VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ
DEPARTMENT OF COMPUTER SYSTEMS

TVORBA SIMULAČNÍCH MODELŮ JAZYKA P4

CREATION OF SIMULATION MODELS FOR P4 LANGUAGE

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — **MIROSLAV BULIČKA**
AUTHOR

VEDOUCÍ PRÁCE — **Ing. RADEK IŠA**
SUPERVISOR

BRNO 2018

Abstrakt

Sdružení CESNET vyvíjí nástroj umožňující spouštět P4 programy na programovatelném hradlovém poli, tento nástroj je využit při návrhu digitálního obvodu, který prochází procesem verifikace. Právě za účelem verifikace je v této práci vytvářen simulační model.

Abstract

Do tohoto odstavce bude zapsán výtah (abstrakt) práce v anglickém jazyce.

Klíčová slova

jazyk P4, simulační model,...

Keywords

P4 language, simulation model,...

Citace

BULIČKA, Miroslav. *Tvorba simulačních modelů jazyka P4*. Brno, 2018. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Radek Iša

Tvorba simulačních modelů jazyka P4

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Radka Išy Další informace mi poskytl Ing. Pavel Benáček, Ph.D.. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Miroslav Bulička
28. května 2019

Poděkování

V této sekci je možno uvést poděkování vedoucímu práce a těm, kteří poskytli odbornou pomoc (externí zadavatel, konzultant, apod.).

Obsah

Kapitola 1

Úvod

Dnešní doba je typická rychlým vývojem ve všech oblastech techniky. Jinak na tom není ani odvětví počítačových sítí, kde neustále rostou požadavky na rychlost přenosu, jednodušší správu sítí a podobně. Řešení těchto věcí přináší Software-defined networking,zkráceně SDN (viz. sekce 2.1), což je architektura umožňující vytvořit flexibilnější, dynamičtější a centralizovanější síť. Jedním ze standardů SDN je OpenFlow (viz. sekce 2.1.2). Jeho verze však mají fixně definováné množiny podporovaných protokolů a akcí, které se dají aplikovat na paket a tím vzniká potřeba pořízení switche podporujícího danou verzi OpenFlow. Toto fixní chování vyžadovalo řešení, jelikož podpora více protokolů by znamenala vyměnit zařízení.

Z tohoto důvodu vznikl jazyk P4 (viz. sekce 2.2), který umožňuje definovat chování dataplane. Toto chování si definuje sám administrátor. Jazyk P4 je platformě nezávislý a tak umožňuje spustit P4 program na různých platformách. Sdružení CESNET vytváří nástroj, který umožňuje spouštět P4 programy na programovatelném hradlovém poli (Field Programable Gate Array - FPGA). Tento nástroj se používá při návrhu digitálního obvodu, který prochází i procesem verifikate. V současné době však neexistuje přesnější model pro ověření funkcionality P4 programu Z toho důvodu je hlavním úkolem této práce navrhnout a implementovat softwarový model, podle kterého se bude ověřovat vygenerovaný popis pro FPGA.

Práce je členěna na kapitoly. Kapitola 2 se zabývá úvodem do teorie. Popisuje SDN, jazyk P4 a programování v něm. Dále popisuje využívané programy a nástroje. V poslední řadě popisuje funkcionalitu referenčního cílového P4 zařízení. Kapitola 3 popisuje návrh na úpravy referenčního P4 zařízení tak, aby splňoval požadavky zadání. Kapitola 4 popisuje postup při provádění úprav z předchozí kapitoly. Kapitola 6 popisuje průběh testování a výsledky dosažené v této práci.

Kapitola 2

Úvod do teorie

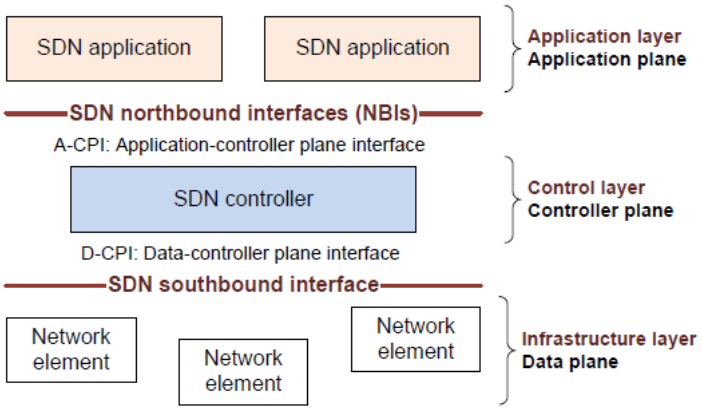
Před samotným návrhem je třeba seznámit se s technologiemi, které jsou v této práci využívány. Tyto technologije byly již zmíněny v úvodu a v této kapitole se jimi budeme zabývat podrobněji. Zmíněno je zde Software-defined networking (sekce 2.1) a s tím související technologie, kterými jsou OpenFlow (sekce 2.1.2) a ONOS (sekce 2.1.1). Následující podkapitolou je nejdůležitější část a to seznámení se s jazykem P4 (sekce 2.2), kde jsou popsány jeho výhody, existující projekty. Dále je v této sekci uveden příklad modelu P4 architektury s její implementací. V sekci 2.3 je popis použitých programů a nástrojů a důvod jejich použití při práci na bakalářské práci. v Sekci 2.4 je popsána funkcionalita některých částí cílového P4 zaříznení simple_switch, jelikož je využito jako referenční zařízení.

V této kapitole jsou využívány pojmy data plane, control plane. **Data plane**, někdy nazývaná forwarding plane, je část architektury routeru, která přeposílá pakety na další skok po cestě do cílové destinace, vybrané na základě logiky control plane. Pakety cestují skrz data plane routeru, místo toho, aby cestovaly do něj a od něj. **Control plane** je část architektury routeru, která rozhoduje, kam budou příchozí pakety odeslány. Zajišťuje také konfiguraci, řízení a výměnu informací routovacích tabulek. V SDN je tato část součástí softwaru[1]. Třetí a poslední vrstvou je management plane, která není v této práci potřebná.

2.1 Software-Defined Networking

Software-defined networking je architektura, která dělá síť více flexibilní, dynamickou a centralizovanou. Tato architektura od sebe odděluje control plane a data plane. Hlavním prvkem sítě je SDN-kontrolér, který poskytuje centralizovaný pohled na celou síť a umožňuje administrátorům konfigurovat, jak mají jednotlivé switche a routery směrovat provoz na síti, bez toho, aby musel konfigurovat několik jednotlivých switchů. Další výhodou je bezpečnost, kdy administrátor může na kontroléru sledovat provoz v síťi a nasazovat bezpečnostní politiky. Naříklad pokud kontrolér usoudí, že je provoz na síti podezřelý, může pakety přesměrovat, či zahodit.

Jak lze vidět v obrázku (2.1) vrstva infrastruktury se skládá z hardwarových prvků sítě, které poskytují svoje služby kontrolní vrstvě skrz rozhraní zvané southbound rozhraní, jehož první standard je OpenFlow (sekce 2.1.2). SDN aplikace se nacházejí v aplikační vrstvě a zasílají svoje síťové požadavky do kontrolní vrstvy skrz northbound rozhraní. SDN kontrolér přeloží požadavky aplikací a vykonává nízkoúrovňovou kontrolu nad prvky sítě, zatímco poskytuje relevantní informace SDN aplikacím. SDN konroler může organizovat konkurenční požadavky aplikací pro omezený počet síťových zařízení.



Obrázek 2.1: Základní komponenty SDN. Převzato z https://www.researchgate.net/figure/Basic-SDN-components_fig3_281979574

2.1.1 Open Network Operating System

Open Network Operting System (ONOS) je opearční systém ,vyvíjený v rámci Open Network Foundation (ONF), vytvořený, tak aby plnil požadavky poskytovatelů internetového připojení,jako jsou vysoká škálovatelnost, dostupnost a výkon, ale může sloužit i jako SDN control plane pro lokální sítě. Skládá se ze tří vrstev, kterými jsou Northbound aplikace, distribuované jádro a southbound protokol.

Northbound aplikace - Stejně jako u ostatních kontrolerů, aplikace pracují pomocí zasílání požadavků a příjmání odpovědí. Northbound rozhraní je abstraktní, tudíž není ovlivněné konkrétními protokoly a slouží pro komunikaci s jádrem.

Jádro - Jádro je distribuované za účelem dosažitelnosti všech zařízení v síti a je založeno na modulární architektuře, která umožňuje přizpůsobení celého systému. Za účelem škálování sítí ONOS umožňuje jednoduché přidání dalších zařízení, bez rušení zbytku sítě. Důležitým prvkem jádra je Intent Framework podsystém, který umožňuje aplikacím specifikovat co potřebují. Tyto požadavky jsou následně přeloženy a kompilovány do specifických instrukcí, které jsou instalovány do síťového zařízení. Další vlastností jádra je uchovávání topologie sítě, kterou uchovává v podobě síťového grafu.

Southbound rozhraní - Toto rozhraní je určeno pro komunikaci mezi jádrem a síťovými zařízeními. ONOS podporuje několik southbound protokolů, mezi které patří například OpenFlow. O zjištění, které protokoly mohou být využity pro interakci se zařízením se stará jádro.

2.1.2 OpenFlow

OpenFlow je považován za jeden z prvních SDN standardů. Základem specifikace je definice prvku, který je abstraktním strojem pro zpracování paketů, zvaný switch. Switch zpracovává pakety pomocí využití obsahu paketů a konfigurace switche. Protokol umožňuje konfiguraci switche a příjmání určitých událostí. Dalším prvkem je kontrolér, který umožňuje modifikovat konfiguraci mnoha switchů a odpovídat na události.

OpenFlow switch obsahuje dvě komponenty a to switch-agent a data plane. Switch-agent komunikuje s jedním nebo více kontroléry. Dále komunikuje s data plane switche pomocí vnitřního protokolu. Switch-agent překládá požadavky kontroléru do potřebných

nízkoúrovňových příkazů, které dále posílá data plane. Vnitřní data plane upozornění pak dále přeposílá kontroléru.

OpenFlow protokol se stará o komunikaci mezi kontrolérem a switchi. Může být rozdělen do čtyř částí:

- **Message layer** - definuje strukturu a sémantiku všech zpráv. Může vytvářet, kopírovat, porovnávat, vypisovat a upravovat zprávy.
- **Konečný automat** - definuje nízkoúrovňové chování protokolu.
- **Rozhraní** - ukazuje jak protokol interaguje s okolím.
- **Konfigurace** - téměř všechny aspekty protokolu mají konfiguraci a počáteční hodnoty.
- **Model dat** - každý switch uchovává relační model dat, který obsahuje atributy pro každou abstrakci OpenFlow.

  – rozšířit podle toho kolik stránek teorie bude po simple switchi

2.2 Jazyk P4

Název P4 vychází z "Programming Protocol-independent Packet Processors". Je to tedy protokolově nezávislý programovací jazyk, který vyjadřuje jakým způsobem jsou zpracovávány pakety v data plane síťových zařízení, jako jsou například: routery, switche, síťové karty atd. Tento jazyk byl původně navržen pro programování switchů, avšak jeho rozsah byl rozšířen na velké množství sítových zařízení, které jsou dále nazývány obecným pojmem target.

Spousta targetů implementuje i control plane, avšak P4 specifikuje pouze funkcionalitu data plane a částečně může definovat komunikaci mezi data plane a control plane, avšak ne konkrétní funkcionalitu.

P4 zařízení pracují ve dvou režimech: konfigurace a běžný chod. Při konfiguraci se do zařízení nahrává P4 program, který se interně kompiluje do reprezentace nejvhodnější pro dané zařízení. Po konfiguraci zařízení přejde do režimu běžného chodu.

P4 programovatelný target se liší od tradičního targetu ve dvou podstatných věcech.

- Funkcionalita data plane není fixní, ale je definována P4 programem. Data plane je konfigurována při inicializaci tak, aby implementovala funkcionalitu popsanou v P4 programu a nemá žádnou vestavěnou znalost existujících síťových protokolů.
- Control plane komunikuje s data plane stejně jako u tradičního targetu, avšak tabulky a ostatní objekty v data plane nejsou fixovány, jelikož jsou definovány P4 programem. P4 program generuje API, které je využito pro komunikaci mezi data plane a control plane

2.2.1 Výhody jazyka P4

Oproti tradičnímu zpracování paketů (zpravování založené na psaní mikrokódu nad vlastním hardwarem), poskytuje P4 tyto výhody:

- **Flexibilita** - P4 umožňuje vyjádřit mnoho politik přesměrování paketů jako programy.

- **Expresivita** - P4 umožňuje vyjádřit hardwarově nezávislé algoritmy zpracování paketů využívající výhradně obecné operace a tabulkové vyhledávání. Tyto programy jsou přenosné na targety implementující stejnou architekturu.
- **Mapování a správa zdrojů** - P4 programy popisují prostředky pro ukládání dat abstraktně. Překladač mapuje tyto uživatelem definované pole na dostupné hardwarové zdroje a spravuje nízkoúrovňové detaily jako je alokace a plánování.
- **Softwarové inženýrství** - P4 programy poskytují výhody jako kontrola typů, schování informací a odmítnutí softwaru v případě neúspěchu při mapování.
- **Knihovny komponent** - Knihovny komponent poskytované výrobcem mohou být užity k zabalení hardwarově specifických funkcí do přenosných vysokoúrovňových P4 konstrukcí.
- **Oddělení vývoje hardwaru a softwaru** - Výrobci targetů mohou použít abstraktní architekrury k odělení vývoje nízkoúrovňových architektonických detailů od vysokoúrovňového zpracování.
- **Debuggování** - Výrobci mohou poskytnout softwarové modely architektury k pomoci ve vývoji a debuggování P4 programů.

2.2.2 Model architektury

P4 architektura identifikuje programovatelné bloky, z nichž hlavní jsou parser, vstupní řízení toku, výstupní řízení toku a deparser, a jejich data plane rozhraní. Výrobce musí poskytovat P4 překladač a definici architektury pro jejich target.

Target interaguje s P4 programem přes sadu kontrolních registrů nebo signálů. Vstupní řízení poskytuje P4 programu informace (například vstupní port, na kterém byl přijat paket). Výstupní řízení může ovlivnit chování targetu (například změnou výstupního portu pro paket). Registry a signály jsou reprezentovány v P4 jako intrinsic metadata. P4 program však může ukládat a pracovat s uživatelem definovanými metadaty, která přísluší každému paketu.

Chování P4 programu může být plně popsáno ve smyslu tranformací, které mapují vektory bitů na vektory bitů. Ke zpracování paketu model architektury interpretuje bity, které P4 program zapisuje do intrinsic metadat. Například pro předání paketu specifickému výstupnímu portu bude muset P4 program zapsat index výstupního portu do vyhrazeného kontrolního registru. Detaily interpretace intrinsic metadat jsou však specifické pro každou architekturu.

P4 program může využívat služby implementováné externími objekty poskytované architekturou. Jejich implementace není v P4 programu specifikována, avšak jejich rozhraní ano. Rozhraní pro externí objekt popisuje každou operaci, kterou poskytuje i s jejími parametry a návratovými typy.

Obecně P4 programy nepředpokládájí přenositelnost na jiné architektury, avšak měl by být přenositelný na všechny targety implementující stejnou architekturu.

2.2.3 Příkald modelu architektury a její implementace

V následující kapitole je popsán model architektury ukázkového zařízení Very Simple Switch (VSS). Jak bylo již zmíněno model architektury by byl dodán výrobcem VSS. Nejedná se o

kompletní kód modelu architektury, ale o útžky z kódu popisující nejdůležitější části architektury. Dále bude popsána implementace daných částí architektury. O tuto implementaci se stará programátor VSS.

Nejprve s v modelu nachází deklarace konstant a struktur, mezi něž patří i PortId, které určuje číslo portu, nebo případně speciální akce prováděné s paketem.

```
// deklarace typu PortId - 4bitove cislo
typedef bit<4> PortId;
// 8 realnych portu
const PortId REAL_PORT_COUNT = 4w8;
// metadata doprovazejici vstupni paket - vstupni port
struct InControl { PortId inputPort; }
// metadata urcena pro vystupni paket - vystupni port
struct OutControl { PortId outputPort; }
// specialni hodnoty vstupnich a vystupnich portu,
   tyto hodnoty specifikuji specialni akce provadene s paketem
const PortId RECIRCULATE_IN_PORT = 0xD;
const PortId CPU_IN_PORT = 0xE;
const PortId DROP_PORT = 0xF;
const PortId CPU_OUT_PORT = 0xE;
const PortId RECIRCULATE_OUT_PORT = 0xD;
```

Následuje deklarace parseru

```
parser Parser<H>(packet_in b, out H parsedHeaders);
```

Tato deklarace popisuje rozhraní pro parser. Parser čte jeho vstup z packet_in, což je předdefinovaný P4 externí objekt, reprezentující vstupní paket. Parser dále zapisuje výstup do argumentu parsedHeaders. <H> určuje typ vstupní a výstupní hlavičky.

Deklarace popisující Match-Action pipeline

```
control Pipe<H>(inout H headers,
             in error parseError,
             in InControl inCtrl,
             out OutControl outCtrl);
```

Pipeline obdrží tři vstupy: hlavičky, parser error, a vstupní kontrolní data. Po zpracování zapíše výstupní kontrolní data a upraví hlavičky. Následně kód pokračuje do deparseru.

```
package VSS<H>(Parser<H> p,
              Pipe<H> map,
              Deparser<H> d);
```

Za účelem programování VSS bude muset uživatel instanciovat tento balíček. V případě tohoto balíčku je typ <H> množina hlaviček.

Nyní bude popsána implementace výše zmíněných částí architektury.

```
typedef bit<48> EthernetAddress;
typedef bit<32> IPv4Address;
// Standardni Ethernetova hlavicka
header Ethernet_h {
    EthernetAddress dstAddr;
    EthernetAddress srcAddr;
```

```
    bit<16> etherType;
}
// IPv4 hlavicka - bez nepovinnych moznosti
header IPv4_h {
    bit<4> version;
    bit<4> ihl;
    bit<8> diffserv;
    bit<16> totalLen;
    bit<16> identification;
    bit<3> flags;
    bit<13> fragOffset;
    bit<8> ttl;
    bit<8> protocol;
    bit<16> hdrChecksum;
    IPv4Address srcAddr;
    IPv4Address dstAddr;
}
// Strukutura naparsovanych hlavicek
struct Parsed_packet {
    Ethernet_h ethernet;
    IPv4_h ip;
}
// Uzivatelem definovane chyby, ktere se mohou objevit behem parsovani
error {
    IPv4OptionsNotSupported,
    IPv4IncorrectVersion,
    IPv4ChecksumError
}
```

Tato část kódu popisuje strukturu hlaviček, které P4 program využívá. Jedná se o standardní ethernetovou hlavičku a IPv4 hlavičku. Dále obsahuje strukturu využívanou k parsování hlaviček a v poslední řadě chyby, které se mohou objevit v průběhu parsování.

Následující kód je implementací parseru.

```
parser TopParser(packet_in b, out Parsed_packet p) {
    Checksum16() ck; // instanciace kontrolniho souctu
    state start {
        b.extract(p.ethernet);
        transition select(p.ethernet.etherType) {
            0x0800: parse_ipv4;
        }
    }
    state parse_ipv4 {
        b.extract(p.ip);
        verify(p.ip.version == 4w4, error.IPv4IncorrectVersion);
        verify(p.ip.ihl == 4w5, error.IPv4OptionsNotSupported);
        ck.clear();
        ck.update(p.ip);
        verify(ck.get() == 16w0, error.IPv4ChecksumError);
```

```
        transition accept;
    }
}
```

Parsování začíná stavem start, kde se extrahuje ethernetová hlavička a následně je vybráno, do jakého dalšího stavu parser přejde. Kód 0x0800 značí IPv4 protokol, tudíž pakety obsahující tento protokol přechází do stavu parse_ipv4. Jelikož začáteční stav neobsahuje možnost přechodu u jiných protokolů a zároveň neobsahuje defaultní pravidlo, všechny ostatní paketu budou odmítnuty. Ve stavu parse_ipv4 se extrahuje IPv4 hlavička. Následně je zkontrolována verze protokolu IP a případně je nastaven error IPv4IncorrectVersion. Následuje kontrola pole IHL značící délku hlavičky. Hodnota 5 značí, že se v této hlavičce nevyskytují nepovinné možnosti, což je pro tuto architekturu žádoucí. V opačném případě je nastaven error IPv4OptionNotSupported. Nakonec je ověřen kontrolní součet. Je-li tento součet 0, pak paket nebyl poškozen a je přijat k dalšímu zpracování do Match-Action pipeline.

```
control TopPipe(inout Parsed_packet headers,
                in error parseError, // parser error
                in InControl inCtrl, // vstupni port
                out OutControl outCtrl) {
      IPv4Address nextHop; //lokalni promenna
      // Akce nastavujici priznak zahozeni paketu
      action Drop_action() {
          outCtrl.outputPort = DROP_PORT;
      }
      // Akce nastavujici dalsi skok paketu, vystupni port a snizujici ttl paketu
      action Set_nhop(IPv4Address ipv4_dest, PortId port) {
          nextHop = ipv4_dest;
          headers.ip.ttl = headers.ip.ttl - 1;
          outCtrl.outputPort = port;
      }
      /* Tabulka urcujici dalsi skok paketu a vystupni port v zavislosti
         na cilove adrese paketu. Podle shody je vyuzita prislusna akce*/
      table ipv4_match {
          key = { headers.ip.dstAddr: lpm; } // longest-prefix match
          actions = {
               Drop_action;
               Set_nhop;
          }
          size = 1024;
          default_action = Drop_action;
      }
      // Akce nastavujici priznak zaslani paketu na CPU port
      action Send_to_cpu() {
          outCtrl.outputPort = CPU_OUT_PORT;
      }
      // Tabulka urcujici, zda bude paket poslan na CPU port
      table check_ttl {
          key = { headers.ip.ttl: exact; }
          actions = { Send_to_cpu; NoAction; }
```

```
    const default_action = NoAction; // defined in core.p4
}
// Akce nastavujici vystupni MAC adresu v ethernet hlavicce
action Set_dmac(EthernetAddress dmac) {
    headers.ethernet.dstAddr = dmac;
}
/* Tabulka ve ktere je vyhledana vystupni MAC adresa podle
   IPv4 adresy dalsiho skoku paketu */
table dmac {
    key = { nextHop: exact; }
    actions = {
        Drop_action;
        Set_dmac;
    }
    size = 1024;
    default_action = Drop_action;
}
// Akce nastavujici vstupni MAC adresu paketu v ethernet hlavicce
action Set_smac(EthernetAddress smac) {
    headers.ethernet.srcAddr = smac;
}
/* Tabulka ve ktere je vyhledana zdrojova MAC adresa podle
   cisla vystupniho portu */
table smac {
     key = { outCtrl.outputPort: exact; }
     actions = {
         Drop_action;
         Set_smac;
    }
    size = 16;
    default_action = Drop_action;
}
/* Provedeni vyhledavani v tabulkach a provedeni
   akci podle nastavenych priznaku. */
apply {
    if (parseError != error.NoError) {
        Drop_action();
        return;
    }
    ipv4_match.apply();
    if (outCtrl.outputPort == DROP_PORT) return;

    check_ttl.apply();
    if (outCtrl.outputPort == CPU_OUT_PORT) return;

    dmac.apply();
    if (outCtrl.outputPort == DROP_PORT) return;
```

```
        smac.apply();
    }
}
```

V Match-action pipeline jsou implementovány tabulky, ve kterých je vyhledáván požadovaný údaj.Tyto tabulky využívají akce, které jsou popsány nad nimi. Sekce apply spouší vyhledávání v těchto tabulkách a podle navrácených příznaků provádí příslušné akce. Následuje zpracování v deparseru.

```
control TopDeparser(inout Parsed_packet p, packet_out b) {
    Checksum16() ck;
    apply {
        b.emit(p.ethernet);
        if (p.ip.isValid()) {
            ck.clear();
            p.ip.hdrChecksum = 16w0;
            ck.update(p.ip);
            p.ip.hdrChecksum = ck.get();
        }
        b.emit(p.ip);
    }
}
```

Deparser k paketu znovu přidá hlavičky, a provede kontrolní součet.

2.2.4 BMv2

BMv2 je behaviorální model jazyka P4 napsaný v jazyce C++. Narozdíl od svého předchůdce, nazývajícího se p4c-behavioral, neobsahuje žádný automaticky generovaný kód a je úplně nezávislý na targetu. Tím umožňuje rychlejší a jednodušší změny P4 programu a jeho následné testování.

P4 program je nejprve zkompilován do JSON reprezentace. Tato reprezentace určuje, které tabulky mají být inicializovány, jak nakonfigurovat parser, atd. JSON reprezentace je vytvořena nástrojem p4c-bm. Následně je tento JSON soubor načtený do BMv2 a spuštěn. Celý projekt je dostupný ve formě zdrojových kódů a slouží jako referenční prostředí pro vývoj a běh P4 programů.

TODO: citace———————————————-

2.3 Použité programy

V této části bude popsán účel použití a funkcionalita programů, které byly v rámci bakalářské práce využívány.

2.3.1 Oracle VM VitrualBox

Jelikož instalace všech potřebných knihoven a nástrojů na vlastní počítač nebylo možná, z důvodu nedostatku místa a velké časové náročnosti, byl zvolen vývoj na serveru sdružení CESNET, avšak v průběhu bylo zjištěno, že na serveru nejsou aktualizované některé potřebné nástroje. Čekáním na aktualizaci nástrojů by došlo ke zdržení, z toho důvodu byl školitelem Ing. Benáčkem Ph.D. poskytnut virtuální počítač, na kterém byly nainstalovány

všechny potřebné knihovny a nástroje v aktuálních verzích. Tento počítač byl využíván pozbytek vývoje simulačního modelu.

Pro spouštění virtuálního počítače byl využíván Oracle VM VirtualBox vyvíjený firmou Oracle, sloužící ke spouštění virtuálního počítače. Funguje na principu tzv. plné virualizace, kdy jsou virtualizovány všechny součásti počítače. V takovémto prostředí nemůže virtualizovaný operační systém žádným způsobem poznat, že nemá přístup k fyzickému hardware. Jelikož virtualizace není smyslem této práce, nebude dále rozebírána. Virtualizovaným operačním systémem je Ubuntu 16.04.6 LTS.

2.3.2 ifconfig

IFCONFIG je nástroj sloužící ke konfiguraci síťových rozhraní v unixových systémech. Tento nástroj lze použít pro nastavení IP adresy, síťové masky popřípadě vypnutí/zapnutí síťového rozhraní.

V průběhu práce na bakalářské práci byl tento nástroj využit k vytvoření virtuálních síťových rozhraní, které byly následně využívány k testování targetu.

Příklad použití v rámci bakalářské práce:

```
ifconfig lo:0 127.0.0.2 netmask 255.0.0.0 up
```

Tetno příkaz vytvoří virtuální rozhraní založené na fyzickém rozhraní lo. Jeho IP adresa bude 127.0.0.2 a síťová maska 255.0.0.0. up slouží k zapnutí tohoto virtuálního rozhraní.

2.3.3 tcpdump

TCPDUMP je program, jenž se řadí mezi tzv. network sniffery, což jsou programy, které umožňují zachytávat, třídit a podle toho zaznamenávat komunikaci po síti, nebo v její části.TCPDUMP se řadí mezi základní network sniffery, avšak je velmi dobře použitelný a pro účely běžného zachycení komunikace je dostatečný. TCPDUMP umožňuje zachycenou komunikaci ukládat do tzv. PCAP souborů, což jsou datové soubory obsahující zachycené pakety spolu časovou značkou, jeho výskytu na síťovém rozhraní.

V průběhu práce na bakalářské práci byl využíván k zachytávání a následnému uložení komunikace na síťovém rozhraní do pcap souboru. Uložení kounikace je nutné, jelikož pomocí těchto souborů probíhalo testování výsledného targetu.

Příklad použití v rámci bakalářské práce:

```
tcpdump -i lo -w ./file.pcap
```

Argument -i slouží k výběru rozhraní, na němž bude zachytávána komunikace. Argument -w slouží k určení souboru, do kterého bude zachycená komunikace ukládána.

2.3.4 p4c-bm2-ss

p4c je modulární překladač pro programovací jazyk P4. Poskytuje standardní frontend a midend, který může být kombinován s backendem podle specifického targetu. V rámci bakalářské práce je používán backend p4c-bm2-ss sloužící k překladu pro target simple_switch napsaný s využitím BMv2 behaviorálního modelu.

Překladači je poskytnut P4 program napsaný programátorem. Tento program je následně převeden do JavaScript Object Notation (JSON), což je způsob zápisu dat využívaný pro výměnu dat na webu. Je založen na podmnožině programovacíjo jazyka JavaScript. Je založen na dvou strukturách.

- **kolekce párů název/honota** - v programovacích jazycích realizován jako objekt, záznam, struktura, hash tabulka, asociativní pole atd..
- **seřazený seznam hodot** - realizován jako pole, vektor, seznam, posloupnost atd..

****jak behaviorální model ten JSON používá****tady, nebo jinde?;
Příklad použití v rámci bakalářské práce:

```
p4c-bm2-ss -o ./file.json ./basic.p4 --emit-externs
```

Argument -o specifikuje soubor, do kterého bude JSON reprezentace P4 kódu uložena. Argument ./basic.p4 udává, jaký p4 kód bude překládán. Argument –emit-externs slouží ke korektnímu převedení p4 extenů do JSON reprezentace.

2.3.5 runtime_CLI.py

Tento program slouží jako příkazový řádek pomocí kterého lze naplnit tabulky targetu pravidly. Program se připojí k Thrift RPC (Remote Procedure Call, neboli vzdálené volání procedur) serveru, který běží v každé instanci targetu. Defaultní port, na kterém Thrift RPC server běží je 9090, port však může být specifikován. Jedna instance CLI může být připojena pouze k jednomu targetu.

Příklad spuštění:

```
./runtime_CLI.py --thrift-port 9090
```

CLI je realizováno využitím cmd modulu programovacího jazky Python. Nejdůležitějšími příkazy jsou:

- **table_set_default <jméno tabulky> <jméno akce> <parametry akce>** - nastaví defaultní akci pro danou tabulku. Příkladem defaultní akce může být zahození paketu. Příklad využití v targetu simple_switch:

```
table_set_default MyIngress.ipv4_lpm MyIngress.drop
```

- **table_add <jméno tabulky> <jméno akce> <IP adresa s prefixem> => <parametry akce>** [**priorita**] - přidá konkrétní pravidlo do dané tabulky. IP adresou s prefixem, jsou definované adresy, kterých se pravidlo týká. Akcí může být například přeposlání paketu a v tom případě jsou parametry akce IP adresa a port, na který bude paket přesměrován. IP adresy se píší v hexadecimálním vyjádření. Příklad využití v targetu simple_switch:

```
table_add MyIngress.ipv4_lpm MyIngress.ipv4_forward
                          0x7f000001/8 => 0x7f000001 0
```

- **table_delete <jméno tabulky> <pořadí přidání do tabulky>** - odstraní pravidlo ze zadané tabulky. Pravidlo se vybírá podle, pořadí přidání do tabulky s tím, že indexace začíná od nuly.. Příklad využití:

```
table_delete MyIngress.ipv4_lpm 0
```

Je možno používat i několik dalších příkazů. Tyto příkazy jsou však nejsou potřeba při práci na bakalářské práci.

2.4 simple_switch

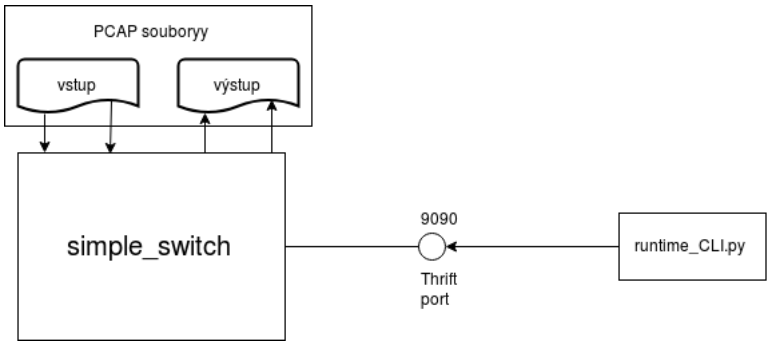
Target simple_switch je v bakalářské práci využit jako výchozí target, který je postupně upravován dle zadání. Z tohoto důvodu zde bude popsána jeho funkcionalita.

Nejprve je simple_switch spuštěn například takto:

```
./simple_switch --use-files 5 --log-console -i 0@lo ./basic.json
```

Argument –use-files s hodnotou 5 určuje, že na vstup a výstup budou použity soubory se jmény "názevRozhrani_in.pcap"a "názevRozhrani_out.pcap". v tomto konkrétním případě tedy lo_in.pcap a lo_out.pcap. Hodnota 5 tohoto argumentu udává čas v sekundách, po který simple_switch čeká, než začne ze souborů číst pakety. Tento čas je nutný kvůli naplnení tabulek přes runtime_CLI. Argument –log-console povoluje logování na stdout. Argument -i specifikuje jedno rozhraní targetu. Nakonec je specifikován JSON soubor, se kterým bude target pracovat.

Následně je spuštěn runtime_CLI.py (viz. 2.3.5) a jsou naplněny tabulky targetu. Po uplynutí času specifikovaného argumentem –use-files začne target číst všechny pakety ze souborů a řadit je podle časových značek paketu. Dále jsou pakety posílány na určený port, provedou se akce specifikované v tabulkách a nakonec je paket odeslán na příslušný port, nebo zahozen. Po zpracování paketů však simple_switch zůstane zapnutý.



Obrázek 2.2: simple_switch, napojení na runtime_CLI a vstupní/výstupní soubory

Simple_switch obsahuje oproti požadavkům několik funkcionalit navíc. Jedná se o možnost klonovat paket, znovu zpracovat již zpracovaný paket a to buď v podobě před tímto zpracováním, nebo po tomto zpracování. Dále umožňuje vytvářet pakety podle multicastových skupin.

***popis toho jak běží???

2.4.1 Čtení paketů

Čtení paketů ze vstupního souboru není funkcionalitou konkrétního cílového P4 zařízení, ale je součástí tříd FilesDevMgrImp a PcapFilesReader, které jsou společné pro většinu P4 zařízení. Třída FilesDevMgrImp spouští vlákno pro čtení a řídí čtení. Funkce scan() třídy PcapFilesReader následně pakety ze vstupních PCAP souborů serializuje a seřadí podle časových značek paketů. Seřazené pakety jsou poté v čase daném časovou značkou zaslány do vstupního bufferu třídy SimpleSwitch.

2.4.2 Vstupní zpracování

Pro zpracování je důležitá třída PHV, která uchovává všechny informace získané z paketu pomocí parsování. Skládá se z vektoru hlaviček paketu s tím že každá hlavička obsahuje vektor polí hlavičky. Paket je vyjmut ze vstupního bufferu a následně mu je přiřazena instance PHV, aby mohl být parsován. Po naparsování je paket poslán do ingress pipeline, kde je zpracován ve správných match-action tabulkách, které určí, co se s paketem bude dít dále. Match-action tabulky nejsou součástí implementace simple_switche, ale jsou dle potřeby definovány P4 programem, který je překládán a poskytován cílovému zařízení. Dalším krokem je provedení příslušných operací, které byly určeny zpracováním v match-action tabulkách. Těmito operacemi jsou: klonování paketu na vstupu, učení, opětovné zaslání nezpracovaného paketu, multicast a zahození paketu. Dále je paket přiřazen do fronty pro výstupní zpracování v konkrétním vlákně daném výstupním portem, nebo zahozen. Vlákno pro vstupní zpracování je ukončeno spolu s ukončením činnosti celého simple_switche.

2.4.3 Výstupní zpracování

Ve vlákně výstupního zpracovánní se provádí operace určené match-action tabulkami, které nebylo možné provést ve vstupním vlákně a to jsou: klonování paketu na výstupu a opětovné zaslání zpracovaného paketu. Před opětovným zasláním zpracovaného paketu se provádí deparsování, které změněný parsovaný paket složí. Tento paket je následně zařazen do fronty pro zapsání do výstupního souboru. Vlákno výstupního zpracování končí spolu s ukončením činnosti celého simple_switche.

2.4.4 Metadata

Simple_switch, stejně jako všechny targety, pracuje s metadaty. Metadata, se kterými může target pracovat jsou definována v P4 kódu, který je targetem využíván. V kódu targetu je možno specifikovat povinnost některých polí metadat. Pokud tato povinná pole nejsou specifikována v P4 kódu, target nebude možné spustit. Metadata jsou součástí každého paketu a umožňují změny a práci s paketem.

Simple_switch má možnost pracovat se třemi hlavičkami metadat. Jedná se o Intrinsic_metadata, obsahující informace potřebné k multicastu a časové značky určující, kdy se paket objevil na vstupu a kdy byl paket přijat pro výstupní zpracování. Dalšími metadaty jsou Queueing_metadata obsahující informace a časové značky týkající se zařazení paketu do fronty. Posledními metadat jsou Standard_metadata. Z těchto metadat je 5 polí povinných a to:

- **ingress_port** - číslo portu na kterém byl paket přijat
- **packet_length** - délka paketu v bytech
- **instance_type** - podle této položky simple_switch určí co se má s paketem stát. Například recirkulace, klonování atd..
- **egress_spec** - do této položky může být v kódu targetu zapsáno, na jaký port bude paket odeslán
- **egress_port** - určeno pro přístup ve výstupním zpracování pro zjištění výstupního portu. Tato položka je pouze pro čtení

2.4.5 Externí objekty

Jedním z požadavků je pohodlnější práce s externímy objekty. Z toho důvodu zde bude popsáno co externí objekty jsou, jak se definují, jak se programuje jejich funkcionalita a jak se volají.

Externí objekty jsou objekty se kterými může P4 program interagovat a které jsou poskytovány architekturou. Z toho vyplívá, že definice externí třídy musí být uvedena v P4 kódu architektury.

Následující kód je implementací třídy ExternCounter, na které budou vysvětleny náležitosti, kterí je třeba dodržet při implementaci externí třídy.

```
class ExternCounter : public ExternType {
public:

  BM_EXTERN_ATTRIBUTES {
    BM_EXTERN_ATTRIBUTE_ADD(initCount);
  }

  void init() override{
    initCount_ = initCount.get<size_t>();
    reset();
  }

  void reset() {
    count = initCount_;
  }

  void setInitCount(const Data &d){
    initCount = d;
  }

  void increment() {
      count++;
  }

  void incrementBy(const Data &d) {
    count += d.get<size_t>();
  }

  size_t get() const {
    return count;
  }

 private:
  Data initCount{0};

  size_t initCount_{0};
  size_t count{0};
};
```

```
BM_REGISTER_EXTERN(ExternCounter);
BM_REGISTER_EXTERN_METHOD(ExternCounter, increment);
BM_REGISTER_EXTERN_METHOD(ExternCounter, init);
BM_REGISTER_EXTERN_METHOD(ExternCounter, incrementBy, const Data &);
BM_REGISTER_EXTERN_METHOD(ExternCounter, reset);
BM_REGISTER_EXTERN_METHOD(ExternCounter, setInitCount, const Data &);
BM_REGISTER_EXTERN_METHOD(ExternCounter, get);

int import_extern_example(){ return 0;}
```

Jak je možno vidět na ukázce kódu třídy, třída musí dědit od třídy ExternType. Dále musí být atributy, ke kterým je potřeba přistupovat přímo a ne přes metody, registrovány pomocí makra BM_EXTERN_ATTRIBUTES, ve kterém se konkrétnímu atributu registrují pomocí makra BM_EXTERN_ATRIBUTTE_ADD(jméno atributu). Přímý přístup k atributům je možný pouze v kódu targetu a ne v P4 kódu, ve kterém nelze přistupovat přímo k atributům z externích tříd, ale je třeba implementace metod get a set. Implementace vlastního konstruktoru je možná, avšak konstruktor nesmí mít argumenty, jelikož při registraci externu musí být dodržena podmínka std::is_default_constructible, která značí, že typ musí být vytvořen bez inicializačních hodnot, nebo arguentů. V sekci private, lze vidět proměnné initCount a initCount_. Rozdíl mezi nimi je ten, že k initCount lze přistupovat a initCount_ je vnitřní proměnná třídy, se kterou pracují pouze její metody. Za implementací třídy, je nutno externí třídu a všechny její metody registrovat.

Funkce import_extern_example() musí být volána v kódu targetu, jako externí, aby bylo vynuceno propojení k souboru, ve kterém se externí třída nachází.

Popis implementovaných metod:

- **void init()** - tato metoda slouží k inicializaci čítače. Nastaví vnitřní inicializační proměnnou, na hodnotu, registrované proměnné, která byla získána uživatelem a zavolá metodu resetující čítač. Tato metoda musí být implementována v každém vlastním externu.
- **void reset()** - metoda nastavující proměnnou počítadla na inicializační hodnotu.
- **void setInitCount()** - metoda nastavující inicializační proměnnou. Tato metoda je využívána pouze v P4 kódu.
- **void increment()** - metoda navyšující hodnotu čítače o jedna.
- **void incrementBy(const Data &d)** - metoda navyšující hodnotu čítače o hodnotu předanou v argumentu.
- **void get()** - metoda navracející aktuální hodnotu čítače.

Následující kód je definicí externu v modelu P4 architektury. Metody této třídy lze využívat v P4 kódu

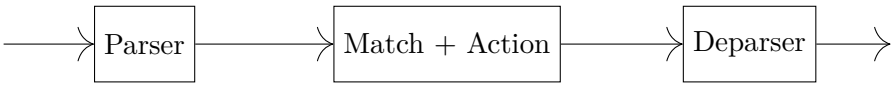
```
extern ExternCounter {
    ExternCounter();
    void setInitCount(bit<32> data);
    void init();
    void increment();
    void increment_by(bit<32> data);
```

```
    bit <64> get();
}
```

Kapitola 3

Návrh implementace

Cílem tohoto návrhu je vytvořit cílové P4 zařízení, které bude splňovat zadané požadavky a které bude sloužit pro účely verifikace jako model. Těmito požadavky jsou: podpora metadat vytvořených sdružním CESNET, okamžité ukončení programu po zpracování paketů, vytvoření flexibilního mechanismu pro přidávání funkcionality nových externích modulů.



Obrázek 3.1: struktura targetu

Na obrázku 3.1 je znázorněno zpracování paketů P4 zařízením. Paket postupně projde třemi základními bloky a to parserem, match-action a nakonec deparserem. Parser je popsán pomocí stavového automatu. Úkolem parseru je extrahovat hlavičky paketu. Následně probíhá zpracování paketu v Match-Action bloku, kde se extrahované hlavičky porovnávají vůči datům uloženým v match-action tabulkách. Nakonec je paket opět složen v deparseru a zapsán do výstupního PCAP souboru, nebo zahozen.

Návrh implementace vychází z již vytvořeného targetu simple_switch, jelikož jeho struktura je velmi podobná požadovanému targetu. Je tedy časově výhodnější tento target upravit, než začínat s tvorbou targetu nového. Úpravy, které budou provedeny jsou následující:

- **Odstranění nepotředného kódu** - Simple_switch implementuje funkce, které nebudou v cílovém zařízení využívány. Tyto funkce jsou: klonování packetů, opětovné zpracování packetu, multicast, prioritní fronty. Odebráním kódu dojde ke zjednodušení celého modelu a zkrácení doby simulace.
- **Úprava metadat** - Výchozí target obsahuje několik hlaviček s metadaty. Standard metadata, Intrinsic metadata, Queueing metadata. Avšak P4 zařízení vyvíjené v rámci projektu liberouter sdružením CESNET podporuje rozšířenou množinu metadat, která nejsou implementována v rámci zařízení simple_switch.
- **Okamžité ukončení programu** - Simple_switch nepodporuje ukončení programu po zpracování všech paketů. Pro účely plánované verifikace je vhodné, aby se implementované zařízení po zpracování všech vstupních paketů z PCAP souboru korektně ukončilo.

- **Načítání externích modulů** - Simple_switch umožňuje vytváření externích objektů, funkcí atd. Avšak výsledný target by měl umožňovat jejich dynamické načítání, obecné volání metod a jednoduchou implementaci nových modulů bez nutnosti většího zásahu do kódu targetu.

3.1 Konkrétní úpravy

V následující sekci je popsán návrh na konkrétní úpravy simple_switche tak, aby odpovídal požadavkům bakalářské práce. V ideálním případě je vhodné vytvořit nový target, a k němu jeho vlastní model. ***jak se tvoří novej model******

3.1.1 Metadata

Sdružení CESNET má pro svoje potřeby vytvořena vlastní Intrinsic_metadata, která kombinují položky Standard a Intrinsic metadat simple_switche. Oproti metadatům simple_switche jsou rozšířena o položky hash, user16 a user4, které jsou využívány za vygenerovaným P4 jádrem pro distribuci na jádra procesoru. Položky metadat jsou následující:

- **ingress_timestamp** - Časový údaj o příchodu paketu na vstup.
- **ingress_port** - Číslo vstupního portu.
- **egress_port** - Číslo výstupního portu.
- **packet_len** - Délka paketu.
- **hash**
- **user16 a user4** - uživatelem definovaná data.

S těmito metadaty musí být vytvořen nový model architektury podobný souboru v1model.p4. Následně musí být v kódu targetu upraveny části kódu, které pracují s původními metadaty.

3.2 Ukončování programu

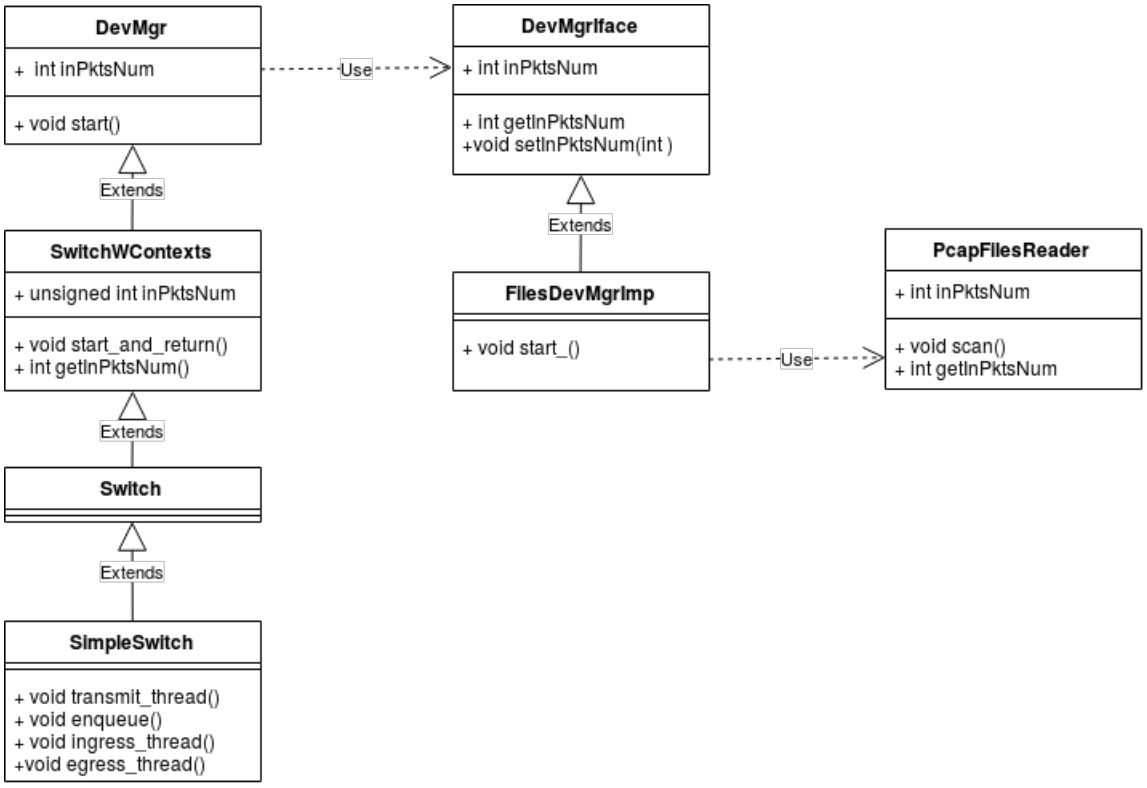
Současná implementace simple_switche nepodporuje okamžité ukončení programu. Místo toho je na konci programu hlavního vlákna tento kód:

```
while (true) std::this_thread::sleep_for(std::chrono::seconds(100));
```

Tento kód je nekonečná smyčka, ve které je hlavní vlákno vždy uspáno na sto sekund. Toto řešení zbytečně zpomaluje simulaci a především je nutno program ukončit pomocí signálu SIGINT (Ctrl+C). Pro účely simulace je tedy vhodnější automatické ukončení programu po zpracování všech paketů ze vstupu.

Toho bude dosaženo pomocí semaforů. Do třídy SimpleSwitch bude vložen semafor, který bude v místech nekonečné smyčky uzamčen. Následně bude nutno zjistit celkový počet paketů, které mají být zpracovány. Z toho důvodu bude do třídy SimpleSwitch přidána proměnná pro tento účel. V poslední řadě je nutno udržovat si počet zpracovaných paketů, což znamená paketů odeslaných, nebo případně zahozených. Z tohoto důvodu bude do třídy SimpleSwitch přidána i tato proměnná. V každé části kódu, která obstarává odeslání nebo zahození paketu, bude nutné po jeho zpracování porovnat proměnou obsahující celkový

počet paketů ke zpracování a počet aktuálně zpracovaných paketů. Budou-li tato čísla stejná, znamená to, že všechny pakety byly zpracovány a tudíž je možné program ukončit. Odemkne se tedy dříve uzamčený semafor, zavolá se destruktor třídy SimpleSwitch, který se stará o ukončení všech vytvořených vláken a následně bude program ukončen.



Obrázek 3.2: UML diagram - ukončování programu

Celkový počet paketů bude získán z funkce scan(), který je součástí třídy PcapFilesReader. Počet zahozených a odeslaných paketů bude počítán v metodách třídy SimpleSwitch, které se o tyto události starají. V místech kde dochází k počítání odeslaných a zahozených paketů se bude při každém průchodu paketu dotazovat na podmínku zmíněnou v předchozím odstavci, při jejím splnění bude program ukončen.

3.3 Externí moduly

Simple switch neposkytuje dostatečnou abstrakci pro pohodlné vytváření externích modulů a jejich následné použití. Z toho důvodu bude implementována knihovna, ke které budou připojeny všechny souboy s vlastnímy externy. Všechny tyto soubory budou umístěny v jedné složce. Bude implementována funkce pro jednodušší spouštění exterů.

rozšířitjak??

Kapitola 4

Implementace

Při implementaci byl měněn především kód targetu a knihovny, které využívá. Model architektury nemusel být zásadně měněn, jelikož architektura zůstala stejná. Změny byly provedeny pouze s metadaty a byla přidána registrace externí třídy pro testování jejího spouštění. Dále bylo třeba lehce zasáhnout do kódu překladače, jelikož jeho implementace neumožňovala registraci vlastních externů.

4.1 Odstranění nepotřebného kódu

Byly odstraněny části kódu určené pro klonování paketu, opětovné zaslání paketu, opětovné zpracování paketu a multicast.

Kompletní odstranění kódu pro prioritní fronty a zrcadnlení relace by se neobešlo bez hlubšího zásahu do kódu a opětovného generování souborů. Z toho důvodu jsou metody využíváné tyto účely ponechány, avšak jejich kód je vymazán a místo toho metody vypisují chybové hlášení, že jejich funkcionalita není implementována.

4.2 Ukončování programu

Za účelem ukončování programu po zpracování všech paketů byla ve třídě PcapFilesReader Nacházející se v souboru pcap_file.cpp deklarována proměnná inPktsNum, která uchovává celkový počet paketů ke zpracování ze všech vstupních souborů. Počet paketů je získáván z funkce PcapFilesReader::scan(). Tato funkce prochází všechny soubory na vstupu a řadí pakety podle jejich časové značky. Dále je potřeba proměnnou inPktsNum Předat třídě SimpleSwitch. Toho je dosaženo nejdříve předáním třídě DevMgrIface v metodě FilesDevMgrImp::start_(), která volá metodu PcapFilesReader::scan(). Metoda PcapFilesReader::scan() je volána metodou PcapFilesReader::start(). Metoda FilesDevMgrImp::start_() je následně volána v metodě DevMgr::start() i zde tedy probíhá předaní proměnné. V poslední řadě je v metodě SwitchWContexts::start_and_return() volána metoda DevMgr::start(). Zde probíhá předání do třídy SwitchWContexts. Z této třídy už není potřeba předávat dále, jelikož instance třídy SimpleSwitch má k této proměnné přístup.

Dále je ve třídě SimpleSwitch deklarována proměnná monitorMutex, což je semafor, který je uzamknut na začátku kódu destruktoru třídy SimpleSwitch. Destruktor je volán ihned po provedení metody start_and_return(), která spouští běh targetu.

Jako další je ve třídě SimpleSwitch deklarována proměnná outPktsNum, která v sobě udržuje aktuální počet zpracovaných paketů. Zpracováním paketů je myšleno odeslání, nebo

zahození, což se řeší v těchto metodách třídy SimpleSwitch: transmit_thread, enqueue, ingress_thread a egress_thread. Za inkrementací počtu zpracovaných paketů je testována podmínínka na dotaz, zda-li se počet zpracovaných paketů rovná počtu paketů na vstupu. Je-li tato podmínka kladná, je odemknut semafor a destruktor korektně ukončí všechna vlákna targetu a poté ukončí celý program

4.3 Metadata

Při implementaci metadat bylo zjištěno, že implementace nového modelu není možná bez hlubšího zásahu do překladače. Z toho důvodu byl upraven původní v1model, ve kterém byly přepsány původní metadata na metadata sdužení CESNET.

Dále byly v kódu targetu přepsány všechny části kódu, který původní metadata využíval. Jelikož si jsou metadata velmi podobná, úpravy nebyly nijak složité.

4.4 Externí třídy

Při práci na implementaci jednoduššího volání externích tříd bylo zjištěno, že překladač p4c-bm2-ss, nezapisuje vytvořené externí instance do výstupního JSON souboru. Bylo tedy potřeba zasáhnout do kódu překladače a tuto chybu opravit.V souboru kódu překladače extern.cpp v metodě ExternConverter::convertExternInstance(ConversionContext* ctxt, const IR::Declaration* c, const IR::ExternBlock* eb, const bool& emitExterns) bylo implementováno pouze ošetření pokusu o zápis neznámé externí instance, samotný zápis do JSON souboru však implementován nebyl. Další chybou byl nekorektní zápis názvu proměnné externí instance.

Zápis externí instance je implementován následovně.

```
auto primitive = new Util::JsonObject();
primitive->emplace("id",nextId("extern_instances"));
primitive->emplace("type",eb->type->name);
primitive->emplace("name",c->getName().toString());
primitive->emplace_non_null("source_info", c->sourceInfoJsonObj());
ctxt->json->externs->append(primitive);
```

V JSON souboru zápis instance vypadá následovně.

```
{
  "id" : 0,
  "type" : "ExternCounter",
  "name" : "ek",
  "source_info" : {
    "filename" : "./basic.p4",
    "line" : 90,
    "column" : 20,
    "source_fragment" : "ek"
  }
```

- id - jednoznačný identifikátor externí instance
- type - typ externí instance

- name - název proměnné externí instance
- source_info - informace o kódu instanciace

Další chybou byl nesprávný název proměnné instance při volání metod v P4 kódu. Jednalo se o chybu v metodě Util::IJson* ExternConverter::convertExternObject(ConversionContext* ctxt, const P4::ExternMethod* em, const IR::MethodCallExpression* mc, const IR::StatOrDecl*, const bool& emitExterns). Tato chyba byla způsobena řádkem etr->emplace("value", em->object->getName());. Tento řádek vkládal do JSON souboru hodnotu typu IR::ID, bylo tedy potřeba vkládanou hodnotu převést na string. Proto byla provedena následující úprava: etr->emplace("value", em->object->getName().toString());

Pro volání externích metod v P4 kódu nebylo potřeba žádných úprav. Pro volání metody stačí instanciovat instanci externí třídy a poté volat její metody. Například takto:

```
ExternCounter() ek;
ek.increment();
```

Volání externích metod v kódu targetu však tak jednoduché není a bylo potřeba imentovat metodu, zjednodušující instanciaci externích instancí.

```
template<typename T>
T ExternRunner::externInit(string externClass, map<string, Data> atributy){

    auto externInstance = ExternFactoryMap::get_instance()->get_extern_instance(
       string(extern_class));
    externInstance->_register_attributes();

    for (std::map<string, Data>::iterator it = atributy.begin(); it != atributy.end();
        externInstance->_set_attribute<Data>(it->first, Data(it->second));
    }
    externInstance->init();

    T specificInstance = dynamic_cast <T> (externInstance.get());
    return specificInstance;
}
```

Tato metoda nejprve vytvoří externí instanci. Následně registruje její atributy. Poté tyto registrované atributy nastaví na požadované hodnoty. Následně je volána inicializační funkce a nakonec je externí instance převedena na instanci konkrétní externí třídy. S touto inicializovanou instancí pak lze v kódu targetu pracovat a volat její metody.

Kapitola 5

Použití programu

Target

Spouštění targetu je totožné se spouštěním simple switche viz. sekce 2.4 Pro korektní funkci je nutno ve vlastním p4 programu pracovat s v1model.p4, který se nachází ve složce behavioral-model/targets/simple_switch. Ve stejné složce se nachází příklad .P4 programu basic.p4. Příklad vytváření externů viz. sekce 2.4.5 Externy musí mít konstruktor bez argumentů. K atributům nelze přistupovat v .p4 kódu přímo. Při vytvoření nového souboru s externí třídou je nutno vytvořit funkci totožnou s funkcí import_extern_example() v souboru extern1.cpp a tu následně volat v konstruktoru třídy ExternRunner. V opačném případě překladač o externí třídě neví. Nové soubory je nutno přidat do Makefile.am ve složce simple_switch do libsimpleswitch_la_SOURCES

Překlad P4 souboru

Překlad P4 souborů pomocí p4c-bm2-ss viz. sekce 2.3.4 Pro správnou funkcnost vlastních externů je nutno přepsat soubor p4c/backends/bmv2/common/extern.cpp souborem, který se nachází v přiložených soborech v totožné cestě a následně tento překladač opětovně přeložit a nainstalovat.

Další možná rozšíření

Vytvořit nový target, místo úpravy již vytvořeného simpleSwitche

úprava překladače p4c-bm2-ss tak, aby bylo možno přistupovat k atributům externích instancí a aby bylo možno vytvářet konstruktory s argumenty. (bude-li tato úprava proveditelná)

Kapitola 6

Testování a dosažené výsledky

Kapitola 7

Závěr

Literatura

[1] McGahan, B.: *Control plane a Data plane*. Červen 2011, [Online].
URL https://blog.ine.com/2011/06/15/control-plane-vs-data-plane