráv z agentů. Jak je již výše napsáno Vyvažovač zátěže byl zvolen typu L4, takže metoda Přepínaní podle obsahu L7 hlaviček nelze implementovat. Zbývají už jen dvě metody.

Rozhodl jsem se pro metodu 5-hashování. Hash je v tomto případě počítán z 5 políček a to ze: Zdrojové a cílové IP adresy, Zdrojového a cílového portu a typu Protokolu (TCP/UDP). Tato metoda poskytuje větší flexibilitu než Round Robin, protože již sama o sobě může implementovat váhy serverů. Tyto váhy poskytují možnost korigovat poměry rozložení na serverech (nastavit váhu serverů) a tak například posílat více spojení na vý-

konnější server. Díky metodě 5-hashování je zajištěno, že stejný tok vždy dojde na stejný server. I když to je pro DNS službu naprosto irelevantní, tak to poskytuje lepší základ pro budoucí rozšíření či přesun ke stavovým službám.

4.2 Návrh

Návrh je přizpůsoben situaci kdy je jeden port Vyvažovače připojen do Internetu a ostatní jsou připojeny přímo k serverům. Základním principem Vyvažovače zátěže je přeposlat paket. Vzít paket, který doputoval z Internetu a poslat jej na jeden z portů připojených k serverům.. Toho lze docílit pouhým posláním paketu výstupním portem, který je připojen ethernetovou linkou k serveru. Ale předtím než je paket odeslán výstupním portem, je třeba změnit cílové adresy paketu. Jinak by se mohlo stát, že by například ethernetová karta serveru paket zahodila, protože by cílová adresa neseděla s MAC adresou karty.

Obdobně funguje i opačný směr. Pokud server odpoví, odpověď doputuje na Vyvažovač portem připojeným k serveru. Vyvažovač musí opět paket přeposlat, ale tentokrát do Internetu. Pro tento paket nechci měnit cílové adresy jako v předešlém případě. Jinak by totiž paket nedošel ke správnému klientovi. Nyní je potřeba nastavit zdrojové adresy tak, aby klient a Internet nevěděl nic o vnitřní struktuře za Vyvažovačem. Zaměním tedy zdrojovou adresu serveru za adresu Vyvažovače. Paket je poté vyslán výstupním portem do Internetu.

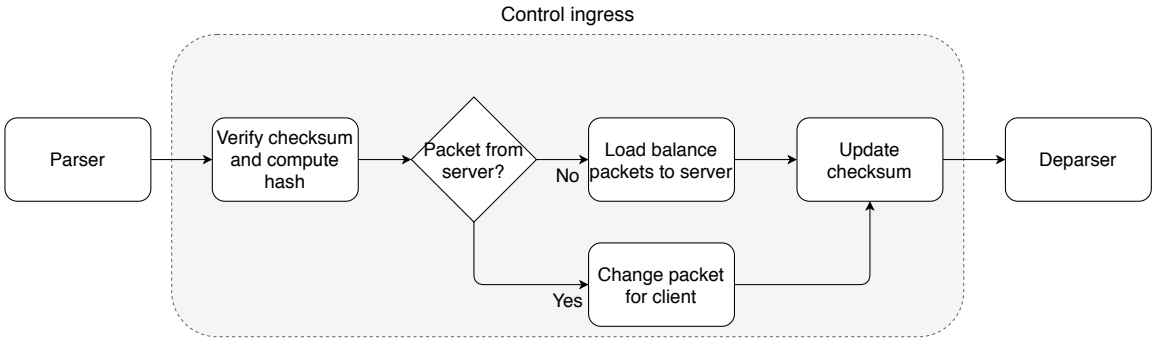
K tomuto přeposílání paketů přidám schopnost vyvažování. Paket doputuje z Internetu na Vyvažovač a ten musí rozhodnout, na který server paket pošle. K tomu je využíván algoritmus 5-hash. U každého paketu se vypočítá tato hodnota, ke které je již přiřazen adekvátní server, na který budou pakety s takovouto hodnotou 5-hash přeposílány.

Pro jednoduchost a rychlou implementaci jsem se rozhodl, že vlastnost Objevování služby (Service Discovery) Vyvažovače zátěže budu řešit staticky. Vyvažovač zátěže musí znát počet serverů a jejich adresy. V tomto případě stačí když tabulky z P4 jazyka budou mít správné záznamy o serverech mezi kterými bude zátěž vyvažována. Podobně se řeší i vlastnost Kontrola živosti serverů (Health Checking), kdy z konfiguračního souboru, tedy z pravidel v tabulkách P4 programu, Vyvažovač zátěže předpokládá, že všechny servery jsou schopny přijmout nové spojení. Pokud by došlo k pádu nějakého serveru, je třeba přepsat a nahrát nová pravidla, ve kterých již nebude tento server obsažen.

Vývoj probíhal nejdříve v jazyce $P4_{16}$ a později byl program přepsán do verze $P4_{14}$. Z důvodu přizpůsobení programu specifikacím překladače P4VHDL od sdružení CESNET byl zdrojový kód upravován, aby splnil jak požadavky tohoto překladače, tak aby byl stále funkční v Behaviorálním modelu jazyka P4 a dal se tak spustit běžném CPU. Na obrázku 4.2 je vidět logické schéma programu. Ten se skládá z:

- Parser a reverzní deperaser extrahují a následně skládají hlavičky protokolů každého paketu.
- Control ingress je jediný kontrolní blok programu. Jsou v něm obsaženy všechny tabulky. Lze ho rozdělit do několika logických částí:
  - Blok kódu pro ověření kontrolního součtu (checksum) v hlavičkách. Pro každý paket je vypočítán kontrolní součet a následně porovnán s kontrolním součtem uloženým v paketu. Pokud by se lišili, je paket zahozen. V této části dochází i výpočtu 5-hash hodnoty a kontroly TTL paketu.

– Tabulka, která zjistí, zda paket přichází od serveru nebo od klienta. Pokud najdeme shodu v této tabulce, znamená to, že paket bude vysílán Vyvažovače ke klientovi a je třeba tedy změnit zdrojové adresy paketu.
– Tabulka, která podle hodnoty 5-hash vyvažuje zátěž na servery. Již vypočítaná hodnota 5-hash paketu je porovnána se vstupy v tabulce. Záznam s touto hodnotou klíče má nastavenou akci a její parametry tak, aby přeposlal paket na právě jeden server.
– Tabulka, které mění paket ze serveru pro klienta. Pokud je zjištěno, že je paket vyslán ze serveru, je třeba změnit zdrojovou adresu paketu. Jelikož je do Internetu zapojen jen jeden port, je tato adresa jen jedna pro všechny pakety vyslané serverem.
– Blok kódu, který vypočítá nové hodnoty kontrolního součtu. Pro každý paket, který je nějak pozměněn musí být aktualizované hodnoty kontrolních součtů, aby nedošlo k zahození paketů na směrovačích po cestě paketu nebo přímo u koncového zařízení klienta.



Obrázek 4.2: **Schéma** logického toku programu.

4.3 Implementace a popis komponent v jazyce P4

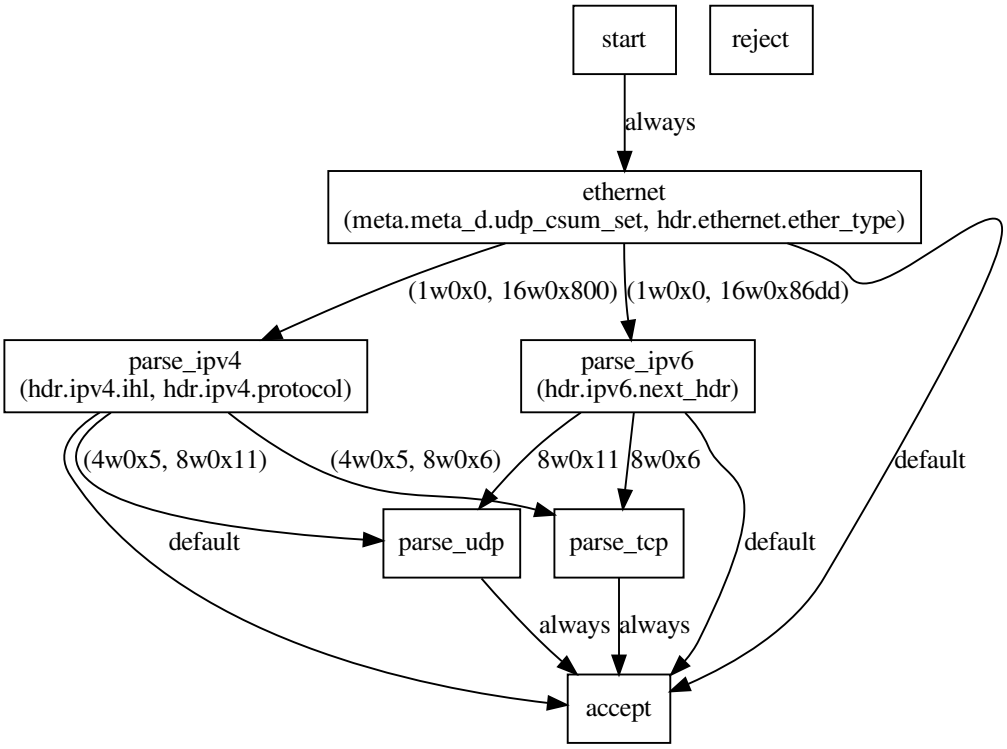
V této sekci popíši některé základní komponenty, které lze vidět na obrázku 4.2 a jejich implementaci v jazyce P4.

Parser a deparser

Základní a první komponentou pro P4 aplikaci je Parser. Jelikož Vyvažovač zátěže operuje na L4 vrstvě, potřebuje protokoly jen do této úrovně. Extrahují se zde Ethernetová hlavička, IP protokoly verze 4 i 6 a na transportní úrovni TCP a UDP protokoly viz obrázek 4.3. Parser je skutečně jednoduchý. Pro DNS službu nepředpokládám žádné zabalení hlaviček do VLAN tagu ani žádných rozšiřujících možností hlaviček.

V poslední úrovni parsování, tedy u UDP nebo TCP protokolu je v parseru nastaven příznak *fully_parsed* v mnou vytvořených metadatech pomocí funkce *set_metadata()*. Pomocí tohoto příznaku posléze v kontrolním bloku poznám, zda je paket rozparsovaný celý a mohu na něj aplikovat tabulky. Pokud příznak není nastaven, znamená to, že paket se nevyparsoval úspěšně do L4 vrstvy a tím pádem směrování nemá smysl.

Ve výše zmíněných lokálních metadatech je několik příznakových polí, jenž se nastavují při určitých akcí (například *dropped* při vykonání akce pro zahození paketu). Nachází se zde i políčko *hash*, do kterého se ukládá vypočítaná hodnota 5-hashe, která je pak vstupem do tabulky zajišťující vyvažování.



Obrázek 4.3: **Graf parseru** vytvořen pomocí nástroje *p4c-graphs*, který je obsažen v překladači P4C [9]

Deparser pracuje reverzně k parseru. Není třeba jej explicitně definovat. Začíná od koncového stavu a pokračuje směrem k počátečnímu. Skládá hlavičky, které jsou přítomny v paketu (ověřit přítomnost hlavičky lze pomocí funkce *valid*), postupně za sebe. Jakmile se dostane do počátečního stavu, je paket opět kompletně složen.

Kontrolní součet a hash výpočet

Pro výpočet hodnoty z několika políček existuje v jazyce P4 konstrukce *field_list* spolu s *field_list_calculation*. Tyto výpočty jsou důležité právě pro aktualizaci a ověření kontrolního součtu (checksum) hlaviček paketu a také pro výpočty hash funkcí. Pokud jsou hodnoty v paketu změněny mělo by vždy dojít k aktualizaci kontrolního součtu, aby nedošlo k zahození paketu na síťových zařízeních (některé OS zahazují pakety se špatným kontrolním součtem).

Následující část kódu ukazuje definici seznamu polí za využití konstrukce *field_list název_seznamu*. V této definici jsou vypsána všechna políčka z hlaviček, nad kterými bude

aplikován výpočet. Když je nadefinovaný seznam, je třeba definovat výpočet nad tímto seznamem pomocí bloku *field_list_calculation název_výpočtu*. V tomto bloku je třeba nadefinovat tři prvky.

- Prvním je vstup tohoto výpočtu pomocí klíčového slova *input*. Zde jako vstup poslouží již nadefinovaný seznam polí.
- Druhým je výběr algoritmu pro výpočet hodnoty ze seznamu. Je podporováno několik metod (xor16, csum16, crc16, crc32).
- Třetím prvkem, který je třeba pro výpočet definovat je výstupní šířka tohoto výpočtu.

```
#define HASH_RANGE 128 //Maximal range of hash function
//IPv4-UDP hash computation
field_list ipv4_udp_hash_list {
    ipv4.src_addr;
    ipv4.dst_addr;
    ipv4.protocol;
    udp.src_port;
    udp.dst_port;
}
field_list_calculation ipv4_udp_hash {
    input {
        ipv4_udp_hash_list;
    }
    algorithm : crc16;
    output_width : 16;
}
action compute_ipv4_udp_hash_action(){
    modify_field_with_hash_based_offset(meta_d.hash, 0,
         ipv4_udp_hash, HASH_RANGE);
}
table t_compute_ipv4_udp_hash_table {
    actions {
        compute_ipv4_udp_hash_action;
    }
}
```

Výpis 4.1: **Definice seznamu polí a výpočet** hash hodnoty pro paket s IPv4 a UDP hlavičkami pomocí tabulky.

Tyto výpočty se provádějí pomocí tabulek s jednou akcí. V této akci je jediná primitivní akce a to *modify_field_with_hash_based_offset(dest, base, field_list_calc, size)*. Funkce aplikuje *field_list_calculation*. Parametr *size* udává rozsah výsledku hash funkce. K mezivýsledku je připočtena hodnota *base* a toto celé je následně uloženo do políčka *dest*.

Tabulky na výpočet takovýchto hash funkcí jsou na začátku kontrolního programu ingress společně s tabulkami na výpočet kontrolního součtu přijatých paketů. Nad paketem je vypočítán kontrolní součet a hash funkce. Následně je zkontrolováno, zda se vypočítaná hodnota kontrolního součtu shoduje s hodnotou v paketu. Pokud si nejsou rovny, nad paketem je provedena akce pro zahození paketu. V této části porovnávání kontrolních součtů

dochází i k porovnání hodnot TTL paketu. Pokud by byla hodnota menší než jedna, je paket také určen k zahození.

Aktualizace hodnot kontrolního součtu je prováděna na konci kontrolního ingress programu stejným způsobem jako výpočet pro ověření správnosti.

Tabulka pro zjištění zda paket pochází ze serveru

Tato tabulka se nachází před tabulkou pro vyvažování zátěže. Její princip je velice jednoduchý. Pouze zkontroluje zdrojové IP adresy paketů a pokud se adresa shoduje s adresou nějakého serveru, provedeme akci, která nastaví příznak metadat. Pomocí tohoto příznaku posléze vím, jestli má být paket směrován na servery (aplikace tabulky t_load_balance) nebo naopak poupraven pro komunikaci s klientem (aplikace tabulky t_answer).

```
action mark_from_server(){
    modify_field(meta_d.from_server, 1);
}

table t_check_from_server {
    reads {
        ipv4.src_addr: exact;
        ipv6.src_addr: exact;
    }
    actions {
        mark_from_server;
        nop;
    }
    //default_action: nop;
}
```

Výpis 4.2: **Tabulka t_check_from_server** zkontroluje zda je paket poslaný od klienta nebo od serveru.

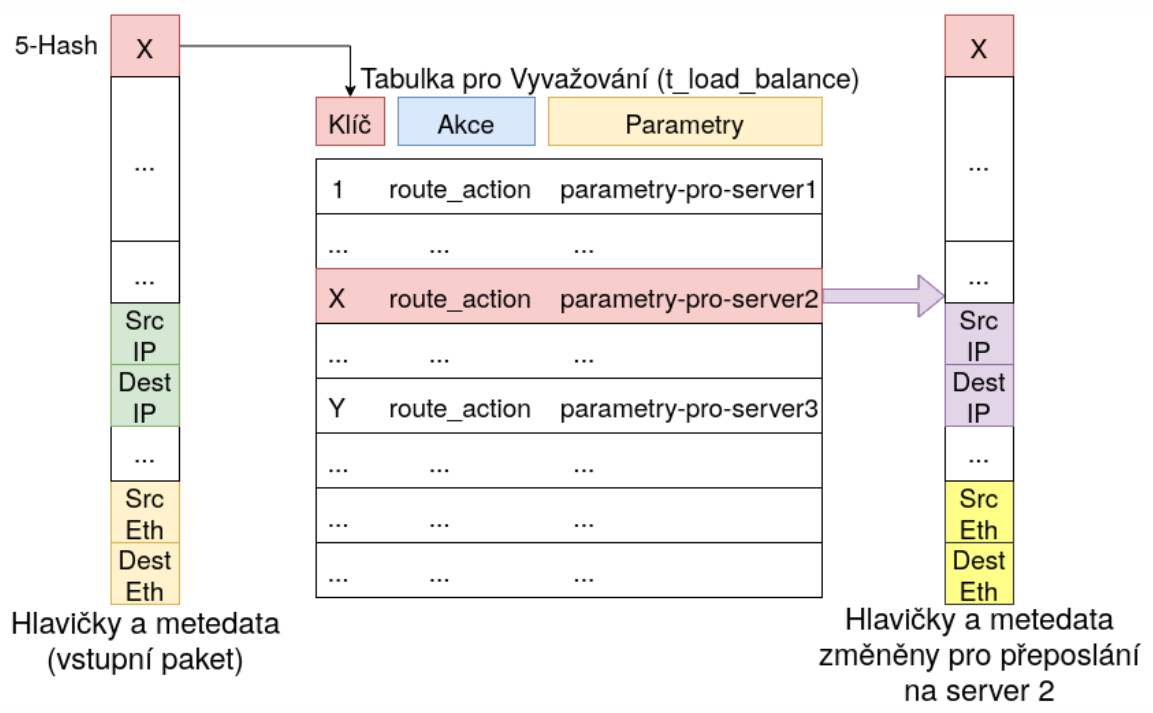
Tabulka pro vyvažování zátěže

Prvotním krokem bylo rozložit zátěž a správně směrovat, proto tato tabulka byla vyvíjena jako první. Tabulka pro vyvažování je jádrem celého programu.

Tabulka bere jako klíč vypočítanou hash hodnotu z pětice: Zdrojová i cílová IP adresa, zdrojový i cílový port a typ protokolu. Podle této hodnoty rozhoduje, na který server pošle paket. Má pouze dvě akce a to:

- drop_action akce poznačí v metadatech, že je paket určen k zahození a provede primitivní akci *drop()*. tato akce je nastavena jakou defaultní. To znamená, že se aplikuje pouze pokud se nenajde záznam v tabulce.
- route_action je akce, která pozměňuje hlavičky paketu a nastavuje výstupní port vedoucí k vybranému serveru. Pro správné fungování musí být přidané všechny pravidla pro rozsah 5-hash hodnoty. Pak by nikdy nemělo dojít k akci drop_action.

Na obrázku 4.4 je znázorněn proces vyhledávání záznamu v tabulce. K záznamu je přiražena adekvátní akce a všechny potřebné parametry sloužící k nahrazení políček v hlavičkách.



Obrázek 4.4: **Proces vyhledání záznamu** v Match+Action tabulce t_load_balance. Záznam obsahuje parametry potřebné pro přeposlání paketu k danému serveru.

Jelikož se hodnota 5-hashe počítá pomocí algoritmu CRC16 je výsledek na 16 bitech. Uvážil jsem, že je to zbytečně velký rozsah, a tak lze vidět v ukázce kódu 4.1, že parametr *size* ve funkci *modify_field_with_hash_based_offset* je nastaven na hodnoto 128 (makro HASH_RANGE). Což ve skutečnosti na vypočítanou hodnotu aplikuje operaci modulo (hash % size). Tím je zajištěn rozsah na maximální číslo 128. Proto je právě v následující ukázce nastaven počet pravidel na hodnotu 128. Tím pádem pro tuto tabulku potřebujeme 128 záznamů, aby směrování správně fungovalo.

```
action route_action(dest_eth, src_eth,
           new_ipv4_dst_addr,
           new_ipv6_dst_addr,
           port) {
    subtract(ipv4.ttl, ipv4.ttl, 1); //TTL
    subtract(ipv6.hop_limit, ipv6.hop_limit, 1); //Hop limit
    modify_field(ethernet.src_addr, src_eth); //Ethernet src
    modify_field(ethernet.dst_addr, dest_eth); //Ethernet dst
    modify_field(ipv4.dst_addr, new_ipv4_dst_addr); //IPv4 dst
    modify_field(ipv6.dst_addr, new_ipv6_dst_addr); //IPV6 dst
    modify_field(standard_metadata.egress_spec, port); // Egress port
}
//Main table for Load Balancing. Needs rules to be added during runtime.
table t_load_balance {
    reads {
        meta_d.hash: exact;
    }
```

```
    actions {
        drop_action;
        route_action;
    }
    size: HASH_RANGE;
}
```

Výpis 4.3: **Tabulka t_load_balance** rozešle pakety na servery podle vypočítaného hashe.

Akce route_action mění hlavičky paketů a nastavuje výchozí port, kterým bude paket odeslán. Parametry, které je potřeba této akci předat jsou:

- desteth je cílová MAC adresa serveru.
- srceth je zdrojvá MAC adresa Vyvažovače zátěže.
- new_ipv4_dst_addr je IP adresa verze 4 serveru, na který bude paket přesměrován.
- new_ipv6_dst_addr je IP adresa verze 6 serveru, na který bude paket přesměrován.
- port určuje výstupní port Vyvažovače zátěže, kterým paket odejde.

Jazyk P4 funguje tak, že může upravovat políčka pouze platných hlaviček. Takže pokud máme například paket s verzí 4 IP protokolu, jakékoliv primitivní akce na hlavičce IPv6 protokolu jsou ignorovány. A naopak.

Nejdříve je v této akci provedena dekrementace TTL (Time To Live) hodnoty paketu, jejíž kontrola byla provedena v sekci ověřování kontrolního součtu paketů v ingress kontrolním programu. Následně jsou nastavený Ethernetové adresy. Poté je změněna cílová IP adresa na adresu serveru. Poslední, snad nejdůležitější primitivní akcí, je nastavení cílového portu pro paket. Port určuje, kterým výstupním rozhraním bude paket vyslán.

Tabulka pro pro vrácení odpovědi klientovi

Tabulka t_answer má pouze jednu akci answer_action, která je téměř totožná s akcí route_action tabulky t_load_balance. Jediná změna v definici je, že neupravuje cílovou IP adresu, nýbrž zdrojovou. Od tabulky t_load_balance se liší také nahranými pravidly. Obsahuje totiž jediné defaultní pravidlo pro vykonání této akce s takovými parametry, že zdrojovou IP adresu serveru přepíše svou, aby zachoval zastínění infrastruktury.

Kapitola 5

Testování

V této kapitole bude popsán způsob testování a dostupné programy/skripty pro ulehčení testování a lepší manipulaci s testovacím prostředím. Testování probíhalo spíše funkcionální, za účelem verifikace programu a ověření správnosti. Následně byla snaha o dosažení požadované propustnosti 40Gb/s.

Ověřování nejdříve probíhalo pouze softwarově s využitím Behaviorálního modelu P4 BMv2. Za tímto účelem je vytvořeno několik skriptů, které napomáhají lepšímu utváření testovacího prostředí a spouštění celého programu. V pozdější fázi se testování a verifikace přesunula do prostředí sdružení CESNET, kde pomocí dostupných nástrojů od společnosti Netcope byla proveden funkční test na kartě NFB-200G2QL. V tomto stádiu testování bylo cílem dosažení požadované rychlosti zpracování a přeposílání paketů.

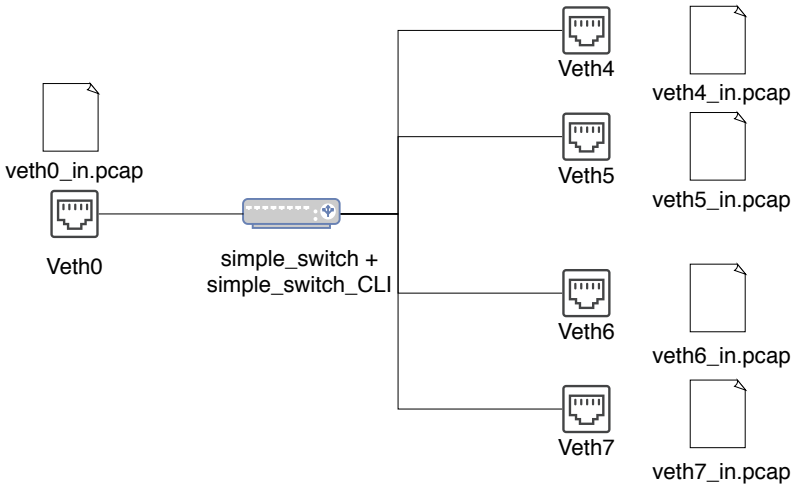
5.1 Simulace v BMv2

V této sekci popíši způsob simulace v Behaviorálním modelu a mnou vyvinuté skripty pro správné a snadné spouštění aplikace.

Behaviorální model a dostupné překladače pro něj umožňují okamžité spuštění P4 aplikace na procesoru běžného počítače. Různé nástroje umožňují přímé nahrávání pravidel do aplikace a dokonce posílat pakety a zachytávat výstupní pakety. V rámci práce jsem vyvinul několik malých skriptů, které mají zlepšit manipulaci, případně opravit některé nedostatky překladače.

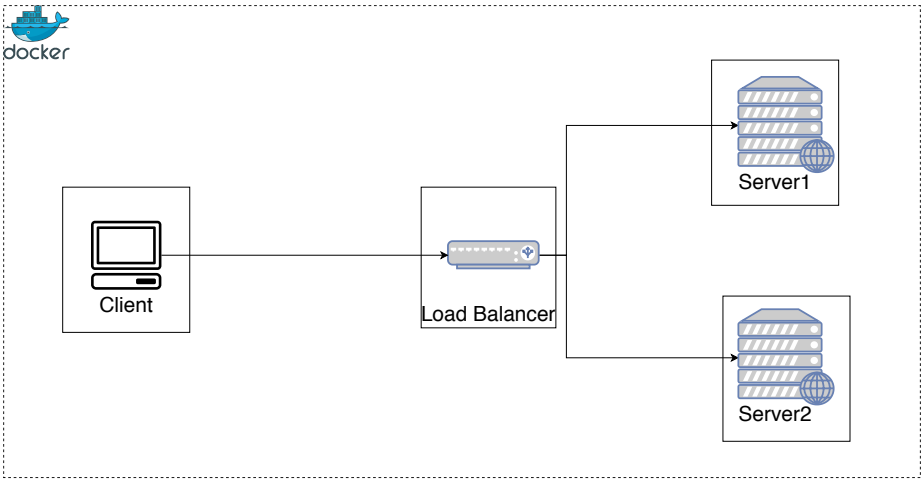
Zapojení

Obrázek 5.1 znázorňuje schéma mého behaviorálního testování. Toto testování bylo spouštěno pouze na jednom stroji za pomocí virtuálních rozhraních (veth) a paketů uložených v pcap souborech. V tomto případě není ani nutné vytvářet virtuální ethernetová rozhraní. Po zpracování pcap souborů jsou výstupní pakety uloženy do <iface0_out>.pcap souborů pro každé rozhraní.



Obrázek 5.1: Schéma testovacího **zapojení** aplikace Vyvažovače v **Behaviorálním modelu** P4 jazyka pomocí virtuálních ethernetových rozhraních.

Pro druhou fázi testování jsem použil prostředí společnosti Docker. Zde jsem měl zapojen pouze jednoho klienta a dva servery. Každý prvek včetně Vyvažovače zátěže běžel ve vlastním kontejneru. Jako základ kontejneru byl použit vzor (image) operačního systému Ubuntu 14.04. Pro funkční testování bylo využito jednoduchých aplikací v jazyce C poskytnutých v rámci studia předmětu ISA (Síťové aplikace a správa sítí). Klientský program posílal data ve formě textu zadaného klientem. Vyvažovač zátěže směroval dotazy na servery podle vypočítané hash hodnoty. Server přijatý text dal do formátu velkých písmen a poslal zpět. Klient přijal paket a vypsal překonvertovaný text. Platformu docker jsem zvolil z důvodu snahy co nejvíce odstínit aplikaci Vyvažovače zátěže od operačního systému mého počítače, ale zároveň pro ověření přijímaní paketů operačními systémy klientů a serveru (správné kontrolní součty a hlavičky paketů).



Obrázek 5.2: Schéma **zapojení** klientů a serverů přes Vyvažovač zátěže pomocí **kontejnerů** v prostředí Docker.

Automatizace vyhodnocení takto komplexních testů je dosti obtížné, proto k vyhodnocení správnosti aplikace musím po každém testu zhodnotit výsledek rozložení zátěže

manuálně. Teprve po analýze výstupních pcapů pomocí nástroje Wireshark nebo tcpdump lze posoudit, zda pakety z veth0 (klienti) byli úspěšně rozděleny mezi rozhraní veth4-7 (servery) a naopak, že pakety ze serverů odešly na klienty. Kromě kontroly ethernetových portů pro komunikaci server-klient, je třeba kontrolovat i port pro Software. Do něj putují pakety, které nevyhovují specifikaci DNS požadavku (nepodařilo se je rozparsovat na úroveň transportní vrstvy nebo mají špatně nastavené porty transportní vrstvy).

Testování

Testování v Behaviorální modelu sloužilo především za účelem verifikace. Vzhledem k pomalému zpracování paketů nemělo smysl sledovat propustnost paketů. Já jsem při testování používal již vyhotovené pcap soubory obsahující mnou požadované pakety. Ty je samozřejmě nutné vytvořit ještě před spouštěním aplikace. Avšak je možné posílat pakety rovnou například skrze nástroj Scapy napsaný v jazyce Python. Právě pomocí Scapy vytvářím a ukládám pakety do pcap souboru.

Pro kompilování jsem použil komunitou poskytovaný překladač P4C. Tento překladač vygeneruje dva soubory. Jeden s příponou p4i. A druhý s příponou JSON a právě ten je použit pro spouštění P4 aplikace. Příklad příkazu pro překlad je následující:

p4c –target bmv2 –arch v1model –std p4-14 <prog>.p4 -o

Parametry jsou použity podle specifikace Behaviorálního modelu 2.3. Kvůli neúplné podpoře překladu P4C kompilátoru pro výpočet kontrolního součtu musím aplikovat skript, který zajistí správný výpočet kontrolního součtu 5.1. Následujícím krokem je vytvoření pravidel pro aplikaci. Je možné za tímto účelem použít mnou vytvořený skript rules_generator.py 5.1, který vygeneruje všechna potřebná pravidla. Avšak lze si pravidla také napsat ručně nebo je přímo za běhu vytvářet a přidávat do tabulek.

Pro samotné spuštění aplikace používám nástroj simple_switch. Je nutné tomuto nástroji předložit seznam ethernetových rozhraních, které chceme použít, pomocí parametru "-i 0<iface0>". Pomocí argumentu –log-console se aktivuje vypisování údajů o zpracovávaném paketu na standardní výstup. Podrobně sděluje, kterou tabulkou prošel, jaké akce na něj byly aplikovány a s jakými hodnotami. Jsou zde i údaje o rozhodování ve větvících příkazech (zda paket vstoupil do if větve nebo ne).

Na běžící aplikaci se nyní můžeme připojit pomocí nástroje simple_switch_CLI. Aplikace simple_switch si totiž vytvoří server RPC a nástroj simple_switch_CLI se připojí na port tohoto serveru, aby mohl komunikovat s procesem aplikace. Pomocí tohoto nástroje nahrajeme pravidla do potřebných tabulek. Po tomto kroku již aplikace pracuje jak má a bude přeposílat pakety na virtuální rozhraní podle pravidel přidaných do tabulky t_load_balance.

Parametrem "–use-files seconds"zadanému nástroji simple_switch nastavujeme za jak dlouho po spuštění se začnou zpracovávat pcap soubory. Nutno podotknout, že pro každé rozhraní v příkazu simple_switch (parametr -i) musíme mít připravený vstupní pcap, klidně i prázdný. Zpracované pakety budou následně uloženy do výstupních pcap souborů daných rozhraních, díky čemuž můžeme kontrolovat správnost aplikace. Je potřeba nastavit vhodnou dobu, za kterou se začnou pakety zpracovávat. Musí to být vždy poté, co se nahrají pravidla do tabulek, jinak program nebude fungovat podle očekávání.

Payload v kontrolním součtu

Kontrolní součty se počítají stejně jako hash hodnoty (viz 4.1. Například transportní protokoly (UDP a TCP) do těchto výpočtů zahrnují i data paketu. Pro to existuje klíčové

slovo *payload*, které představuje nevyextrahovaná data paketu. Toto klíčové slovo musí být napsáno v definici seznamu políček, ze kterých se počítá výsledný kontrolní součet. Ve specifikaci jazyka $P4_{14}$ je uvedeno, že se kontrolní součty počítají pomocí definice bloku *calculated_field*. Avšak pro kompatibilitu s překladačem pro FPGA karty P4VHDL byly výpočty implementovány stejně jako výpočty hash funkcí (pomocí tabulky a defaultní akce pro výpočet).

V tomto použití za pomocí tabulek však překladač P4C nepodporuje definování paylodu paketu v seznamu políček pro výpočty. Proto jsem vytvořil jednoduchý skript **json_parse.py**, který je aplikován hned po překladu P4C překladačem. Jeho funkce je najít JSON reprezentaci seznamu políček pro výpočty, které mají obsahovat tento payload a přidat jej do definice. Díky tomu je při následném spuštění zajištěno správné vypočítání kontrolního součtu v paketech.

```
"calculations": [
    ...
    {
    "algo": "csum16",
    "id": 5,
    "name": "calc_4",
    "input": [
        ...
        {
          "type": "payload",
          "value": "None"
        }
        ]
    }
    ...
]
```

Výpis 5.1: **Skript json_parse.py** přidá do definice seznamu políček při výpočtu položku *payload* v json reprezentaci programu.

Generování pravidel

Při každém spuštění se musí do aplikace nahrát pravidla pro tabulky. Pomocí těchto pravidel se řídí celý tok P4 programu a aplikují se určité akce na paket. Pro lepší konfigurovatelnost a automatické vytváření pravidel jsem vytvořil jednoduchý modul.

Skript **rules_generator.py** bere na vstup 2 argumenty. První je JSON soubor s definovanou strukturou parametru pro vytváření pravidel (viz níže) a druhým je výstupní soubor. Tento skript prochází JSON reprezentaci, ve které jsou nadefinované servery s adresami a adresy Vyvažovače zátěže a vytváří adekvátní pravidla. Díky tomuto není třeba vypisovat například 128 pravidel ručně pro tabulku t_load_balance, kde každý vstup potřebuje 5 parametrů.

Následující ukázka představuje část obsahu JSON souboru pro inicializaci pravidel pro tabulku t_load_balance. V popisu mého Vyvažovače zátěže 4.1 jsem uvedl, že vlastnost Objevování služby a Kontrola živosti se implementuji staticky. Tím je myšlen právě tento soubor. V něm si mohu nadefinovat na kolik serverů budu rozkládat zátěž a jejich adresy. Po spuštění skriptu se vytvoří pravidla pro rovnoměrné rozložení (každý server má

"128/počet_serverů"pravidel). Tento způsob rozkládání lze snadno upravit pro jiné rozložení zajišťující váhy serverů.

```
"t_load_balance":[
     {"dstEth": "f8:f2:1e:39:89:a1",
      "srcEth": "02:42:ac:12:10:03",
       "dstIp": "11.11.11.11",
       "dstIp6": "2001:4f8:011:011:2e0:81ff:fe52:9a11",
       "port": 4
     },...
],
```

Výpis 5.2: **Ukázka části JSON souboru**, pomocí kterého se vytvářejí pravidla. Slouží jako staticky konfigurátor počtu serverů a jejich adres.

Vygenerovaná pravidla jsou již v adekvátním formátu a lze je přímo použít a nahrát do P4 aplikace. Pomocí tohoto skriptu se generují veškerá potřebná pravidla včetně defaultních pravidel. Zde je definice přidání pravidla do tabulky v pseudokódu:

```
table_add <table_name> <action> <keys...> => <parametres...>
```

5.2 Ověření na kartě

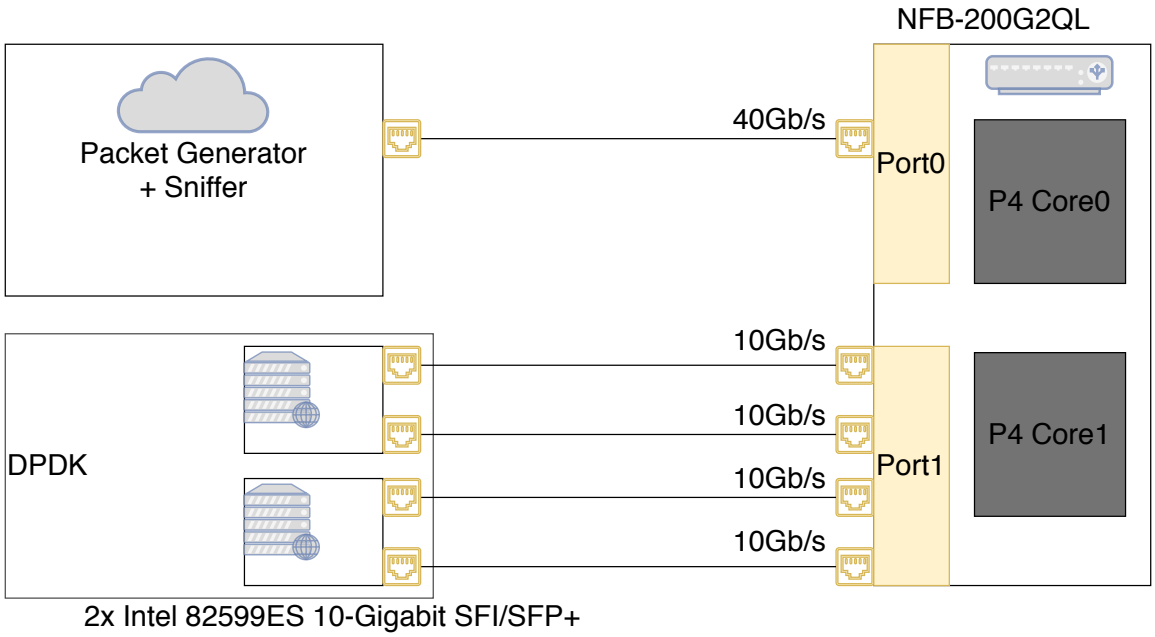
Po dostatečné verifikaci programu a dostatečném vývoji aplikace, byla práce přesunuta do prostředí sdružení CESNET na kartu NFB-200G2QL. Překladač P4VHDL převede reprezentaci z vysokoúrovňového jazyka P4 do jazyka VHDL. Následně je potřeba vybrat design naší karty a implementovat naše řešení.

Pro obsluhu samotné karty jsem používal nástroje nfb a ndp 2.4. Pro nahrávání pravidel jsem použil knihovnu libp4dev a nástroje p4test. Generování pravidel pro tabulky funguje totožně s generátorem pravidel pro behaviorální model, jen je změněn syntax pravidel podle konvencí p4testu. Pakety opět posílám pomocí již dříve vytvořených pcap souborů.

Zapojení prostředí

Karta NFB200G2QL má dva ethernetové porty dosahující maximální rychlosti 100Gb/s. V mém případě je jeden port zapojen na rychlosti 40Gb/s a druhý port rozdělen pomocí transcieverů na 4 porty o rychlosti 10Gb/s. Z tohoto návrhu zapojení vycházelo i testování v Behaviorálním modelu.

Vize testu byla taková, že by se generoval tok paketů (nebo se použil pcap soubor) v druhé NFB kartě. Ta by byla spojená s mou kartou portem s rychlostí 40Gb/s. P4 program by zpracoval pakety a rozdělil zátěž mezi desítkové porty. Naproti těmto čtyř portům karty NFB by byly zapojeny dvě dvou-portové ethernetové karty Intel. Pro ověření plné rychlosti ve full-duplex zpracování paketů P4 aplikací jsem se seznámil s DPDK projektem. DPDK program byl použit pro běh na ethernetových rozhraních Intel karet. Jeho funkce je prostá, vymění zdrojové a cílové adresy a porty v hlavičkách paketu a přidá DNS odpověď. Výsledný paket odešle zpět stejným rozhraním, jakým ho přijal. Pakety z DPDK aplikace by doputovaly zpět do P4 jádra a to by poznalo, že má pakety směrovat zpět na původní rozhraní do Internetu (40Gb/s port). Zapojení celého aparátu lze vidět na obrázku 5.3 .



Obrázek 5.3: Schéma **zapojení v hardware na kartě NFB-200G2QL**. Proti ni jsou připojeny dvou-portové karty Intel, na nichž běží DPDK aplikace pro simulaci odpovědí ze serverů. Na port s rychlostí 40Gb/s je připojen generátor DNS dotazů (při testu druhá karta NFB-200G2QL).

Syntéza a implementace firmwaru

Samotné implementací předchází syntéza. Při ní dochází k převodu VHDL kódu na reprezentaci pomocí logických prvků. Následuje implementace tohoto logického obvodu, kdy se logické prvky mapují na fyzické zdroje dané architektury. Výsledkem je zpráva o využití zdrojů na zařízení, zpráva o splnění časování a požadované frekvence a bitstream (sekvence bitů = konfigurační soubor). Pomocí výstupního bitstreamu se následně inicializuje (bootuje) vybraná karta.

Výňatek ze zprávy o využití zdrojů na FPGA čipu je znázorněn v tabulce 5.1. V tabulce jsou hodnoty využití zdrojů Vyhledávacích Tabulek (LookUp Table), registrů a Blokové paměti RAM (Block RAM). V levém sloupci se nachází požadované zdroje celého firmwaru. V pravém sloupci jsou pak hodnoty, které zabírá P4 aplikace. V těchto hodnotách jsou zahrnuty požadavky pro obě dostupná jádra na kartě NFB-200G2QL. V obou sloupcích je k hodnotám v závorce uvedeno procentuální využití zdrojů vzhledem k celkovým dostupným prvkům karty.

Tabulka 5.1: **Využití zdrojů** FPGA čipu po implementaci.

	Celkový Firmware	P4 Top (aplikace P4)
LUT (LookUp Table)	475864 (60%)	211444 (26%)
Registry	572238 (36%)	294542 (18%)
BRAM	941 (65.35%)	118 (18%)

Testování

Testovací prostředí bylo úspěšně zapojeno a podařilo se mi zprovoznit všechny dílčí prvky. Po nahrání designu na kartu je třeba aktivovat vysílání a přijímaní paketů rozhraními na kartě pomocí *nfb-eth -e 1*. Následně je třeba nahrát pravidla do obou P4 jader pomocí nástroje *p4test* a zapnout DPDK aplikaci na definovaných rozhraních.

Pomocí nástroje ndp-transmit jsem poslal pakety přes jinou kartu NFB na mou aplikaci. V následující testu jsem zvolil nižší rychlost přenosu pro demonstraci správnosti mé aplikace spolu s plnou funkčností prostředí.

```
$ ndp-transmit -i 0 -f flow0.pcap -l 7
------------------------------ NDP transmit stats ----
Packets                     :                28672
Bytes                       :              2093056
Avg speed [Mpps]            :                 38.851
Avg speed L1 [Mb/s]         :                30148.336
Avg speed L2 [Mb/s]         :                23932.184
Time                        :                    0.001
```

Výpis 5.3: **Nástroj ndp-transmit** přepošle pakety přes DMA kanály do karty, která pakety následně přepošle na kartu NFB s aplikací Vyvažovače zátěže v jazyce P4. V tomto příkladu testu přenos dosahuje rychlosti přibližně 30Gb/s.

Stejný počet paketů, který doputoval na mou mou kartu s P4 programem, se podařilo aplikaci rozhodit na 4 rozhraní s rychlostí 10Gb/s. Tudíž v P4 programu nedošlo k zahlcení a zpracovaly se všechny pakety jak měly. Kontrola spočívala ve správnosti počtu paketů na rozhraních karty. To jak mezi jednotlivými rozhraními, tak i v rámci jednoho rozhraní u počtu přijatých a odeslaných. Na výňatku z výstupu nástroje nfb-eth lze vidět, že na rozhraní 4 byl odeslán určitý počet paketů do DPDK aplikace (TXMAC Status). Stejný počet byl poté přijat (RXMAC status). Tedy na každý dotaz přišla z DPDK aplikace odpověď. Pakety se liší ve velikosti, protože DPDK aplikace k paketům přidává DNS odpověď. Proto jsou odeslané pakety (dotazy) menší než ty přijaté (odpovědi).

```
---------------------------------- Ethernet interface 4 ----
Speed                           : 10 Gb/s
Transceiver status              : OK
Transceiver cage                : QSFP-1
---------------------------------------- RXMAC Status ----
Received octets                 :               666624
Processed                       :               7168
Received                        :               7168
---------------------------------------- TXMAC Status ----
Transmitted octets              :               551936
Processed                       :               7168
Transmitted                     :               7168
```

Výpis 5.4: **Pomocí nástroje nfb-eth** lze sledovat statistiky přijatých a odeslaných paketů na rozhraních karty NFB. Zde je stejný počet odeslaných i přijatý paketů na rozhraní 4 karty NFB-200G2QL.

Knihovna libp4dev a p4test

Multiplatformní knihovna libp4dev je napsána v jazyce C. Slouží pro komunikaci mezi softwarem a FPGA zařízením. Každý prvek P4 jazyka zde má vytvořený vlastní modul. Všechna potřebná data jako adresy a offsety prvků (tabulek a akcí) v P4 jádře jsou uložena v souboru *devicetree* ke každému designu zařízení. Knihovna tedy mapuje prvky P4 programu na jednotlivé komponenty FPGA čipu popsané v devicetree. Poskytuje konfiguraci Match+Action tabulek pravidly a spravuje P4 jádro.

Nástroj p4test umožňuje snadné nahrání pravidel pro Match+Action tabulky do karet s P4 jádrem podporující libp4dev. Funguje formou příkazového interpreta, kdy příkazy zpracuje a předá rovnou knihovně libp4dev. Specifikováním parametrů –device a –core můžeme vybrat jaké zařízení a jaké jádro tohoto zařízení chceme konfigurovat. Díky parametru –core můžeme docílit jiného chování mezi jádry a tak i větší flexibility našeho zařízení. Názvy tabulek jsou stejné jako v P4 programu a klíče a parametry se definují vždy ve dvojici s názvem políčka <header.field>=<value>. Syntax přidání pravidla se mírně liší oproti behaviorálnímu modelu.

```
table <table_name> add-rule <keys...> perform <action> <parametres...>
```

Je třeba použít speciální příkazy pro správný běh a zapnutí P4 aplikace. Důležitými příkazy pro P4 jádro jsou:

- reset - Příkaz vynuluje stav P4 zařízení (jednoho P4 jádra) a vyprázdní tabulky. Uvede zařízení do tzv. známého stavu.
- enable / disable - Zapne resp. vypne zpracování paketů P4 jádrem. Pro správné fungování provozu je potřeba zapnout všechna jádra dostupná na kartě.

5.3 Shrnutí výsledků

Analýza výsledků probíhala čistě manuálně pomocí kontroly výstupních pcap souborů a čítačů paketů. Testováním P4 aplikace jsem ověřil správnost návrhu a funkčnost programu. Program pracoval podle specifikací a fungoval jak v Behaviorálním modelu, tak v testovacím prostředí na kartě NFB-200G2QL. Na kartě se mi podařilo dosáhnout rychlosti zpracování dotazů od klientů v P4 aplikaci až 40Gb/s

Avšak jsem neprokázal tuto rychlost při zpracování dotazů i odpovědí zároveň kvůli problémům s komunikací jednotlivých komponent. S určitostí se mi nepodařilo lokalizovat tento problém. Ale jelikož jsem všechny pakety zpracované P4 jádrem na DMA kanálu (port 128 a výše) zachytil, není problém ve zpracování paketů P4 aplikací.

Při větším množství poslaných paketů se začaly pakety ztrácet a ani nedoputovaly do DPDK aplikace. Předpokládal jsem že tento problém souvisí s rychlostí posílání paketů, avšak testováním jsem zjistil, že problémovým faktorem je v množství. Po zkoumání a konzultacích jsem došel závěru, že se chyba vyskytuje v komunikaci mezi desítkovými transcievery a Intel kartami.

Pro dokončení plného testování je třeba eliminovat problém v komunikaci a pokusit se replikovat testovací prostředí. Případně vytvořit jiné prostředí, ve kterém by bylo možno ověřit plnou rychlost zařízení.

Kapitola 6

Závěr

Cílem bakalářské práce bylo vytvořit Vyvažovač zátěže síťového provozu s využitím vysokoúrovňového jazyka P4 a návrh implementovat do čipu s technologií FPGA. Při návrhu řešení byl kladen důraz na rychlost zpracování paketů a co největší propustnost takového zařízení. Výsledná implementace byla úspěšně přenesena v podobě firmwaru na kartu NFB-200G2QL s čipem FPGA. Testovací prostředí bylo úspěšně zapojeno a P4 aplikace správně fungovala.

V počáteční fázi práce jsem se nejprve musel seznámit s problematikou Vyvažování zátěže, vlastnostmi samotného Vyvažovače a různými druhy algoritmů pro rozkládání síťového provozu. Zaměřil jsem se především na principy, které by mohli poskytnout lepší výkonnost. Následně jsem se musel obeznámit se standardy jazyka P4.

Návrh Vyvažovače zátěže byl zaměřen na rychlost zpracování paketů a vysokou propustnost. Návrh byl upravován tak, aby byla možná bezproblémová implementace pomocí jazyka P4. V rámci vývoje byli funkcionální testy prováděny za využití simulace v Behaviorálním modelu. Na základě výsledků simulací byl návrh revidován a upraven do finální podoby.

K přenosu vysokoúrovňového programu P4 na kartu NFB-200G2QL bylo třeba se seznámit s překladačem P4VHDL, který umí P4 program přenést do reprezentace ve VHDL.

Pro přenos P4 aplikace na kartu bylo nutné se seznámit s nástroji platformy NDK jako jsou ndp, nfb nebo knihovna libp4dev. Tuto knihovnu používá projekt p4test pro konfiguraci zařízení za běhu skrze naplňování tabulek P4 aplikace. Výsledný firmware byl za použití těchto nástrojů odzkoušen v definovaném prostředí na kartě NFB-200G2QL. Vyvažovač zátěže splnil funkční testování a pracoval správně.

Výsledkem práce je tedy funkční Vyvažovač síťového provozu s možností redefinování počtu koncových serverů za běhu pomocí externího nástroje p4test, který je v podobě firmwaru nasazen na kartě NFB. Implementace na této kartě dosahuje rychlosti zpracování paketů 40Gb/s. Tento způsob řešení na FPGA čipu poskytuje mnohanásobně levnější řešení než drahé hardwarové Vyvažovače tvořené výkonnými serverovými stroji. Navíc SmartNIC karty nesoucí FPGA čipy již dosahují rychlosti 100Gb/s, takže je zde možnost i rychlostního zlepšení.

Pro plné dokončení testování by bylo dobré vyřešit problém s komunikací mezi transcievery a Intel kartami, a ověřit tak plnou propustnost P4 aplikace. Následným zlepšením by mohlo být pokusit se aplikovat toto řešení na vyšší rychlost zpracování paketů, například 100Gb/s. Dalším zajímavým pokračování by mohl být posun zařízení k flexibilnímu přizpůsobení se změnám aktuálního sítového prostředí například poznání výpadku jednoho z koncových serverů a přizpůsobení se této změně.

Literatura

[1] *CESNET* [online]. [cit. 2020-30-4]. Dostupné z: https://www.cesnet.cz/.

[2] Aghdai, A. *Transport-Layer Load Balancing at Core and Edge Clouds*. 2019. Disertační práce. New York University Tandon School of Engineering.

[3] a.s., N. T. *Netcope P4: User Guide*. Netcope, 2020.

[4] Benáček, P., Puš, V., Kořenek, J. a Kekely, M. Line rate programmable packet processing in 100Gb networks. In: IEEE. *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*. 2017, s. 1–1. ISSN 946-1488.

[5] Bosshart, P., Daly, D., Gibb, G., Izzard, M., McKeown, N. et al. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review*. ACM New York, NY, USA. 2014, sv. 44, č. 3, s. 87–95. ISSN 0146-4833.

[6] Chapman, R. *How to balance traffic without server getting crashed?* [online]. Lime Proxies, červen 2018 [cit. 2020-24-4]. Dostupné z: https://limeproxies.com/blog/how-to-balance-traffic-without-server-getting-crashed/#load.

[7] Consortium, T. P. L. *P4-HLIR* [online]. [cit. 2020-24-4]. Dostupné z: https://github.com/p4lang/p4-hlir.

[8] Consortium, T. P. L. *P4 Language Consortium* [online]. [cit. 2020-24-4]. Dostupné z: https://p4.org/.

[9] Consortium, T. P. L. *P4C* [online]. [cit. 2020-24-4]. Dostupné z: https://github.com/p4lang/p4c.

[10] Consortium, T. P. L. *P4C-BM* [online]. [cit. 2020-30-4]. Dostupné z: https://github.com/p4lang/p4c-bm.

[11] Consortium, T. P. L. *Behavioral Model (bmv2)* [online]. 2014 [cit. 2020-30-4]. Dostupné z: https://github.com/p4lang/behavioral-model.

[12] Consortium, T. P. L. *P4 Language Tutorial* [online]. 2017 [cit. 2020-30-4]. Dostupné z: https://p4.org/assets/P4_tutorial_01_basics.gslide.pdf.

[13] Consortium, T. P. L. *The P4 Language Specification* [online]. The P4 Language Consortium, listopad 2018 [cit. 2020-30-4]. Dostupné z: https://p4.org/p4-spec/p4-14/v1.0.5/tex/p4.pdf.

[14] Kekely, L. *NDK Platforma: Školenie FW nováčikov TMC*. Božetěchova 1/2, 612 00 Brno-Královo Pole: Cesnet, 2019.

[15] Klein, M. *Introduction to modern network load balancing and proxying* [online]. Envoy, prosinec 2017 [cit. 2020-24-4]. Dostupné z: https://blog.envoyproxy.io/introduction-to-modern-network-load-balancing-and-proxying-a57f6ff80236.

[16] Liberouter. *NetCOPE* [online]. [cit. 2020-30-4]. Dostupné z: https://www.cesnet.cz/.

[17] LoadBalancer.org. *Load Balancer* [online]. [cit. 2020-30-4]. Dostupné z: https://www.loadbalancer.org/.

[18] Miao, R., Zeng, H., Kim, C., Lee, J. a Yu, M. Silkroad: Making stateful layer-4 load balancing fast and cheap using switching asics. In: *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. 2017, s. 15–28. ISBN 9781450346535.

[19] Openstack. *NetworkLoadBalancingIntegrationsWithQuantum* [online]. [cit. 2020-30-4]. Dostupné z: https://wiki.openstack.org/wiki/NetworkLoadBalancingIntegrationsWithQuantum.

[20] Postel, J. *User Datagram Protocol* [online]. Srpen 1980 [cit. 2020-30-4]. Dostupné z: https://tools.ietf.org/html/rfc768.

[21] Postel, J. *Internet Protocol* [online]. Září 1981 [cit. 2020-30-4]. Dostupné z: https://tools.ietf.org/html/rfc791.

[22] Postel, J. *Transmission Control Protoco* [online]. Září 1981 [cit. 2020-30-4]. Dostupné z: https://tools.ietf.org/html/rfc793.

[23] Technologies, K. *Compare Kemp LoadMaster, F5 Big-IP I& Citrix Netscaler* [online]. [cit. 2020-30-4]. Dostupné z: https://kemptechnologies.com/emea/compare-kemp-f5-big-ip-citrix-netscaler-virtual-load-balancers/.

[24] Technologies, K. *Kemp* [online]. [cit. 2020-30-4]. Dostupné z: https://kemptechnologies.com/.