Eötvös Loránd Tudományegyetem
Informatikai Kar
Információs Rendszerek Tanszék

NS-3 szimulátor kiegészítése P4 programozható switch támogatással

Laki Sándor
Egyetemi adjunktus

Rosanics Márton
Programtervező Informatikus BSc

Budapest, 2018

Tartalomjegyzék

1. Bevezető

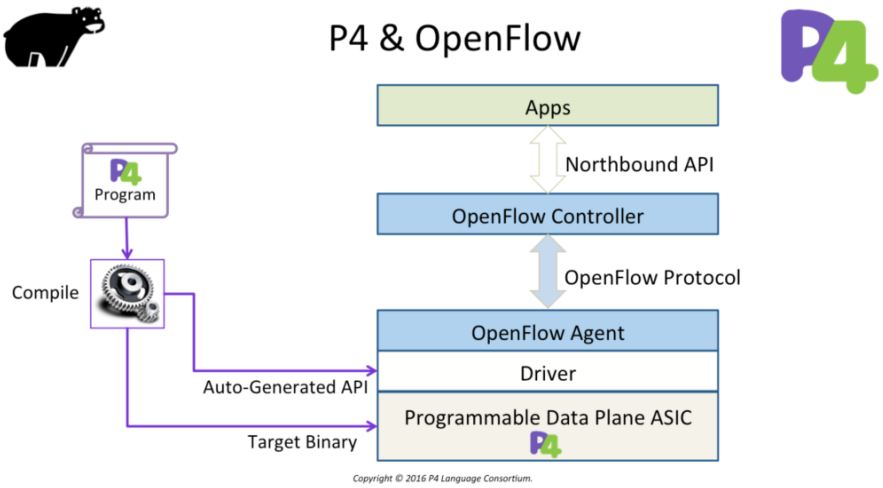
1.1. Motiváció

A hálózati kutatások terén napjaink egyik felkapott témája az absztrakt adatsíkok programozása. Ezen területen kiemelkedik a P4 nyelv és annak továbbítási modellje, mely rugalmas adatsíkok kialakítását teszi lehetővé SDN hálózatokban. Az elérhető néhány prototípuson kívül azonban ezen megoldás nagy skálájú vizsgálata eddig nem volt lehetséges, hiszen nem létezett olyan csomagszintű hálózat szimulátor, ami támogatta volna ezt az új paradigmát.

Dolgozatom célja a P4 képes hálózatok vizsgálatának lehetővé tétele csomagszintű szimulációk segítségével. Ehhez a népszerű NS-3 hálózat szimulátor jó alapot nyújt, mely P4 képes eszközökkel való kibővítése nagyobb szakmai közönség számára is hasznos lehet.

1.2. P4-ről röviden

A P4 egy nyílt forráskódú nyelv ami hálózati eszközök programozására alkalmatos: switch-csek, routerek, hálózati kártyák működését lehet meghatározni vele. A hálózatok többségében ez alulról-felfele (bottoms-up) módszerrel történik, rögzített működésű switch-csekkel amik csak egyféle képpen továbbítanak csomagot. Ezzel szemben a P4-programozható hálózatok felülről lefele (top-down) megközelítésével bármilyen funkcionalitás telepíthető amit a felhasználó szeretne. Ez a módszer hasonlatos a népszerű OpenFlow switch működéséhez, azonban (az OpenFlowval ellentétben) a P4 az adatkapcsolati rétegre fókuszál.



További információ erről a P4 konzorcium honlapján [1] található.

1.3. NS-3-ról röviden

Az NS-3 egy nyílt forráskódú, diszkrét eseményű hálózat szimulátor. C++ nyelven íródott, de Python szkriptelési lehetőségekkel is bír. Moduláris szerkezetének köszönhetően könnyen bővíthető, számos külső adatelemző és vizualizációs eszköz használható vele. A szimulátor célja egy olyan környezet biztosítása, ahol a valóságban nehezen kivitelezhető hálózatok tanulmányozása és tesztelése folyhat.

További információk az NS-3 hálózat szimulátor honlapján [2] találhatóak.

1.4. A probléma megközelítése

Az implementáció megkezdése előtt több megközelítés közül választhattam. Az egyik lehetséges megoldás az lett volna, ha a P4 nyelv egy már létező interpreterjét, a BMv2-t integráltam volna az NS-3 környezetbe. Végül azonban amellett döntöttem, hogy az ELTE-P4 compiler segítségével oldom meg a feladatot. A compiler a P4 kódot C nyelvre fordítja, melyet az általam létrehozott NS-3 modul képes használni.

Az új NS-3 modul lényegében egy ns3::NetDevice-ból származtatott osztályt valósít meg (`P4SwitchNetDevice`), és ez az osztály fogja magában foglalni az interfészt a P4 switch-hez.

2. Felhasználói dokumentáció

Az a programozó aki az NS-3-as szimulációjához P4 nyelven írt switch-cset akar használni (továbbiakban: felhasználó) a modulom használatához több szoftverrel is kell, hogy rendelkezzen. Habár feltételezhető, hogy a felhasználó a modulom használatakor már ismeri az NS-3 szimulátort, a teljesség kedvéért erről is biztosítok rövid magyarázatot az alábbi szekciókban.

2.1. Használati előfeltételek

2.1.1. Az NS-3 rendszer

Az NS-3 rendszer egy meglehetősen komplex rendszer, számos függőséggel és egyéb komponenssel rendelkezik. Az NS-3 elsősorban GNU/Linux platformra lett fejlesztve, dolgozatom is ilyen környezetben (Ubuntu 16.04-en) készült.

Az NS-3 hivatalos oldala tartalmaz egy részletes listát azokról a programokról és csomagokról amik az installáció *előtt* szükségesek [3]. Minimális előfeltétel egy gcc compiler és egy Python interpreter, de az oldal 20 további szoftvercsomag telepítését is részletezni. Emellett még létezik egy telepítési utasítás magához az NS-3-hoz is [4], mely végigvezet a szoftverrel kapcsolatos kezdeti lépéseken.

Szakdolgozatom az NS-3-nak a 3.26-os verziójával készült. Bár valószínű, hogy későbbi verziókkal is ugyan úgy fog működni a program, a továbbiakban leírtak helyessége csak ilyen körülmények között garantált.

2.1.2. A P4 fordítóprogram

A felhasználó által megírt P4 switch-cset le is kell tudni fordítani C nyelvre. Ehhez a t4p4s elnevezésű ELTE P4 compilerre van szükség, mely egy Github tárolóról [5] érhető el, telepítési utasításokkal együtt. Megjegyzendő, hogy bár a t4p4s teljes körű működéséhez DPDK [6] szükséges, azonban erre a felhasználónak tulajdonképpen nem lesz nagy szüksége, hiszen gyakorlatilag a DPDK szerepét az NS-3 veszi majd át.

A P4 compilerről bővebben itt [7] lehet még olvasni.

A szakdolgozatom készítése idején a t4p4s fordítóprogram a P4 nyelvnek csak a $P4_{14}$-es változatát támogatta, $P4_{16}$-ot nem. Ezen túl pedig mint ahogy az NS-3-nál, itt sem garantált, hogy a t4p4s későbbi verzióinál minden ugyan úgy fog működni. Dolgozatom az ELTE P4 compiler 2017-es verziójával készült.

2.1.3. A P4 modul hozzáadása a szimulátorhoz

Ahhoz, hogy a felhasználó az általam írt interfészt használni is tudja, kézileg kell új modult létrehoznia és elhelyeznie a fájlokat a megfelelő helyre. Az alábbi lépések tekinthetőek a P4 modulom telepítési útmutatójának:

1. A terminálban az NS-3 `src` mappájába navigálva az alábbi utasítással készíthető új modul (a `P4` nevet adjuk a modulnak a helyes működés érdekében):

```
$ ./create-module.py P4
```

2. Dolgozatom `p4` mappájában mellékelt fájlokat és almappákat az új modul mappájába kell másolni (az alapértelmezettek helyébe). Az almappák mellett található `wsript` fájlról sem szabad elfeledkezni; az NS-3 ez alapján fogja tudni mely fájlok tartoznak a modulhoz.

3. A szimulátort ezután konfigurálni és összeállítani (nem az `src`-ből, hanem a főmappából) a következő utasításokkal lehet:

```
$ ./waf configure
$ ./waf build
```

Új modul létrehozásáról bővebben az NS-3 dokumentációban is lehet olvasni [8].

2.2. Használat

Feltételezhető az a kiinduló állapot, hogy a felhasználónak van egy P4 nyelven leprogramozott switch-cse, illetve egy NS-3 szimulációja (C++-ban írt állománya) amelyben használni akarja azt. Ekkor a következő teendők adódnak:

2.2.1. Előkészületek

A felhasználó által megírt P4 kódból C állományokat kell generálnia, például a t4p4s compiler mappájából indított

```
$ ./compile.sh examples/l2_example_switch.p4
```

utasítással.

További információt a fordítás mikéntjeiről és a compiler használatáról a t4p4s GitHub tárolóján [5] lehet találni.

A fordítás után nyolc C-ben íródott állományt kellett a felhasználónak kapnia:
`actions.c`

`action.h`
`controlplane.c`
`data_plane_data.h`
`dataplane.c`
`parser.c`
`parser.h`
`tables.c`

Ezeken a fájlokon csupán egy apró módosítást kell elvégezni használat előtt. Az `actions.c, controlplane.c, dataplane.c, parser.c, tables.c` valamint az `actions.h` fájloknál a

```
#include "dpdk_lib.h"
```

sort le kell cserélni arra, hogy:

```
#include "p4-interface.h"
```

Ezt követően mindegyik fájlt az új modul `model` mappájába kell helyezni. Az elérési útvonal az NS-3 mappájából tehát a következőképpen fog kinézni: `/src/P4/model`

2.2.2. A switch elhelyezése a szimulációban

A NS-3-as szimulációs állományt a felhasználónak callback függvényekkel maga ki kell egészítenie. A függvényeket C (nem C++) nyelven kell megírni, hiszen a P4 switch is C kódra van fordítva; az interfész csak így lesz képes használni a függvényt. Első lépésként include-olni kell az új P4 modult:

```
extern "C"
{
  #include "ns3/p4-module.h"
}
```

A callback függvényeknél továbbá a felhasználónak valószínűleg szüksége lehet kapcsoló-tábla kezelő függvényekre, melyeket használat esetén `extern`-ként kell deklaráln. Ezeket a `p4_interface.h` fájlban meg lehet találni, kigyűjtöttem ide azokat a műveleteket, melyek a leggyakrabban előfordulhatnak:

```
extern int exact_add (lookup_table_t* t, uint8_t* key, uint8_t* ↩
    value);
extern int lpm_add (lookup_table_t* t, uint8_t* key, uint8_t ↩
    prefix_length, uint8_t* value);
extern int ternary_add (lookup_table_t* t, uint8_t* key, uint8_t* ↩
    mask, uint8_t* value);
extern uint8_t* exact_lookup (lookup_table_t* t, uint8_t* key);
extern uint8_t* lpm_lookup (lookup_table_t* t, uint8_t* key);
extern uint8_t* ternary_lookup (lookup_table_t* t, uint8_t* key);
extern int exact_remove (lookup_table_t* t, uint8_t* key);
extern int lpm_remove (lookup_table_t* t, uint8_t* key, uint8_t ↩
    prefix_length );
extern int ternary_remove (lookup_table_t* t, uint8_t* key);
extern void free_entries (table_entry_t* t);
```

Az elnevezések önmagukért beszélnek, azonban további információ ezekről a függvényekről a Fejlesztői dokumentációban (3) még található, illetve használatukról a P4 interfész kódján belül, kommentek formájában is lehet olvasni.

Ezek után jöhet a két callback függvény, melyeket nevezzünk most `init_tables`-nek és `msg_digest`-nek, és az alábbi az alábbi szignatúrával rendelkezzenek:

```
int init_tables(lookup_table_t** t);
int msg_digest(lookup_table_t** t, char* name, int receiver, ↩
    struct type_field_list* digest_field_list);
```

Az `init_tables` függvény feladata, hogy inicializálja a switch kapcsolótábláit. A másik függvény azonban sokoldalúbb; a `msg_digest` úgynevezett digest függvényként funkcionál, mely egy karakter sorozatot (P4 switch által küldött üzenet) vár, és ezt kell a felhasználónak itt lekezelnie. (A karaktersorozat a felhasználó által megírt P4 switch-től függ.) A felhasználó ebben a függvényben tudja például

meghatározni, hogy mi történjen ha a switch egy új címmel találkozik (legyen szó akár L2, akár L3 forgalomirányításról); itt tudja megadni, hogy miként tanulja meg az új címet a switch.

A callback függvények megírása után már csak annyi teendője maradt a felhasználónak, hogy magában a szimulációban példányosítsa és használja a P4SwitchNetDevice-ot. Ehhez a P4 modul helper osztálya nyújt segítséget, mely segítségével lényegében két sorral beilleszthető a P4 switch:

```
P4SwitchHelper switch;
switch.Install(switchNode, switchDevices, init_tables, msg_digest);
```

Az `Install` metódus négy paramétert vár, az utolsó kettő a korábban említett callback függvények, míg az első kettő egy `Ptr<Node>` illetve egy `NetDeviceContainer` típusú változó. Az előbbi egy általános `Node` típusú objektumra mutató pointer (az a node amire a switch lesz installálva) míg az utóbbi egy, a switch portjait tartalmazó konténer.

A P4SwitchHelper osztálynak van még egy *setter* metódusa is (`AllowPacketDrop`), mellyel azt lehet megadni, hogy a P4-es switch kezel-e csomageldobást. Ha ez az érték igaz, akkor a modul számítani fog arra, hogy a generált fájlok beállítják a `packet_descriptor dropped` mezőjét. Alapértelmezetten ez az érték hamis.

Mindezek után pedig a szimulációt leíró állományt az NS-3 által elérhető helyre kell helyezni, majd az NS-3 szimuláció a szokványos módon futtatható. (A fordításhoz és futtatáshoz megint csak a hivatalos NS-3 dokumentáció nyújt segítséget és néhány mintát [9]).

2.2.3. Tudnivalók a P4SwitchNetDevice használatához

Ebben az alszekcióban szeretném részletezni a modulomnak azokat a funkcióit illetve függvényeit, melyek ismeretére a felhasználónak szüksége lesz az `init_tables` és a `msg_digest` callback függvények írása során.

Mindenek előtt hangsúlyozni szeretném, hogy a felhasználónak ismernie kell milyen actionöket (switch utasításokat) programozott le a P4 switch-csébe, illetve milyen kapcsoló-táblákat határozott ott meg. Elkerülhetetlen ugyanis, hogy a felhasználó saját maga írja meg a táblafeltöltéssel kapcsolatos függvényeket. A modul önmagában nem tudja őket lekezelni, de nem is az a cél: a P4 switch rugalmassága éppen abban rejlik, hogy nincsenek előre megszabva milyen switch

actionök léteznek. A P4 switch működése "utólag", a szimuláción belülről is módosítható.

Az `init_tables`-ről elsősorban a paraméterének típusát érdemes ismerni. A `lookup_table_t` egy struktúra, mely a következőképpen van meghatározva:

```
typedef struct lookup_table_s {
    char* name;
    unsigned id;
    uint8_t type;
    uint8_t key_size;
    uint8_t val_size;
    int min_size;
    int max_size;
    void* default_val;
    table_entry_t* table;
} lookup_table_t;
```

Ez a struktúra írja le a kapcsoló-táblát. Részletesebben erről a 3.2.5. szekcióban írok, de amit innen a használathoz érdemes észben tartani, az a név és az azonosító (`name` és `id`), valamint a `default_val` pointer. Ez utóbbi fog a arra az actionre mutatni, melyet a switch alapértelmezetten végrehajt ha nincs találata a lookup táblában.

Az alábbi három sor egy példa a beállítás mikéntjére:

```
struct smac_action def_smac_action;
def_smac_action.action_id = 0; //action_mac_learn
memcpy(t[0]->default_val, (uint8_t*)&def_smac_action, ↩
    sizeof(struct smac_action));
```

Ebben a kódrészletben az 1. táblának (`t[0]`-nak) az alapértelmezett actionjét állítottam be. A táblák generálásakor automatikusan le lesz foglalva a megfelelő méretű hely a memóriában (és a végén fel is lesz szabadítva), így az alapértelmezett érték eltárolásához csak egy `memcpy` szükséges. A tárolandó action típusa (jelen esetben az `smac_action` rekord) a P4 kód által van meghatározva: ezt a generált C fájlok közül az `actions.h` tartalmazza. Ugyanitt találhatóak az actionök típusai is; a példámnál ezek így néztek ki az `actions.h` fájlban:

```
enum actions {
  action_mac_learn,
  action__nop,
  action_forward,
  action_bcast,
};
```

```
struct smac_action {
    int action_id;
    union {
    };
};
```

Az alapértelmezett érték mellett még igen valószínű, hogy a felhasználó a táblákat további értékekkel szeretné feltölteni. Ehhez például használható az `exact_add` függvény, mely használatát egy példán szeretném először szemléltetni:

```
uint8_t key[6] = {255,255,255,255,255,255};
struct action_forward_params bcast_param;
bcast_param.port[0] = (uint8_t) BROADCAST_PORT;
bcast_param.port[1] = 0;
struct dmac_action dmac_action_val;
dmac_action_val.action_id = 3; //action_bcast
dmac_action_val.forward_params = bcast_param;
exact_add(t[1],key,(uint8_t*)&dmac_action_val);
```

Ebben a kódrészletben a 2. táblát (`t[1]`-t) egészítem ki úgy, hogy a FF:FF:FF:FF:FF:FF MAC címre a switch broadcast utasítást hajtson végre. A switch actionjeit leíró struktúra ezúttal két egymásba ágyazott `struct`-ból áll, melyet ugyancsak az `action.h`-beli kód szerint állítottam össze. A korábbi példához képest ez a kódrészlet kiegészül egy kulcs (`key`) meghatározásával. A key egy `uint8_t` tömb kell, hogy legyen. Hossza a P4 kódban került a felhasználó által meghatározásra, azonban ez az érték a generált fájlok közül a `tables.c` fájlban is megtalálható.

Az alapértelmezett érték beállításával ellentétben itt `memcpy` helyett az interfészben meghatározott függvények használatára lesz szükség. Az `exact_add` a kulcsnál pontos egyezést vizsgál, adott (`exact_lookup` típusú) táblát, míg `lpm_add` leghosszabb prefix egyezés szerint tölti fel az adott (`lpm_lookup` típusú) táblát. Mindkét függvényhez szükséges bemeneti paraméter egy pointer arra a táblára ahova be szeretnénk illeszteni az új értéket, és egy-egy pointer a kulcsra, valamint az actionre. (A táblában tárolt `key` és `value` számára memóriaterület automatikusan le lesz foglalva és fel lesz szabadítva, így azzal nem kell törődnie a felhasználónak.) Az `lpm_add` esetén még szükség van egy `uint8_t` számértékre is, mely a beillesztés mélységét, azaz a prefix hosszát adja meg bitekben. Jelenleg a mélységek felosztása bájtonként történik, tehát az elfogadható számértékek egy 4 bájt hosszú IP cím esetén: 8, 16, 24, 32. Eltérő számértékek esetén a függvény lefelé kerekít, azonban a számnak így is 0-nál nagyobbnak és 32-nél nem nagyobbnak kell lennie.

A `msg_digest`-re is vonatkoznak az `init_tables`-ről leírt tudnivalók, de mellettük még magyarázatra szorul az `msg_digest char* name` és `struct type_field_list* digest_field_list` függvényparaméterek is. A `name` változóban fogja a digest megkapni a lekezelendő üzenetet. Ez alapján fogja tudni a felhasználó megszabni milyen tevékenységet hajtson végre a switch. (Ide is érvényes az, hogy az üzenetet a P4 kódban kellett megszabnia a felhasználónak; itt ugyan azokat a karaktersorozatokat fogja visszakapni.) Az üzenethez társulhatnak bizonyos adatok is, ezeket pedig a `struct type_field_list* digest_field_list` fogja tárolni. A `struct type_field_list` a következő képpen néz ki a `P4SwitchNetDevice`-on belül:

```
struct type_field_list {
    uint8_t fields_quantity;
    uint8_t** field_offsets;
    uint8_t* field_widths;
};
```

A `fields_quantity` megadja hány mezőben vannak az adatok tárolva (azaz hogy mennyi adat van), a `field_widths` megadja a mezők hosszát (például, a `field_widths[0]` megadja az 1. mező hozzát) és a `field_offsets` tartalmazza a mezőket; ezek a csatolt adatok.

További segítséget nyújthat még a dolgozatomhoz mellékelt kódban található `lookup-tables-test.cc` állomány (melyről bővebben a 3.4.4. alszekcióban írok), ami számos példát kínál a kapcsoló-tábla műveletek használatára, illetve a táblák kezelésére.

2.3. Jellegzetes hibák és kezelésük

Mivel szakdolgozatomnál alapvetően egy szimuláció programozásáról lehet beszélni *felhasználói esetek* címszó alatt, sokféle hibaeset előfordulhat. A P4 modul csak egy része a nagy egésznek, az NS-3-ban megírt szimuláción és a P4 kódból generált switch-csen is sok múlik. Éppen ezért lehetetlen lenne ebben a szekcióban minden hibalehetőséget felsorolni, néhányat viszont az esetleges esetekből feltüntetnék. A hibák lekövetéséről és a debuggolásról a 3.4. szekcióban írok majd bővebben.

2.3.1. Fordítási hibák

- Statikus könyvtárak: Ha a felhasználó a szimuláció fordításakor a statikus könyvtáraknál hibát kap, előfordulhat, hogy a WAF-ot konfigurálnia kell előtte. A `./waf configure -enable-static` utasítás megoldhatja a problémát.

- Különböző szimulációk: Használat során előállhat az az állapot, hogy a felhasználónak több szimulációja van megírva, különböző switch-csekkel. Ekkor figyelni kell arra, hogy a kívánt szimuláció fordításakor a WAF további szimulációs fájlokat is megpróbálhat lefordítani; például a `scratch` mappában találhatókat. Ilyenkor ha azok a további szimulációk másmilyen P4 switch-cset használnak, fordítási hiba léphet fel, hiszen a generált C fájlok bár azonos nevűek, más switch-cset írnak le.
  Ez gyakorlatilag azt jelenti, hogy fordítás során csak egy féle P4 switch-hez lehetséges szimulációt fordítani. Bővebben erről még a 3.3.4. alszekcióban írok.

- Egyéb, P4 kódból eredő hibák: A P4 switch-től (és a t4p4s fordítóprogramtól) sok minden függhet, bizonyára lehetséges olyan switch kódot generálni, melyet a jelenlegi modul nem tud megfelelően lekezelni. (Ez akár futási hiba is lehet.) Például, ha a t4p4s compiler egy újabb verziójában a generált kódban már máshogy van elnevezve egy régebben használt függvény, az hibát eredményezhet az interfész fordításakor. A P4 switch-csel való kompatibilitásról bővebben a fejlesztési lehetőségek 3.3.5. szekciójában írok.

2.3.2. Futási hibák

- Port számok: A t4p4s fordító (jelenlegi verziója) a switch actionjein belül a port számot két bájton tárolja. Az első bájton érdemes tárolni a port számát (0-tól 255-ig), míg a második bájtot érdemes kinullázni. Ez azt jelenti, hogy például egy `param` elnevezésű kapcsoló-tábla utasítást leíró változó `port` paraméterét (nagy valószínűséggel) a következő képpen kell az 50-es portra beállítani:

```
param.port[0] = 50;
param.port[1] = 0;
```

Erre a felhasználónak a callback függvények írásánál, `exact_add` és `lpm_add` függvények használatakor kell leginkább figyelni.

- Broadcast port: A t4p4s fordító (jelenlegi verziója) a generált C kódban a switchben a 100-as portot automatikusan broadcast portként kezeli. Ennek megfelelően található a P4 interfészben egy `#define BROADCAST_PORT = 100` makró, ami azonban nincs összeköttetésben a generált fájlban található (beégetett) 100-as értékkel. A felhasználónak ügyelnie kell erre a callback függvények írásánál és a port számok kiosztásánál.

- Egyező kulcsok kapcsoló-tábla bővítésnél: Az `exact_add` és `lpm_add` függvények során előfordulhat, hogy a felhasználó többször ugyan ahhoz a kulcshoz akar értéket párosítani. Ilyen esetekben a régi érték felülírásra kerül, így ha a felhasználó nem figyel, nem kívánatos működést érhet el a switch-csében. (A felülírásról azonban az alapértelmezett kimeneten értesít a program.)

- Kulcsok hossza kapcsoló-tábla kezelésnél: A felhasználónak ügyelnie kell arra, hogy az `_add` és `_lookup` függvényeknél megfelelő kulcsméretet használjon. A modul (pontosabban a P4 interfész) a `tables.c` állományból olvassa ki a kulcsméretet,vagyis a P4 kódban meghatározott, táblákhoz hozzárendelt kulcs hosszt vár.

3. Fejlesztői dokumentáció

Mielőtt nekiálltam volna bármit is implementálni, fontos első lépés volt jobban megismerni az adott technológiákat, hiszen dolgozatom írása előtt se NS-3-al, se P4-el nem találkoztam még. Míg a P4 nyelv használatához felületes ismeretek elegek voltak (hiszen feladatom nem P4 switch-csek elkészítéséből áll, csupán azok NS-3 környezetbe való integrálásából), addig a hálózat szimulátort kicsit jobban meg kellett ismernem.

Az NS-3 tanulmányozásához nagy segítség volt a hivatalos honlapról is elérhető oktatóanyag [10]. Megismertem az NS-3 által nyújtott szimulációk alapelemeit, és megtanultam hogyan lehet egyszerű virtuális hálózatokat kialakítani, futtatni, és nyomon követni.

Jelentős időt fordítottam a P4 switch-csek megismerésére is, pontosabban az ELTE-P4 fordítóprogram által generált C kód analizálására. Ahhoz, hogy az NS-3 modulom használni tudja a P4 switch-cseket, pontosan meg kellett értenem mi-hogy zajlik le a switch-csen belül, melyek azok a feladatok amiket a switch "magától megold", és melyek azok amiket le kell implementálnom az interface készítése során.

3.1. Megvalósítási terv

Az NS-3 szimulátorhoz létezik egy (gyakorta használatos) switch támogatás, az OpenFlow switch. Habár bizonyos területeken az OpenFlow switch eltér a P4 switch-től, elég sok volt a hasonlóság ahhoz, hogy az NS-3-as OpenFlow modul kiváló kiinduló pontot jelentsen számomra. Ennek megfelelően az első lépés az volt, hogy az OpenFlowSwitchNetDevice osztályt mintául véve készítsem el a saját P4SwitchNetDevice osztályom.

A modul beüzemelését ezek után három nagy részre tudtam osztani:

1. Egy minta P4 switch hiba nélküli "bekötése". A cél az volt, hogy a programot hiba nélkül tudjam fordítani. Ehhez implementálnom kellett a P4SwitchNetDevice modul alapvető metódusait, valamint szintaktikusan helyessé kellett tennem a kódot. A P4 switch által meghatározott függvényeket deklarálnom kellett, a használt típusokat meg kellett adnom.

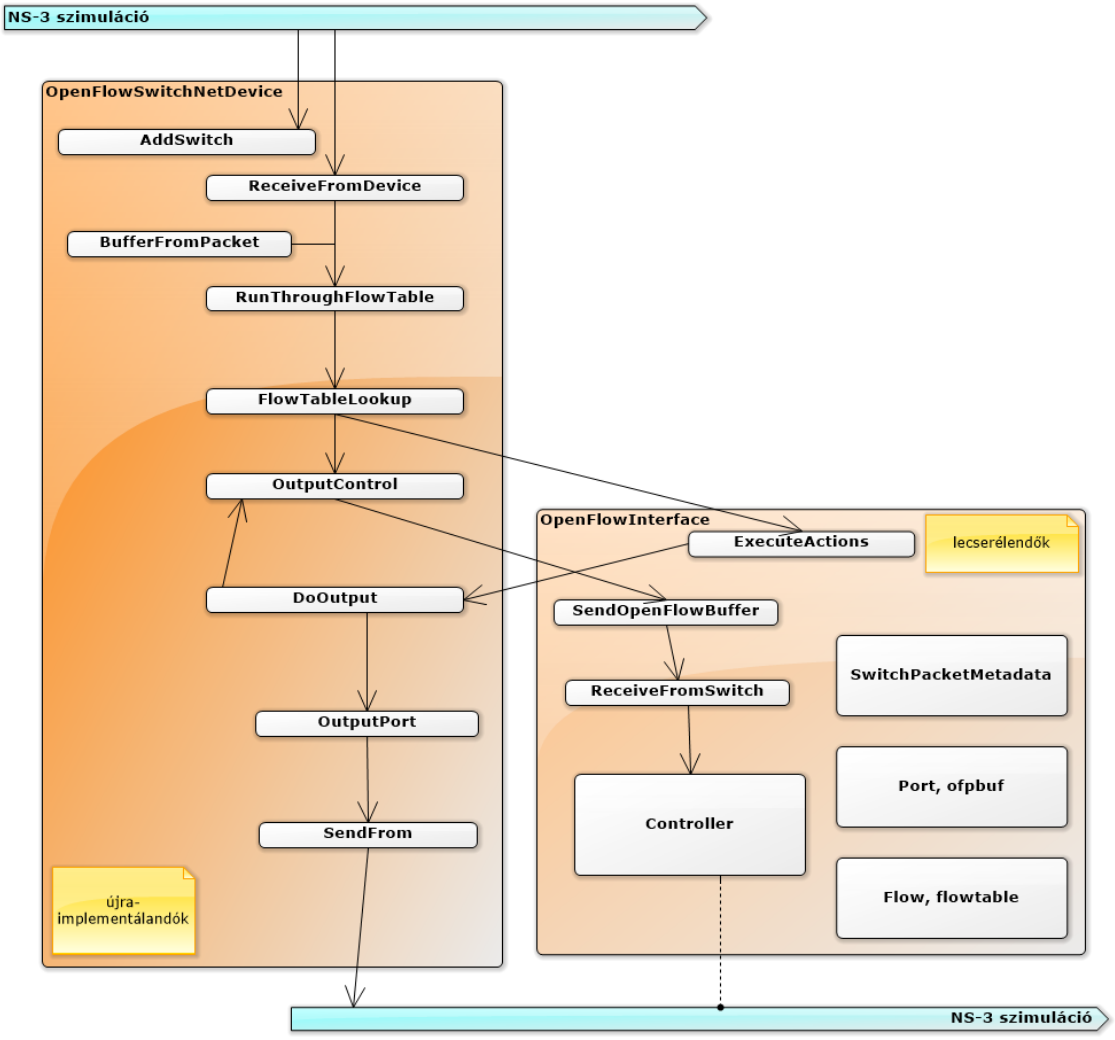
2. A kód szemantikusan helyessé tétele. Itt a cél az volt, hogy a csomagküldés sikeresen és helyesen történjen meg az adott példára; összevetve az OpenFlow switch-hez tartozó minta szimulációval, ugyan azt az eredményt kellett hogy kapjam.

3. A funkcionalitások kibővítése. Ne csak a korábban használt minta (Layer 2-es) switch működjön hibátlanul, hanem több mindenre is képes lehessen a P4 switch támogatás (például Layer 3 switch használta, több router/switch egymás utáni összekapcsolása, stb).

Az alábbi három alszekcióban a kezdeti megoldási tervet (tehát a fenti három rész közül az elsőt) részletezem bővebben, illetve ismertetem, hogy pontosan hogyan is nézett ki a feladat az implementáció megkezdése előtt.

3.1.1. OpenFlow switch

Az OpenFlow modul két jelentős osztályt tartalmaz, ezek működését megvizsgáltam. Az alábbi ábrát afféle leegyszerűsített folyamatábraként készítettem annak érdekében, hogy kiemeljem a szimuláció futásakor számomra fontos részeket. Ez sokat segített a modul megértésében és a P4SwitchNetDevice tervezésében is.



1. ábra

Az 1. ábra szemlélteti azt is, hogy a hálózati csomagok "milyen pályát futnak be", azaz mely metódusokon megy keresztül a csomag mielőtt továbbításra kerül. Míg az OpenFlowSwitchNetDevice osztály metódusait a saját modulomban a megfelelő módosításokkal (többnyire) fel tudtam használni, addig az OpenFlowInterface osztályt teljesen le kellett cserélni. Az alábbiakban egy rövid áttekintést szeretnék írni néhányukról, illetve arról, hogy miként terveztem megfeleltetni őket a P4SwitchNetDevice-omban.

`ReceiveFromDevice`: Az első "állomása" a csomagnak. Itt kerül lekezelésre az, hogy a csomagot milyen szándékkal küldték és ki a címzettje. A csomagtípus többféle lehet (például broadcast vagy multicast), de amit itt fontos kiszűrni, hogy a csomag címzettje (vagyis a destination MAC-cím) nem maga a switch-e. Itt történik továbbá annak a beazonosítása is, hogy a switch mely portján érkezett a csomag. Értelemszerűen minderre nem csak az OpenFlow switch-nél, hanem a saját switch támogatásomnál is szükség van.

`BufferFromPacket`: Szintén egy olyan metódus, amelyre szükségem van a saját modulomban is. Implementációjának miképtjére a későbbiekben (3.2.9. alszekció) még kitérek, de a szükségességéről itt ejtenék pár szót. Mikor egy NS-3 szimuláció során a csomag arra a pontra jut, hogy megérkezik a *ReceiveFromDevice* metódushoz, az NS-3 már automatikusan elvégzett néhány módosítást a csomagon. Legfontosabb ezek közül az, hogy leválasztotta az ethernet fejlécet. Mivel erre az OpenFlow switch-nek (és az én esetemben a P4 switch-nek is) szüksége van, ezért ezt kézileg kell visszahelyezni a csomag elejére. Emellett ráadásul az OpenFlow switch egy saját adattípust is használ, az *ofpbuf*-ot. Ezt különböző L2, L3 és L4 adattal (meta-adattal) tölti fel a csomag alapján, erre azonban nekem nem volt szükségem, szóval ezen egyszerűsíteni tudtam.

`RunThroughFlowTable`: Kezdődik a csomag feldolgozása. Az OF switch bizonyos feltételek mellett eldobja itt a csomagot, illetve kivárja a megadott késleltetési időt. Ezek az én esetemben a P4 switch-be vannak kódolva, így ezzel nem kellett foglalkoznom.

`FlowTableLookup/ExecuteActions`: A kapcsoló-táblákból megkeresi a switch a megfelelő utasítást a csomagnak. Ha ilyet nem talál, akkor egy sor függvényhíváson keresztül végül a Controllerhez fog eljutni a csomag, ahol aztán eldől, hogy milyen actiont kell végrehajtani. A saját modulomban én ezt a viselkedést (a lookup-okat) az interface állományomban definiálom.

`Controller`: Ez valójában nem egy metódus, hanem egy osztály (melyből további osztályok vannak származtatva) az OpenFlow switch-csen belül. A P4SwitchNetDevice-on belül ezt s struktúrát a `digest` callback függvény fogja helyettesíteni.

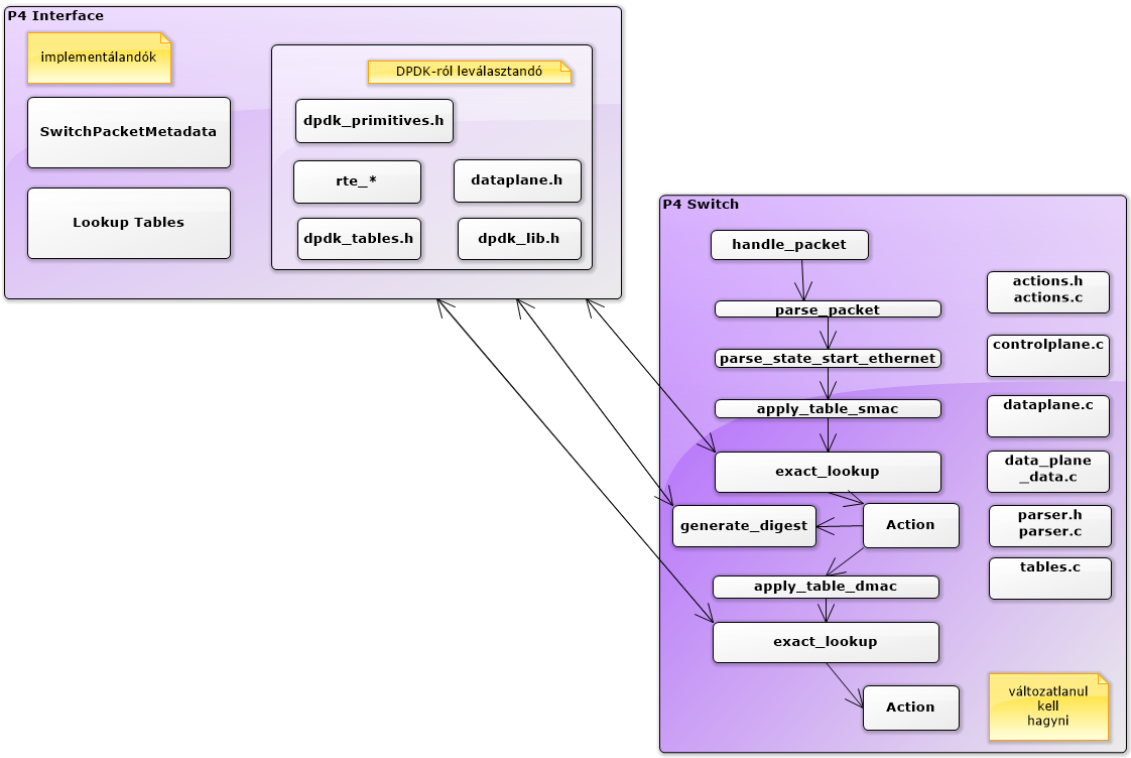
`SendFrom`: Innentől már ismét egyezés van a P4SwitchNetDevice és az OpenFlowSwitchNetDevice között. A kimeneti porton (melyet korábban a lookupnál meghatároztunk) továbbítjuk a csomagot a következő állomást reprezentáló Node-nak.

3.1.2. L2 minta switch

A következő lépés a P4 switch vizsgálata volt. Ez az OpenFlow switch vizsgálatánál egy fokkal bonyolultabbnak bizonyult, ugyanis automatikusan íródott C fájlokat kellett átnéznem (dokumentáció és kommentek nélkül), illetve kellett a működésüket megállapítanom. Kezdetnek a t4p4s fordítóprogram mellé nyújtott Layer 2-es switch mintával foglalkoztam.

A nyolc generált fájlon túl viszont még számos más állományt is meg kellett néznem a t4p4s fordítóprogramon belül. Alapértelmezetten ugyanis DPDK-s backendre támaszkodva lehetséges a generált switch-cset használni, így a típusok és függvényhívások igen gyakran "kimutatnak" az emlegetett nyolc állományon kívülre. Ezen típus- és függvénydefiníciók egy része megtalálható volt a t4p4s fordítóprogram fájljai között, ám egy részük DPDK-s állományok által volt definiálva. A DPDK-s könyvtárrendszer elemzésébe túlságosan nem merültem bele, hiszen tudtam, hogy azt a részt nem fogom használni.

Ehhez megintcsak készítettem egy kvázi-folyamatábrát (2. ábra), azonban itt a függvényhívások sorrendje meglehetősen lineáris. A programrész bonyolultságát inkább csak a hívások sokasága adja, illetve azok nehezen átlátható sorrendje.



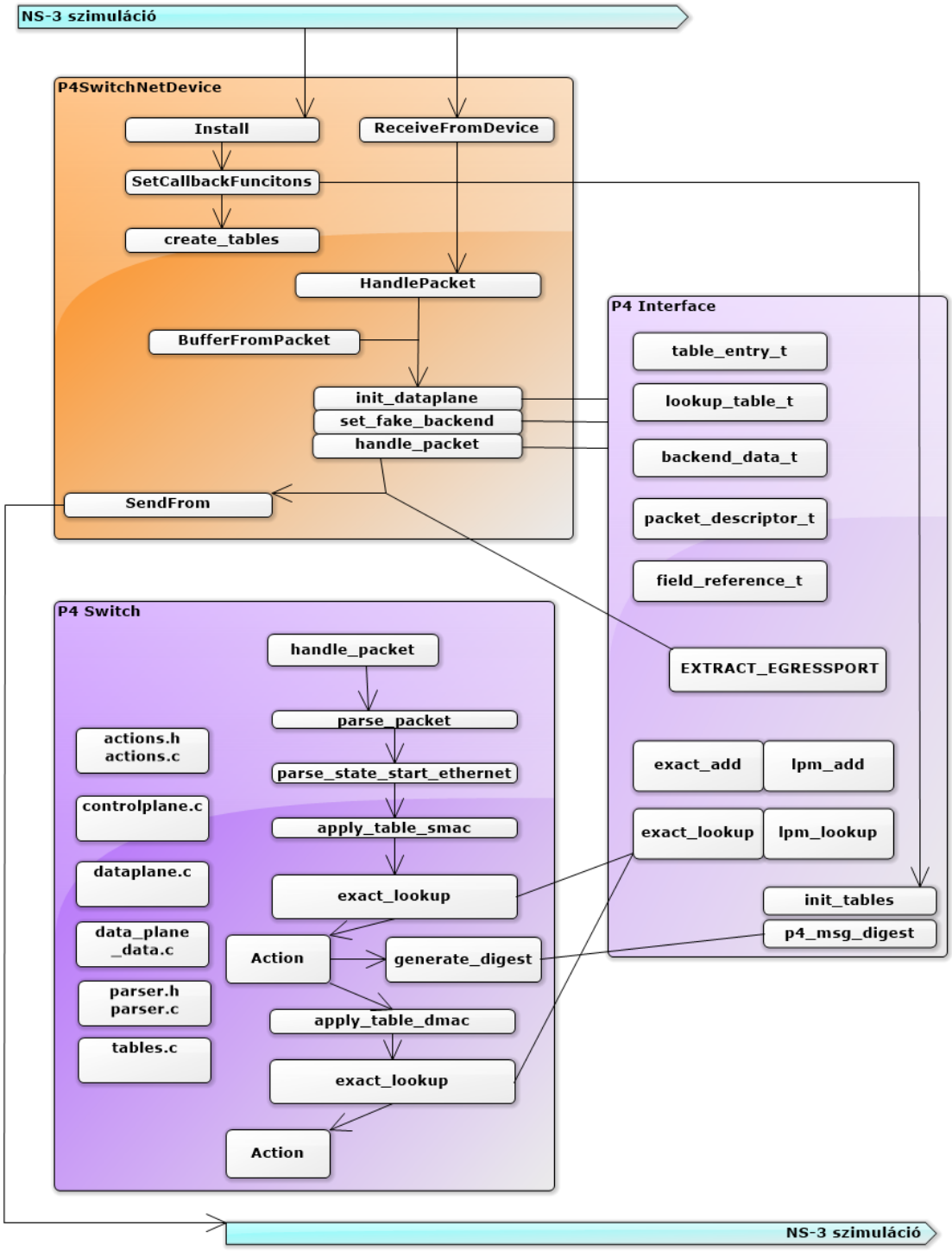
2. ábra

A fenti ábra "P4 Switch" része a generált fájlok által alkotott rendszert reprezentálja (a nyolc fájl az ábra jobb oldalán van felsorolva). A programnak ennél a részénél az volt az elsődleges célom, hogy minél kevesebb módosítást hajtsak végre az adott fájlokon: a modul használásának egyszerűsítése érdekében igyekeztem arra törekedni, hogy a felhasználónak ne kelljen a generált fájlokon még külön, kézileg is módosításokat végrehajtani. A képen látható folyamatábra nem teljes, igyekeztem csak a lényeget kiragadni az átláthatóság érdekében. Jelen példánál a switch *handle_packet* utáni részénél kezdődtek a számomra igazán érdekes dolgok: a switch két (MAC-címeket tartalmazó) tábláján való lookup-függvények, illetve a táblafeltöltést definiáló digest függvények lefutása.

A 2. ábra "P4 Interface" részén címszavakban feltüntettem néhány állományt (illetve adatstruktúrát is), melyek a generált fájlokon kívülre estek. Ezeket saját magam terveztem újra-implementálni, habár emellett meglehetős részük még újra-felhasználható is volt. Még ha nem is egy-az-egyben a kódomba átmásolható kódról volt itt szó, működésük alapos vizsgálatával rengeteg ötletet tudtam belőlük meríteni.

3.1.3. Célterv

Az alábbi ábrán pedig a végeredményt szemléltetem: a P4SwitchNetDevice-ot és a P4 switch-cset egy interfésszel terveztem összekapcsolni. Ez az ábra sem teljes, az átláthatóság kedvéért csak a fontosabb elemeket jelenítettem meg.



3. ábra

A három fő komponensből a P4 switch rész az előzőekhez képest változatlan, hiszen a függőségeket NS-oldalról kellett megoldani. A P4SwitchNetDevice az OpenFlow változathoz képest egyszerűsödött, működését bővebben a 3.2. szekcióban fogom taglalni. A 3. ábrán található utolsó komponens pedig az interfész: NS-3-as névteren kívüli, C nyelven íródott állomány, mely összeköttetést biztosít a másik két komponens között. Az ábrán felsoroltam a nevezetesebb egyéni adatstruktúrákat amiket használ, továbbá azokat a függvényeket amelyek kulcsfontosságúak a működéshez.

A 3. ábrát röviden úgy lehetne összefoglalni, hogy a hálózati adatcsomag továbbítását a P4SwitchNetDevice három fő lépésben fogja megtenni:

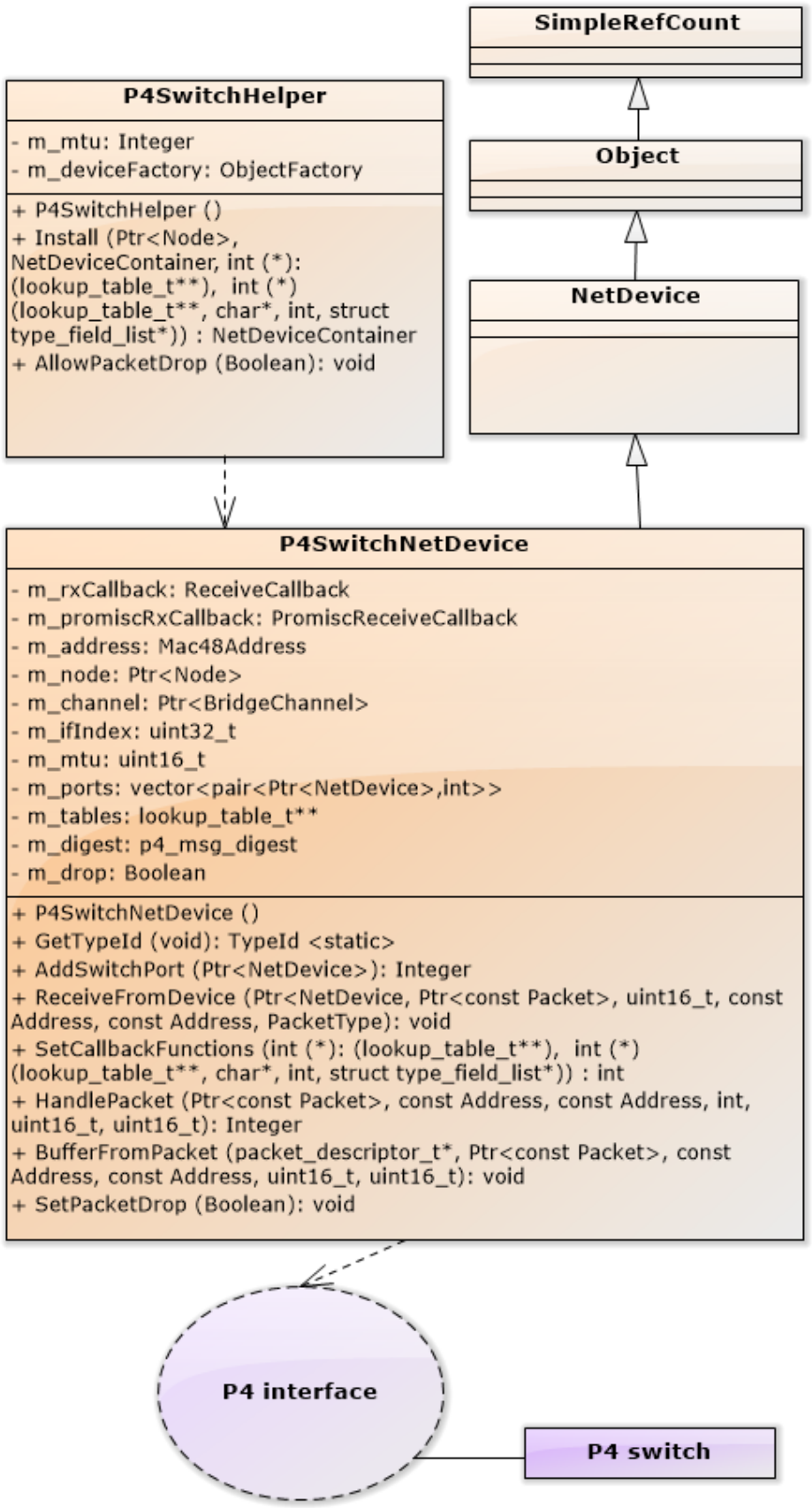
1. *Előkészületek.* A kapcsoló-táblák felállítása, callback függvények beállítása, adatkapcsolati réteg inicializálása, csomag konvertálása.

2. *Csomagfeldolgozás.* A csomag átadása a P4 switch-nek, amely a kapcsoló-táblák használatával megkeresi és végrehajtja a megfelelő switch utasításokat. Eldől, hogy mit kell csinálni az adott csomaggal, megállapítja a kimeneti port számát.

3. *Továbbítás.* A P4 switch-től a kimenő port lekérdezése, majd ezen port(ok)on kereszül a csomag továbbítása, esetleg eldobása.

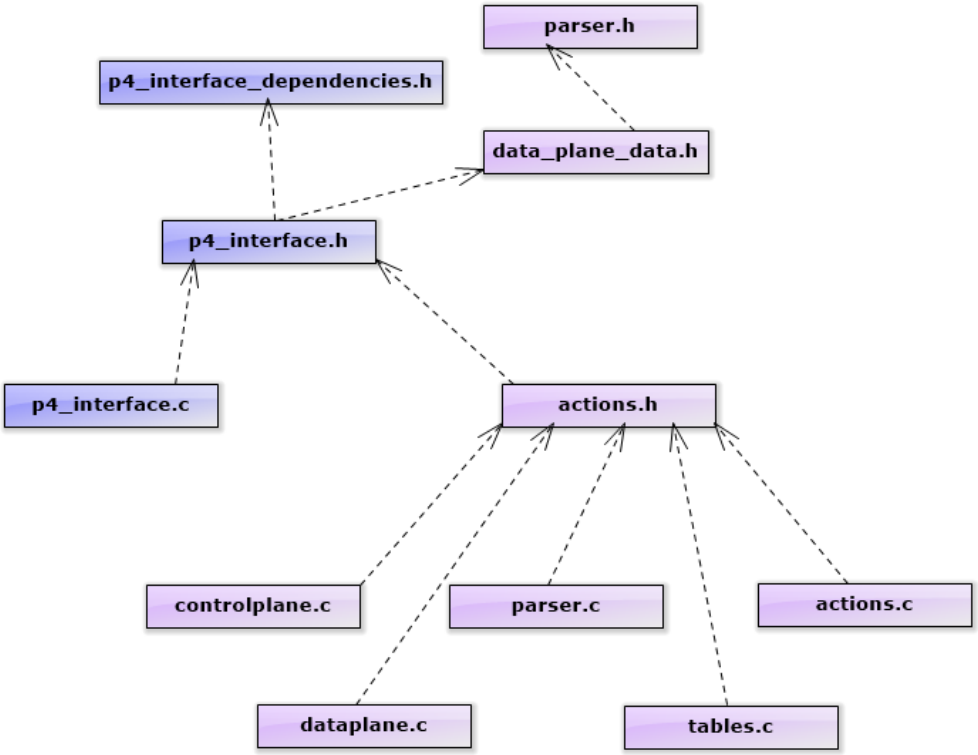
3.1.4. A modul szerkezete

Végezetül pedig szeretnék kitérni a P4 switch modulom szerkezetére. Míg a 3. ábrán jól szemlélteti mi-hogyan történik a modulon belül, az osztályok kapcsolatának ez nem pontos reprezentációja. A következő ábrán osztálydiagrammon kíséreltem meg bemutatni a modul struktúráját. Tekintettel arra, hogy a lényegi rész (maga a switch) C nyelven íródott, az ábra főként csak az "NS-3-as részét" írja le a modulnak. Érdemes tudni még, hogy ezen osztályok egy úgynevezett `ns3` névteren belül helyezkednek el, míg a P4 interface és a P4 switch azon kívül.


SimpleRefCount
Object
NetDevice
P4SwitchHelper
- m_mtu: Integer
- m_deviceFactory: ObjectFactory
+ P4SwitchHelper ()
+ Install (Ptr<Node>, NetDeviceContainer, int (*): (lookup_table_t**), int (*) (lookup_table_t**, char*, int, struct type_field_list*)) : NetDeviceContainer
+ AllowPacketDrop (Boolean): void
P4SwitchNetDevice
- m_rxCallback: ReceiveCallback
- m_promiscRxCallback: PromiscReceiveCallback
- m_address: Mac48Address
- m_node: Ptr<Node>
- m_channel: Ptr<BridgeChannel>
- m_ifIndex: uint32_t
- m_mtu: uint16_t
- m_ports: vector<pair<Ptr<NetDevice>,int>>
- m_tables: lookup_table_t**
- m_digest: p4_msg_digest
- m_drop: Boolean
+ P4SwitchNetDevice ()
+ GetTypeId (void): TypeId <static>
+ AddSwitchPort (Ptr<NetDevice>): Integer
+ ReceiveFromDevice (Ptr<NetDevice, Ptr<const Packet>, uint16_t, const Address, const Address, PacketType): void
+ SetCallbackFunctions (int (*): (lookup_table_t**), int (*) (lookup_table_t**, char*, int, struct type_field_list*)) : int
+ HandlePacket (Ptr<const Packet>, const Address, const Address, int, uint16_t, uint16_t): Integer
+ BufferFromPacket (packet_descriptor_t*, Ptr<const Packet>, const Address, const Address, uint16_t, uint16_t): void
+ SetPacketDrop (Boolean): void
P4 interface
P4 switch

4. ábra

A P4 interface fájljai közötti hierarchia leegyszerűsítve:



5. ábra

Ez azonban csak az ideális eset, a valóságban a különböző állományok között előfordul redundancia include-olások során. Például, mint azt a használati előkészületeknél említettem, a `controlplane.c, dataplane.c, parser.c, tables.c, actions.c` fájloknál a sikeres fordítás érdekében a p4_interface.h fájlt külön include-olni kellhet. Továbbá mivel generált fájlokról van szó, ezért többször is előfordul hogy ezek a fájlok egymást is include-olják.

3.2. Megoldás menete, implementációs döntések

Ebben a szekcióban részletezem, hogy a modulom egyes részeit miként készítettem el, illetve mi mindenre kellett odafigyelnem implementáció során. Az alszekciók sorrendjét igyekeztem a megvalósításuk időrendjéhez mérten meghatározni.

3.2.1. A P4SwitchHelper osztály

Amiatt, hogy a szimulációban egyszerűbben lehessen használni a switch-cset (és a könnyebb bővíthetőség érdekében is), létrehoztam egy P4SwitchHelper osztályt. Ez egyébként konvenció az NS-3-as modulok körében: a különböző hálózati eszközöket úgynevezett Helper-osztályokon keresztül is hozzá lehet adni a szimulációhoz. A P4SwitchHelper osztály felelős a P4SwitchnetDevice létrehozásáért, illetve konfigurálásáért.

Mint ahogy az a 4. ábrán is megfigyelhető volt, a P4SwitchHelper jelenlegi szerkezete nem túl bonyolult. Első nekifutásra ennek az osztálynak a konstruktorában közvetlenül példányosítottam a P4SwitchNetDevice osztályomat, ezen azonban később módosítottam. Úgy figyeltem meg, hogy az NS-3-as környezetben az a szokás, hogy a Helper-osztályokon belül először egy általános objektumot hozunk létre (`ObjectFactory m_deviceFactory;`) majd pedig később megadjuk neki, hogy pontosan milyen eszközt képvisel:

```
m_deviceFactory.SetTypeId ("ns3::P4SwitchNetDevice");
```

Ehhez persze a P4SwitchNetDevice osztályon belül tudatni kell, hogy ilyen TypeId létezik, azaz definiálni kellett a P4SwitchNetDevice osztályban megadni az alábbiakat:

```
TypeId
P4SwitchNetDevice::GetTypeId (void)
{
  static TypeId tid = TypeId ("ns3::P4SwitchNetDevice")
    .SetParent<NetDevice> ()
    .SetGroupName ("P4")
    .AddConstructor<P4SwitchNetDevice> ()
  ;
  return tid;
}
```

Az Install metódus sem volt túl bonyolult: lényegében annyi történik itt, hogy NS-3-as eljárások segítségével hozzákapcsolom a megadott portokat a switch eszközhöz;

ezeken keresztül fog a switch csomagokat fogadni és továbbítani. Ehhez nem kellett külön semmit sem létrehoznom a P4SwitchNetDevice osztályomban, a NetDevice-ból való öröklődésnek köszönhetően minden megvolt hozzá. Emellett pedig még a callback függvényeket is továbbadom (tárolásra) a P4SwitchNetDevice-nak.

3.2.2. A P4SwitchNetDevice alapjai

Ez a lépés megint csak egyszerűen ment, köszönhetően az OpenFlowSwitch-nek, melyet mintául tudtam venni. A konstruktort később, a switch pontosabb ismerete mellett egészítettem csak ki, viszont az eszköz működéséhez elengedhetetlen keret-metódusokat gond nélkül létrehoztam.

3.2.3. A switch hozzáillesztése a modulhoz

A nyolc generált C fájlban számos olyan típus, struktúra, illetve függvény volt, melyek implementációja ezen fájlokon kívülre esett; ezeket nekem kellett definiálni. Hogy ezt megtehessem, létrehoztam egy p4_interface.h és egy p4_interface.c állományt. Azonban mielőtt ezeket elkezdhettem volna feltölteni, meg kellett oldanom azt, hogy az NS-3 szimuláció fordításakor a `waf` fordító ezeket a C fájlokat bele tudja fordítani a programba.

Az NS-3 a WAF (Python-alapú fordítórendszer) segítségével fordul, így a feladat nem volt annyira egyszerű, mint egyébként környezetben az szokott lenni. Szóba került többféle megoldás is, például a fájlok dinamikus programkönyvtárrá alakítása, és annak használata. Végül azonban különböző, ezt a témát tárgyaló források [12] átnézése után sikerült megoldanom a problémát. A program így most statikus könyvtárként kezeli a P4 interface-t. Ehhez a modul `wscript` fájlját kellett kiegészítenem a megfelelő flagekkel, illetve a csatolandó állományok nevével. Ugyanígy a nyolc generált fájl is hozzáillesztésre került, majd ezek után jöhetett a fordítási hibák feloldása: az interface-ben elkezdtem deklarálni mindent, ami a t4p4s fordítónál a generált fájlokon kívülre esett.

Ez leginkább azt jelentette, hogy át kellett néznem a t4p4s `src` mappájából a hardver-függőként megjelölt állományokat. A t4p4s DPDK és Freescale backendet is támogat, mindkettő hasznos volt abban, hogy jobban megértsem miként működik a switch és használni tudjam a szimulációban. Kiemelt fontosságúak a switch által (csomagfeldolgozásra) használt makrók, melyeket az állományból nagyrészt át tudtam emelni. (Erre később, a 3.2.9. alszekcióban még kitérek.) A hardware-dependent részben számos típus- és függvénydefiníciót azonban nem a DPDK vagy

a Freescale mappákban, hanem az `src/hardware_dep/shared` mappa almappáiban találtam.

A különböző állományokból egybegyűjtött kód viszonylag hosszú volt, ezért a könnyebb átláthatóság érdekében létrehoztam egy `p4_interface_dependencies.h` állományt, melyben elhelyeztem azokat kódrészleteket, amelyekhez az interfész írása során viszonylag kevesebbszer kellett hozzányúlnom. Csak a fontosabb és/vagy gyakrabban használt elemeket hagytam meg a `p4_interface.h`-ban.

Miután minden fordítási hibát sikerült megszüntetnem, kezdődhetett a switch működésre bírása.

3.2.4. Kapcsoló-táblák létrehozása

Habár a felhasználó adhat meg tábla-inicializáló callback függvényt, ez nem azt jelenti, hogy mindent kézileg kell létrehoznia. A program indulásakor lefutó `create_tables` eljárásban beállítok több dolgot is a `tables.c` alapján. Többek között itt zajlik a:

- tábláknak memória-terület allokálása
- tábla-azonosítók és táblatípus megadása
- `key` és `value` méretének rögzítése
- alapértelmezett `value` memória-területének allokálása

Ezen felül pedig ebben az eljárásban lesz beállítva az a `backend`, amit a switch eredetileg használt volna. Erről annyit kell tudni, hogy ez egy globális változóként tárolt pointer gyakorlatilag, a DPDK-s (illetve Freescale-es) P4 switch eredetileg ezt használta a táblák elérésére. Az én esetemben habár a `backend`-et az NS-3-as rész képviseli, mivel a generált kód ezt a backend-pointert használja, ezért nem tudtam ezt egy az egyben kikerülni. Megoldásként létrehoztam egy `backend_data_t` típust az interfészben:

```
typedef struct backend_data_s {
    lookup_table_t** t;
    p4_msg_digest pdc;
    uint8_t* metadata;
} backend_data_t;
```

Egy ilyen struktúrára állítom a `backend` pointert, aminek a memória-területét itt, a `create_tables`-ben foglalom le – majd nem sokkal később be is állítom: a `backend` tartalmazni fog a kapcsoló-táblákra egy pointert, a `digest` callback függvényre is

egy pointert, illetve további meta-információ is átadható rajta. Mivel mindez egy globális változóban van elhelyezve, ügyelni kell arra hogy ne keletkezzen konfliktus: a változóban a táblára mutató pointert cserélni kell, ha switch-eszköz váltás történik. Ez azonban magától adódik, hiszen egyébként is egyszerre csak egy "switch-csel dolgozunk", azaz egyszerre csak egy switch dolgozza fel az aktuális hálózati csomagot.

3.2.5. Kapcsoló-táblák szerkezete

A kapcsoló-tábla struktúrája már a 2.2.3. szekcióban szóba került, viszont a lényegi rész, azaz maga az adatrekord még nem szerepelt. Ezt a struktúrát próbáltam úgy definiáltni, hogy több különböző típusú (például exact vagy lpm) kapcsoló-tábla is használni tudja.

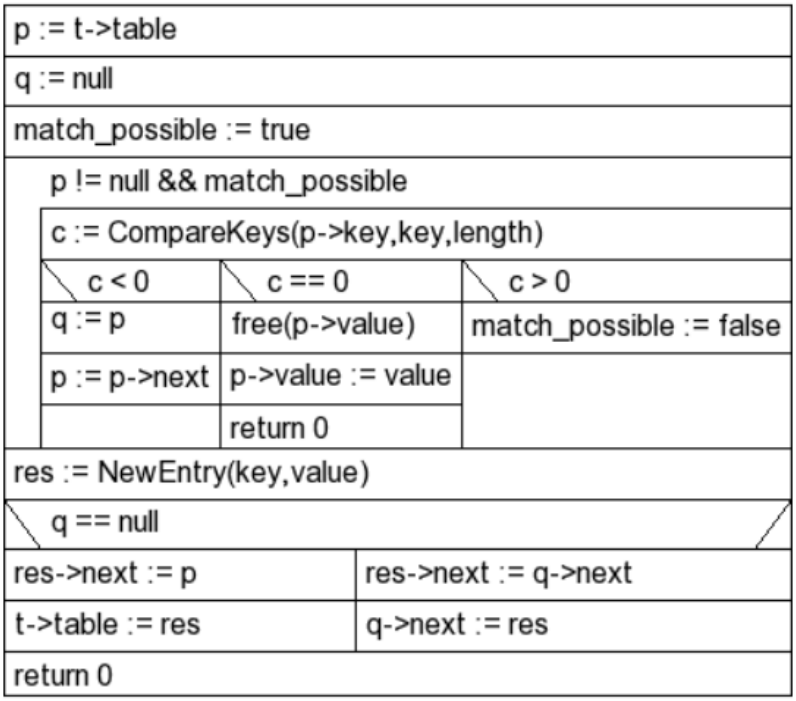
```
typedef struct table_entry table_entry_t;
struct table_entry {
    uint8_t* key;
    uint8_t* value;
    uint8_t* mask;
    table_entry_t* next;
    table_entry_t* child;
};
```

A legelső minta switch-nél (mely exact_lookup-ot használ) nem használom a mask és a child pointereket, de a későbbiekben ezekre még szükség lesz. A `key` és `value` szerepe magától értetődő: a kulcs alapján keresünk a kapcsoló-táblákban, ez lehet például egy MAC vagy egy IP cím. Hossza a P4 switch-ben van definiálva. A `value` tárolja a végrehajtandó utasítást (action-t, szerkezete szintén kiolvasható a lefordított C fájlokból).

3.2.6. Az exact_add és exact_lookup függvények

A kapcsoló-táblák kezeléséhez elengedhetetlen, hogy legyen egy-egy függvényünk új rekord hozzáadására, illetve keresésére. Első körben ez az `exact_add` és `exact_lookup` implementálását jelentette, hiszen ezek azok a függvények melyeket az első minta switch használ. Ezek működésének ismertetésére szavak helyett egy-egy struktogramot szeretnék bemutatni (6. és 7. ábra).

```
int exact_add (lookup_table_t* t, uint8_t* key, uint8_t* value);
```



6. ábra

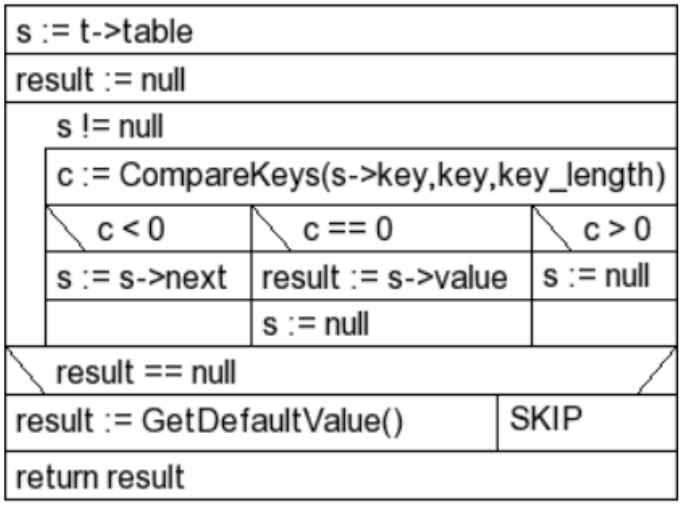
A fenti ábrán `o, p` és `q table_entry_t*` típusú változók, míg a NewEntry függvény új memóriaterületet foglal le és feltölti a key és value értékekkel.

A másik függvényt, mely a CompareKeys névre hallgat, több helyen is használom (6.,7.,8. és 9. ábrák). Segítségével tetszőleges (a 3. paraméterben megadott hosszúságú) kulcsokat hasonlítok össze. A függvény bájtonként vizsgálja az adott kulcsokat (bájtsorozatokat). A visszatérési érték -1, 0, vagy 1 lehet, annak megfelelően, hogy a másik két paraméter egyezett-e. Egyenlőség esetén ez az érték 0, ha az első paraméterben megadott kulcs számértéke kisebb, akkor -1, különben pedig 1.

A CompareKeys segítségével rendezem a kapcsoló-tábla rekordjait (mely itt gyakorlatilag mindössze egy láncolt lista), a keresés lineáris.

Az `exact_lookup` struktogramja (7. ábra) meglehetősen hasonlít a társára, `s` változó itt `table_entry_t*` típusú, `result` viszont `uint8_t*`, mely egy `value`-ra fog mutatni. A switch ezt át fogja *cast*olni a neki megfelelő formátumba (*action value* struktúrába), ennek köszönhetően az értékeket itt, illetve az `exact_add` során nem kellett castolnom, elég csupán bájtsorozatként tárolni őket.

```
uint8_t* exact_lookup (lookup_table_t* t, uint8_t* key);
```



7. ábra

3.2.7. Portok

Ezek után a P4-es programrész nagyjából meg is volt, a szimuláció működésre bírásához azonban a P4SwitchNetDevice osztályban akadtak teendőim. Az osztály konstruktorában a kapcsoló-tábla inicializálásának meghívása mellett még azonosítót is kellett rendelnem a portokhoz. Ennek érdekében *(NetDevice,int)* párokat alakítottam ki, ahol az int az azonosító szám.

A P4SwitchNetDevice AddSwitchPort metódusát (melyet a P4SwitchHelper hív meg) úgy írtam meg, hogy a portok azonosítóit 0-tól kezdve osztja ki, mindig 1-el növelve az azonosító számot. A maximum tárolható portok száma 255, ezt a p4_interface.h fájlban határoztam meg (tehát a szimulációból nem módosítható jelenleg). Ugyancsak "beleégetett" érték a broadcast port száma, ez viszont a t4p4s által generált fájlok által van meghatározva. (Ennek ellenére az értéket a MAX_PORTS-hoz hasonlóan makróként betettem a p4_interface.h fájlba is, pusztán kényelmi szempontból. Ez az érték egyébként is csak a t4p4s compiler módosításával lenne változtatható.) Az AddSwitchPort metódus a broadcast port számát azonosító kiosztáskor automatikusan átugorja.

3.2.8. Hálózati csomagkonfigurációja

A szimuláció során a csomag kezelésekor (HandlePacket metódusában a P4SwitchNetDevice-nak) nem volt elég a csomagot szimplán továbbadni a P4 switch-nek feldolgozásra. Az NS-3 szimulációban mire eljut a csomag a Receive-FromDevice metódushoz (ahhoz a metódushoz amin ekresztül a NetDevice fogadja

a csomagot), az NS-3 már magától leválasztotta az ethernet fejlécet. Megfigyelhető, hogy ez az OpenFlow switch-nél is gondot jelentett.

A csomag konvertálására hoztam létre a BufferFromPacket metódust. Az ethernet header visszaillesztéséhez három adatra volt szükség mindenképp: a címzett MAC címrére, a forrás MAC címrére, illetve az ethernet protokol számra. Ezeket továbbadva a BufferFromPacket metódusomnak, ebben a sorrendben bájtsorozatra konvertálva már kész is volt a (VLAN-tag nélküli) fejléc. Ezek után még magát a csomagot is át kellett alakítanom (ns3::Packet formátumból bájtsorozattá), annak érdekében, hogy létre tudjam hozni a P4 switch által használt `packet_descriptor` típust.

3.2.9. Csomag feldolgozása

Mielőtt a csomagfeldolgozás megkezdődhetne, a P4SwitchNetDevice-ban el kell végezni két műveletet, melyek úgymond "előkészítik a terepet". A P4 switch-nek van egy eljárása mely az `init_dataplane` névre hallgat, és ezt kell meghívni ahhoz, hogy később a switch képes legyen kezelni a beérkező csomagot. Az `init_dataplane` paraméterként várja azt a packet_descriptort, melyet a BufferFromPacketsben már össze lett állítva. A csomagleíró típusnak a következő a szerkezete:

```
typedef struct packet_descriptor_s {
    uint8_t* data;
    header_descriptor_t headers[HEADER_INSTANCE_COUNT+1];
    parsed_fields_t fields;
    void*  wrapper;
    uint8_t dropped;
} packet_descriptor_t;
```

A BufferFromPackets metódusban a `packet_descriptor`nak csupán a *data* adattagja lett feltöltve, az `init_dataplane` azonban ebben a struktúrában kezel mást is: a *headers* és a *fields* mezőket. Ezek típusa a p4_interface.h állományban van meghatározva, a t4p4s compilerből előkeresett fájlok alapján. Felépítésüket és működésüket itt most külön nem részletezném, mivel ez már úgymond a t4p4s compiler hatáskörébe tartozik. A headers és fields mezőket a switch generált C fájljai fogják kezelni, legtöbbször a p4_interface_dependencies.h állományban elhelyezett makrókon keresztül. Ezekbe a makrókba minimálisan bele kellett nyúlnom korábban (a DPDK-s részeket kiszedni ahhoz, hogy lefordulhasson a szimuláció), de azt leszámítva a switch ezen részével nem volt más dolgom.

A `packet_descriptor` tartalmaz még egy wrapper és egy dropped mezőt: előbbit az eredeti csomag (ns3::Packet) hozzáillesztéséhez lehet használni, míg az utóbbit arra használja a switch hogy jelezze, ha feldolgozás közben eldobta a csomagot.

Az `init_dataplane` mellé még szükség volt külön egy függvény meghívására, mely a `packet_descriptor` *field* mezejébe beírja a csomaghoz tartozó bemeneti port számát (tehát azt a portot ahonnan a csomag jött). Erre azért van szükség, mert ezt az `init_dataplane` magától nem teszi meg.

Az inicializálás után kezdődhet a csomagfeldolgozás. Ehhez a P4 switch `handle_packet` eljárását kellett meghívnom, mely után megindul a csomag kezelése (főleg az action.c és a dataplane.c generált állományokban leírtak alapján). Az itt történő dolgok szintén csak a felhasználón fognak múlni, tehát kódot írnom ennél a résznél nem kellett. Ennek ellenére igyekeztem lekövetni a csomag útját a minta switch-csen belül, ez a rész fontos volt számomra az *add*, a *lookup*, és a *digest* függvények helyes működésének ellenőrzésekor, valamint debuggolás során is. Nem minden sikerült elsőre tökéletesen, így szükséges volt a switch különböző pontjain ellenőrizni, hogyan állnak a dolgok.

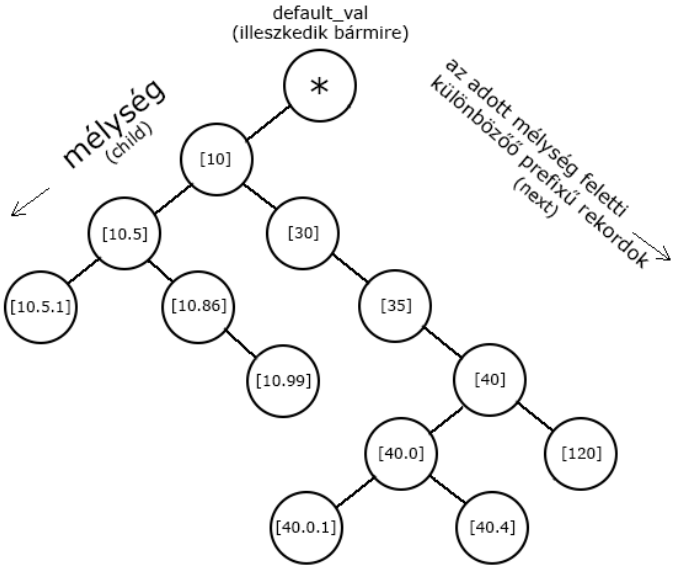
Mindezek után legvégül a P4SwitchNetDevice HandlePacket metódusának végén ki kell olvasni a kapott eredményt, azaz a kimeneti portot. Erre azért van szükség, mert a C kódrészlet nem látja az NS-3-as NetDevice osztályt, és nem tudja elérni sem. (A korábban meghívott handle_packet pedig nem tér vissza értékkel, és ezen módosítani sem lehetett.) Így tehát az interface-en keresztül kell a portszámot lekérdezni, majd azt lekezelni: a csomagot a kapott porton keresztül továbbítom, kivéve ha a port maga a broadcast port, vagy a packet_descriptor `dropped` mezőjének figyelése be van kapcsolva.

3.2.10. Következő lépések

Ezek után már a minta switch amelyet alapul vettem helyesen működött, tehát a hálózati szimuláció P4 switch segítségével képes volt a 2. hálózati rétegen (adatkapcsolati rétegen) való forgalomirányításra. A következő lépések a funkcionalitások kiegészítése volt, kezdve azzal, hogy L3 switch-cset is támogasson a modul. Az ilyen switch-ekhez azonban egy másik, gyakorta használatos kapcsoló-tábla típushoz kellett támogatást írnom. A *Longest Prefix Match* tábláknál a kulcsnál nem feltétlenül szükséges a teljes egyezés (exact match), csupán a megadott prefix hosszban kell, hogy egyezzenek.

3.2.11. Az lpm_add és lpm_lookup függvények

A *longest prefix match* tábla szerkezete eltér az *exact match* tábláétól. Itt már gyakorlatilag egy mátrixról van szó, melynek sorai a különböző mélységeket reprezentálják, oszlopai pedig az adott mélységek található rekordokat tartalmazzák. Erre a struktúrára úgy is lehet gondolni, mint egy bináris fára:



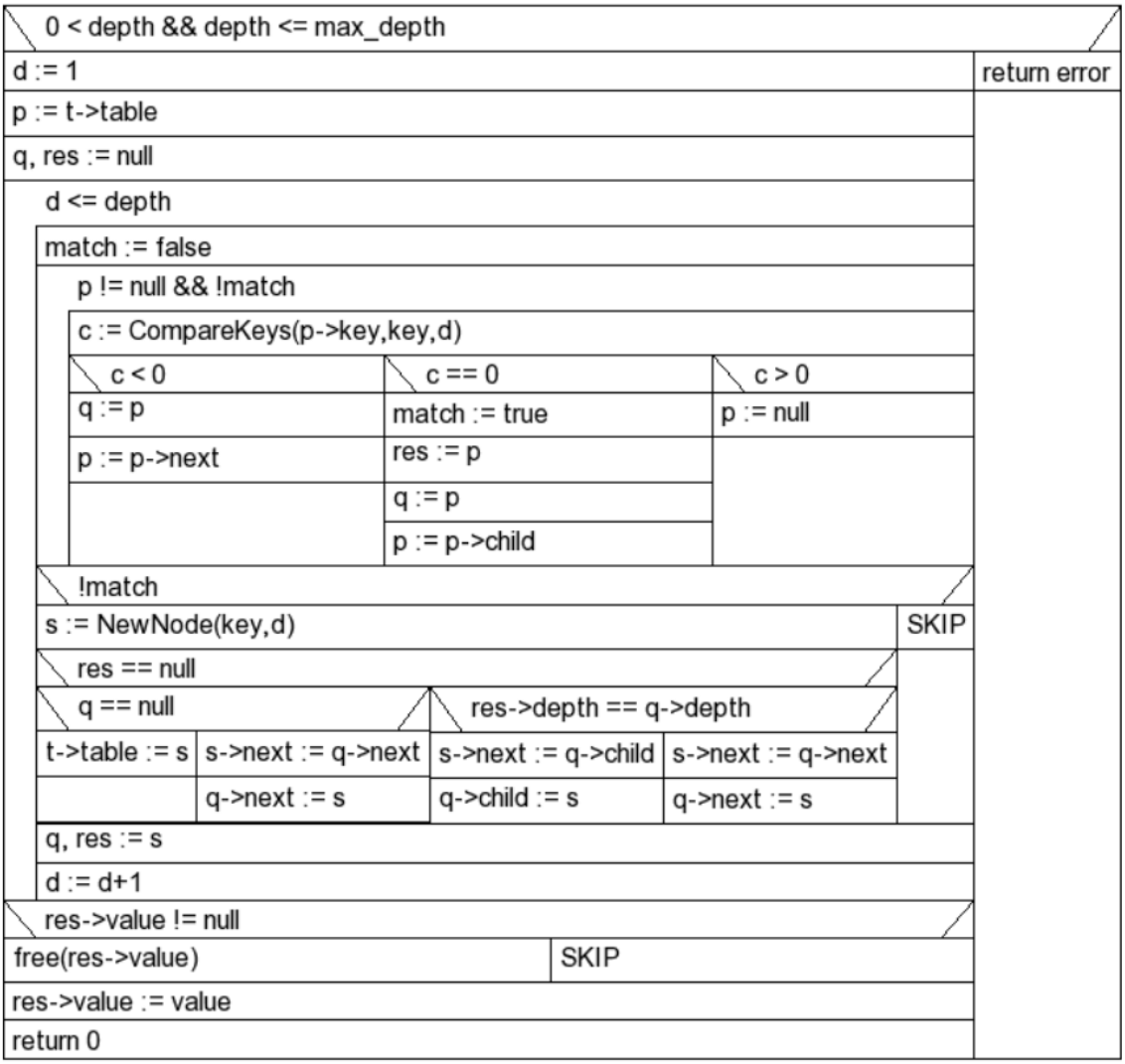
8. ábra

Ennek oka az elérés módja: az lpm keresés, ill. beszúrás kapcsán adódott, hogy fa szerkezetben tároljam az adatokat. Emellett szól az is, hogy a táblát mindenképpen dinamikusan kell tárolni, hiszen rettenetesen sok erőforrást venne idénybe ha minden lehetséges helyet előre lefoglalnék.

Meg kell azonban jegyeznem, hogy az jelenleg implementált lpm fa csupán leegyszerűsített verziója az lpm fáknak: bitek helyett bájtonként vizsgálja és rendezi a kulcsokat. Ez egy limitáció, melyre bővebben a 3.3.3. szekcióban ki fogok térni.

A két függvény struktogramja a következőképpen néz ki:

```
int lpm_add (lookup_table_t* t, uint8_t* key, uint8_t depth, ↩
    uint8_t* value);
```



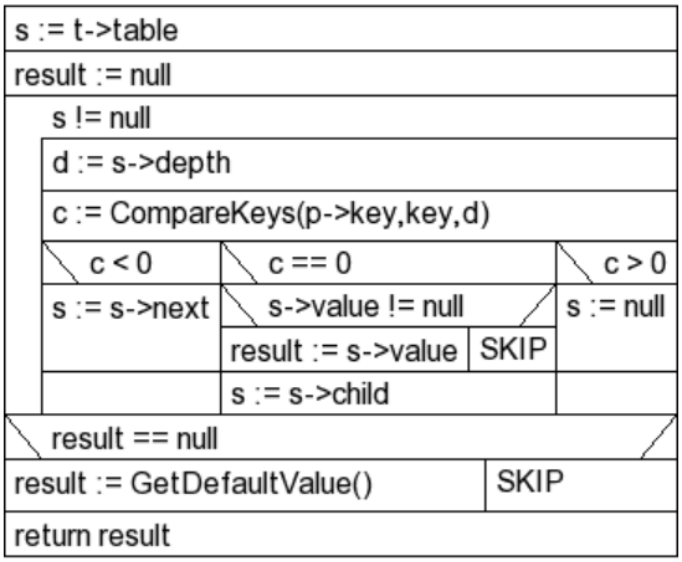
9. ábra

Az lpm_add a megadott mélységben létrehoz egy új node-ot, melyben elmenti az aktuális értéket. Azonban ahhoz, hogy később ez megtalálható legyen, a fa bejárása közben folyamatosan hoz létre újabb node-okat, melyeknek csupán a kulcs részét tölti ki, az értéket nem. Hasonlóan az exact_add függvényhez, abban az esetben ha már létezett a megadott kulcshoz érték, a régi érték felül lesz írva.

A kulcsok egymás mellett növekvő sorrendben vannak elhelyezve (gyorsabb lookup érdekében) valamint csak a szükséges helyet foglalják le (kevesebb memóriaigény érdekében), azaz egy 6 bájt hosszúságú kulcshoz tartozó, 2. mélységi szinten elhelyezkedő rekord csak az első két bájtját tárolja a kulcsnak.

Az lpm_lookup már egyszerűbb mint az lpm_add, a keresés során viszont figyelni kellett, hogy az adott entrynél van-e érték (vagy csak a fentebb említett okokból van lefoglalva). A táblán megfigyelhető egyébként, hogy a 8. ábrán látható reprezentációval ellentétben a gyökeret külön nem tárolom: a tábla egyből az első mélységi szintnél indul. Ha nincs találat akkor a tábla-inicializálásnál eltárolt alapértelmezett értéket adom vissza.

```
uint8_t* lpm_lookup (lookup_table_t* t, uint8_t* key);
```



10. ábra

Megjegyezném, hogy habár a `table_entry_t` -nek nincs `depth` nevű adattagja, a `mask` elnevezésű adattag erre a célre tökéletesen megfelelő volt. Bár implementálhattam volna a prefix-hosszot úgy is, hogy maszkként legyen kezelve (például `[0,0,0,255]`) végül úgy döntöttem, hogy használat szempontjából egyszerűbb, ha egyetlen számértéket (például `24`) kell a felhasználónak megadnia.

3.2.12. controlplane.c

A fejlesztés során el kellett gondolkoznom azon, hogy miként valósítsam meg a *digest*et. Habár az elkerülhetetlen, hogy a felhasználó a digest callback függvény használatával egészítse ki a switch-cset, a t4p4s által fordított switch-ben szintén léteznek kisegítő lehetőségek. A switch controleplane.c állományában rendszerint olyan függvények vannak generálva, melyek egy ún. `p4_ctrl_msg` struktúrán keresztül kezelik a táblák alapértelmezett értékeinek beállítását, illetve új rekord táblához való hozzáadását.

Ez annyit jelentett volna, hogy a felhasználónak a szimulációs fájlban a callback függvény írásakor nem közvetlenül kellett volna az `exact_add` vagy `lpm_add` függvényeket hívnia. Ehelyett egy `p4_ctrl_msg` kellett volna feltöltenie, és azt átadni a `p4_ctrl_msg` megfelelő eljárásának (`recv_from_contoller`).

Ezt a megoldást végül azonban elvetettem, mert habár valamennyire elegánsabb megoldásnak tűnhet (elfed bizonyos függvényeket a felhasználó elől, vagyis a felhasználónak azokkal nem kell foglalkoznia), valójában nem egyszerűsített volna a modul használatán. A `p4_ctrl_msg` struktúrát meg kellett volna ismernie a felhasználónak a helyes használathoz: mezői feltöltésénél nem mindegy, milyen típusú változókkal, rekordokkal teszi azt meg (castolni kell többüket). Ez azt jelentette, hogy a hibalehetőségek számát nem igazán sikerült volna csökkenteni, a gépelt sorok száma még több is volt, és a debuggolás is nehézkesebb úgy, a kapcsoló-tábla feletti extra rétegek miatt.

3.2.13. A ternary_add és ternary_lookup függvények

A ternary típusú kapcsoló-tábla annyit tesz, hogy a kulcsok kezelésénél egy maszkot lehet megadni, azaz kulcsrészlet alapján is lehet keresni. A *longest prefix match* lookup gyakorlatilag egy olyan ternary művelet, melynél a kulcsban egy adott bájttól kezdődően minden bájt le van fedve.

A ternary műveleteket nem bonyolítottam túl, ennek okát a fejlesztési lehetőségeknél, a 3.3.2. alszekcióban fejtem ki bővebben. A ternary műveletek jelen implementációban nagyon hasonlatosak az *exact* típusú táblák műveleteihez, pár apró eltérésben: a kulcsok nincsenek növekvő sorrendbe helyezve (a hozzáadás mindig a legelső pozícióra szúrja be a kulcsot), illetve saját kulcs-összehasonlító függvényt használ (mely a maszkot is figyelembe veszi).

Minden egyes rekord a kulcs mellett tartalmaz egy maszkot is (`mask` adattag), mely meghatározza, hogy a megadott kulcsnak csak mely részét hasonlítja össze a switch lookup során.

3.2.14. A _remove függvények

Ahhoz, hogy a felhasználó könnyebben tudja kezelni a kapcsoló-táblákat, írtam függvényeket külön a bejegyzések kulcs-szerinti törléséhez is. Ezt már korábban is meg tudta tenni a felhasználó bizonyos mértékben: a lookup függvények az értékekre mutató pointerrel térnek vissza, melyeket aztán fel lehet szabadítani (ám

körülményes volt ügyelni arra hogy hibamentes maradjon a program). A _remove függvények ezt igyekeznek megkönnyíteni, viszont érdemes észben tartani, hogy viselkedésük táblatípustól függően eltérő lehet.

- `exact_remove`: A megadott kulcs alapján törli a legelső (és egyetlen) illeszkedő rekordot.
- `lpm_remove`: A megadott kulcs és mélység alapján törli a megfelelő rekord *értékét*, ha az létezik. Magát a bejegyzést nem törli, kivéve, ha a rekordnak nincs "gyereke". (Ha van, akkor nem törölhető a bejegyzés, hiszen a hozzá kapcsolódó értékek elérhetetlenné válnának.) A teljes rekord törlése esetén az eljárás nem távolítja el a rekord felett álló, esetlegesen üres (érték nélküli) rekordokat.
- `ternary_remove`: A megadott kulcs alapján törli az *összes* olyan rekordot, amelyhez a kulcs illeszkedik. Ilyen rekordból több is lehet, hiszen a kulcsok maszkokkal együtt vannak tárolva a táblában.

3.3. Fejlesztési lehetőségek

3.3.1. Kapcsoló-táblák optimalizálása

A program futását gyorsítani lehetne, például ha a jelenlegi táblaszerkezeteket lecserélnénk hash-táblákra; ez azonban C nyelven kell, hogy legyen megírva (C++ nem jó, hiszen a switch programrész nem tudná használni), és nagy valószínűséggel külső könyvtárként kellene csatolni a modulhoz. Felmerül néhány érv viszont, hogy a modul jelenlegi verziójában ezt miért nem így csináltam:

1. Mint ahogy azt már a 3.2.3. szekcióban említettem, külső könyvtárak csatolása az NS-3 modulhoz meglehetősen nehézkes. Próbáltam keresni megoldást egész könyvtárrendszerek include-okására, de jelenlegi tudásom szerint csak úgy sikerült ezt megoldani, hogy minden egyes állományt név szerint felsorolok a modul fő `wscript` fájljában. Ez azt jelenteni, hogy ha hash-seléshez egy olyan C könyvtárat szeretnék használni, ami 30 különböző állományból áll, akkor mindegyiket kézileg a `wscript` fájlban fel kellene tüntetnem. Nem lehetetlen, csak nem túl praktikus.

2. Ténylegesen nem számít a program futási ideje. Az NS-3 szimulációt mérési szempontból nem érinti; a szimulációs időt nem befolyásolja a futási idő. A valós switch-csekkel ellentétben itt a kapcsoló-táblák kezelésének nem kell a legoptimálisabb lennie, hiszen a szimulációban használatos idő értékeket a felhasználó állítja be és a szimuláció azok szerint jár el.

3. A mostani, egyszerűbb adatstruktúra számomra jobban átlátható volt, ami elősegítette a debuggolást. Könnyebb ideiglenes módosításokat végrehajtani tesztelés érdekében.

Az 1. pontban leírtakat figyelembe véve, megfelelő C-ben íródott könyvtárak után keresgélve végül az uthash-t [15] találtam, ami ígéretesnek tűnt. További optimalizálás során ebbe az irányba haladnék tovább.

3.3.2. Ternary search

Habár a program támogat ternary típusú kapcsoló-táblákat, az ezekre vonatkozó műveletek (add, lookup, remove) szintén nem a legoptimálisabbak. Ez az előző, 3.3.1. alszekcióban leírtak miatt van; gyorsabb lookup érdekésben más `table_entry` struktúrára lenne szükség. Mivel a szimuláció tényleges futási idejének gyorsítása nem volt prioritás, ezért a ternary műveletek gyorsasága miatt nem módosítottam a teljes interfészt.

3.3.3. Bitenkénti kulcs-összehasonlítás

Fejlesztési lehetőség az is, hogy a *longest prefix match* táblákat a jelenlegi, bájtonkénti összehasonlítás helyett bitenkénti összehasonlítással kezeljük. Egy ilyet leprogramozni jóval nagyobb feladat lett volna mint a jelenlegi algoritmusokat lekódolni. Megoldást jelenthet ehhez egy már meglévő implementáció felhasználása; ebben az esetben azonban a 3.3.1 alszekcióban könyvtár-használattal kapcsolatban leírtak szintúgy érvényesek.

Megoldásként a Judy-tömbök [16] használatát vizsgáltam, azonban az adatstruktúra könyvtárszerkezete számos állományból áll, melyek importálása a modulomba nehézkes lett volna. Így erről végül lemondtam, azonban ha ezen a vonalon dolgoznék tovább és Judy-tömböket szeretnék használni, a P4 konzorcium tárolóján található BMv2-es modelből [17] lehetne ötletet meríteni, miként érdemes a Judy adatstruktúrát a P4 switchen belül használni.

3.3.4. Különböző P4 switch-csek egyidejű használata

Jelen esetben több, egymástól különböző P4 switch-cset használó szimuláció nehézkesen használható: a nyolc generált C fájlnak kötött az elnevezése, így névütközés léphet elő. A probléma megoldható a fájlok átnevezésével, wscript fájl kiegészítésével, valamint a switch-csen belüli include-ok javításával, de ez nem éppen egy praktikus megoldás.
A probléma kikerülhető azzal is, hogy a P4 nyelven megírt switch-be több mindent "belepakol" a felhasználó; egy olyan switch-cset készít, amit univerzálisan több helyen is fel tud használni.

A fent említett két opció közül egyik sem túl elegáns, ezért ezt a témát is fejlesztési lehetőségként tudom be. Tetszőleges számú switch könnyű használatához azonban dinamikus könyvtárakra lenne szükség, illetve meglátásom szerint még egy extra (switch-cseket leíró) callback függvényre is. Azonban még ha a modulom ezeket le is tudná kezelni, megvalósításuk a felhasználókra hárulna (mint ahogy most is rájuk hárul a C fájlok generálása és a callback függvények megírása), így kérdéses, hogy ez mennyit javítana a felhasználói élményen.

3.3.5. P4 switch-hez való kompatibilitás növelése

Bár az interfész le tudja kezelni az általános célokra íródott switch-cseket, speciális esetek (ahol a jelenlegi eszköztár nem elegendő) előfodulhatnak, különösképp a t4p4s compiler újabb verzióival. Például, jelenleg az interfész azt feltételezi, hogy a switch a kulcsoknál fix hosszúságú (tehát nem váltakozó méretű) maszkot

használ (mely jelenleg igaz, de nincs garancia rá, hogy mindig így lesz). Ez a `p4-interface-dependecies.h`-ban, a t4p4s-ból átemelt mező-kezelő makrók között található definiálva, mint ahogy sok hasonló "beépített switch művelet" is. Hátránya ennek, hogy ha a t4p4s compiler megváltozik, akkor az itt található makrók is javításra szorulhatnak.

Egy másik példa a switch által eldobásra megjelölt csomagok kezelése: a packet_descriptor-nak van egy "dropped" nevű flagje. Jelenleg ennek figyelése a P4SwitchNetDevice-omban ki van kapcsolva (kommentezve) mert az L2 teszt switch-csem ilyennel nem foglalkozik, de előfordulhat, hogy erre a felhasználónak szüksége lehet. Mindez attól függ, hogy a switch-cset a felhasználó hogyan írta meg. Fejlesztési lehetőségként ki lehetne egészíteni a modult úgy, hogy a szimulációból a csomageldobás-figyelését ki- és be lehessen kapcsolni.

Még egy probléma forrása lehet az, hogy a modul alapvetően "nem tudja" hogy őt L2 vagy L3 forgalomirányításra szándékozzák-e használni. Mint ahogy az L3 switch-csem tesztelése során (3.4.3. szekció) is felmerült, ha a P4 switch nincs felkészítve ARP requestek fogadására, akkor azt a szimulációban kell megtenni. Itt segíthet például az, ha a `test-switch-l3.cc` szimulációs fájlomban leírt ARP tábla-feltöltő függvény már be van építve a P4SwitchNetDevice modulva (például annak a helperjébe), hogy onnan meg lehessen hívni, ha szükség van rá.

3.3.6. További funkcionalitások bevezetése

Természetesen a fentieken kívül számos más feature-rel is ki lehet egészíteni a P4 interfészt. Definiálni lehetne például további ellenőrző függvényeket (checksum) vagy kényelmi funkciókat a könnyebb naplózás és csomag-követés érdekében (debug függvények). A t4p4s fordítóprogramtól függően egyéb módosítások is bevihetőek a switchbe, illetve NS-3 oldalról is maradtak még olyan funkcionalitások (osztály metódusok) melyekkel ki lehetne egészíteni a P4SwitchNetDevice-ot.

3.4. Tesztelés

A szimulációt teljes egészében is teszteltem (3.4.2 és 3.4.3), illetve egyes komponensei funkcionalitását is ellenőriztem (3.4.4). Ehhez a bevett módszerem a *logging* volt, melyről a következő alszekcióban (3.4.1) írok.

A dolgozatom írásakor hibaeset során alkalmanként használtam még a GDB debuggert is, hogy pontosabban be tudjam határolni hol szállt el a program. Ez a debugger elindítása után, mely a

```
$ ./waf --run <program-name> --command-template=''gdb --args %s ↩
    <args>''
```

utasítással futtatható, a "run" és a "bt" (backtrace) parancsok kiadásával történt. Bővebb információt a GDB használatáról NS-3 szimulációk során itt [18] található.

Az NS-3 biztosít egy általános tracing (nyomkövetési) lehetőséget is, mely a küldött/fogadott csomagokról rögzít adatokat (ASCII tracing). Az általam használt minta szimulációk generálnak egy `.tr` kiterjesztésű fájlt, melyből aztán az alábbihoz hasonló bejegyzések olvashatóak ki:

```
+ 1.0134
/NodeList/1/DeviceList/0/$ns3::CsmaNetDevice/TxQueue/Enqueue
  ns3::EthernetHeader (
    length/type=0x806,
    source=00:00:00:00:00:03,
    destination=00:00:00:00:00:01)
  ns3::ArpHeader ( reply
    source mac: 00-06-00:00:00:00:00:03
    source ipv4: 10.1.1.2
    dest mac: 00-06-00:00:00:00:00:01
    dest ipv4: 10.1.1.1)
  Payload (size=18)
  ns3::EthernetTrailer (fcs=0)

- 1.0134
/NodeList/1/DeviceList/0/$ns3::CsmaNetDevice/TxQueue/Dequeue
  ns3::EthernetHeader (
    length/type=0x806,
    source=00:00:00:00:00:03,
    destination=00:00:00:00:00:01)
  ns3::ArpHeader ( reply
    source mac: 00-06-00:00:00:00:00:03
    source ipv4: 10.1.1.2
    dest mac: 00-06-00:00:00:00:00:01
    dest ipv4: 10.1.1.1)
  Payload (size=18)
```

```
  ns3::EthernetTrailer (fcs=0)

r 1.0134
/NodeList/2/DeviceList/0/$ns3::CsmaNetDevice/MacRx
  ns3::EthernetHeader (
    length/type=0x806,
    source=00:00:00:00:00:01,
    destination=ff:ff:ff:ff:ff:ff)
  ns3::ArpHeader ( request
    source mac: 00-06-00:00:00:00:00:01
    source ipv4: 10.1.1.1
    dest ipv4: 10.1.1.2)
  Payload (size=18)
  ns3::EthernetTrailer (fcs=0)
```

A sorok elején található +, -, r, illetve d karakterek rendre a csomag kiküldését, érkezését, fogadását, illetve elejtését jelölik. Fel vannak tüntetve a különböző fejlécek, MAC- és IP címek, csomag mérete, szimulációs idő, a bejegyzést kiváltó node és annak típusa, valamint a művelet típusa.

Többet az ASCII tracingről a hivatalos útmutató [10] 5.3.1. szekciójában (is) lehet olvasni.

A szimulációimhoz pcap (packet capture) nyomkövetést is használtam, mellyel az egyes terminálok (állomások) tevékenységeit lehet megfigyelni. Böngészésükhöz *Wireshark* vagy *tcpdump* ajánlott, én ez utóbbit használtam.

E két féle nyomkövetést a szimulációban az alábbi sorokkal lehet bekapcsolni,

```
  AsciiTraceHelper ascii;
  csma.EnableAsciiAll (ascii.CreateFileStream ↩
     ("lookup-tables-test.tr"));

  csma.EnablePcapAll ("lookup-tables-test", false);
```

mely után a `.tr` és a `.pcap` kimeneti fájlok az NS-3 főmappájában jelennek meg.

Ellenőrzésképpen a Layer 2-es switch minta eredményeit össze is vetettem az OpenFlow switch eredményeivel; ugyan azt kaptam mindkét esetben. A következő ábrák a pcap mérések eredményeiből mutatnak egy-egy részletet.

```
$ tcpdump -nn -tt -r test-switch-l2-1-0.pcap
```

OpenFlow switch

```
reading from file openflow-switch-1-0.pcap, link-type EN10M
1.013396 ARP, Request who-has 10.1.1.2 (ff:ff:ff:ff:ff:ff) tell 10.1.1.1, length 50
1.013396 ARP, Reply 10.1.1.2 is-at 00:00:00:00:00:03, length 50
1.023389 IP 10.1.1.1.49153 > 10.1.1.2.9: UDP, length 512
1.026358 IP 10.1.1.1.49153 > 10.1.1.2.9: UDP, length 512
1.030361 IP 10.1.1.1.49153 > 10.1.1.2.9: UDP, length 512
1.038553 IP 10.1.1.1.49153 > 10.1.1.2.9: UDP, length 512
1.046745 IP 10.1.1.1.49153 > 10.1.1.2.9: UDP, length 512
1.054937 IP 10.1.1.1.49153 > 10.1.1.2.9: UDP, length 512
1.063129 IP 10.1.1.1.49153 > 10.1.1.2.9: UDP, length 512
1.071321 IP 10.1.1.1.49153 > 10.1.1.2.9: UDP, length 512
1.079513 IP 10.1.1.1.49153 > 10.1.1.2.9: UDP, length 512
1.087705 IP 10.1.1.1.49153 > 10.1.1.2.9: UDP, length 512
1.095897 IP 10.1.1.1.49153 > 10.1.1.2.9: UDP, length 512
1.104089 IP 10.1.1.1.49153 > 10.1.1.2.9: UDP, length 512
1.112281 IP 10.1.1.1.49153 > 10.1.1.2.9: UDP, length 512
1.115396 ARP, Request who-has 10.1.1.1 (ff:ff:ff:ff:ff:ff) tell 10.1.1.4, length 50
1.121539 IP 10.1.1.1.49153 > 10.1.1.2.9: UDP, length 512
1.128665 IP 10.1.1.1.49153 > 10.1.1.2.9: UDP, length 512
1.138227 IP 10.1.1.1.49153 > 10.1.1.2.9: UDP, length 512
1.147774 IP 10.1.1.1.49153 > 10.1.1.2.9: UDP, length 512
1.154646 IP 10.1.1.1.49153 > 10.1.1.2.9: UDP, length 512
1.161433 IP 10.1.1.1.49153 > 10.1.1.2.9: UDP, length 512
```

P4 switch

```
reading from file test-switch-l2-1-0.pcap, link-type EN10M
1.013396 ARP, Request who-has 10.1.1.2 (ff:ff:ff:ff:ff:ff) tell 10.1.1.1, length 50
1.013396 ARP, Reply 10.1.1.2 is-at 00:00:00:00:00:03, length 50
1.023389 IP 10.1.1.1.49153 > 10.1.1.2.9: UDP, length 512
1.026358 IP 10.1.1.1.49153 > 10.1.1.2.9: UDP, length 512
1.030361 IP 10.1.1.1.49153 > 10.1.1.2.9: UDP, length 512
1.038553 IP 10.1.1.1.49153 > 10.1.1.2.9: UDP, length 512
1.046745 IP 10.1.1.1.49153 > 10.1.1.2.9: UDP, length 512
1.054937 IP 10.1.1.1.49153 > 10.1.1.2.9: UDP, length 512
1.063129 IP 10.1.1.1.49153 > 10.1.1.2.9: UDP, length 512
1.071321 IP 10.1.1.1.49153 > 10.1.1.2.9: UDP, length 512
1.079513 IP 10.1.1.1.49153 > 10.1.1.2.9: UDP, length 512
1.087705 IP 10.1.1.1.49153 > 10.1.1.2.9: UDP, length 512
1.095897 IP 10.1.1.1.49153 > 10.1.1.2.9: UDP, length 512
1.104089 IP 10.1.1.1.49153 > 10.1.1.2.9: UDP, length 512
1.112281 IP 10.1.1.1.49153 > 10.1.1.2.9: UDP, length 512
1.115396 ARP, Request who-has 10.1.1.1 (ff:ff:ff:ff:ff:ff) tell 10.1.1.4, length 50
1.121539 IP 10.1.1.1.49153 > 10.1.1.2.9: UDP, length 512
1.128665 IP 10.1.1.1.49153 > 10.1.1.2.9: UDP, length 512
1.138227 IP 10.1.1.1.49153 > 10.1.1.2.9: UDP, length 512
1.147774 IP 10.1.1.1.49153 > 10.1.1.2.9: UDP, length 512
1.154646 IP 10.1.1.1.49153 > 10.1.1.2.9: UDP, length 512
1.161433 IP 10.1.1.1.49153 > 10.1.1.2.9: UDP, length 512
```

```
$ tcpdump -nn -tt -r test-switch-l2-3-0.pcap
```

```
l2-3-0.pcap
reading from file test-switch-l2-3-0.pcap, link-type EN10MB (Ethernet)
1.013396 ARP, Request who-has 10.1.1.2 (ff:ff:ff:ff:ff:ff) tell 10.1.1.1, length 50
1.111191 ARP, Request who-has 10.1.1.1 (ff:ff:ff:ff:ff:ff) tell 10.1.1.4, length 50
1.123231 ARP, Reply 10.1.1.1 is-at 00:00:00:00:00:01, length 50
1.123231 IP 10.1.1.4.49153 > 10.1.1.1.9: UDP, length 512
1.124144 IP 10.1.1.4.49153 > 10.1.1.1.9: UDP, length 512
1.127346 IP 10.1.1.4.49153 > 10.1.1.1.9: UDP, length 512
1.132767 IP 10.1.1.4.49153 > 10.1.1.1.9: UDP, length 512
1.140959 IP 10.1.1.4.49153 > 10.1.1.1.9: UDP, length 512
1.149151 IP 10.1.1.4.49153 > 10.1.1.1.9: UDP, length 512
1.157343 IP 10.1.1.4.49153 > 10.1.1.1.9: UDP, length 512
1.165535 IP 10.1.1.4.49153 > 10.1.1.1.9: UDP, length 512
1.173727 IP 10.1.1.4.49153 > 10.1.1.1.9: UDP, length 512
1.181919 IP 10.1.1.4.49153 > 10.1.1.1.9: UDP, length 512
1.190111 IP 10.1.1.4.49153 > 10.1.1.1.9: UDP, length 512
1.198303 IP 10.1.1.4.49153 > 10.1.1.1.9: UDP, length 512
```

```
$ tcpdump -nn -tt -r test-switch-l2-0-0.pcap
```

```
l2-0-0.pcap
reading from file test-switch-l2-0-0.pcap, link-type EN10MB (Ethernet)
1.009191 ARP, Request who-has 10.1.1.2 (ff:ff:ff:ff:ff:ff) tell 10.1.1.1, length 50
1.017602 ARP, Reply 10.1.1.2 is-at 00:00:00:00:00:03, length 50
1.017602 IP 10.1.1.1.49153 > 10.1.1.2.9: UDP, length 512
1.018515 IP 10.1.1.1.49153 > 10.1.1.2.9: UDP, length 512
1.024575 IP 10.1.1.1.49153 > 10.1.1.2.9: UDP, length 512
1.032767 IP 10.1.1.1.49153 > 10.1.1.2.9: UDP, length 512
1.040959 IP 10.1.1.1.49153 > 10.1.1.2.9: UDP, length 512
1.049151 IP 10.1.1.1.49153 > 10.1.1.2.9: UDP, length 512
1.057343 IP 10.1.1.1.49153 > 10.1.1.2.9: UDP, length 512
1.065535 IP 10.1.1.1.49153 > 10.1.1.2.9: UDP, length 512
1.073727 IP 10.1.1.1.49153 > 10.1.1.2.9: UDP, length 512
1.081919 IP 10.1.1.1.49153 > 10.1.1.2.9: UDP, length 512
1.090111 IP 10.1.1.1.49153 > 10.1.1.2.9: UDP, length 512
1.098303 IP 10.1.1.1.49153 > 10.1.1.2.9: UDP, length 512
1.106495 IP 10.1.1.1.49153 > 10.1.1.2.9: UDP, length 512
1.114687 IP 10.1.1.1.49153 > 10.1.1.2.9: UDP, length 512
1.115396 ARP, Request who-has 10.1.1.1 (ff:ff:ff:ff:ff:ff) tell 10.1.1.4, length 50
1.116665 ARP, Reply 10.1.1.1 is-at 00:00:00:00:00:01, length 50
1.122879 IP 10.1.1.1.49153 > 10.1.1.2.9: UDP, length 512
1.129018 IP 10.1.1.4.49153 > 10.1.1.1.9: UDP, length 512
```

3.4.1. Logging

Az NS-3 szimulációs fájljaimban a `-v` illetve a `-verbose` kapcsolókkal lehet engedélyezni a naplózást. A P4SwitchNetDevice írása során megadott logging üzenetek (például `NS_LOG_INFO("Returned from HandlePacket without error.");`) ilyenkor az alapértelmezett outputra kiíródnak. A naplózás azonban csak a szimuláció ns3 névteren belüli részére vonatkozik csupán; a switch C-ben

íródott részénél már csupán `printf` segítségével írattam ki sorokat. (Ez azt jelenti hogy kényelmesen ez a rész nem kapcsolható ki- és be, a p4_interface.c állományban kommentezni kell.)

A logging kezelését a waf fordító konfigurálásával lehet elérni; a naplózás meg lehet szorítani különböző komponensekre (pl csak a P4SwitchNetDevice-ra), illetve különböző naplózási szintek (info, debug, warning, error) is léteznek.

Egy lehetséges futtatási módja a szimulációmnak tehát:

```
$ export NS_LOG=*=level_info
$ ./waf --run "scratch/lookup-table-tests --v" > log.out 2>&1
```

A "`> log.out 2>&1`" rész a naplózás külön állományba való irányításáról szól, melyet én legtöbbször elhagytam, mivel az NS-3 beépített naplózása és a printf művelet nincs szinkronban; a sorok kiíráskor elcsúsznak, ha a P4 interfészen belülről is ki szeretnék íratni valamit.

3.4.2. L2 switch

Az L2 switch egy olyan kapcsoló, mely a 2. hálózati rétegen végez forgalomirányítást; a csomagok továbbítását a MAC címek vizsgálatával teszi. A szimulációs környezetben ahol ezt teszteltem (`test-switch-l2.cc`), a switch-csem négy terminált kapcsol össze. Csomagküldés az 1. terminálból a 2.-ba, valamint a 4. terminálból az elsőbe történik, melyeket először egy-egy ARP-kérés előz meg. A csomagokat periodikusan küldöm egy meghatározott intervallumon belül.

A szimulációban használt P4 switch két kapcsoló-táblát vezet, melyek az `smac` és `dmac` nevet viselik. Az előbbiben a csomag bejövő MAC címe a kulcs, míg az utóbbiban a kimenő MAC címet használja kulcsként. Mindkét tábla *exact* típusú. Az `smac` táblában értékként egyszerűen csak a végrehajtandó feladat (action) azonosítóját tárolja, míg a `dmac` az action-azonodítón kívül egy portszámot is tárol; ezen keresztül kell a csomagot továbbküldeni majd. Az actionök négyfélék lehetnek: cím megtanulása (ilyenkor lefut a digest callback függvény), csomag továbbítása, skip, illetve broadcast.

Konklúzió: az L2-es switch használata a szimulációban hiba nélkül, megfelelően működik.

OpenFlow switch

```
Create nodes.
Build Topology
RegisterProtocolHandler for ns3::CsmaNe
RegisterProtocolHandler for ns3::CsmaNe
RegisterProtocolHandler for ns3::CsmaNe
RegisterProtocolHandler for ns3::CsmaNe
Assign IP Addresses.
Create Applications.
Configure Tracing.
Run Simulation.
---------------------------------------
UID is 1
Received packet from 00:00:00:00:00:01
Creating Openflow buffer from packet.
Parsed EthernetHeader
Parsed ArpHeader
Matching against the flow table.
Flow not matched.
Sending packet to controller
Setting to flood; this packet is a broa
Added new flow.
Flooding over ports.
Sending packet 1 over port 1
Sending packet 1 over port 2
Sending packet 1 over port 3
Learned that 00:00:00:00:00:01 can be f
---------------------------------------
UID is 3
Received packet from 00:00:00:00:00:03
Creating Openflow buffer from packet.
Parsed EthernetHeader
Parsed ArpHeader
Matching against the flow table.
Flow not matched.
Sending packet to controller
Added new flow.
Sending packet 3 over port 0
Learned that 00:00:00:00:00:03 can be f
---------------------------------------
UID is 0
Received packet from 00:00:00:00:00:01
Creating Openflow buffer from packet.
Parsed EthernetHeader
Parsed Ipv4Header
Parsed UdpHeader
Matching against the flow table.
Flow not matched.
Sending packet to controller
Added new flow.
Sending packet 0 over port 1
---------------------------------------
UID is 5
Received packet from 00:00:00:00:00:01
Creating Openflow buffer from packet.
Parsed EthernetHeader
Parsed Ipv4Header
Parsed UdpHeader
Matching against the flow table.
Flow matched
Sending packet 5 over port 1
```

P4 switch

```
Create nodes, 4+1.
Build Topology.
Installing switch device on node 4.
Setting properties for table 0.
Setting properties for table 1.
Adding SwitchPort 0x25b6ab0.
RegisterProtocolHandler for ns3::CsmaNetDevice
Adding SwitchPort 0x25bca90.
RegisterProtocolHandler for ns3::CsmaNetDevice
Adding SwitchPort 0x25ba8a0.
RegisterProtocolHandler for ns3::CsmaNetDevice
Adding SwitchPort 0x25750e0.
RegisterProtocolHandler for ns3::CsmaNetDevice
Setting callback functions.
Assign IP Addresses.
Create Applications.
Configure Tracing.
Run Simulation.
-----------------------------------------
UID is 1
Received packet from 00:00:00:00:00:01 looking for ff:ff:ff:ff:ff:ff
Creating P4 buffer from packet.
Handling packet arriving at port 0.
  :::: executing table smac
   ::: exact_lookup didn't find match for key [0,0,0,0,0,1]
    :: executing action mac_learn...
     : Learned that port 0 belongs to MAC address 00:00:00:00:00:01
  :::: executing table dmac
   ::: exact_lookup was successful with value 3 and key [255,255,255,2
    :: executing action bcast...
(broadcast) Sending packet 1 over port 1
(broadcast) Sending packet 1 over port 2
(broadcast) Sending packet 1 over port 3
-----------------------------------------
UID is 3
Received packet from 00:00:00:00:00:03 looking for 00:00:00:00:00:01
Creating P4 buffer from packet.
Handling packet arriving at port 1.
  :::: executing table smac
   ::: exact_lookup didn't find match for key [0,0,0,0,0,3]
    :: executing action mac_learn...
     : Learned that port 1 belongs to MAC address 00:00:00:00:00:03
  :::: executing table dmac
   ::: exact_lookup was successful with value 2 and key [0,0,0,0,0,1]
    :: executing action forward...
Sending packet 3 over port 0.
-----------------------------------------
UID is 0
Received packet from 00:00:00:00:00:01 looking for 00:00:00:00:00:03
Creating P4 buffer from packet.
Handling packet arriving at port 0.
  :::: executing table smac
   ::: exact_lookup was successful with value 1 and key [0,0,0,0,0,1]
    :: executing action _nop...
  :::: executing table dmac
   ::: exact_lookup was successful with value 2 and key [0,0,0,0,0,3]
    :: executing action forward...
Sending packet 0 over port 1.
-----------------------------------------
UID is 5
```

⋮ ⋮

```
Parsed Ipv4Header
Parsed UdpHeader
Matching against the flow table.
Flow matched
Sending packet 89 over port 0
---------------------------------------
UID is 90
Received packet from 00:00:00:00:00:01
Creating Openflow buffer from packet.
Parsed EthernetHeader
Parsed Ipv4Header
Parsed UdpHeader
Matching against the flow table.
Flow matched
Sending packet 90 over port 1
---------------------------------------
UID is 91
Received packet from 00:00:00:00:00:07
Creating Openflow buffer from packet.
Parsed EthernetHeader
Parsed Ipv4Header
Parsed UdpHeader
Matching against the flow table.
Flow matched
Sending packet 91 over port 0
Done.
```

```
-----------------------------------------
UID is 90
Received packet from 00:00:00:00:00:01 looking for 00:00:00:00:00:03
Creating P4 buffer from packet.
Handling packet arriving at port 0.
  :::: executing table smac
   ::: exact_lookup was successful with value 1 and key [0,0,0,0,0,1]
    :: executing action _nop...
  :::: executing table dmac
   ::: exact_lookup was successful with value 2 and key [0,0,0,0,0,3]
    :: executing action forward...
Sending packet 90 over port 1.
-----------------------------------------
UID is 91
Received packet from 00:00:00:00:00:07 looking for 00:00:00:00:00:01
Creating P4 buffer from packet.
Handling packet arriving at port 3.
  :::: executing table smac
   ::: exact_lookup was successful with value 1 and key [0,0,0,0,0,7]
    :: executing action _nop...
  :::: executing table dmac
   ::: exact_lookup was successful with value 2 and key [0,0,0,0,0,1]
    :: executing action forward...
Sending packet 91 over port 0.
Destructing tables.
Done.
```

3.4.3. L3 switch

Az L3 swtich egy olyan kapcsoló, mely a 3. hálózati rétegen végez forgalomirányítást; a csomagok továbbítását az IP címek vizsgálatával teszi. A `test-switch-l3.cc` tartalmazza a tesztelésre használt szimulációmat.

Ez az L3 switch is két kapcsoló-táblát vezet, melyek az `ipv4_fib_lpm` illetve a `sendout` nevet viselik. Az előbbi "lpm", míg a második "exact" típusú tábla. Az `ipv4_fib_lpm` IPv4 címet használ kulcsként, értékként pedig kimenő MAC cím és port párokat tárol a végrehajtandó action azonosítója mellett. A `sendout` pedig a kimeneti port számát használja kulcsként, értékként pedig az action azonosítója mellett a bejövő MAC címet tárolja. Az actionök háromfélék lehetnek: on miss (nincs találatt a kulcsra), next hop (ugrás a következőre) és rewrite smac (forrás MAC cím átírása).

Az IP címeket (a tábla-inicializálásnál) előre feltöltöttem a `ipv4_fib_lpm` táblába, azok alapján dönti el a switch, merre kell a csomagokat továbbítania. A szimulációmnál első körben gondot okozott, hogy a P4 switchnek nem volt olyan actionje, mely lekezelte volna az ARP kéréseket. Két megoldás merült fel ennek kiküszöbölésére: az egyikhez a P4 switch-cset kellett volna kiegészíteni egy ARP reflectorral (ami az ARP reqestben küldött csomagokat feldolgozza és visszaküldi, mintha ténylegesen elérkezett volna a címzetthez és az válaszolt volna). Én a másik opciót valósítottam meg végül, kiegészítve a szimulációt egy "ARP cache feltöltővel". Lényegében egy listába szedtem a terminálok (Node-ok) címét, majd ezt odaadtam mindegyiknek, mintha mindig is ismerték volna a többi Node-ot.

Ez a szimuláció különösen hasznos volt abból a szempontból, hogy felmerültek olyan problémák amik létezéséről L2-es tesztelés során nem értesültem. Módosítanom kellett a BufferFromPacket osztályt, ugyanis kiderült, hogy az ns3::Packet nem a megszokott csomagfelépítést követi; az IP fejléc másmilyen pozíción áll mint egy hagyományos hálózati csomagnál. Ráadásul az EtherType (NS3-as *protocol* változó) fordított bájtsorrendben szerepel, így ezt is javítanom kellett. Ezek kiküszöbölése után azonban sikeresen lefutott a szimuláció.

Konklúzió: az L3-as forgalomirányítás is megfelelően működik.

3.4.4. Kapcsoló-tábla műveletek

Az interfész különböző kapcsoló-tábla műveleteinek működését egy új szimulációs állományban, a `lookup-table-test.cc`-ben teszteltem. Habár olyan L2-es P4 switchet használtam, mely *exact* típusú táblákkal dolgozik, a `table_entry_t` szerkezetének köszönhetően *lpm* és *ternary* műveleteket is tesztelni tudtam különösebb módosítások nélkül.

- EXACT műveletek

  Add: Teszteltem új rekord hozzáadását a táblához: üres táblába, a tábla (azaz entry-lista) legvégére, a tábla legelejére, illetve meglévő rekordok közé. Teszteltem ugyan annak a bejegyzésnek többszöri hozzáadását a táblához (felülírja-e a meglévő értéket).

  Lookup: Ellenőriztem hogy az új rekordok tényleg sorba vannak-e rendezve, meg lehet-e őket találni. Ellenőriztem, hogy alapértelmezett értéket kapok-e vissza abban az esetben, ha a kulcshoz nem tartozott bejegyzés.

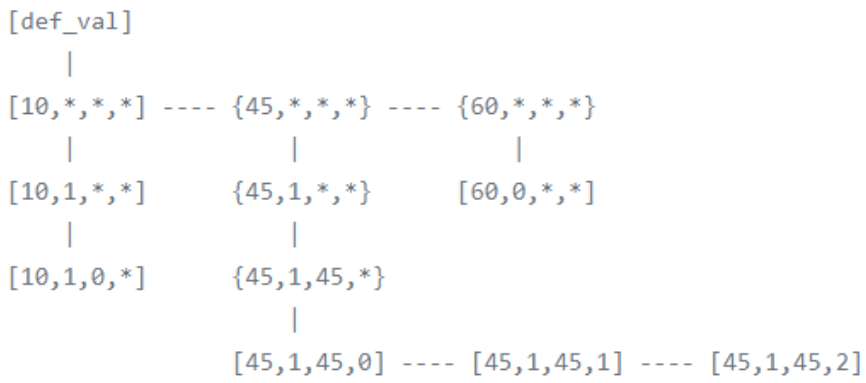
  Remove: Teszteltem a rekordok eltávolítását a tábla legelejéről, legvégéről, közepéről, egy elemű táblából, üres táblából, illetve olyan kulccsal való eltávolítás amelyhez nem tartozott bejegyzés.

- LPM műveletek

  Add: Teszteltem új rekord hozzáadását a táblához: üres táblába, a tábla (fa szerkezet) legaljára illetve legtetejére való beillesztést, ugyan azon kulcs különböző mélységen való használatát. Teszteltem ugyan annak a bejegyzésnek többszöri hozzáadását a táblához (felülírja-e a meglévő értéket), illetve olyan bejegyzés hozzáadását amelynél a rekord már létezett, csak az érték nem volt még megadva.

  Lookup: Ellenőriztem hogy a hozzáadás során tényleg rendezetten lettek-e a rekordok eltárolva. Teszteltem olyan értékek felkeresését melyre nem talált bejegyzést (alapértelmezett értékkel tér vissza) illetve ahol el volt tárolva kulcs, de csak egy felsőbb szinten. Ellenőriztem azt, hogy a lookup során a legjobban illeszkedő rekord értékével tér-e vissza (hiszen az LPM lookup lényege: az alsóbb szinten lévő bejegyzések felülírják a felettük lévőket, azaz mindig a leghosszabb prefix-egyezéssel tér vissza).

A `lookup-table-test.cc`-ben felépített teszt-fa így fog kinézni (a maszkot * jelöli, az üres rekord { }-ben van, míg az értékkel rendelkező bejegyzés [ ]-ben).



Remove: Teszteltem, hogy az értékeket a megadott kulcs és mélység függvényében megfelelően távolítja-e el, illetve hogy nem létező kulcs esetén nem csinál semmit sem.

- TERNARY műveletek

  Az tesztek az EXACT táblához hasonlóak voltak, azzal a kivétellel, hogy a sorrendet nem kellett figyelembe venni. Ellenőriztem emellett azt is, hogy a maszkokat megfelelően kezelték-e a műveletek, vagyis a lefedett kulcsrészleteket nem vették figyelembe.

- Táblatörlés

  Az adott táblát törölni lehet a `free_entries` függvénnyel, bármilyen típusú tábláról is volt szó. Ez nem egy kifejezetten a felhasználó számára íródott függvény, de használható. Nem szabad azonban elfelejteni, hogy a táblák további használathoz a táblához tartozó mutató (`t->table` pointer) értékét NULL-ra kell állítani.

3.5. Fájlok elérhetősége

A dolgozatomhoz tartozó fájlok (P4SwitchNetDevice modul, P4 interfésszel) elérhetőek a github tárolómról [19] is. Ugyan itt található egy L2 illetve egy L3 minta switch is (P4 fájlok és generált fájlok egyaránt), valamint a hozzájuk tartozó minta szimulációk, és a teszteléshez használt teszt szimuláció is.

Ez a project nyílt forráskódú alapokra épülő open source project.

4. Hivatkozásjegyzék

[1] P4 konzorcium hivatalos honlapja
`https://p4.org`
Elérés dátuma: 2018.04.20.

[2] NS-3 hálózat szimulátor hivatalos honlapja
`https://www.nsnam.org/`
Elérés dátuma: 2018.03.08.

[3] NS-3 telepítési előfeltételei
`https://www.nsnam.org/wiki/Installation`
Elérés dátuma: 2018.03.08.

[4] NS-3 telepítési útmutatója
`https://www.nsnam.org/docs/tutorial/html/getting-started.html`
Elérés dátuma: 2018.03.08.

[5] ELTE P4 compiler tárolója
`https://github.com/P4ELTE/t4p4s`
Elérés dátuma: 2018.03.08.

[6] DPDK hivatalos oldala
`http://dpdk.org`
Elérés dátuma: 2018.03.08.

[7] P4@ELTE
`http://p4.elte.hu/`
Elérés dátuma: 2018.03.08.

[8] Segítség új modul létrehozásához NS-3-ban
`https://www.nsnam.org/docs/manual/html/new-modules.html`
Elérés dátuma: 2018.03.10.

[9] NS-3 szimuláció futattásához segítség
`https://www.nsnam.org/docs/tutorial/html/getting-started.html#running-a-script`
NS-3 minta szimuláció
`https://www.nsnam.org/docs/tutorial/html/conceptual-overview.html#a-first-ns-3-script`
Elérések dátuma: 2018.03.10.

[10] NS-3 tutorial (pdf)
https://www.nsnam.org/docs/release/3.26/tutorial/ns-3-tutorial.pdf
Elérés dátuma: 2018.03.17.

[11] NS-3 dokumentáció OpenFlow switch-hez tartozó része
https://www.nsnam.org/docs/release/3.26/models/html/openflow-switch.html
Elérés dátuma: 2018.03.17.

[12] NS-3 modul kiegészítése külső könyvtárakkal

- http://shieldroute.blogspot.hu/2012/08/extending-ns3-with-your-module-and.html
- https://www.nsnam.org/wiki/HOWTO_build_old_versions_of_ns-3_on_newer_compilers
- https://groups.google.com/forum/#!topic/ns-3-users/Ue8EUatLDok
- https://garfield001.wordpress.com/2013/04/16/compile-and-link-ns3-program-with-external-library/
- https://jiaziyi.com/making-use-of-external-libraries-in-ns3/

Elérések dátuma: 2018.03.23.

[13] A osztálydiagrammot és folyamatábrákat a Software Ideas Modeler segítségével készítettem:
https://www.softwareideas.net/
Elérés dátuma: 2018.03.27.

[14] A struktogramokat Nagy Sándor Tibor oldala segítségével állítottam össze:
develop.stuki.hu
Elérés dátuma: 2018.03.27.

[15] Uthash hash-tábla
https://troydhanson.github.io/uthash/
Elérés dátuma: 2018.03.27.

[16] Judy adatstruktúra
http://judy.sourceforge.net/
Elérés dátuma: 2018.03.27.

[17] Judy adatstruktúra használata P4-es BMv2-val
`https://github.com/p4lang/behavioral-model/tree/master/src/bf_lpm_trie`
Elérés dátuma: 2018.03.27.

[18] GDB debugger használata
`https://www.nsnam.org/wiki/HOWTO_use_gdb_to_debug_program_errors`
Elérés dátuma: 2018.04.23.

[19] Szakdolgozatom fájljait tartalmazó GitHub tároló
`https://github.com/mrosan/P4SwitchNetDevice`
Elérés dátuma: 2018.04.25.